Основы математической логики и логического программирования

ЛЕКТОР: В.А. Захаров

Лекция 18.

Отрицание в логическом программировании. Оператор not. Встроенные предикаты и функции. Оператор вычисления значений. Модификация баз данных.

ОТРИЦАНИЕ В ЛОГИЧЕСКОМ ПРОГРАММИРОВАНИИ

Отрицание $\neg$ — это очень полезная логическая связка. Часто мы обращаемся с вопросами, используя отрицание.

Пример

$$
\begin{aligned}
\mathcal{P}: \quad & \text{птица}(\text{орел}) \leftarrow; \\
& \text{птица}(\text{воробей}) \leftarrow; \\
& \text{птица}(\text{пингвин}) \leftarrow; \\
& \text{летает}(\text{орел}) \leftarrow; \\
& \text{летает}(\text{воробей}) \leftarrow; \\
& \text{летает}(\text{самолет}) \leftarrow;
\end{aligned}
$$

Сформулируем вопрос: какая птица не летает?

$$G: ?\ \text{птица}(X), \neg\text{летает}(X)$$

Какой ответ мы ожидаем получить на этот запрос?

ОТРИЦАНИЕ В ЛОГИЧЕСКОМ ПРОГРАММИРОВАНИИ

Пример

С точки зрения декларативной семантики, для получения ответа на этот запрос нужно выяснить, выполняется ли логическое следование

$$\mathcal{P} \models \exists X(\text{птица}(X)\&\neg\text{летает}(X)),$$

$$\text{где } \mathcal{P} = \{\text{птица}(\text{орел}), \text{птица}(\text{воробей}), \text{птица}(\text{пингвин}), \text{летает}(\text{орел}), \text{летает}(\text{воробей}), \text{летает}(\text{самолет})\}$$

Оно не выполняется, т. к.

$$B_{\mathcal{P}} \models \mathcal{P},\ B_{\mathcal{P}} \not\models \exists X(\text{птица}(X)\&\neg\text{летает}(X)).$$

Значит, ответ на этот запрос должен быть отрицательным: такой птицы нет .

ОТРИЦАНИЕ В ЛОГИЧЕСКОМ ПРОГРАММИРОВАНИИ

Пример

Однако в обыденной жизни мы в таких случаях даем совсем другой ответ.

Обнаружив, что в нашей базе знаний $\mathcal{P}$
есть сведение $\text{птица}(\text{пингвин})$ и
нет никаких сведений о том, что $\text{летает}(\text{пингвин})$,
мы отвечаем:

«Насколько позволяет судить наша база знаний, нелетающая птица — это пингвин».

В юрипруденции такой подход к решению вопроса о виновности человек называется **презумпцией невиновности** :

в случае отсутствия свидетельств виновности
человек считается невиновным.

ОТРИЦАНИЕ В ЛОГИЧЕСКОМ ПРОГРАММИРОВАНИИ

Мы можем распространить принцип презумпции невиновности и на логические программы.

Допущение Замкнутости Мира

Пусть имеется некоторое непротиворечивое множество замкнутых формул Γ (например, хорновская логическая программа) и замкнутая формула φ (например, запрос или отдельная подцель).

Тогда формула $\neg\varphi$ является логическим следствием множества Γ в допущении замкнутости мира

$$\Gamma \models_{CWA} \neg\varphi,$$

если **неверно**, что φ логически следует из Γ, т. е. $\Gamma \not\models \varphi$.

Здесь CWA — аббревиатура Closed World Assumption.

ОТРИЦАНИЕ В ЛОГИЧЕСКОМ ПРОГРАММИРОВАНИИ

Суть Допущения Замкнутости Мира (CWA) состоит в том, что при извлечении CWA-логических следствий из базы знаний Г ($\models_{CWA}$) нужно рассматривать не все модели для Г, а только такую минимальную модель, в которой истинными являются одни лишь классические следствия ($\models$) из Г.

Такая минимальная модель существует, вообще говоря, не всегда (например, если $\Gamma = \{A \vee B\}$).

Но в случае хорновских логических программ, минимальной моделью для программы $\mathcal{P}$ является наименьшая эрбрановская модель $\mathbf{M}_{\mathcal{P}}$.

ОТРИЦАНИЕ В ЛОГИЧЕСКОМ ПРОГРАММИРОВАНИИ

Обратимся к орнитологическому примеру.

Пример

$$\begin{aligned}\mathcal{P} &= \{\text{птица}(\text{орел}), \text{птица}(\text{воробей}), \text{птица}(\text{пингвин}),\\ &\quad \text{летает}(\text{орел}), \text{летает}(\text{воробей}), \text{летает}(\text{самолет})\}\end{aligned}$$

$$\varphi = \text{птица}(\text{пингвин}) \ \& \ \neg\text{летает}(\text{пингвин})$$

Тогда

$\mathcal{P} \models_{CWA} \text{птица}(\text{пингвин})$, поскольку $\mathbf{M}_{\mathcal{P}} \models \text{птица}(\text{пингвин})$,

$\mathcal{P} \models_{CWA} \neg\text{летает}(\text{пингвин})$, поскольку $\mathbf{M}_{\mathcal{P}} \not\models \text{летает}(\text{пингвин})$.

Значит, $\mathcal{P} \models_{CWA} \text{птица}(\text{пингвин}) \ \& \ \neg\text{летает}(\text{пингвин})$.

ОПЕРАТОР not

Итак, теперь к логическим программам можно обращаться с запросами, содержащими отрицания подцелей, и декларативная семантика умеет разумно обращаться с таким отрицательными подцелями.

Нам остается научить этому обращению компьютер, т. е. ввести в операционную семантику правила для обработки отрицательных подцелей.

