Суперкомпиляция

Обобщение, зацикливание, гомеоморфизм

Илья Ключников Сергей Романенко

Институт прикладной математики им.М.В.Келдыша РАН

Ноябрь 2009

План доклада

Суперкомпиляция

Язык SLL

Прогонка – “символическое” вычисление

Зацикливание

Обобщение

Гомеоморфизм

Контроль – глобальный и локальный

Происхождение суперкомпиляции

Автор идеи: *Валентин Фёдорович Турчин*

`http://pespmc1.vub.ac.be/TURCHIN.html`)

Происхождение суперкомпиляции

- Турчин В.Ф. 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.
- Турчин В.Ф. Эквивалентные преобразования программ на РЕФАЛе. Автоматизированная система управления строительством. Труды ЦНИПИАСС, N 6. ЦНИПИАСС. Москва, 1974. Стр. 36-68.
- Базисный Рефал и его реализация на вычислительных машинах. М.: ЦНИПИАСС, 1977. - 258 с. Стр. 92-95.

Происхождение суперкомпиляции

- V.F.Turchin. The Language Refal – The Theory of Compilation and Metasystem Analysis. Curant Institute of Mathematics. Technical report #018, NY, 1980.
- V.F.Turchin. The concept of a supercompiler. ACM Transactions on Programming Languages and Systems, 8, pp.292-325, 1986.
- V.F.Turchin. Program transformation with metasystem transitions. Journal of Functional Programming 3(3) pp.283-313, July 1993.
- V.F.Turchin. Supercompilation: Techniques and Results. In: Perspectives of System Informatics, pp. 229-248, LNCS, vol 1181, Springer, 1996.
- V.F.Turchin. Metacomputation: Metasystem Transitions plus Supercompilation. In: Partial Evaluation, pp. 481-510, LNCS, vol. 1110, Springer, 1996.

Сущность суперкомпиляции

Чем занимается суперкомпилятор

- На входе: (P, C), где P - исходная программа, а C - ограничения на условия эксплуатации P.
- На выходе: P' - остаточная программа, эквивалентная P (*при условиях* C).
- Цель: убрать избыточность из P (при условиях C).
- Надежды и ожидания:
  - P' будет эффективнее, чем P.
  - P' будет меньше, чем P.
  - P' будет проще, чем P.
- Реальность: ожидания иногда сбываются. :-)

Что даёт суперкомпиляция

Какие “упрощения” могут получаться

- Устранение бесполезных частей программы.
- Выполнение в процессе суперкомпиляции тех операций, исходные даные для которых (достаточно хорошо) известны во время суперкомпиляции.
- Превращение многопроходных алгоритмов в однопроходные.
- Удаление промежуточных структур данных.

Возможные применения

- Оптимизация программ.
- Анализ программ.

Как устроена суперкомпиляция

Составные части суперкомпиляции

- Прогонка. Символическое исполнение программы с отслеживанием всех возможных вариантов трассы вычислений. Результат - бесконечное “дерево процесса”. В каждой вершине - “конфигурация”, описывающая целый класс возможных состояний вычисления.
- Обобщение и зацикливание. “Подрезание” дерева и превращение его в конечный (вообще говоря) циклический “граф конфигураций”. При этом может потребоваться *обобщение* конфигураций.
- Генерация остаточной программы из графа конфигураций.

Частный случай: “позитивная” суперкомпиляция

“Позитив”: конечный набор случаев

$x = 25$

$x \in \{25, 100\}$

“Негатив”: бесконечный набор случаев

$x \neq 25$

$x \notin \{25, 100\}$

Построение трасс вычислений (прогонка)

- “Позитив” – запоминаем.
- “Негатив” – забываем.

За и против позитивной суперкомпиляции

- $\Uparrow$ Легче реализовать, легче объяснить.
- $\Downarrow$ Потеря точности, грубее результаты.

Простота – причина популярности:

- M.H.Sørensen. Turchin's Supercompiler Revisited. Master's Thesis. Department of Computer Science, University of Copenhagen. 1994. DIKU-rapport 94/17.
- M.H.Sørensen, R.Glück, and N.D.Jones. A positive supercompiler. J. Funct. Prog., 6:811-838, 1996.
- M.H.Sørensen and R.Glück. An algorithm of generalization in positive supercompilation. In *J.W. Lloyd, editor, Logic Programming: Proceedings of the 1995 International Symposium*, pages 465–479. MIT Press, 1995.
- M.H.Sørensen. Convergence of program transformers in the metric space of trees. Sci. Comput. Program. 37, 1-3 (May. 2000), 163-205.

“Учебный” суперкомпилятор SPSC

SPSC = A Simple Supercompiler

Для простого “ленивого” функционального языка.

Теоретическая основа

M.H.Sørensen. Convergence of program transformers in the metric space of trees. Sci. Comput. Program. 37, 1-3 (May. 2000), 163-205.

Можно употреблять как

- “Наглядное пособие” по принципам суперкомпиляции.
- “Подопытного кролика” для экспериментов.

Чем хорош

- Прост и понятен (по сравнению с некоторыми другими). :-)
- Выложен в Google Code: http://spsc.googlecode.com/
- Доступен через веб-интерфейс: http://spsc.appspot.com/

SLL: простой “ленивый” язык

- SLL = A Simple Lazy Language.
- SLL – входной/выходной язык суперкомпилятора SPSC.
- Данные: выражения (“термы”) из “конструкторов”.
- Программа: набор “правил редукции”.

Пример данных: представление натуральных чисел:

0	1	2	3	...
`Z`	`S(Z)`	`S(S(Z))`	`S(S(S(Z)))`	...

Пример программы: сложение и умножение

`add(Z,y) = y;`	$0 + y = y$
`add(S(x),y) = S(add(x,y));`	$(x + 1) + y = (x + y) + 1$
`mult(Z,y) = Z;`	$0 * y = 0$
`mult(S(x),y) = add(mult(x,y),y);`	$(x + 1) * y = x * y + y$
`square(x) = mult(x,x);`	$x^2 = x * z$

SLL: Пример вычисления

Программа

```
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
double(x) = add(x, add(x, Z));
```

Вычисление $(1 + (1 + 0))$ — "редукция"

"ленивое" (**SLL**, Haskell)

```
double(S(Z))
add(S(Z), add(S(Z), Z))
S(add(Z, add(S(Z), Z)))
S(add(S(Z), Z))
S(S(add(Z, Z)))
S(S(Z))
```

"строгое" (**Refal**, Standard ML, C)

```
double(S(Z))
add(S(Z), add(S(Z), Z))
add(S(Z), S(add(Z, Z))
add(S(Z), S(Z))
S(add(Z, S(Z)))
S(S(Z))
```

SLL: Пример вычисления

Программа

```
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
double(x) = add(x, add(x, Z));
```

Вычисление $(1 + (1 + 0))$ — "редукция"

"ленивое" (**SLL**, Haskell)

```
double(S(Z))
add(S(Z), add(S(Z), Z))
S(add(Z, add(S(Z), Z)))
S(add(S(Z), Z))
S(S(add(Z, Z)))
S(S(Z))
```

"строгое" (**Refal**, Standard ML, C)

```
double(S(Z))
add(S(Z), add(S(Z), Z))
add(S(Z), S(add(Z, Z))
add(S(Z), S(Z))
S(add(Z, S(Z)))
S(S(Z))
```

SLL: Пример вычисления

Программа

```
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
double(x) = add(x, add(x, Z));
```

Вычисление $(1 + (1 + 0))$ — "редукция"

"ленивое" (**SLL**, Haskell)

```
double(S(Z))
add(S(Z), add(S(Z), Z))
S(add(Z, add(S(Z), Z)))
S(add(S(Z), Z))
S(S(add(Z, Z)))
S(S(Z))
```

"строгое" (**Refal**, Standard ML, C)

```
double(S(Z))
add(S(Z), add(S(Z), Z))
add(S(Z), S(add(Z, Z))
add(S(Z), S(Z))
S(add(Z, S(Z)))
S(S(Z))
```

SLL: Пример вычисления

Программа

```
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
double(x) = add(x, add(x, Z));
```

Вычисление $(1 + (1 + 0))$ — "редукция"

"ленивое" (**SLL**, Haskell)

```
double(S(Z))
add(S(Z), add(S(Z), Z))
S(add(Z, add(S(Z), Z)))
S(add(S(Z), Z))
S(S(add(Z, Z)))
S(S(Z))
```

"строгое" (**Refal**, Standard ML, C)

```
double(S(Z))
add(S(Z), add(S(Z), Z))
add(S(Z), S(add(Z, Z))
add(S(Z), S(Z))
S(add(Z, S(Z)))
S(S(Z))
```

SLL: Пример вычисления

Программа

```
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
double(x) = add(x, add(x, Z));
```

Вычисление $(1 + (1 + 0))$ — "редукция"

"ленивое" (**SLL**, Haskell)

```
double(S(Z))
add(S(Z), add(S(Z), Z))
S(add(Z, add(S(Z), Z)))
S(add(S(Z), Z))
S(S(add(Z, Z)))
S(S(Z))
```

"строгое" (**Refal**, Standard ML, C)

```
double(S(Z))
add(S(Z), add(S(Z), Z))
add(S(Z), S(add(Z, Z))
add(S(Z), S(Z))
S(add(Z, S(Z)))
S(S(Z))
```

SLL: Пример вычисления

Программа

```
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
double(x) = add(x, add(x, Z));
```

Вычисление $(1 + (1 + 0))$ — "редукция"

"ленивое" (**SLL**, Haskell)

```
double(S(Z))
add(S(Z), add(S(Z), Z))
S(add(Z, add(S(Z), Z)))
S(add(S(Z), Z))
S(S(add(Z, Z)))
S(S(Z))
```

"строгое" (**Refal**, Standard ML, C)

```
double(S(Z))
add(S(Z), add(S(Z), Z))
add(S(Z), S(add(Z, Z))
add(S(Z), S(Z))
S(add(Z, S(Z)))
S(S(Z))
```

SLL: синтаксис программ

$prog ::= d_1 \ \ldots \ d_n$	program
$d ::= f(x_1, \ \ldots \ , x_n) = \mathrm{e};$	f-function
$\mid g(p_1, x_1, \ \ldots \ , x_n) = e_1;$	g-function
$\ldots$	
$g(p_m, x_1, \ \ldots \ , x_n) = e_m;$	
$e ::= x$	variable
$\mid C(e_1, \ \ldots \ , e_n)$	constructor
$\mid f(e_1, \ \ldots \ , e_n)$	call to f-function
$\mid g(e_1, \ \ldots \ , e_n)$	call to g-function
$p ::= C(v_1, \ \ldots \ , v_n)$	pattern

Прогонка = “символическое” вычисление

(иногда переменные “не мешают”)

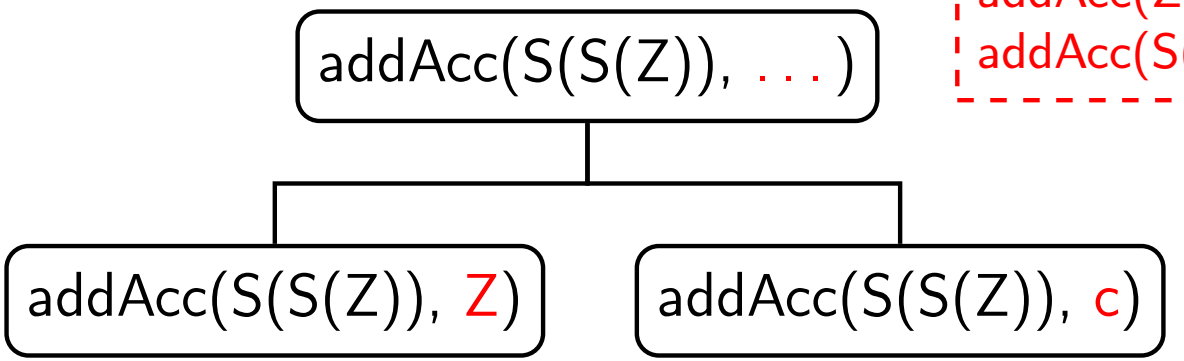
addAcc(S(S(Z)), . . .)

addAcc(Z, y) = y;
addAcc(S(x), y) = addAcc(x, S(y));

Прогонка = “символическое” вычисление

(иногда переменные “не мешают”)

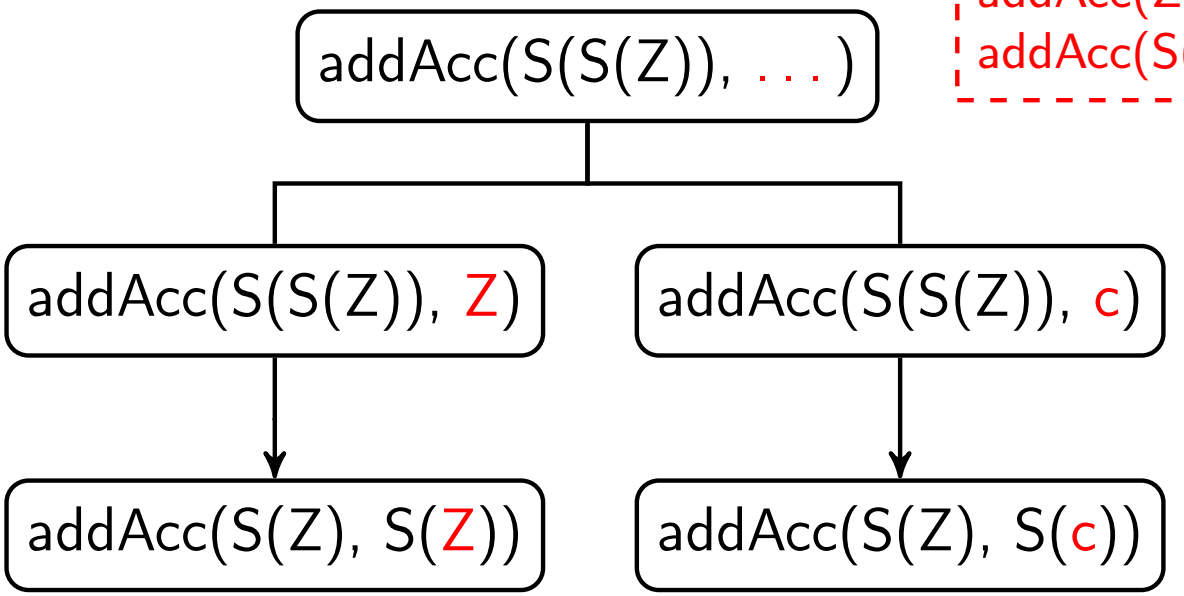
addAcc(Z, y) = y;
addAcc(S(x), y) = addAcc(x, S(y));



Прогонка = “символическое” вычисление

(иногда переменные “не мешают”)

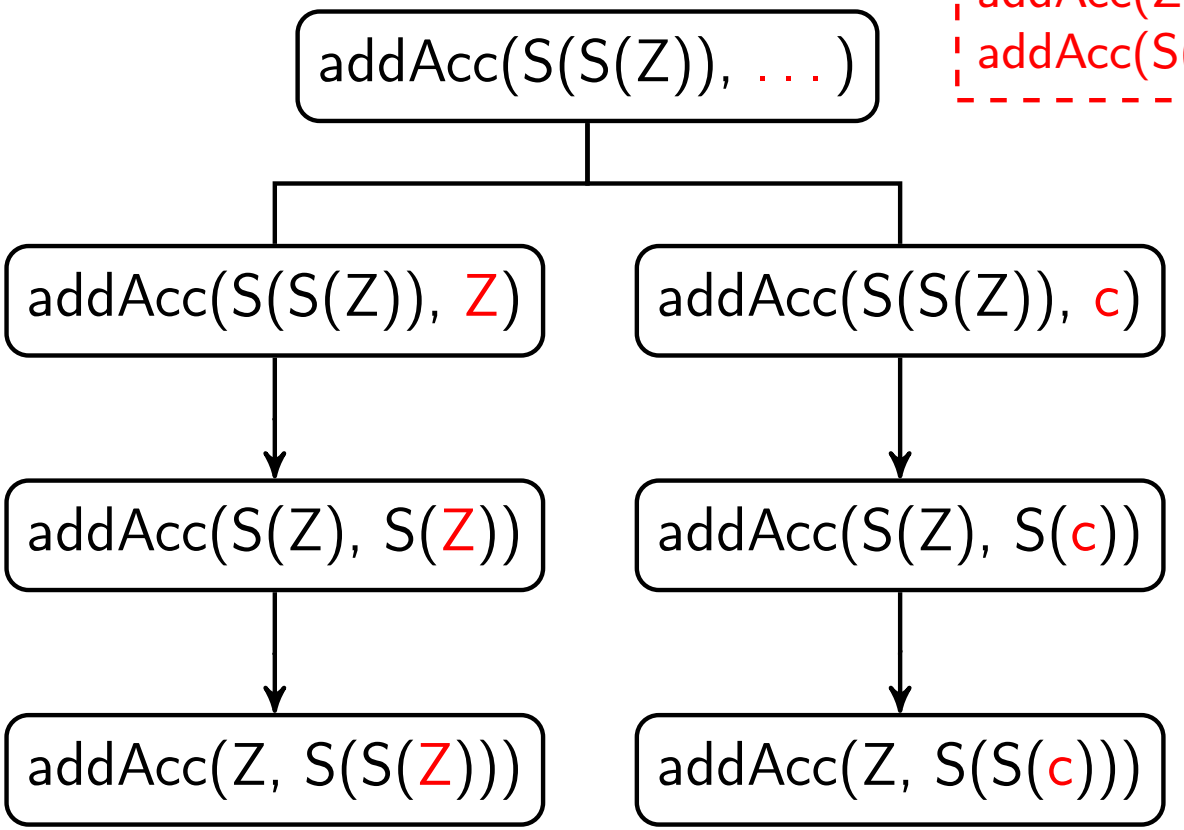
addAcc(Z, y) = y;
addAcc(S(x), y) = addAcc(x, S(y));