Однако здесь нас ожидает неприятный сюрприз.

ОПЕРАТОР not

Предположим, что имеется запрос $G : ?\neg C_0$ к программе $\mathcal{P}$.

Чтобы получить утвердительный ответ на запрос G (т. е. доказать, что $\mathcal{P} \models_{CWA} \neg C_0$), интерпретатор должен проверить, что $\mathbf{M}_{\mathcal{P}} \not\models C_0$.

Для этого интерпретатор должен убедиться, что запрос $G' : ?C_0$ к программе $\mathcal{P}$ **не имеет ни одного успешного вычисления**.

Но эта задача является алгоритмически неразрешимой (почему?).
Да потому, что к ней сводится проблема останова машин Тьюринга.

Следовательно, никакой интерпретатор логических программ не может гарантировать получение утвердительного ответа на запрос $G : ?\neg C_0$ даже в том случае, если этот ответ существует.

ОПЕРАТОР not

Алгоритмическая неразрешимость проблемы существования (отсутствия) успешного вычисления логических программ — это непреодолимое препятствие на пути создания такой операционной семантики, которая была бы адекватна декларативной семантике в рамках Допущения Замкнутости Мира.

Можно лишь построить такой интерпретатор (операционной семантики), который как можно лучше соответствует CWA при обращении с отрицательными подцелями $\neg C_0$.

Для этого в язык логического программирования был введен специальный (встроенный) оператор **not**.

ОПЕРАТОР not

Аргументами оператора **not** являются атомы.

Для вычисления ответов на запрос ? **not**(C_0) вводится правило SLDNF-резолюции (Not as Failure), которое определяется следующим образом.

Правило SLDNF-резолюции

Пусть имеется запрос G_0 : ? **not**$(C_0), C_1, \ldots, C_n$ к программе $\mathcal{P}$.

Для вычисления SLDNF-резольвенты G_1

1. формируется запрос G' : ? C_0 к программе $\mathcal{P}$;
2. проводится построение (обход) дерева вычислений T запроса G' : ? C_0;
3. в зависимости от устройства дерева T возможен один из трех исходов.

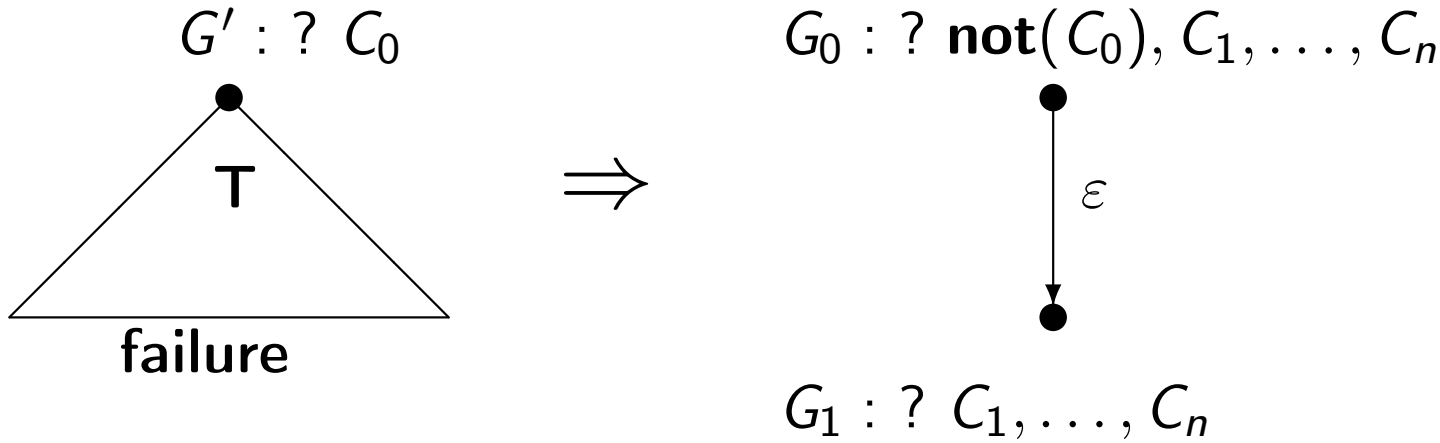
ОПЕРАТОР not

Правило SLDNF-резолюции

Вариант 1: **Успех.**

Дерево T конечно, и все его ветви (SLD-резолютивные вычисления) являются тупиковыми.

Тогда запрос $G_0 : ?\ \mathbf{not}(C_0), C_1, \ldots, C_n$ имеет SLDNF-резольвенту $G_1 = ?\ C_1, \ldots, C_n$, которая получается с использованием пустой подстановки ε.