Прогонка = "символическое" вычисление

(иногда переменные "не мешают")

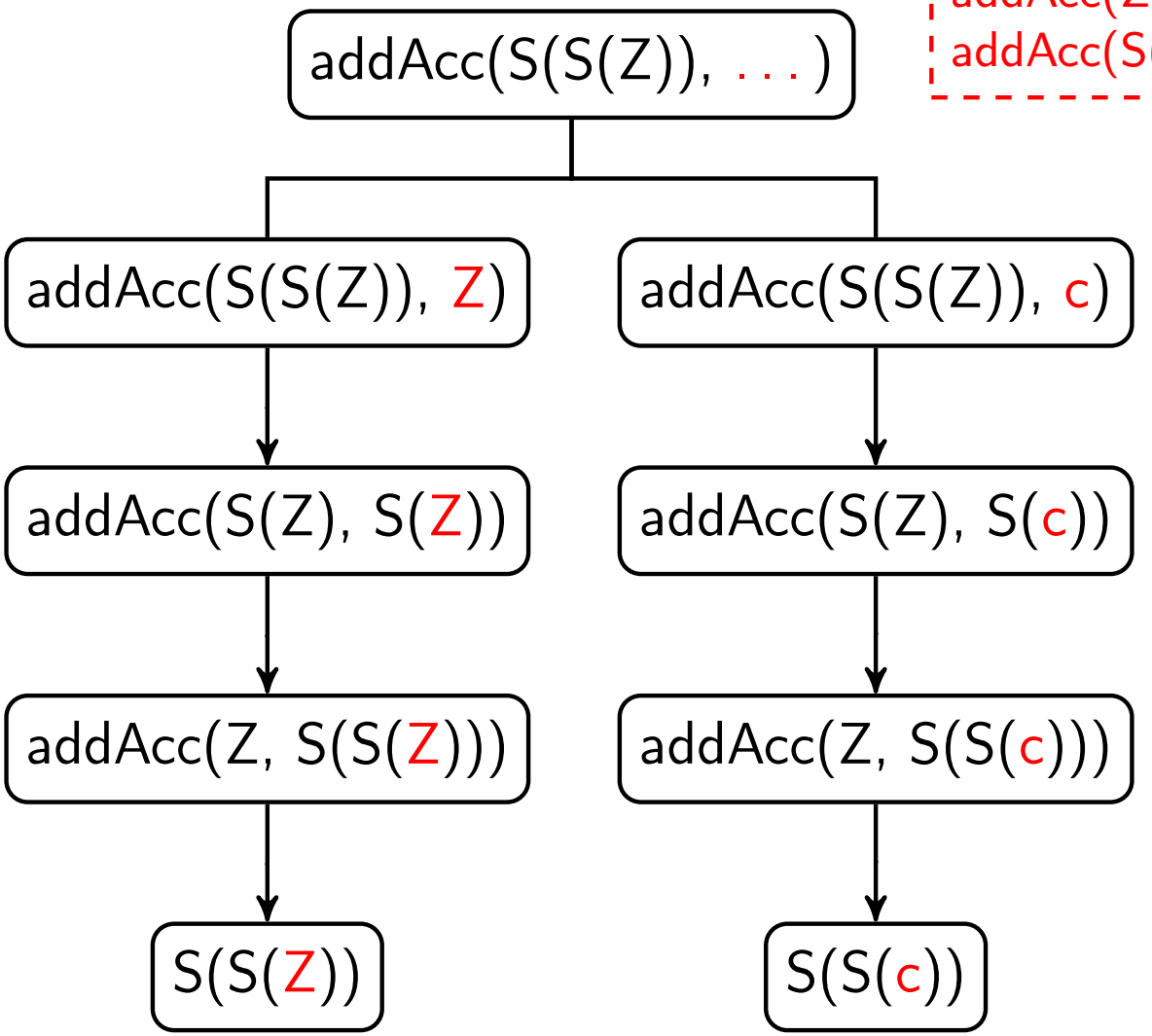
addAcc(Z, y) = y;
addAcc(S(x), y) = addAcc(x, S(y));



Прогонка = "символическое" вычисление

(иногда переменные "не мешают")

addAcc(Z, y) = y;
addAcc(S(x), y) = addAcc(x, S(y));



Прогонка = “символическое” вычисление

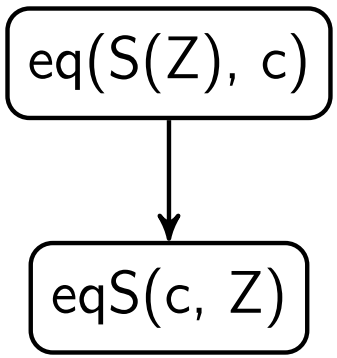
(переменные “мешают” – разбор случаев)

eq(S(Z), c)

```
eq(Z, y) = eqZ(y);
eq(S(x), y) = eqS(y, x);
eqZ(Z) = True;
eqZ(S(y)) = False;
eqS(Z, x) = False;
eqS(S(y), x) = eq(x, y);
```

Прогонка = “символическое” вычисление

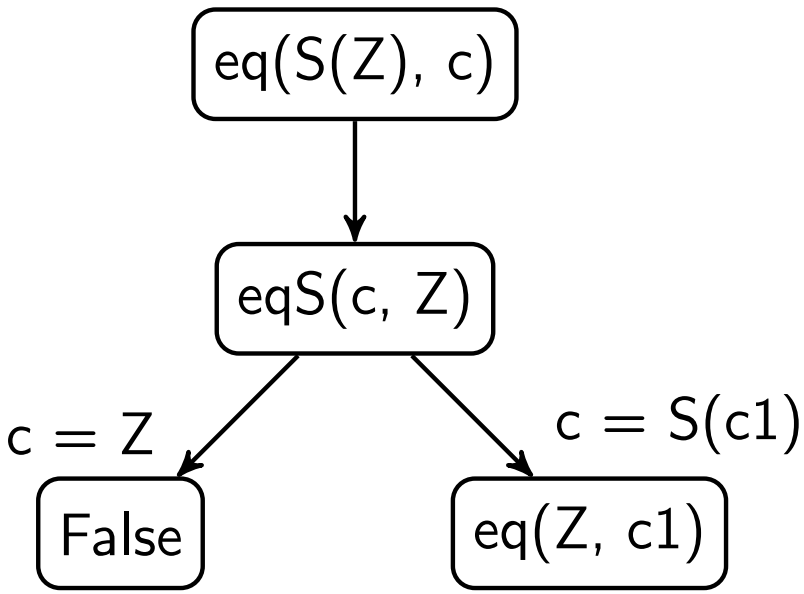
(переменные “мешают” – разбор случаев)




```
eq(Z, y) = eqZ(y);
eq(S(x), y) = eqS(y, x);
eqZ(Z) = True;
eqZ(S(y)) = False;
eqS(Z, x) = False;
eqS(S(y), x) = eq(x, y);
```

Прогонка = “символическое” вычисление

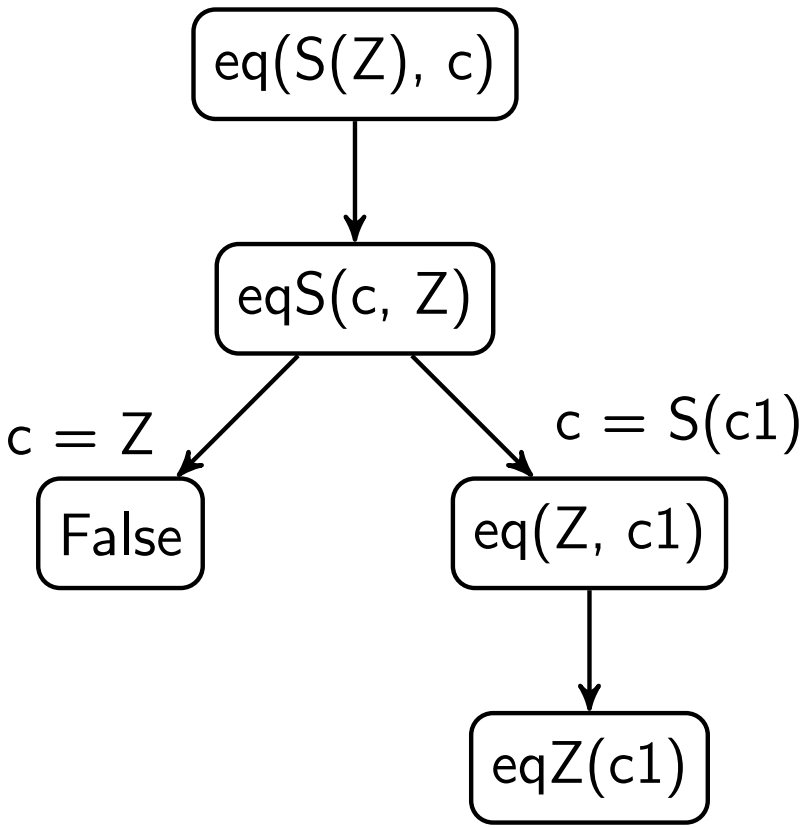
(переменные “мешают” – разбор случаев)




```
eq(Z, y) = eqZ(y);
eq(S(x), y) = eqS(y, x);
eqZ(Z) = True;
eqZ(S(y)) = False;
eqS(Z, x) = False;
eqS(S(y), x) = eq(x, y);
```

Прогонка = “символическое” вычисление

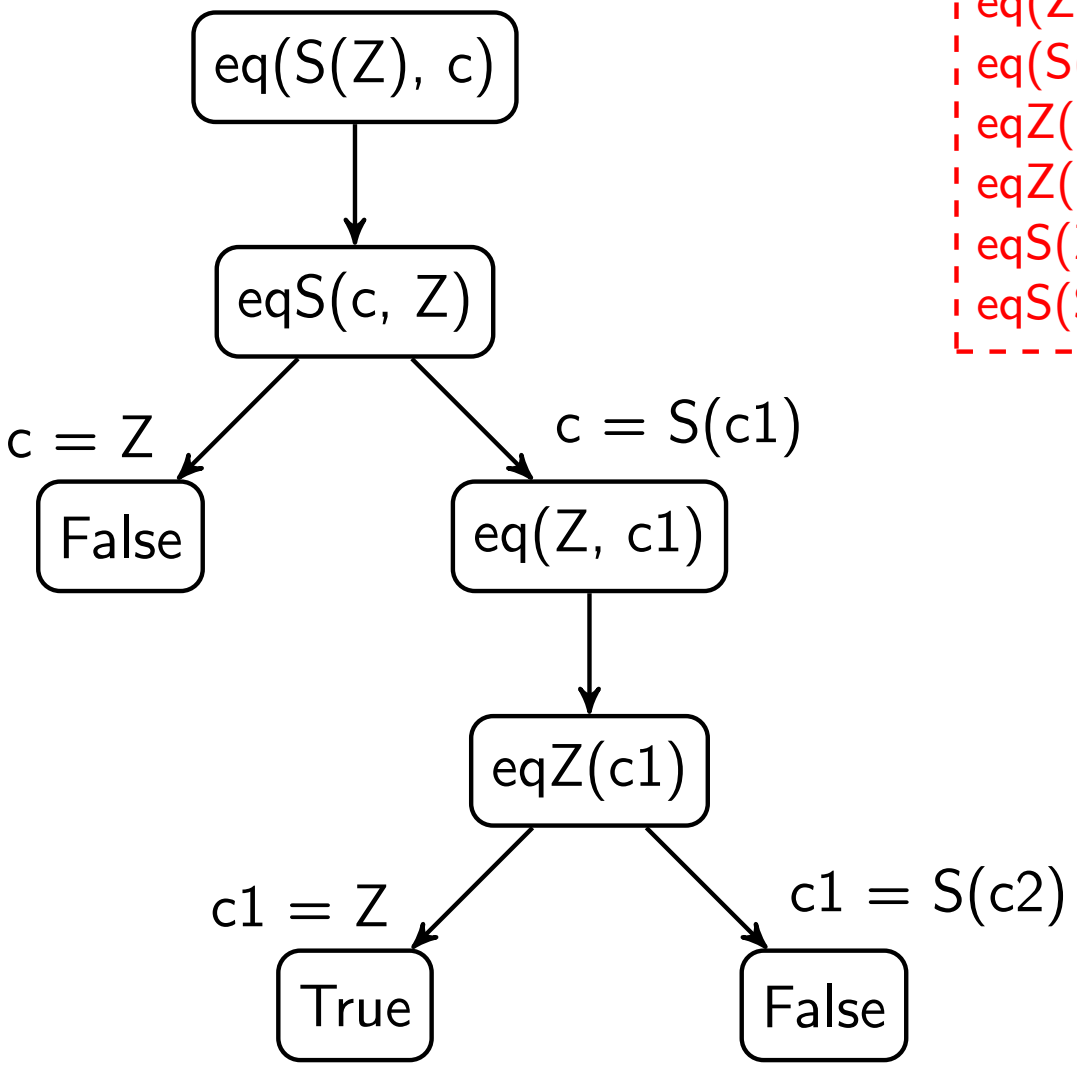
(переменные “мешают” – разбор случаев)




```
eq(Z, y) = eqZ(y);
eq(S(x), y) = eqS(y, x);
eqZ(Z) = True;
eqZ(S(y)) = False;
eqS(Z, x) = False;
eqS(S(y), x) = eq(x, y);
```

Прогонка = “символическое” вычисление

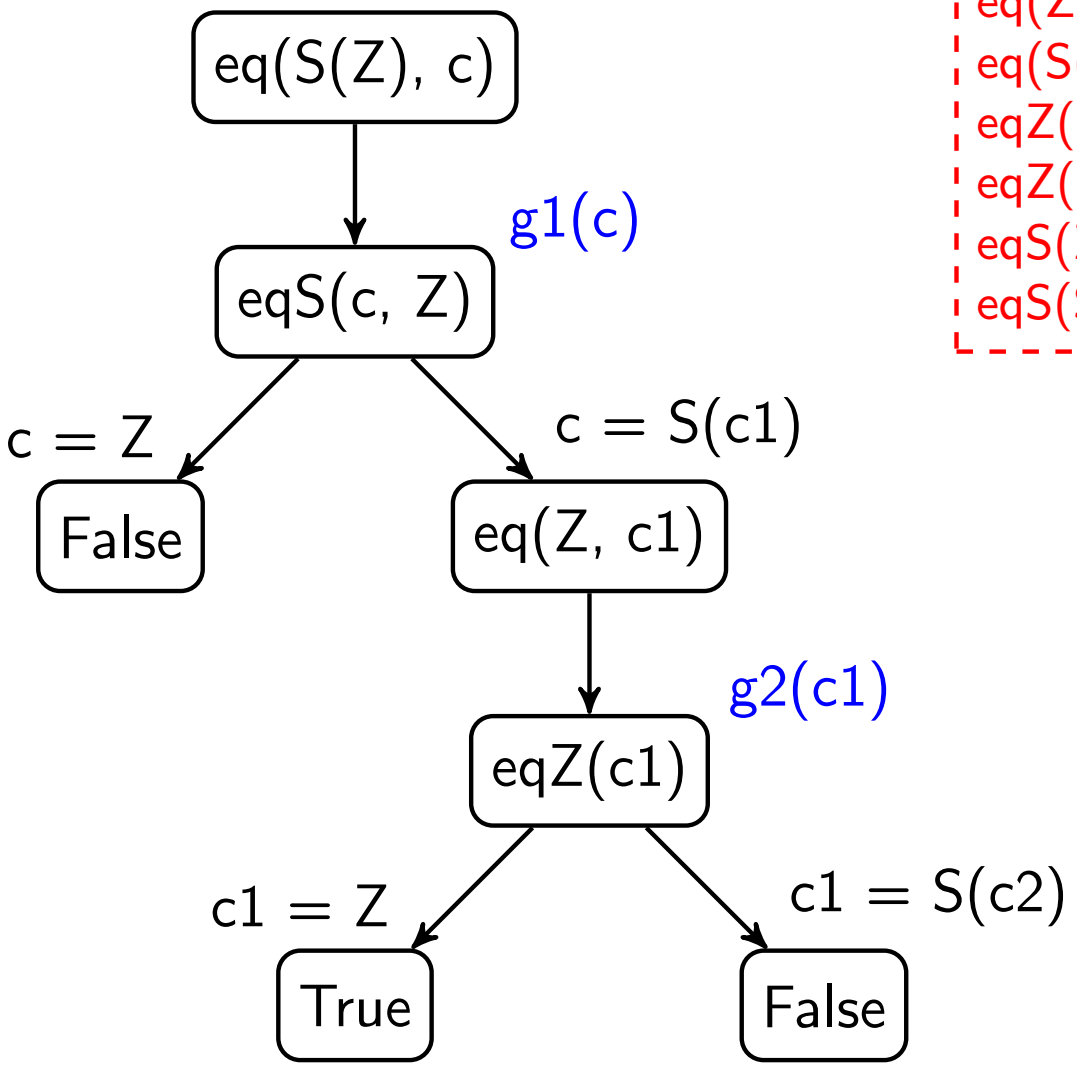
(переменные “мешают” – разбор случаев)




```
eq(Z, y) = eqZ(y);
eq(S(x), y) = eqS(y, x);
eqZ(Z) = True;
eqZ(S(y)) = False;
eqS(Z, x) = False;
eqS(S(y), x) = eq(x, y);
```

Прогонка = "символическое" вычисление

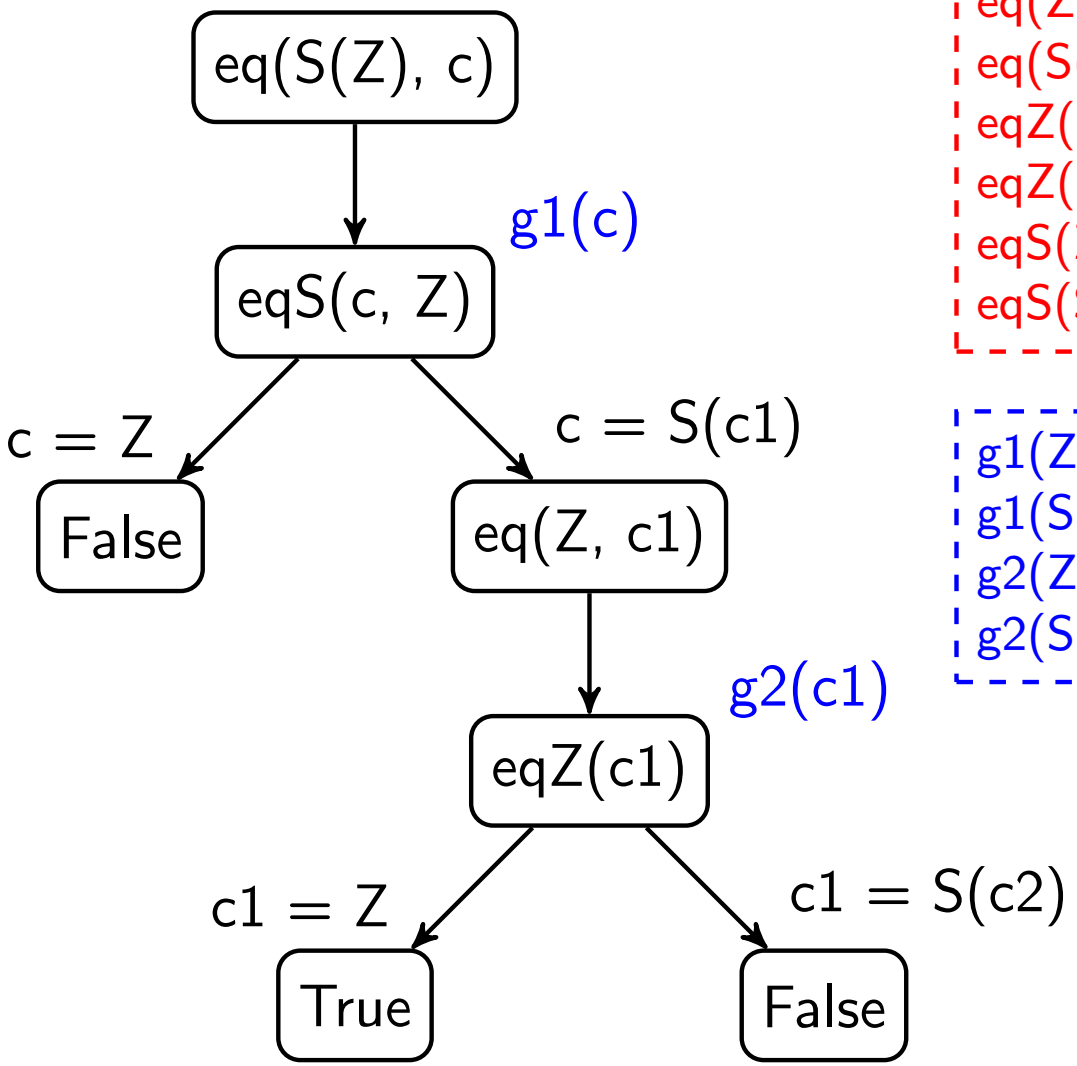
(переменные "мешают" – разбор случаев)




```
eq(Z, y) = eqZ(y);
eq(S(x), y) = eqS(y, x);
eqZ(Z) = True;
eqZ(S(y)) = False;
eqS(Z, x) = False;
eqS(S(y), x) = eq(x, y);
```

Прогонка = "символическое" вычисление

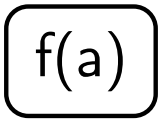
(переменные "мешают" – разбор случаев)




```
eq(Z, y) = eqZ(y);
eq(S(x), y) = eqS(y, x);
eqZ(Z) = True;
eqZ(S(y)) = False;
eqS(Z, x) = False;
eqS(S(y), x) = eq(x, y);
```

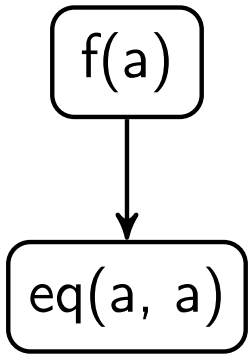
```
g1(Z) = False;
g1(S(c1)) = g2(c1);
g2(Z) = True;
g2(S(c2)) = False;
```

Зацикливание: бесконечное дерево ⇒ конечный циклический граф



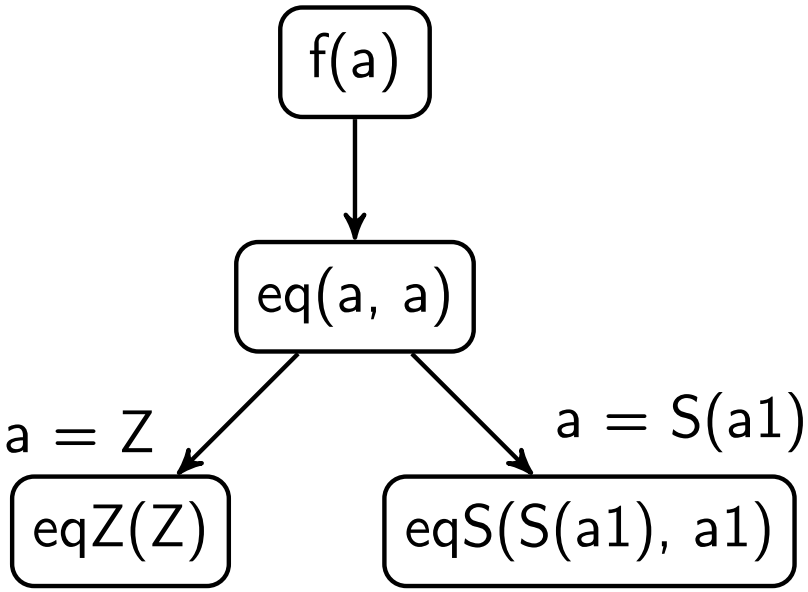

```
f(x) = eq(x, x);
eq(Z, y) = eqZ(y);
eq(S(x), y) = eqS(y, x);
eqZ(Z) = True;
eqZ(S(y)) = False;
eqS(Z, x) = False;
eqS(S(y), x) = eq(x, y);
```

Зацикливание: бесконечное дерево ⇒ конечный циклический граф



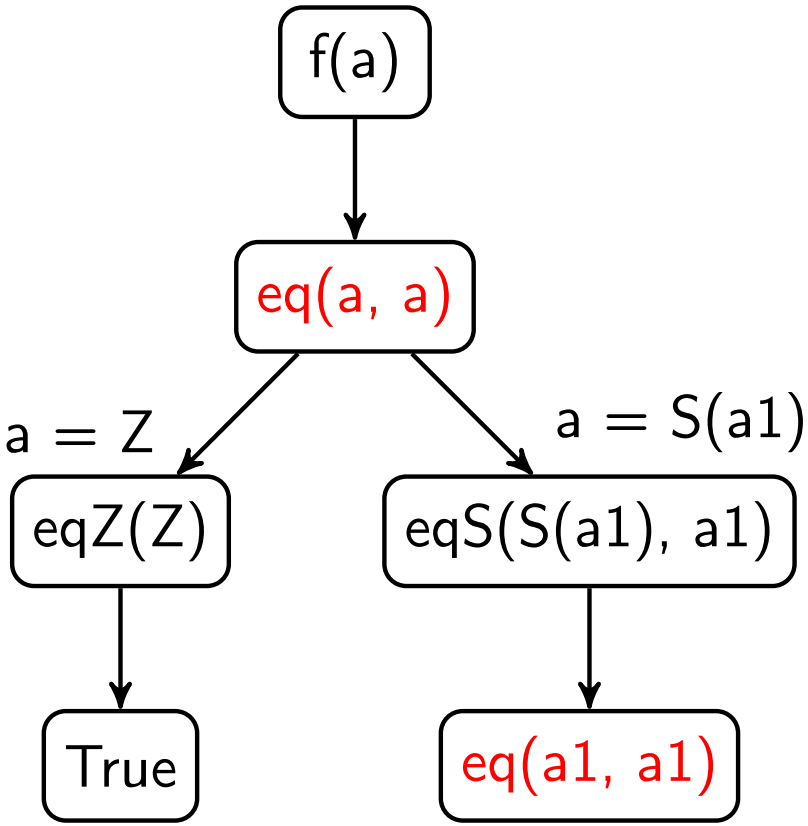

```
f(x) = eq(x, x);
eq(Z, y) = eqZ(y);
eq(S(x), y) = eqS(y, x);
eqZ(Z) = True;
eqZ(S(y)) = False;
eqS(Z, x) = False;
eqS(S(y), x) = eq(x, y);
```

Зацикливание: бесконечное дерево ⇒ конечный циклический граф



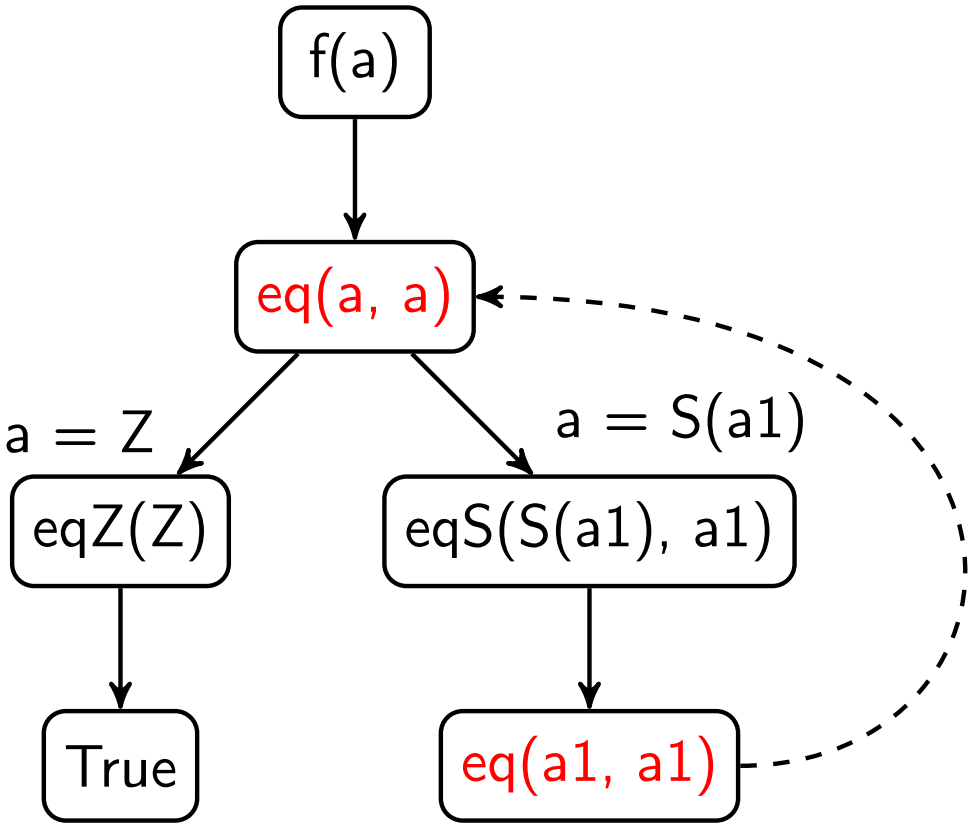

```
f(x) = eq(x, x);
eq(Z, y) = eqZ(y);
eq(S(x), y) = eqS(y, x);
eqZ(Z) = True;
eqZ(S(y)) = False;
eqS(Z, x) = False;
eqS(S(y), x) = eq(x, y);
```

Зацикливание: бесконечное дерево ⇒ конечный циклический граф



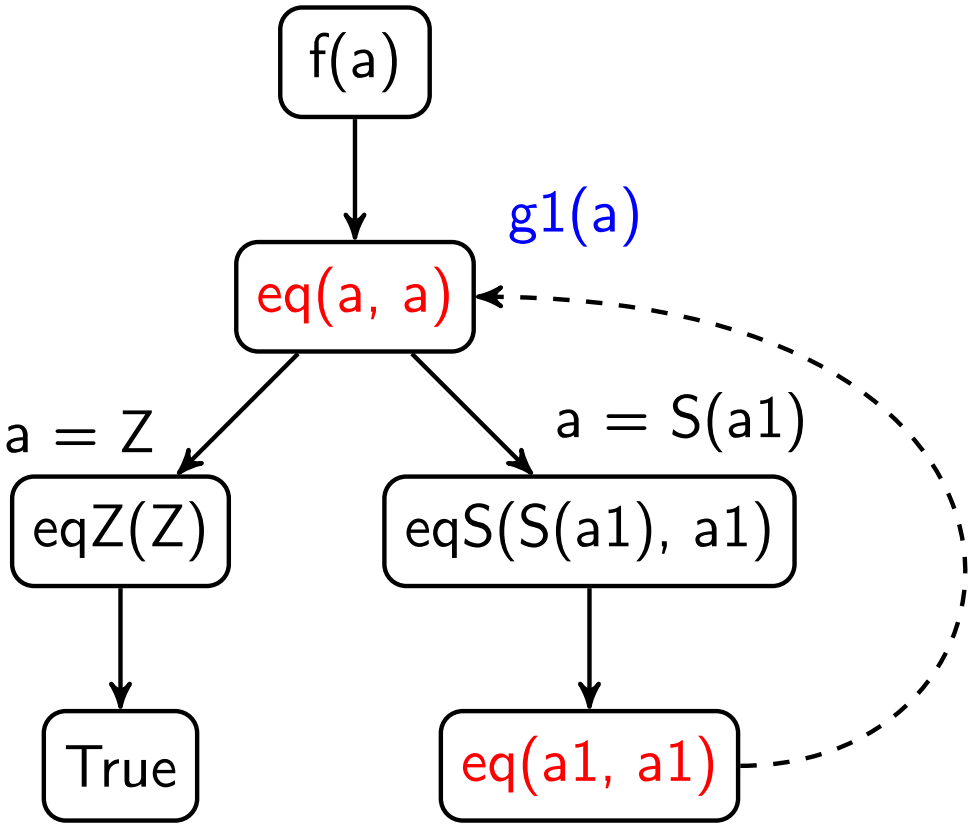

```
f(x) = eq(x, x);
eq(Z, y) = eqZ(y);
eq(S(x), y) = eqS(y, x);
eqZ(Z) = True;
eqZ(S(y)) = False;
eqS(Z, x) = False;
eqS(S(y), x) = eq(x, y);
```