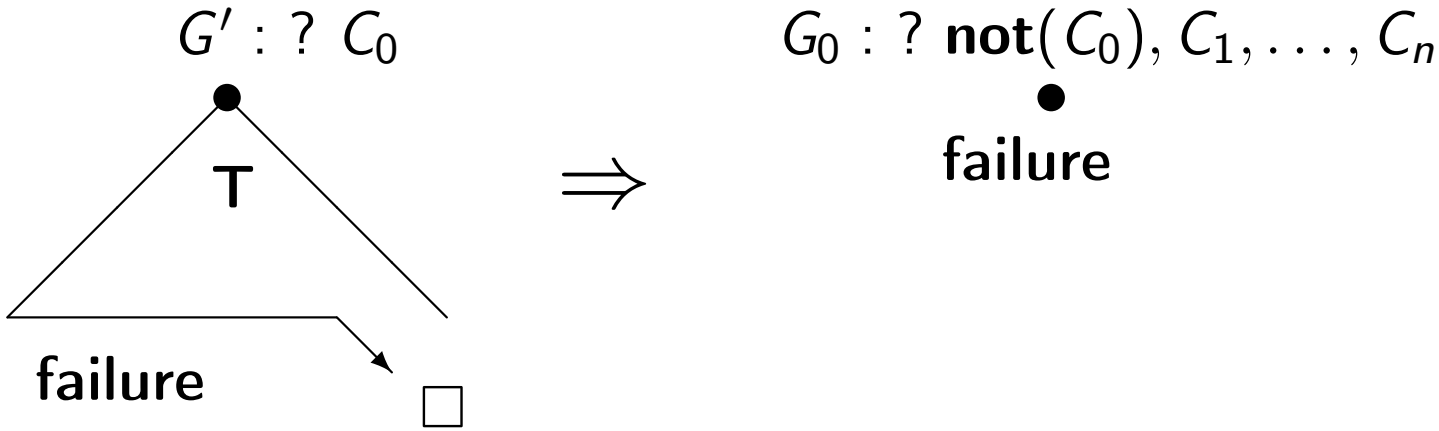
ОПЕРАТОР not

Операционная семантика оператор not

Вариант 2: **Неудача.**

При построении (обходе) дерева T было обнаружено успешное вычисление.

Тогда запрос $G_0 : ?\ \mathbf{not}(C_0), C_1, \ldots, C_n$ не имеет SLDNF-резольвент, и вычисление этого запроса является тупиковым .



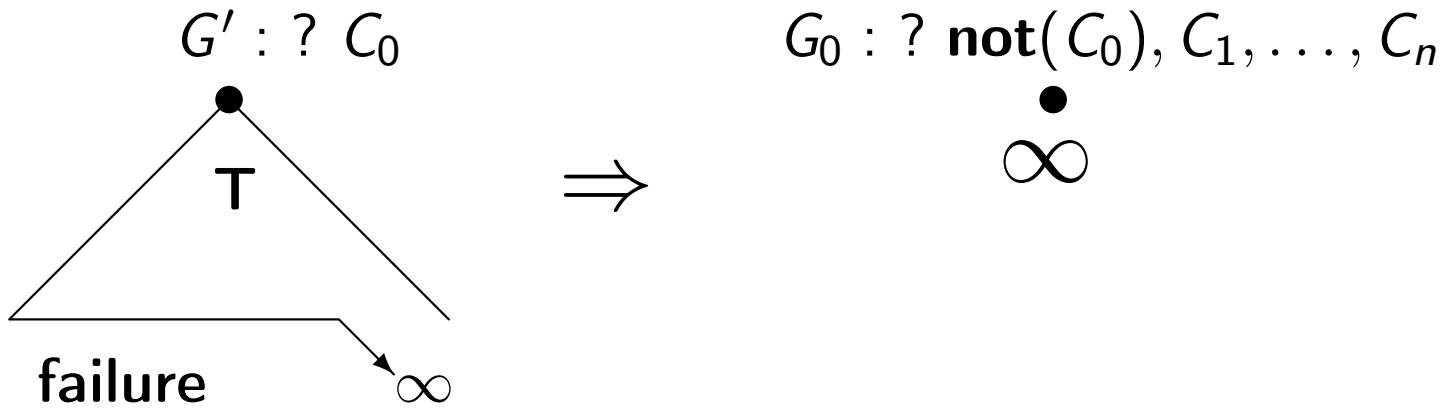
ОПЕРАТОР not

Операционная семантика оператор not

Вариант 3: **Бесконечное вычисление.**

Дерево T бесконечно и при его построении (обходе) не было обнаружено успешных вычислений.

Тогда запрос $G_0 : ?\ \mathbf{not}(C_0), C_1, \ldots, C_n$ не имеет SLDNF-резольвент, и вычисление этого запроса является бесконечным («сингулярная бесконечность»).



ОПЕРАТОР not

Операционная семантика оператор not

Вариант 3: **Бесконечное вычисление.**

Дерево T бесконечно и при его построении (обходе) не было обнаружено успешных вычислений.

Тогда запрос $G_0 : ?\ \mathbf{not}(C_0), C_1, \ldots, C_n$ не имеет SLDNF-резольвент, и вычисление этого запроса является бесконечным («сингулярная бесконечность»).

Условие существования бесконечного вычисления запроса $\mathbf{not}(C_0)$ описано не вполне строго, поскольку обнаружение успешного вычисления существенно зависит от стратегии обхода дерева SLD-резолютивных вычислений. Так, например, стандартная стратегия обхода в глубину может не обнаружить существующего успешного вычисления (почему?).

ОПЕРАТОР not

Теорема (корректности SLDNF-резолюции)

Если запрос G : ? **not**(C_0) к хорновской логической программе $\mathcal{P}$ имеет успешное SLDNF-резолютивное вычисление, то $\mathcal{P} \models_{CWA} \neg\exists\overline{y}\ C_0$.

Доказательство

Самостоятельно .

А вот обратное утверждение (теорема полноты) для SLDNF-резолютивного вычисления будет уже неверно.

ОПЕРАТОР not

Пример.