Зацикливание: бесконечное дерево ⇒ конечный циклический граф



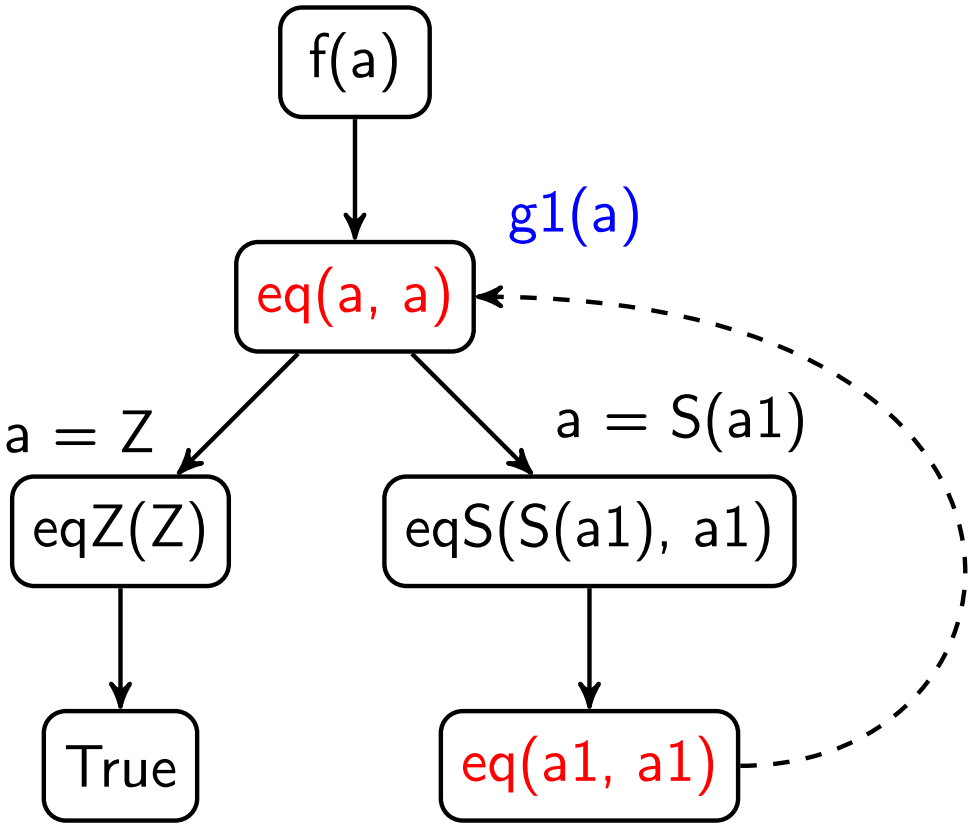

```
f(x) = eq(x, x);
eq(Z, y) = eqZ(y);
eq(S(x), y) = eqS(y, x);
eqZ(Z) = True;
eqZ(S(y)) = False;
eqS(Z, x) = False;
eqS(S(y), x) = eq(x, y);
```

Зацикливание: бесконечное дерево ⇒ конечный циклический граф




```
f(x) = eq(x, x);
eq(Z, y) = eqZ(y);
eq(S(x), y) = eqS(y, x);
eqZ(Z) = True;
eqZ(S(y)) = False;
eqS(Z, x) = False;
eqS(S(y), x) = eq(x, y);
```

Зацикливание: бесконечное дерево ⇒ конечный циклический граф




```
f(x) = eq(x, x);
eq(Z, y) = eqZ(y);
eq(S(x), y) = eqS(y, x);
eqZ(Z) = True;
eqZ(S(y)) = False;
eqS(Z, x) = False;
eqS(S(y), x) = eq(x, y);
```

```
f(a) = g1(a);
g1(Z) = True;
g1(S(a1)) = g1(a1);
```

- `eq(a, a)` не может выдать `False`.
- если `a` - конечное число, `eq(a, a)` завершается и выдаёт `True`.

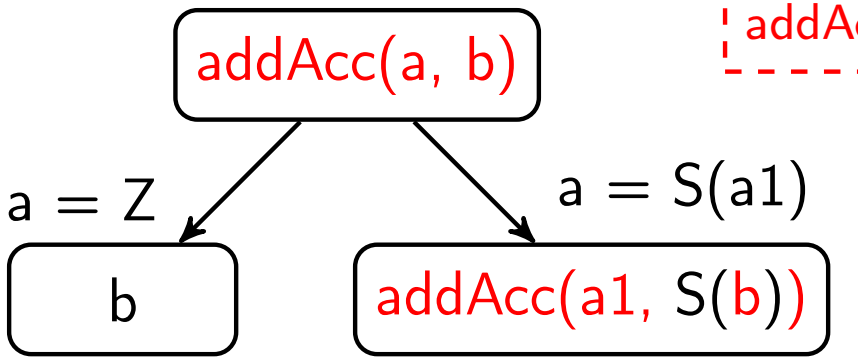
addAcc(a, b): Прогонка – бесконечное дерево

addAcc(a, b)

addAcc(Z, y) = y;
addAcc(S(x), y) = addAcc(x, S(y));

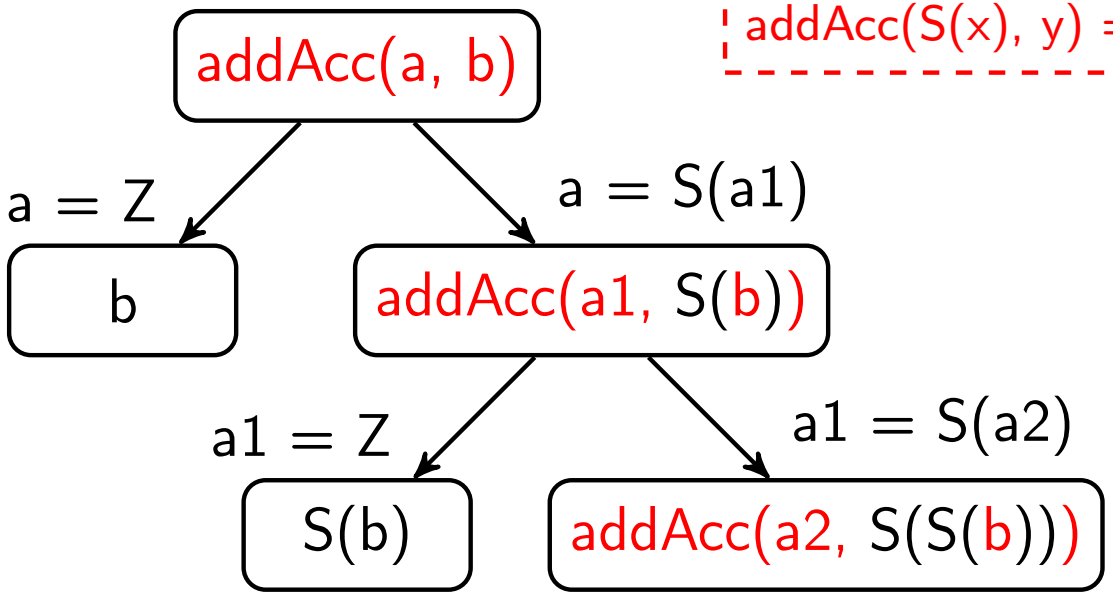
addAcc(a, b): Прогонка – бесконечное дерево

```
addAcc(Z, y) = y;
addAcc(S(x), y) = addAcc(x, S(y));
```



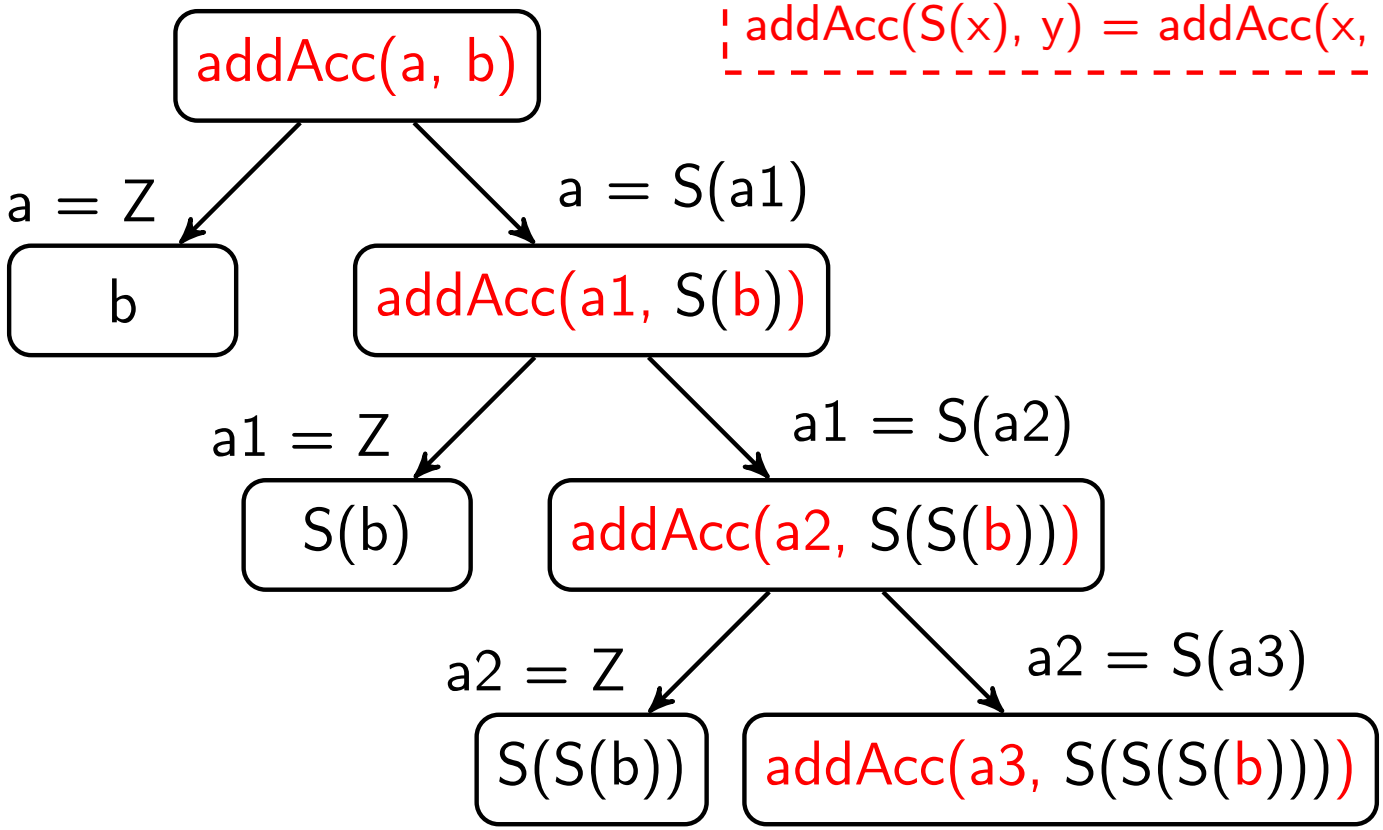
addAcc(a, b): Прогонка – бесконечное дерево

addAcc(Z, y) = y;
addAcc(S(x), y) = addAcc(x, S(y));

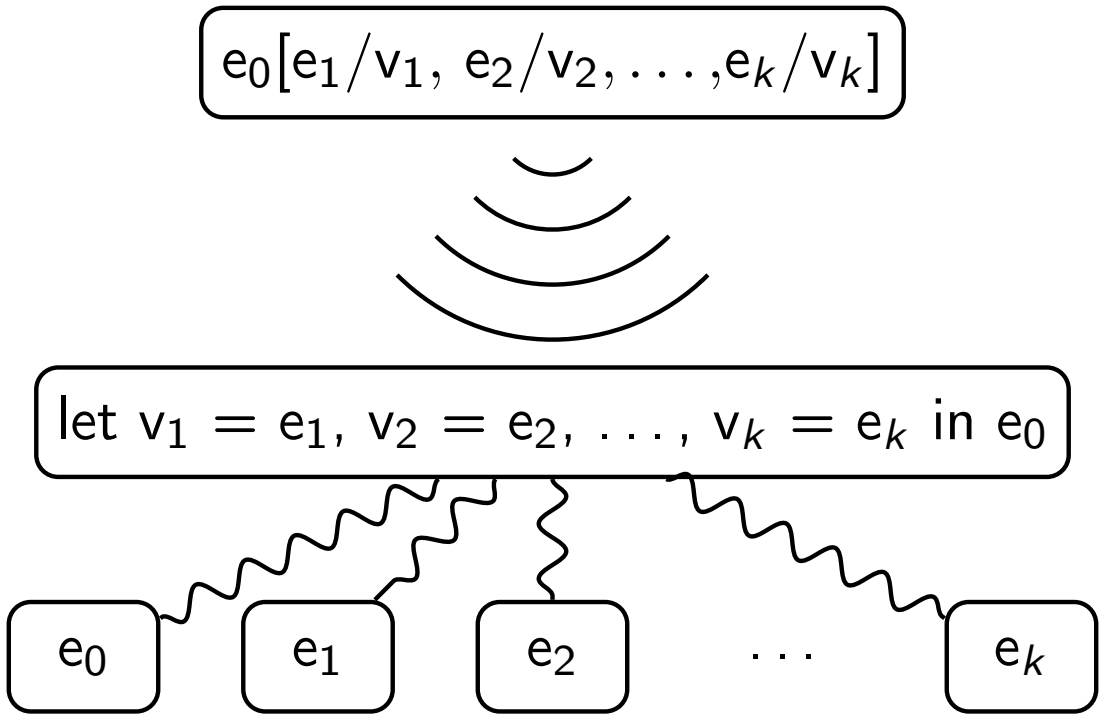


addAcc(a, b): Прогонка – бесконечное дерево

addAcc(Z, y) = y;
addAcc(S(x), y) = addAcc(x, S(y));



Декомпозиция конфигурации: let



- Подменяем узел на let-узел.
- Разбираем let-узел на куски.
- Каждый кусок суперкомпилируем отдельно.

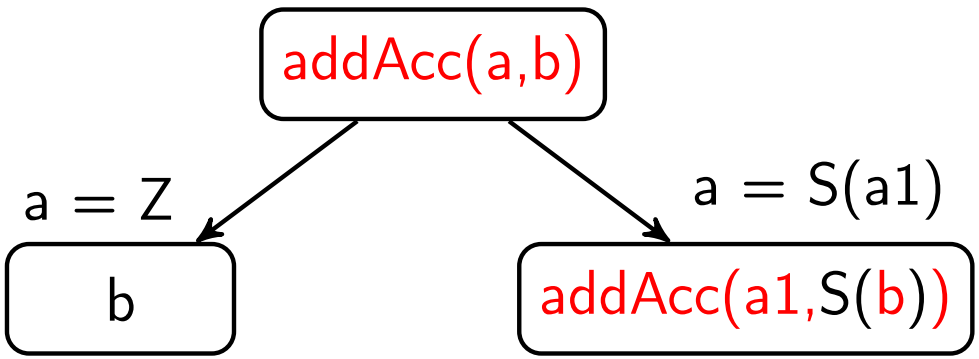
addAcc(a, b): Обобщение и зацикливание

```
addAcc(Z, y) = y;
addAcc(S(x), y) = addAcc(x, S(y));
```

addAcc(a,b)

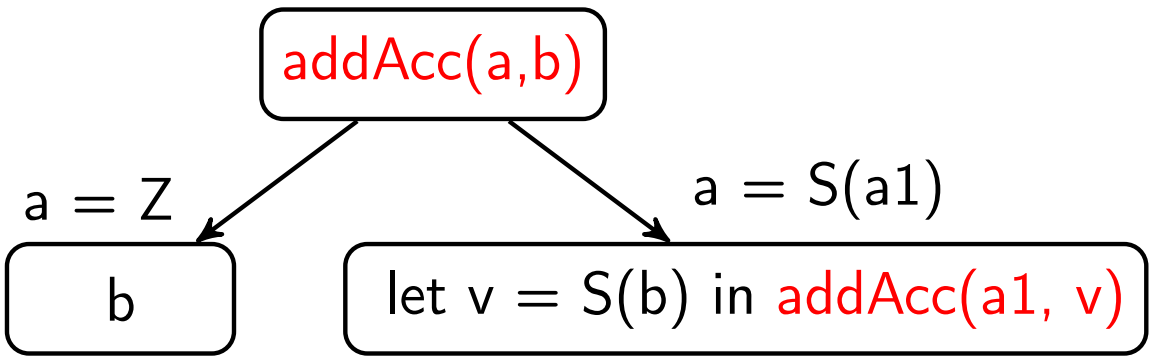
addAcc(a, b): Обобщение и зацикливание

```
addAcc(Z, y) = y;
addAcc(S(x), y) = addAcc(x, S(y));
```



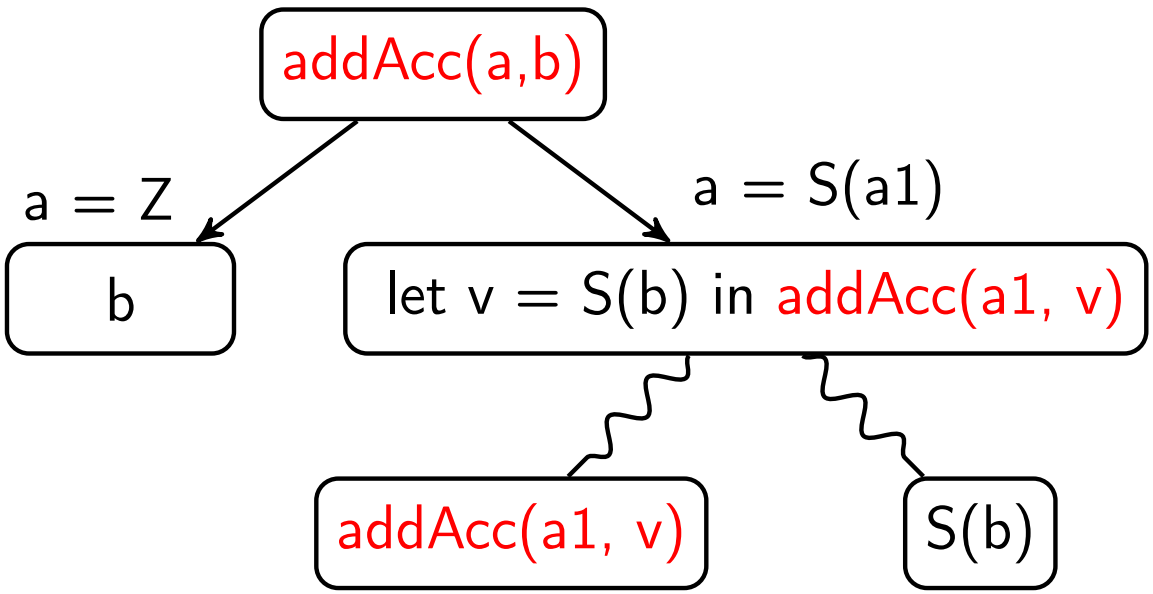
addAcc(a, b): Обобщение и зацикливание

```
addAcc(Z, y) = y;
addAcc(S(x), y) = addAcc(x, S(y));
```



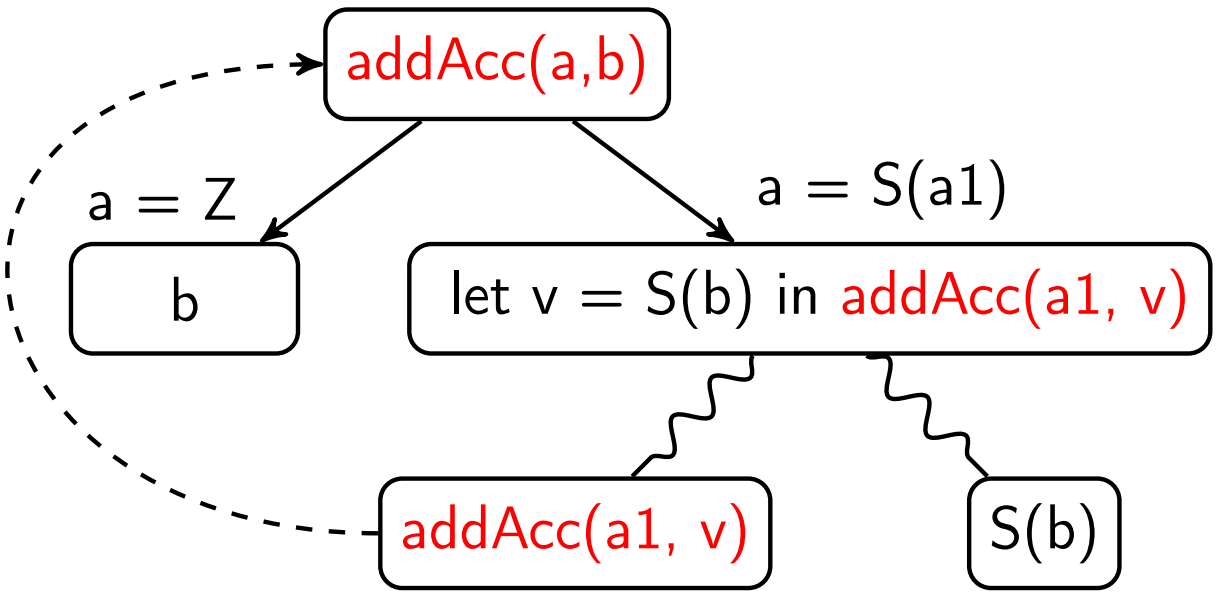
addAcc(a, b): Обобщение и зацикливание

```
addAcc(Z, y) = y;
addAcc(S(x), y) = addAcc(x, S(y));
```



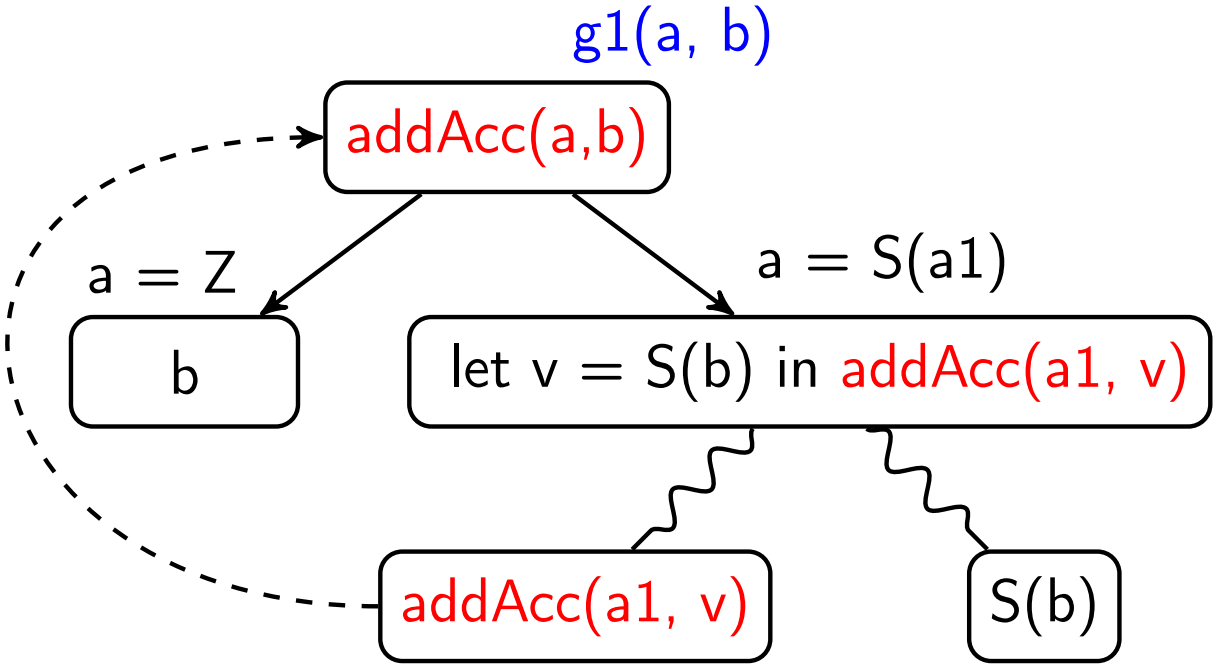
addAcc(a, b): Обобщение и зацикливание

```
addAcc(Z, y) = y;
addAcc(S(x), y) = addAcc(x, S(y));
```



addAcc(a, b): Обобщение и зацикливание

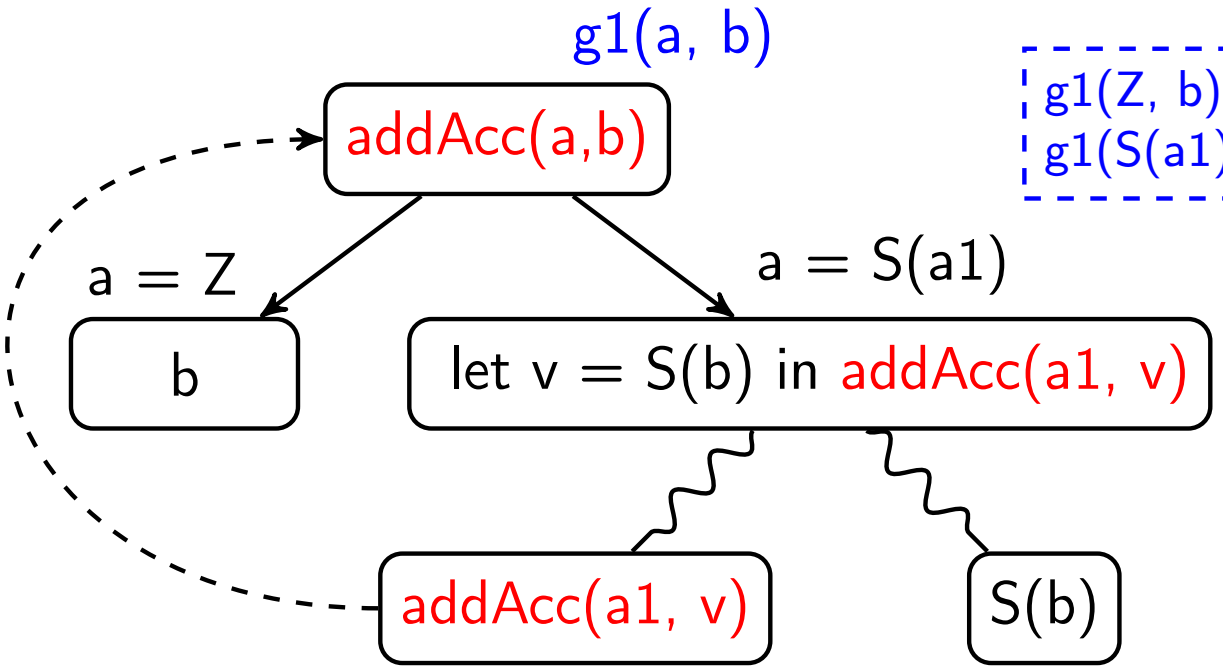
```
addAcc(Z, y) = y;
addAcc(S(x), y) = addAcc(x, S(y));
```



addAcc(a, b): Обобщение и зацикливание

```
addAcc(Z, y) = y;
addAcc(S(x), y) = addAcc(x, S(y));
```

```
g1(Z, b) = b;
g1(S(a1), b) = g1(a1, S(b));
```

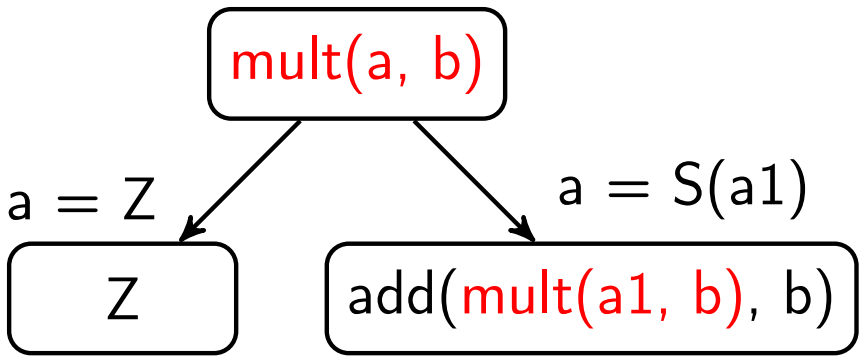


mult(a, b): Бесконечное дерево

mult(a, b)

```
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
mult(Z, y) = Z;
mult(S(x), y) = add(mult(x, y), y);
```

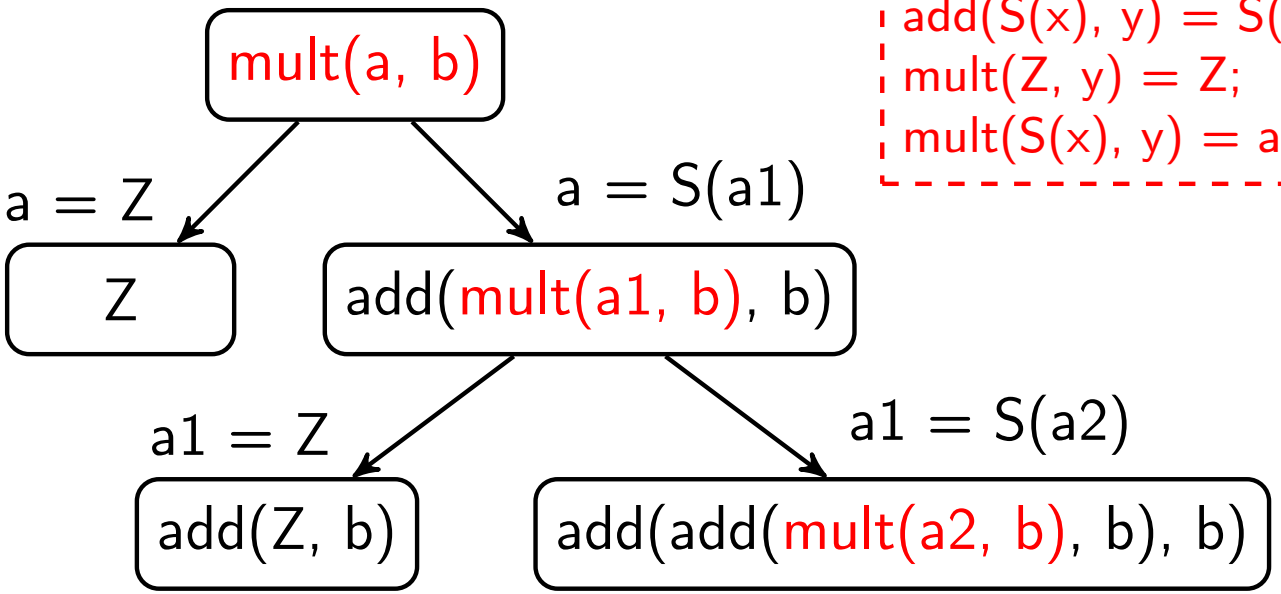
mult(a, b): Бесконечное дерево




```
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
mult(Z, y) = Z;
mult(S(x), y) = add(mult(x, y), y);
```

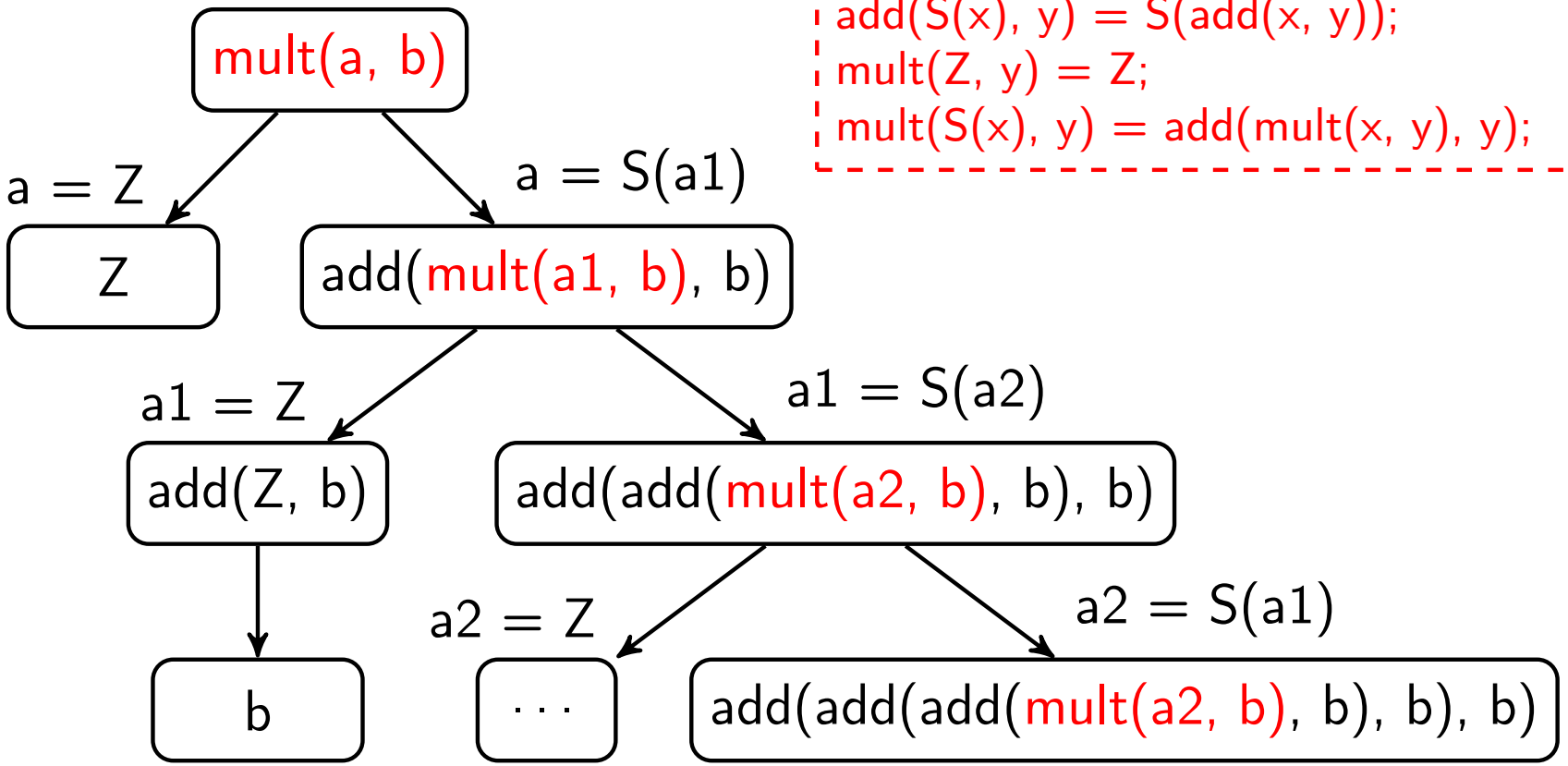
mult(a, b): Бесконечное дерево

```
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
mult(Z, y) = Z;
mult(S(x), y) = add(mult(x, y), y);
```