Рассмотрим программу поиска всех тех букв X слова L', которые не содержатся в слове L''.

$$\begin{array}{lrcl} \mathcal{P}: & S(X, L', L'') & \leftarrow & E(X, L'), \mathbf{not}(E(X, L'')); \\ & E(X, X \,.\, L) & \leftarrow & ; \\ & E(X, Y \,.\, L) & \leftarrow & E(X, L); \end{array}$$

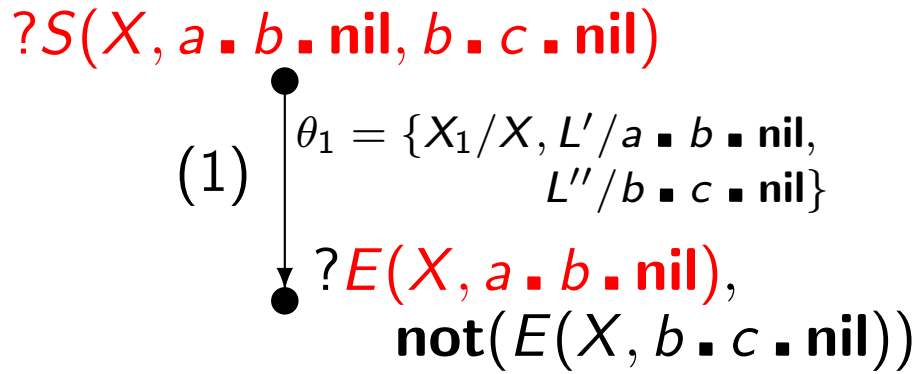
и обратимся к ней с запросом

$$G_0 : ?\ S(X, a \,.\, b \,.\, \mathbf{nil}, b \,.\, c \,.\, \mathbf{nil}).$$

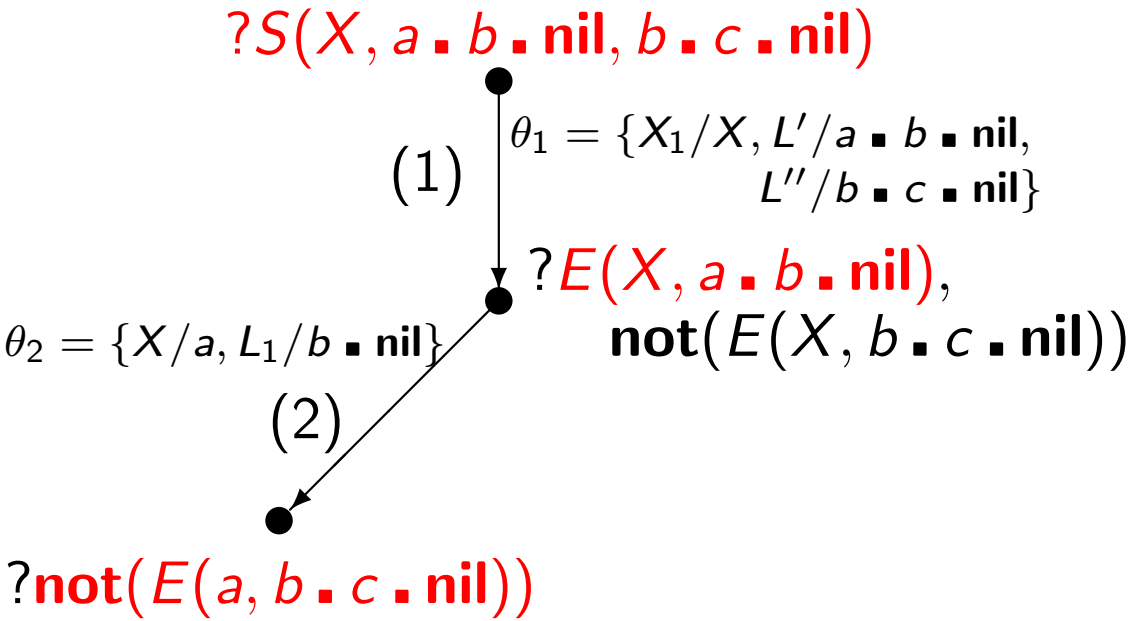
$$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L'')); \quad (1)$$
$$E(X, X \centerdot L) \leftarrow; \quad (2)$$
$$E(X, Y \centerdot L) \leftarrow E(X, L); \quad (3)$$

$$?S(X, a \centerdot b \centerdot \mathbf{nil}, b \centerdot c \centerdot \mathbf{nil})$$

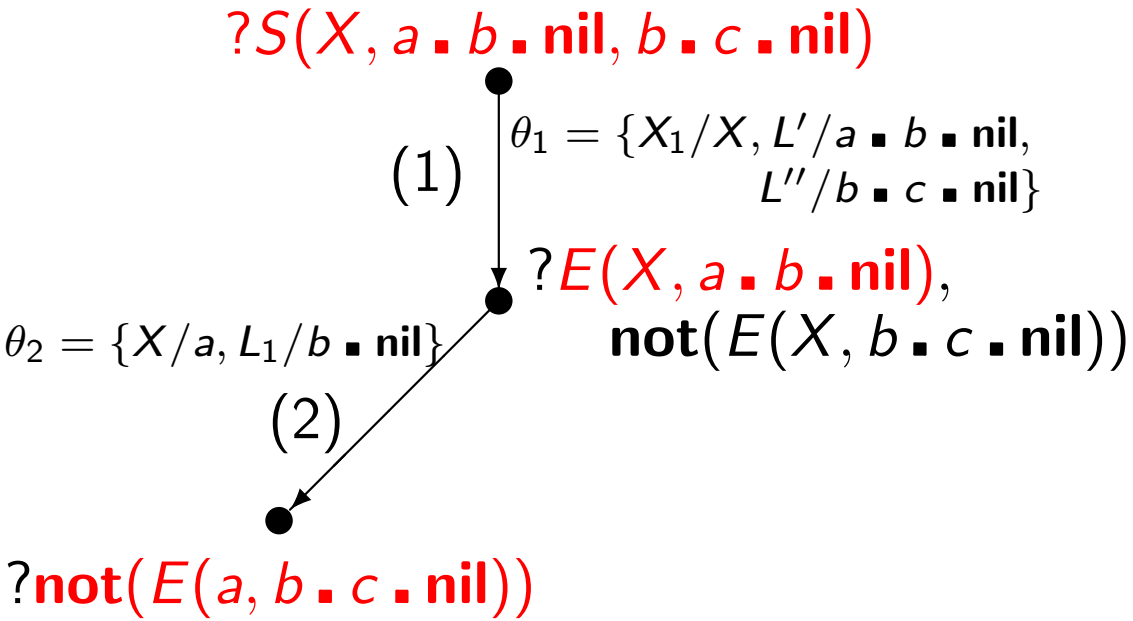
$$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L'')); \quad (1)$$
$$E(X, X \centerdot L) \leftarrow; \quad (2)$$
$$E(X, Y \centerdot L) \leftarrow E(X, L); \quad (3)$$