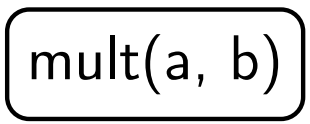
mult(a, b): Бесконечное дерево

```
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
mult(Z, y) = Z;
mult(S(x), y) = add(mult(x, y), y);
```



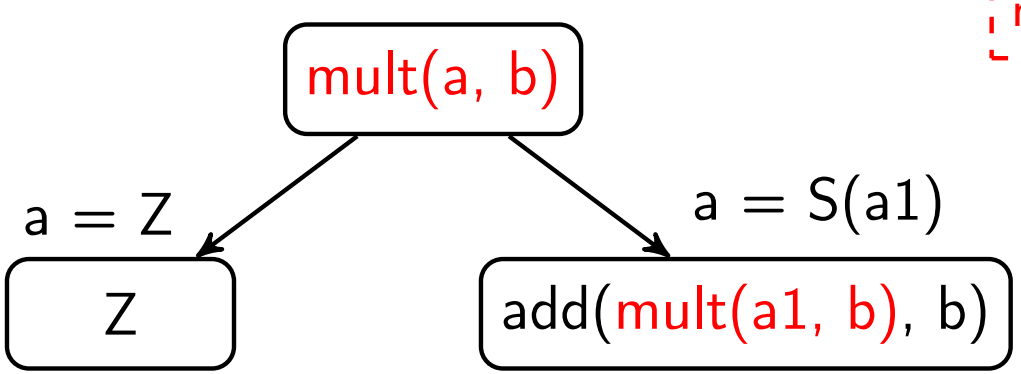
mult(a, b): Обобщение и зацикливание

```
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
mult(Z, y) = Z;
mult(S(x), y) = add(mult(x, y), y);
```



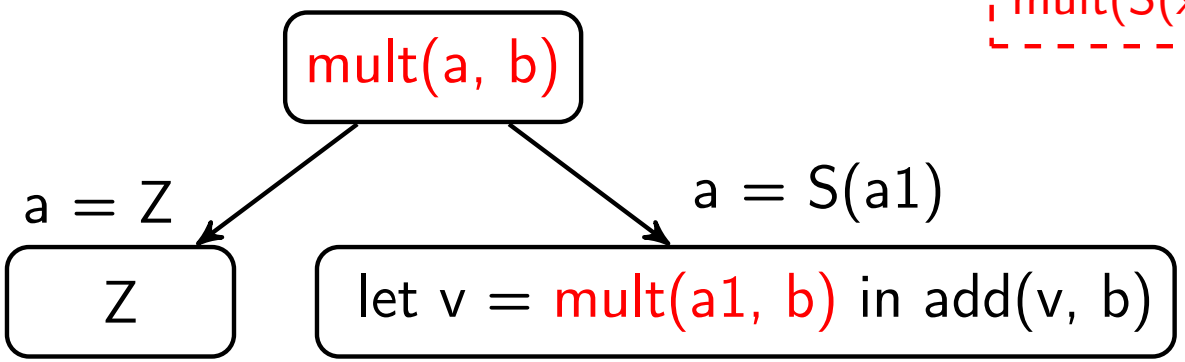
mult(a, b): Обобщение и зацикливание

```
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
mult(Z, y) = Z;
mult(S(x), y) = add(mult(x, y), y);
```



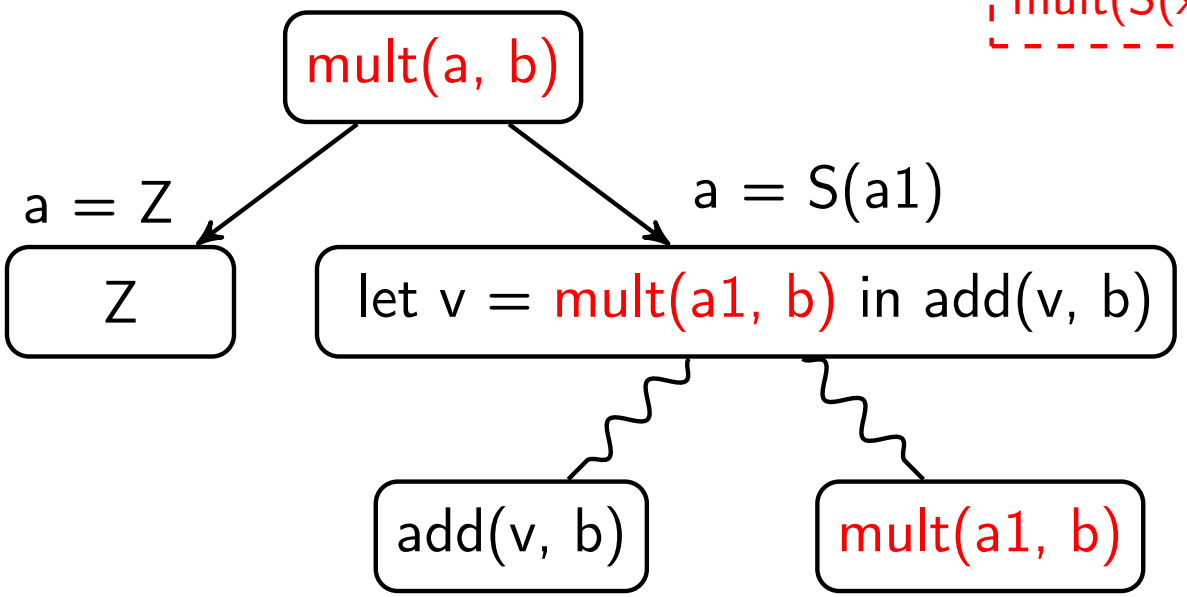
mult(a, b): Обобщение и зацикливание

```
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
mult(Z, y) = Z;
mult(S(x), y) = add(mult(x, y), y);
```



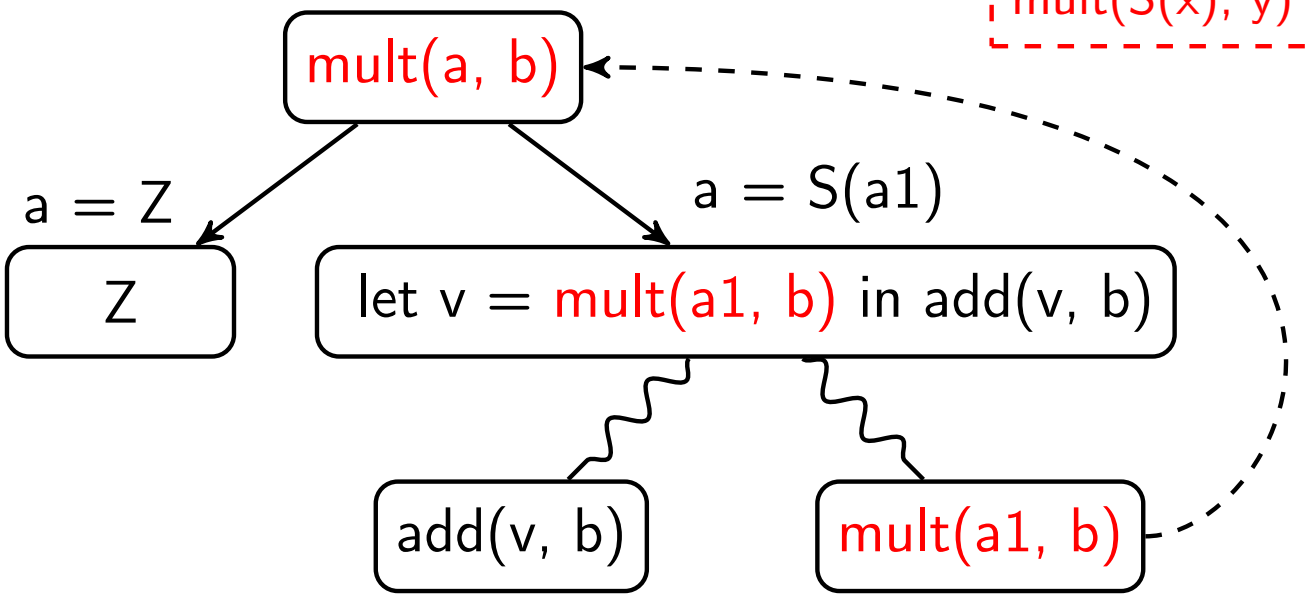
mult(a, b): Обобщение и зацикливание

```
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
mult(Z, y) = Z;
mult(S(x), y) = add(mult(x, y), y);
```

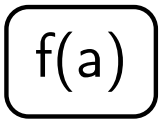


mult(a, b): Обобщение и зацикливание

```
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
mult(Z, y) = Z;
mult(S(x), y) = add(mult(x, y), y);
```

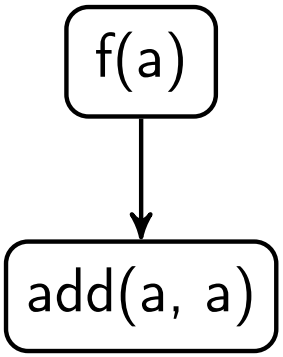


add(a,a): Бесконечное дерево



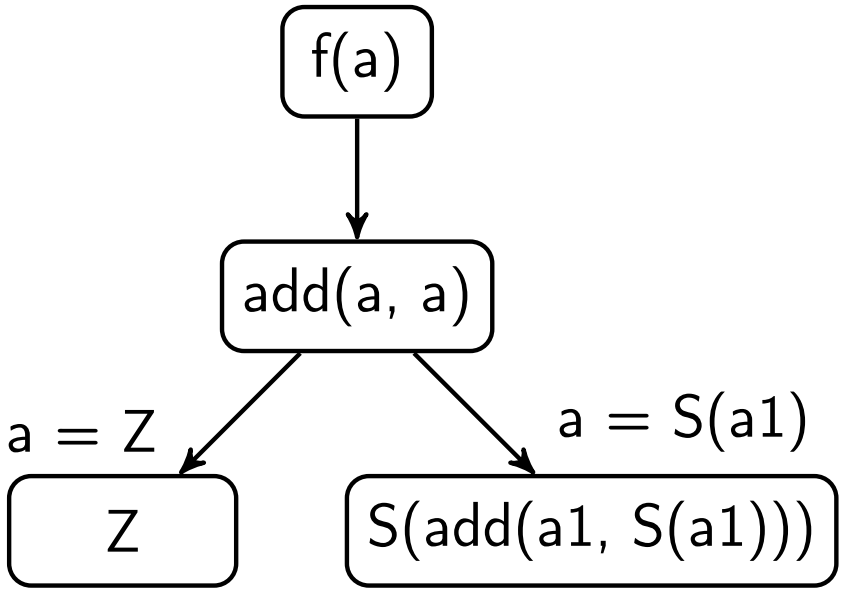

```
f(x) = add(x, x);
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
```

add(a,a): Бесконечное дерево



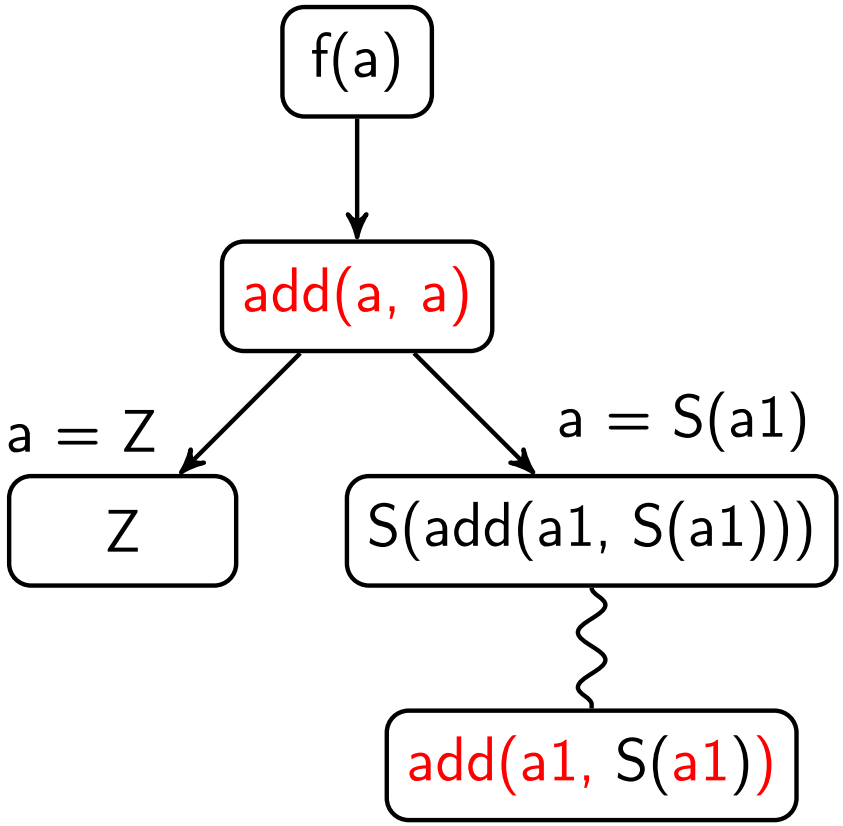

```
f(x) = add(x, x);
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
```

add(a,a): Бесконечное дерево



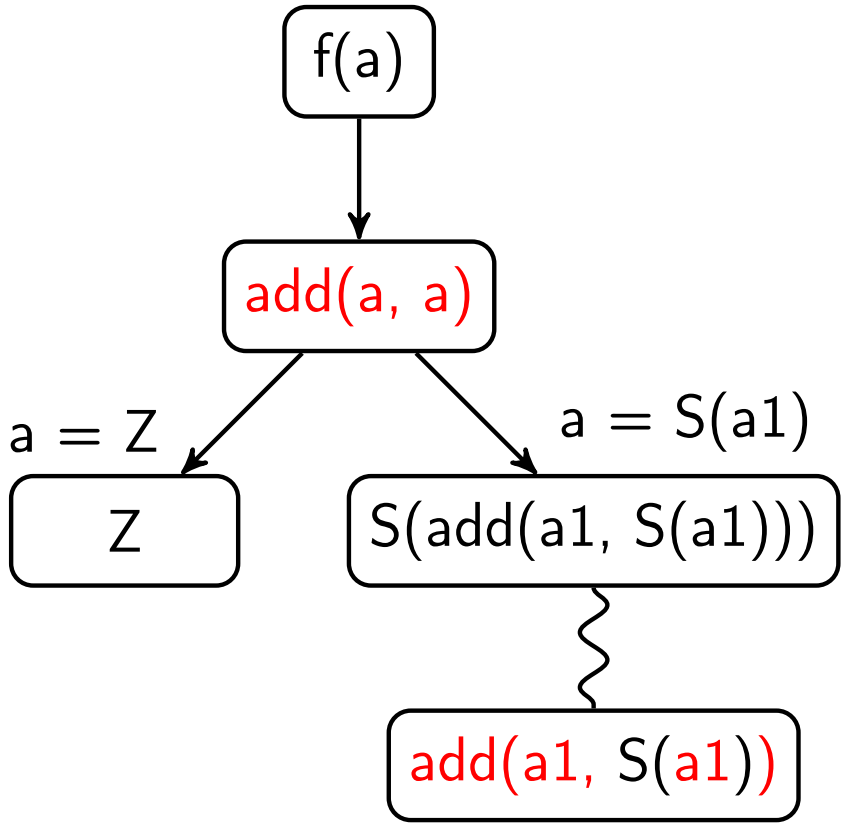

```
f(x) = add(x, x);
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
```

add(a,a): Бесконечное дерево




```
f(x) = add(x, x);
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
```

add(a,a): Бесконечное дерево




```
f(x) = add(x, x);
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
```

Как обобщать? “Верхнюю”, а не “нижнюю” конфигурацию!

$\text{add(a, a)} \cap \text{add(a1, S(a1))} = \emptyset$

$add(a, v) = \text{add(a, a)} \sqcup \text{add(a1, S(a1))}$

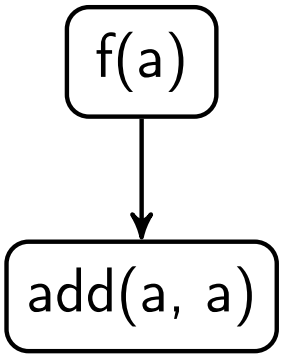
$e_1 \sqcup e_2$ - “наиболее тесное обобщение” e_1 и e_2

add(a,a): Обобщение и зацикливание

f(a)

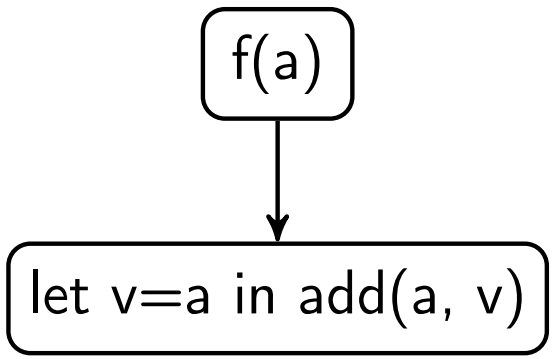
```
f(x) = add(x, x);
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
```

add(a,a): Обобщение и зацикливание



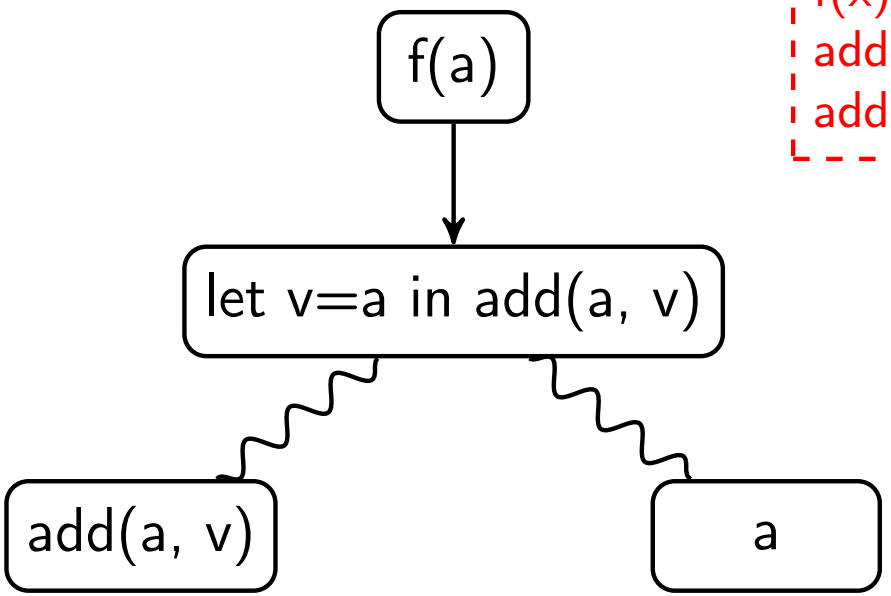

```
f(x) = add(x, x);
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
```

add(a,a): Обобщение и зацикливание




```
f(x) = add(x, x);
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
```

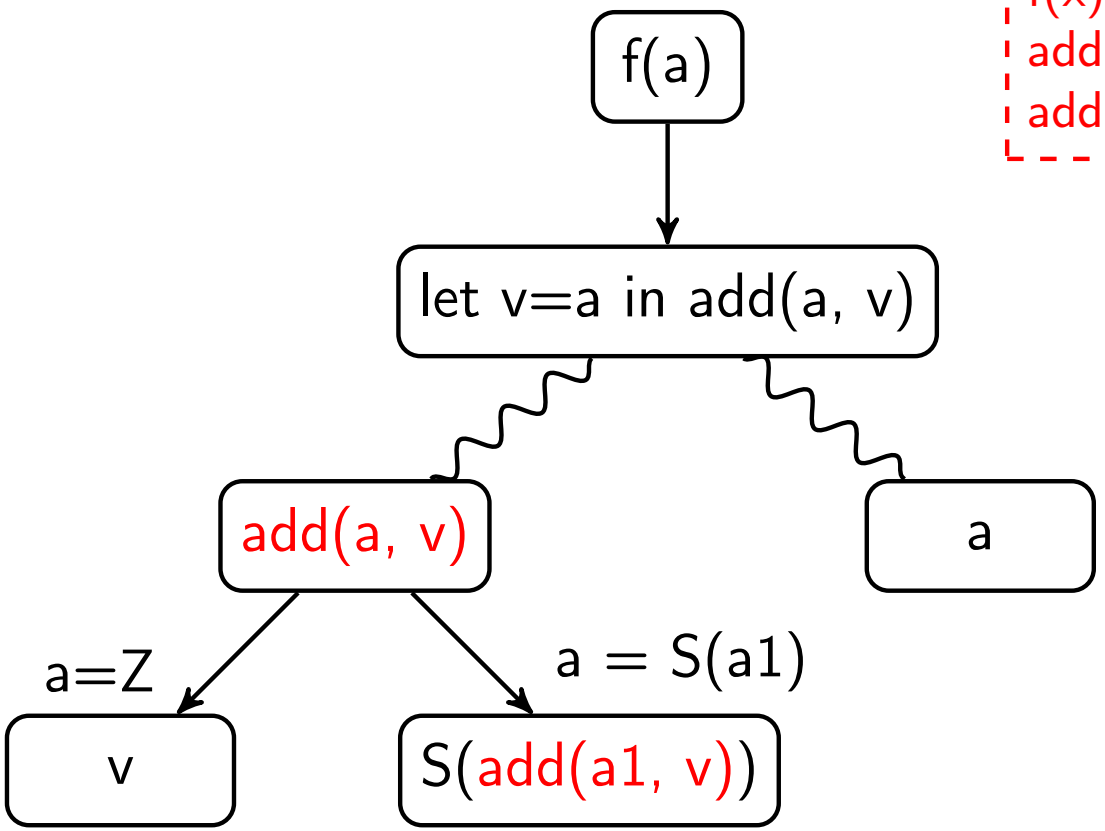
add(a,a): Обобщение и зацикливание




```
f(x) = add(x, x);
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
```

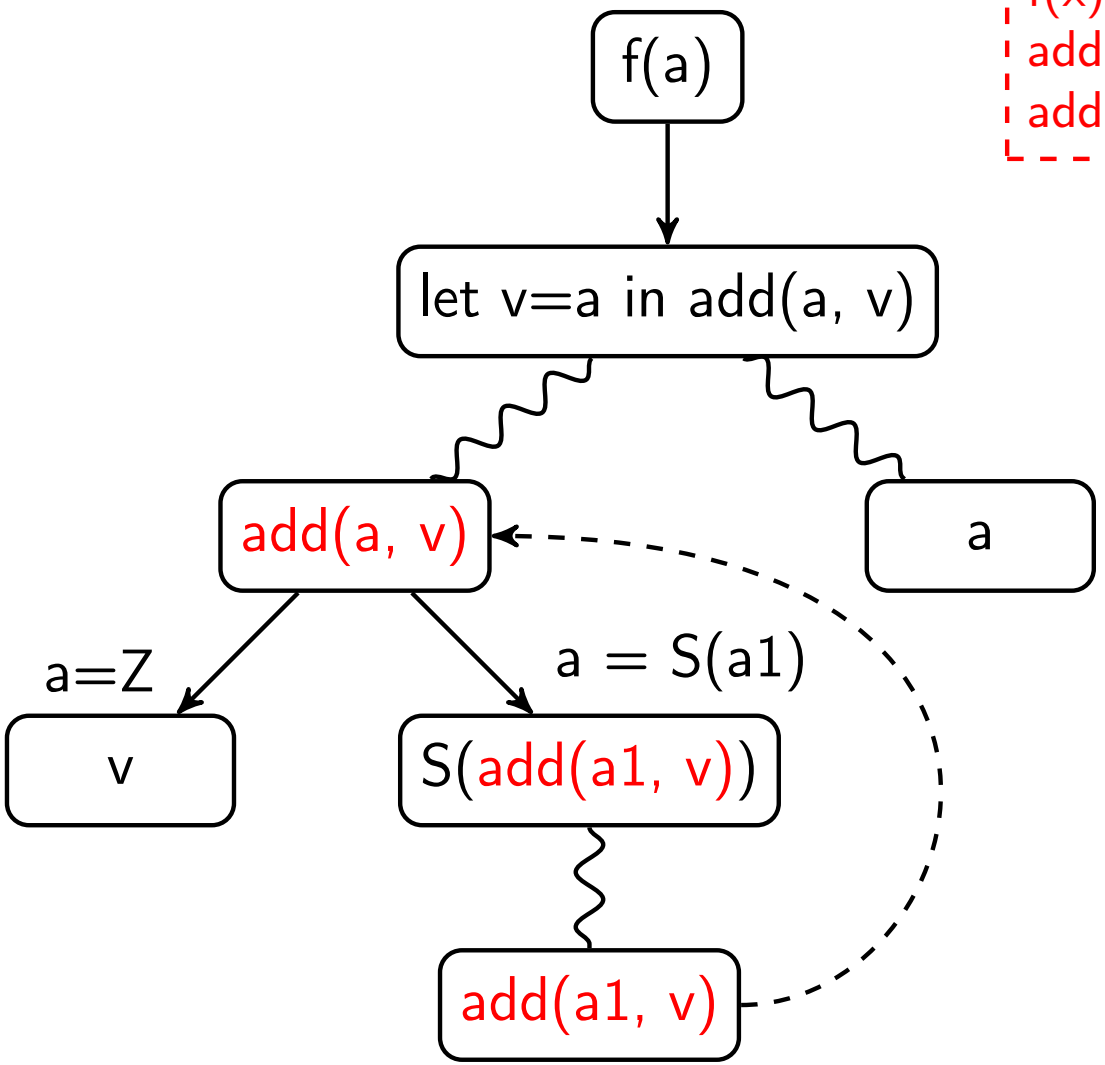
add(a,a): Обобщение и зацикливание

```
f(x) = add(x, x);
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
```



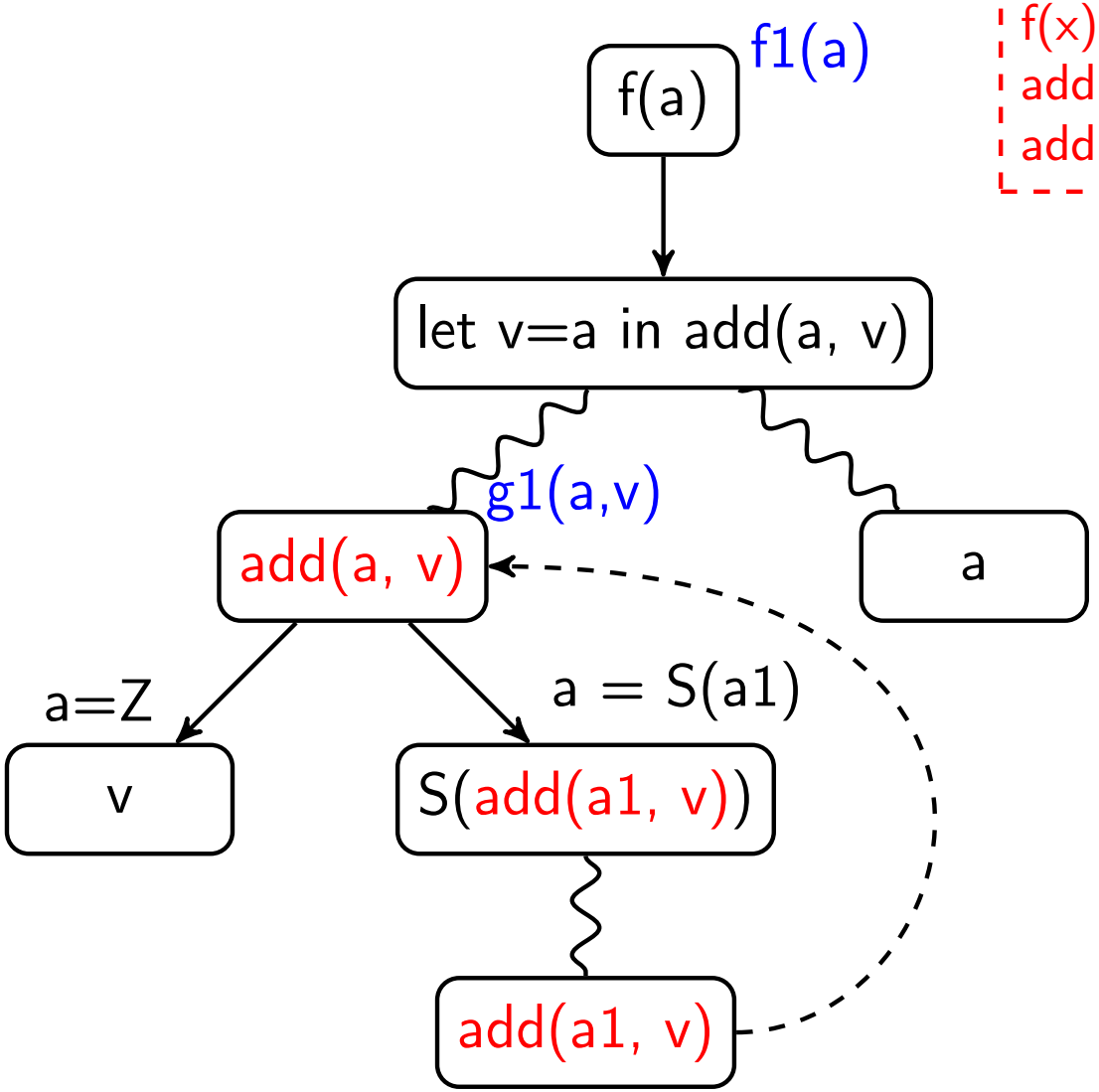
add(a,a): Обобщение и зацикливание

```
f(x) = add(x, x);
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
```

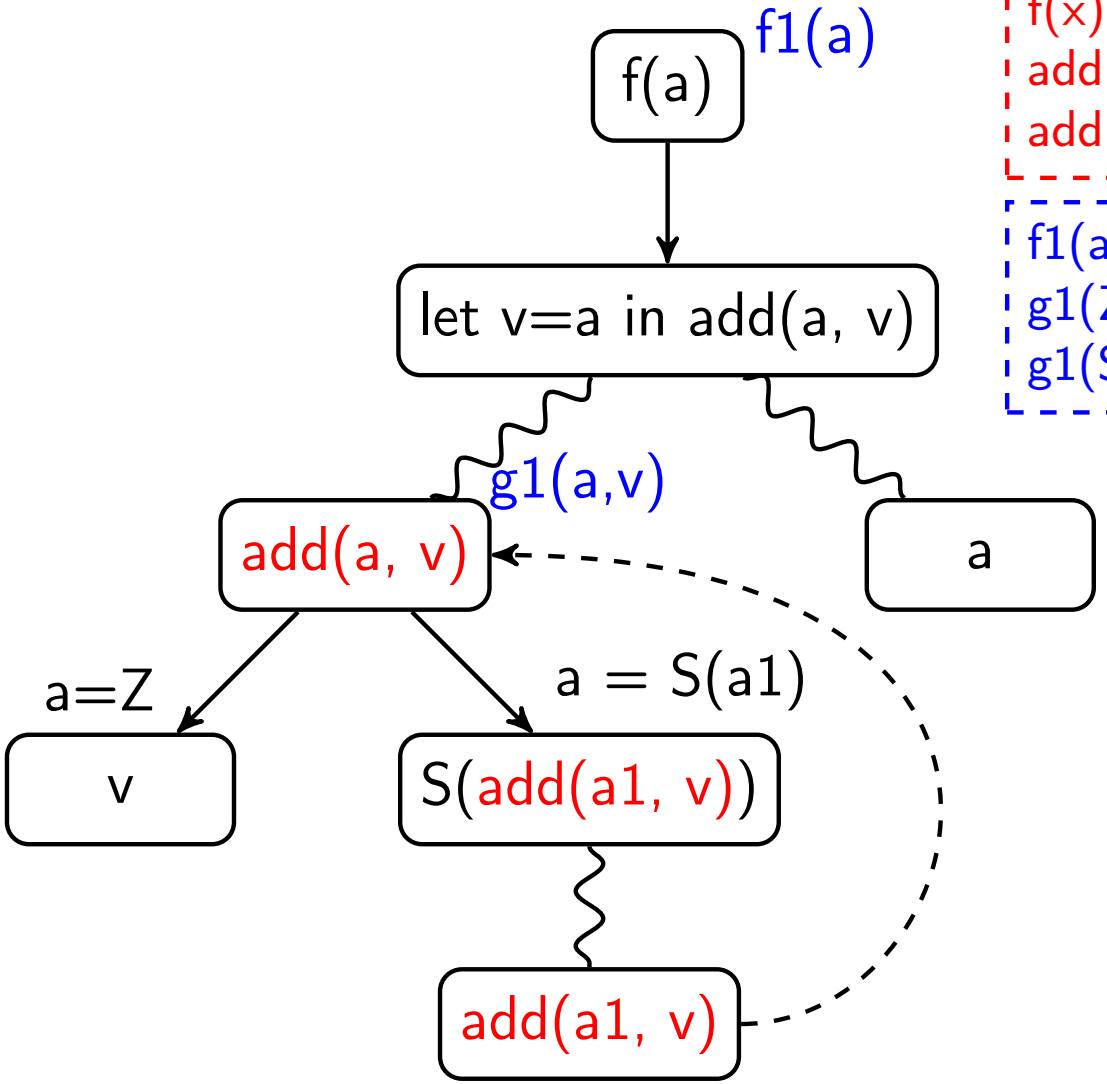


add(a,a): Обобщение и зацикливание

```
f(x) = add(x, x);
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
```



add(a,a): Обобщение и зацикливание




```
f(x) = add(x, x);
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
```

```
f1(a) = g1(a, a);
g1(Z, v) = v;
g1(S(a1), v) = S(g1(a1, v));
```

Гомеоморфное вложение (определение)

Обозначение

$e \trianglelefteq e' \iff e$ гомеоморфно вложено в e'

Определение

1. $v_1 \trianglelefteq v_2$. (Переменные.)
2. $e \trianglelefteq h(e_1, \ldots, e_n)$,
   если $e \trianglelefteq e_i$ для некоторого i. (Ныряние, diving.)
3. $h(e_1, \ldots, e_n) \trianglelefteq h(e'_1, \ldots, e'_n)$,
   если $e_i \trianglelefteq e'_i$ для всех i. (Сочетание, coupling.)