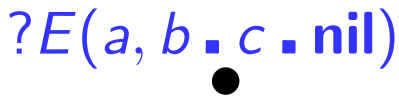


$$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L'')); \quad (1)$$
$$E(X, X \centerdot L) \leftarrow; \quad (2)$$
$$E(X, Y \centerdot L) \leftarrow E(X, L); \quad (3)$$

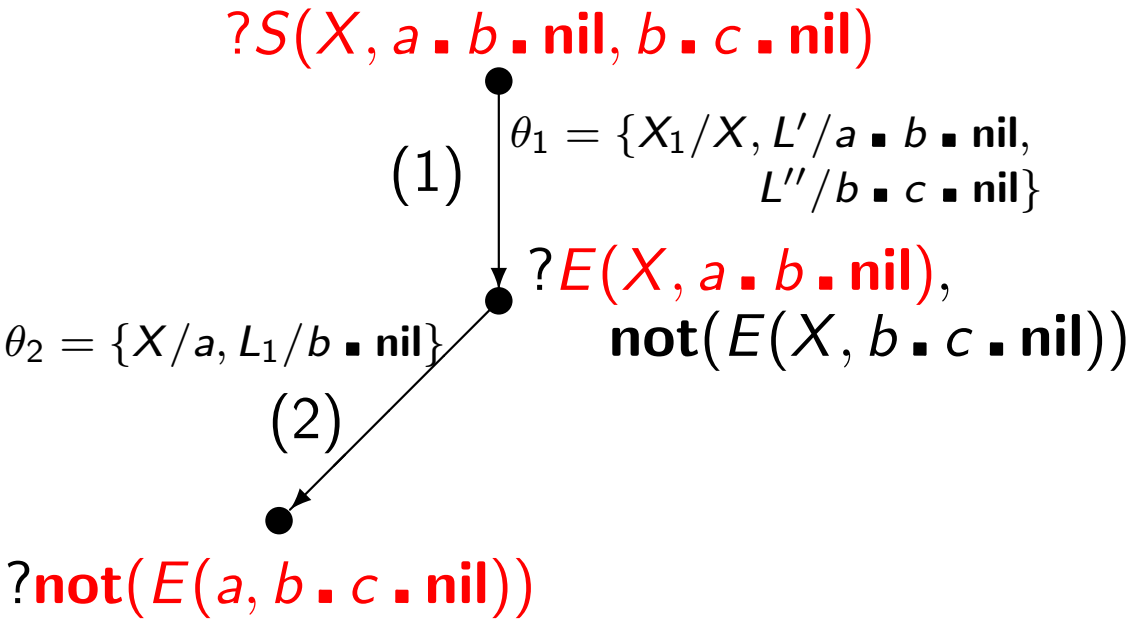


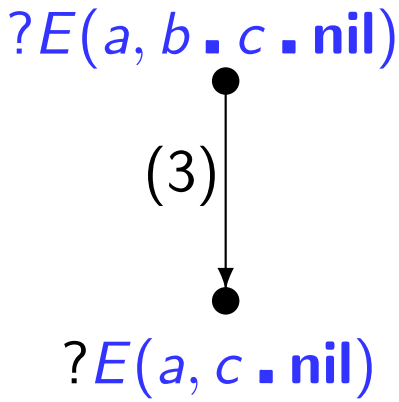
$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''))$; (1)
$E(X, X \centerdot L) \leftarrow$; (2)
$E(X, Y \centerdot L) \leftarrow E(X, L)$; (3)



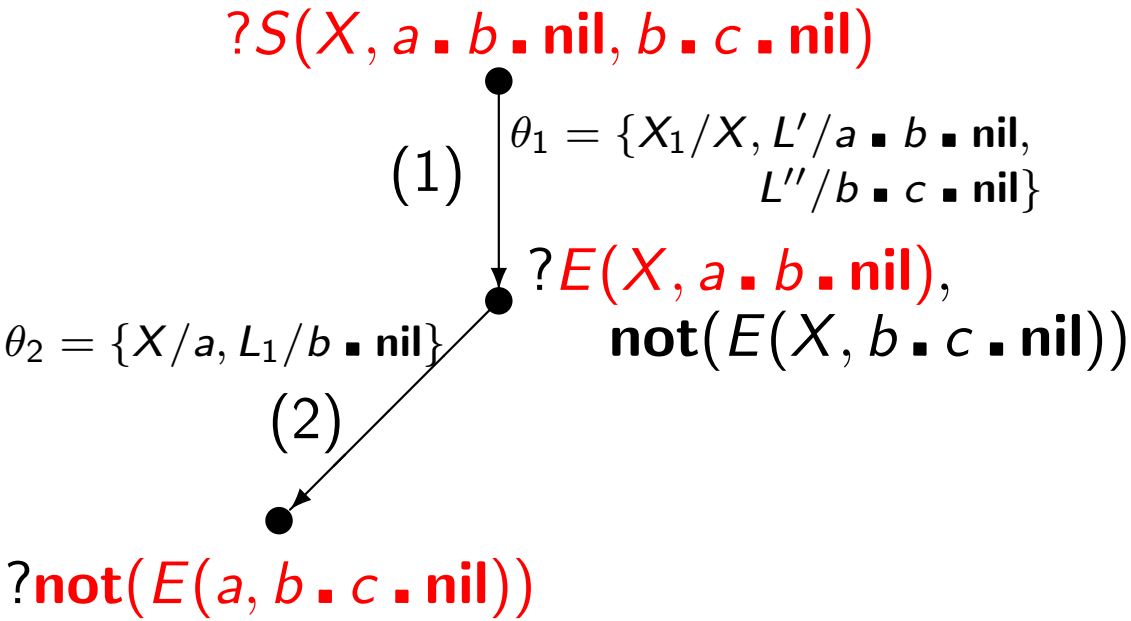


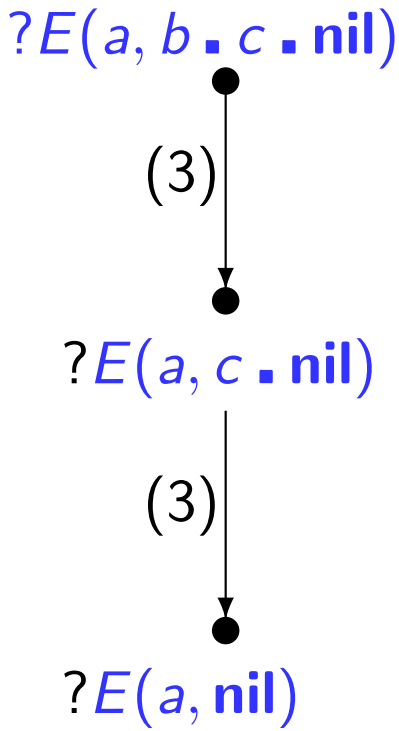
$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''));$ (1)
$E(X, X \blacksquare L) \leftarrow;$ (2)
$E(X, Y \blacksquare L) \leftarrow E(X, L);$ (3)



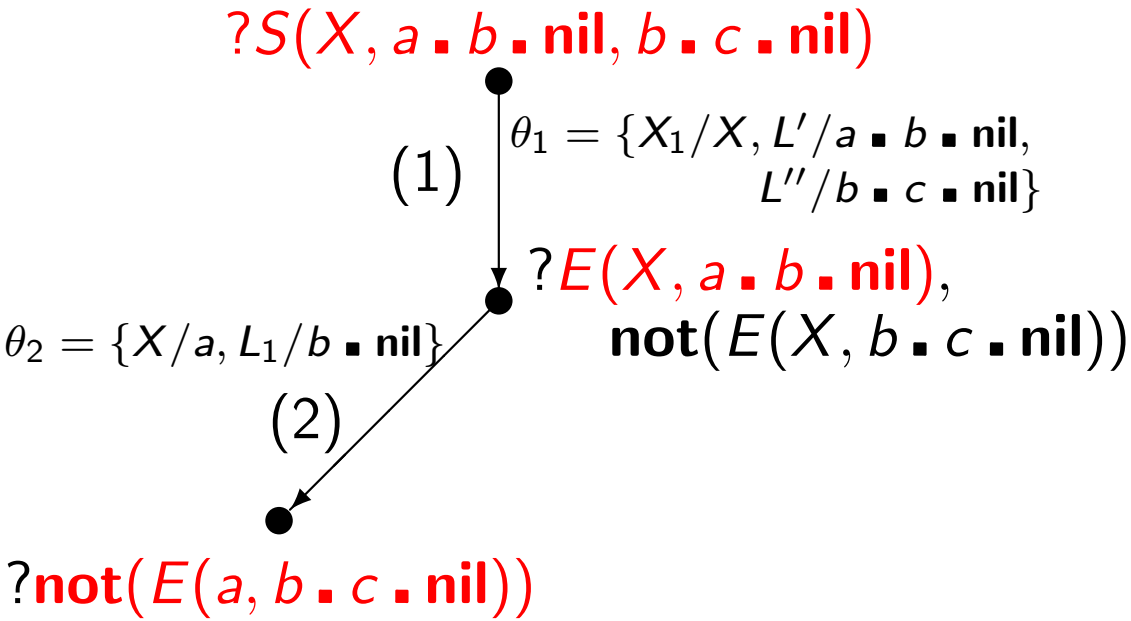


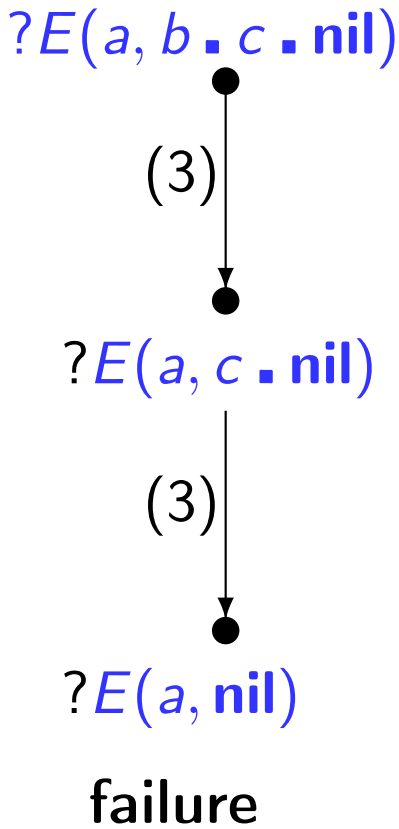
$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''));$ (1)
$E(X, X \centerdot L) \leftarrow;$ (2)
$E(X, Y \centerdot L) \leftarrow E(X, L);$ (3)