где $\mathtt{h}$ – конструктор, имя f-функции или имя g-функции.

Теорема Хигмана-Крускала (Higman-Kruskal)

Для любой ∞ последовательности выражений $e_0, e_1, \ldots$ найдутся такие i и j, что $i < j$ и $e_i \trianglelefteq e_j$.

Контроль: разбиение конфигураций на 3 категории

- T – тривиальные
  - Переменные: v
  - Обобщение: `let` $\mathtt{v}_1$ `=` $\mathtt{e}_1$ `...in` $\mathtt{e}_0$
  - Конструктор: `C(...)`
- G – глобальные (редукция с сужением)
  - `g2(g1(v,...),...)`
- L – локальные (редукция без сужения)
  - `g2(g1(C(...),...),...)`
  - `g2(g1(f(...),...),...)`
  - `f(...)`

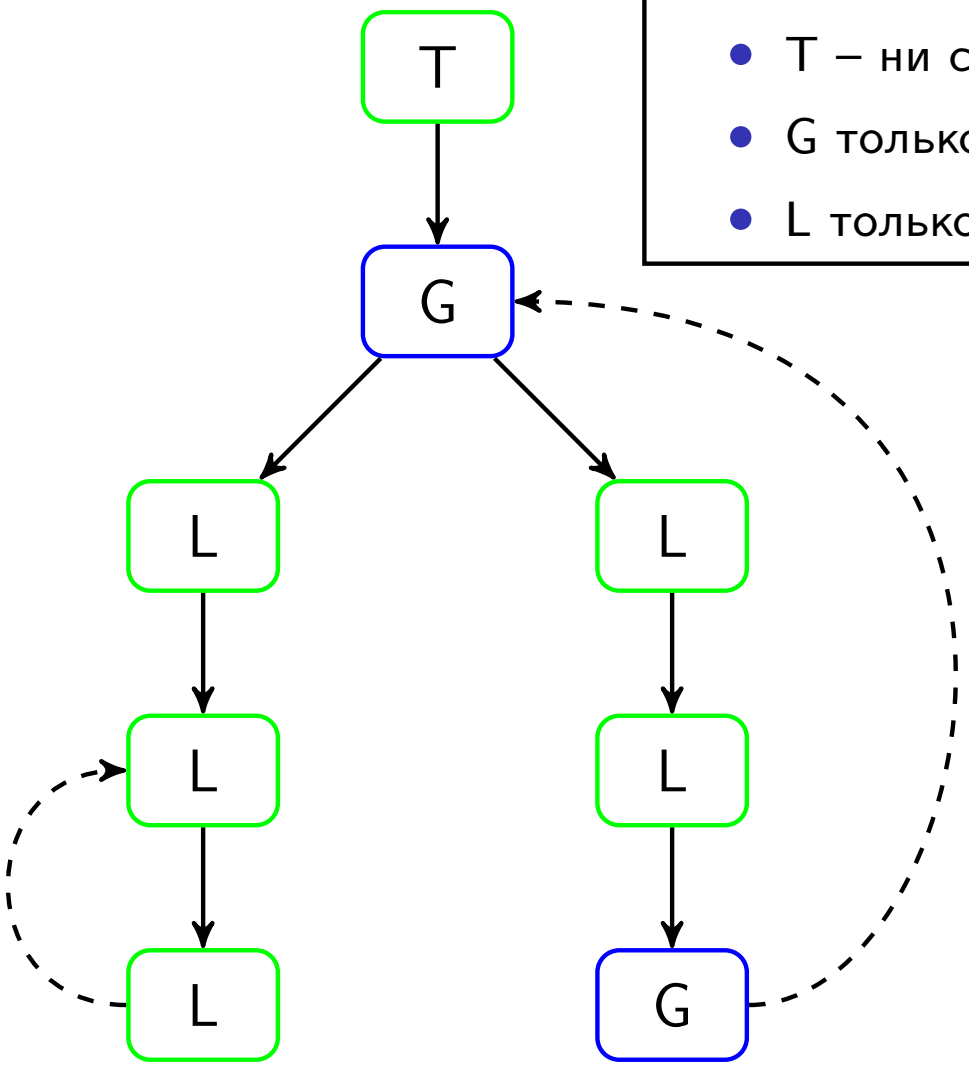
Идея

Если из ∞ последовательности $e_0, e_1, \ldots$ выбрать ∞ подпоследовательность, теорема Хигмана-Крускала всё равно будет выполнена. Можно сравнивать конфигурации только внутри своей категории!

Контроль: раздельное сравнение конфигураций

Сравнение конфигураций:

- T – ни с чем
- G только с G
- L только с L до ближайшего G

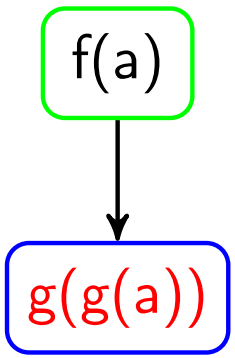


Контроль: пример предотвращения раннего зацикливания

f(a)

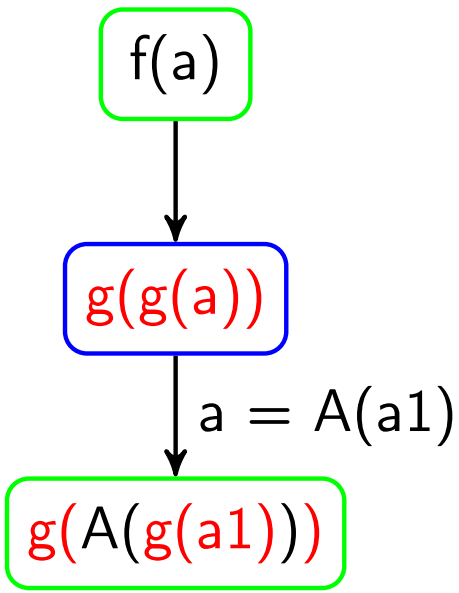
f(x) = g(g(x));
g(A(x)) = A(g(x));

Контроль: пример предотвращения раннего зацикливания



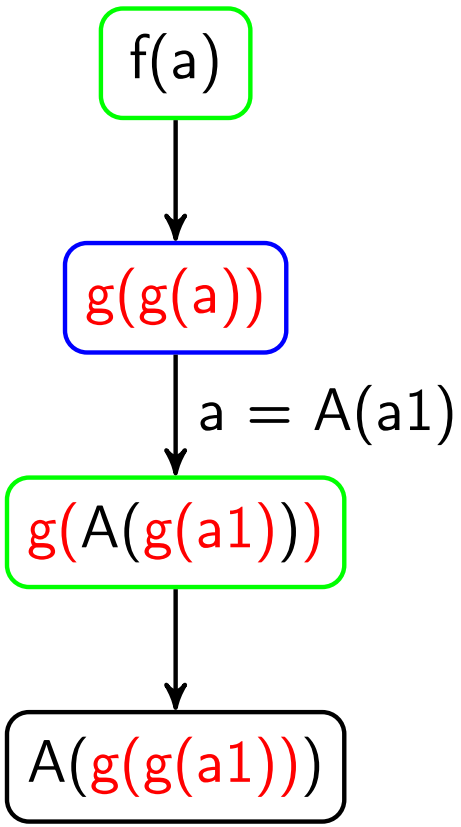
f(x) = g(g(x));
g(A(x)) = A(g(x));

Контроль: пример предотвращения раннего зацикливания



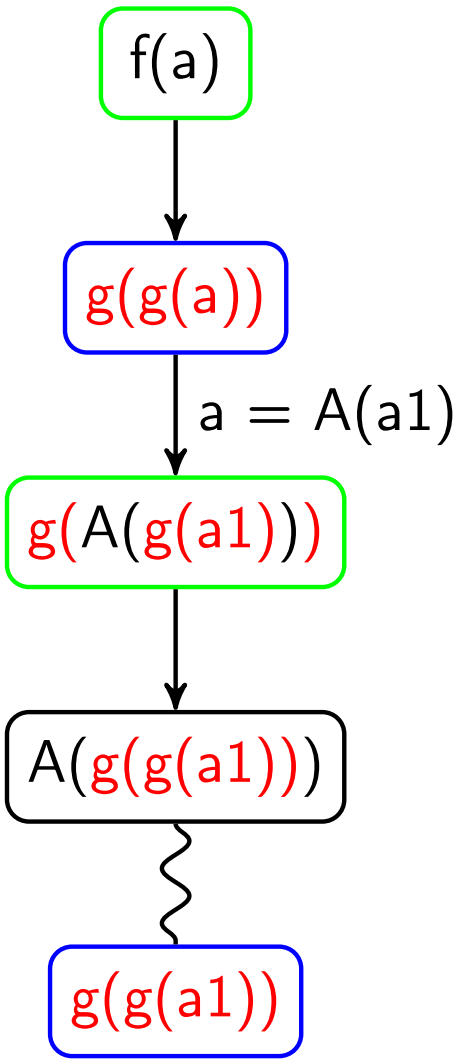
f(x) = g(g(x));
g(A(x)) = A(g(x));

Контроль: пример предотвращения раннего зацикливания



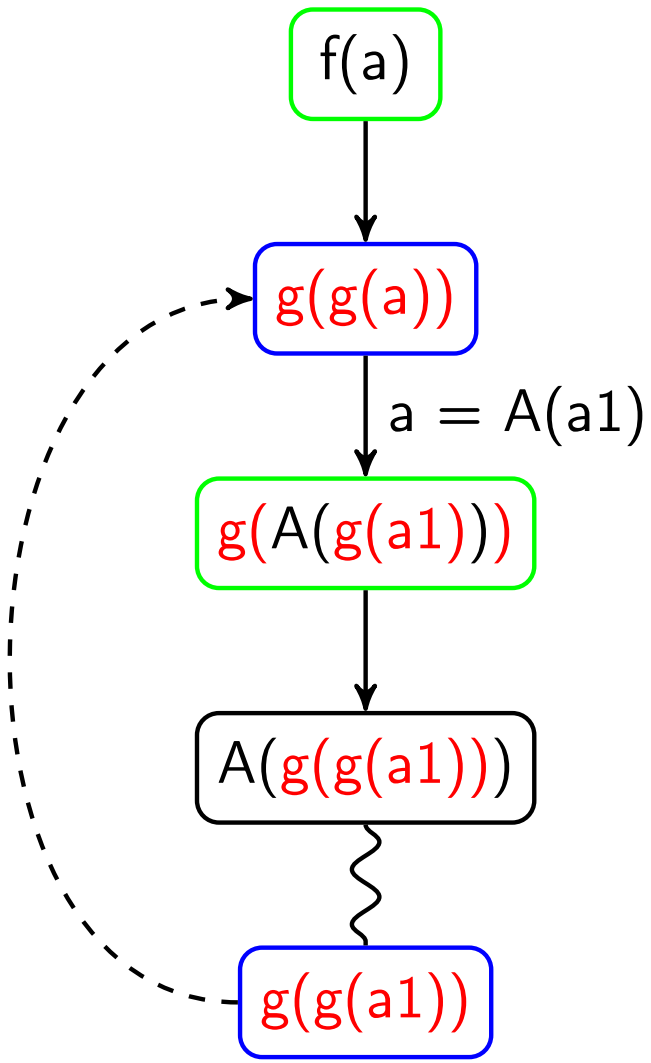
f(x) = g(g(x));
g(A(x)) = A(g(x));

Контроль: пример предотвращения раннего зацикливания



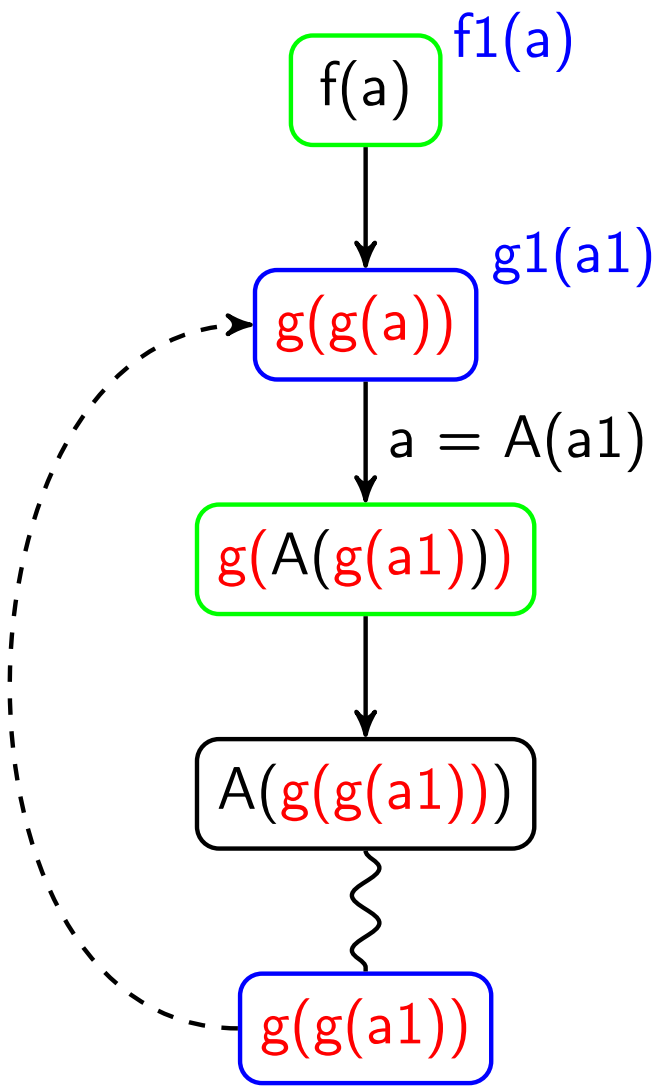
f(x) = g(g(x));
g(A(x)) = A(g(x));

Контроль: пример предотвращения раннего зацикливания



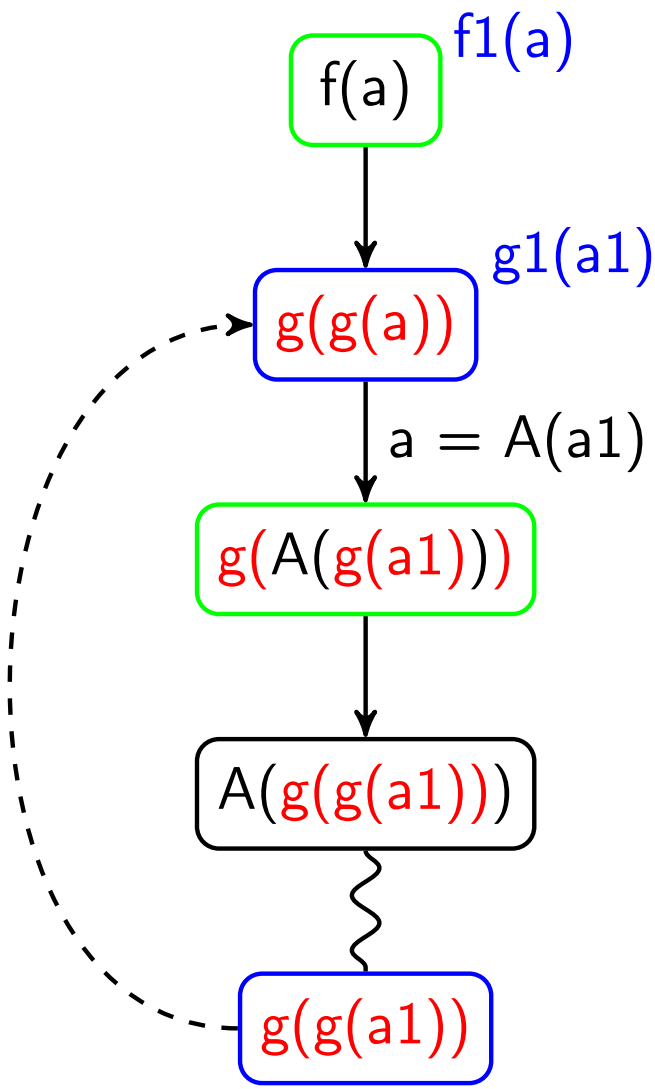
f(x) = g(g(x));
g(A(x)) = A(g(x));

Контроль: пример предотвращения раннего зацикливания



f(x) = g(g(x));
g(A(x)) = A(g(x));

Контроль: пример предотвращения раннего зацикливания



f(x) = g(g(x));
g(A(x)) = A(g(x));

f1(a) = g1(a);
g1(A(a1)) = A(g1(a1));

Спасибо за внимание!

Домашние страницы

- Илья Ключников
  http://pat.keldysh.ru/~ilya/
- Сергей Романенко
  http://pat.keldysh.ru/~roman/

Суперкомпилятор SPSC

- Проект в Google Code: http://spsc.googlecode.com/
- Веб-интерфейс: http://spsc.appspot.com/

Блог

- “Метавычисления и специализация программ”:
  http://metacomputation-ru.blogspot.com/