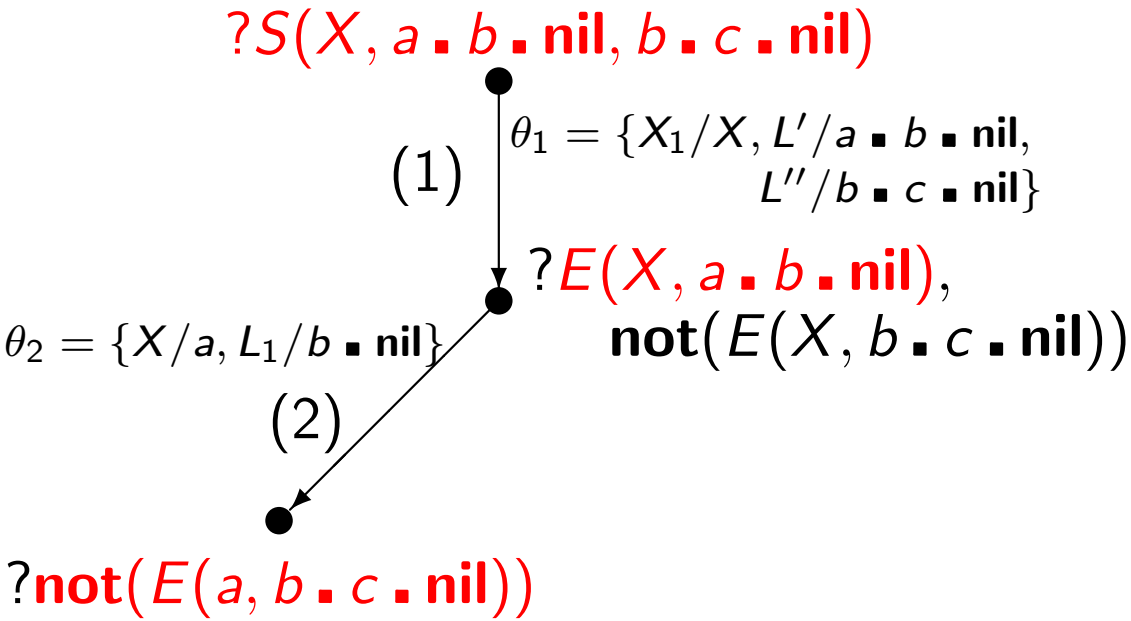


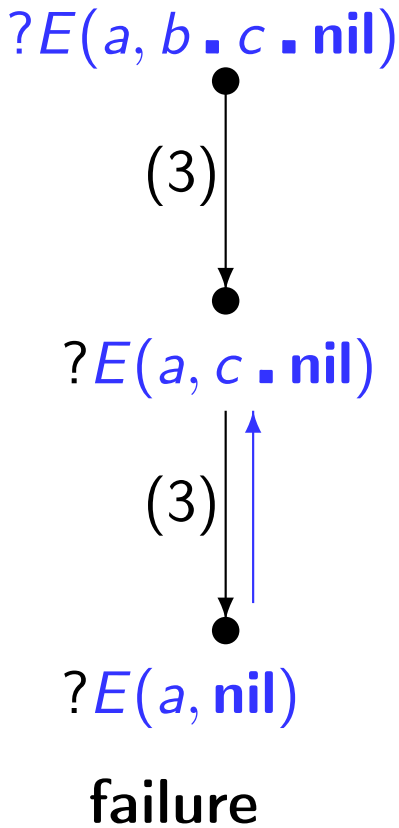
$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''));$ (1)
$E(X, X \centerdot L) \leftarrow;$ (2)
$E(X, Y \centerdot L) \leftarrow E(X, L);$ (3)



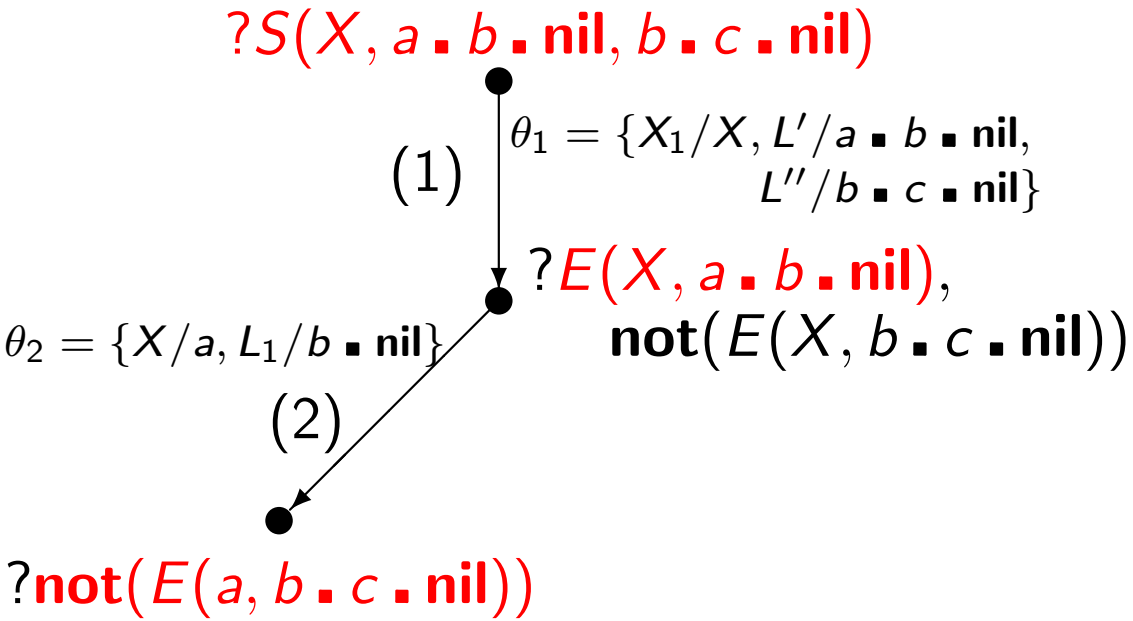


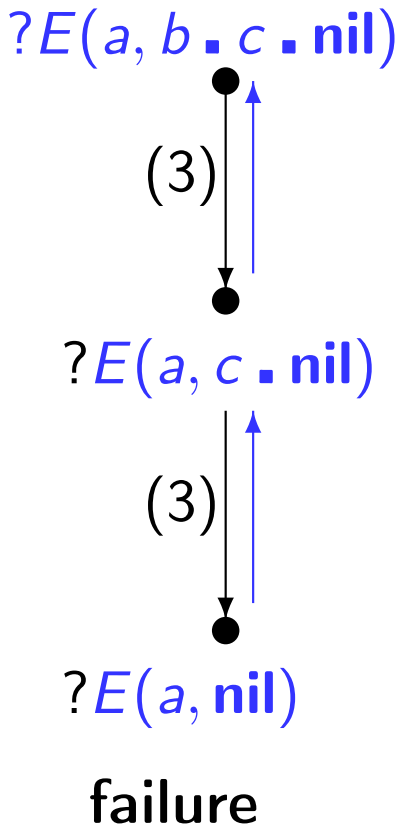
$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''));$ (1)
$E(X, X \centerdot L) \leftarrow;$ (2)
$E(X, Y \centerdot L) \leftarrow E(X, L);$ (3)



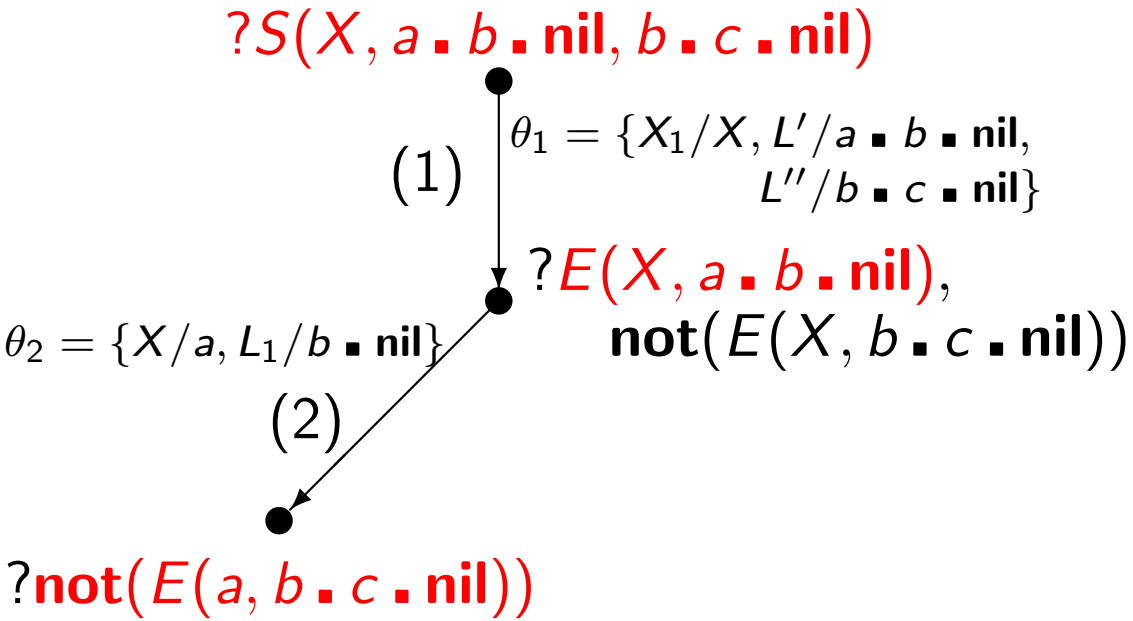


$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''));$ (1)
$E(X, X \centerdot L) \leftarrow;$ (2)
$E(X, Y \centerdot L) \leftarrow E(X, L);$ (3)

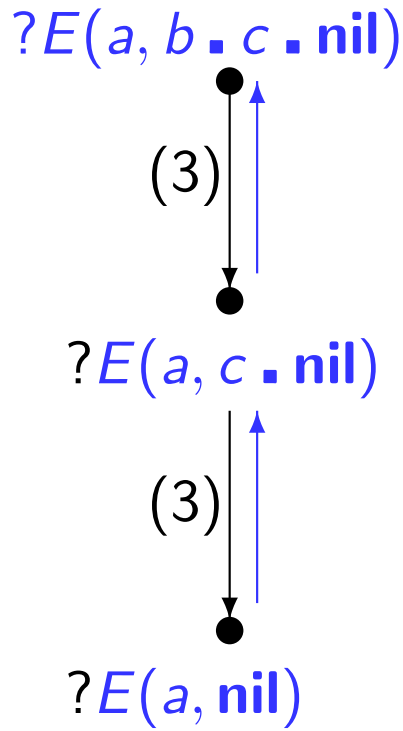




$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''));$ (1)
$E(X, X \centerdot L) \leftarrow;$ (2)
$E(X, Y \centerdot L) \leftarrow E(X, L);$ (3)

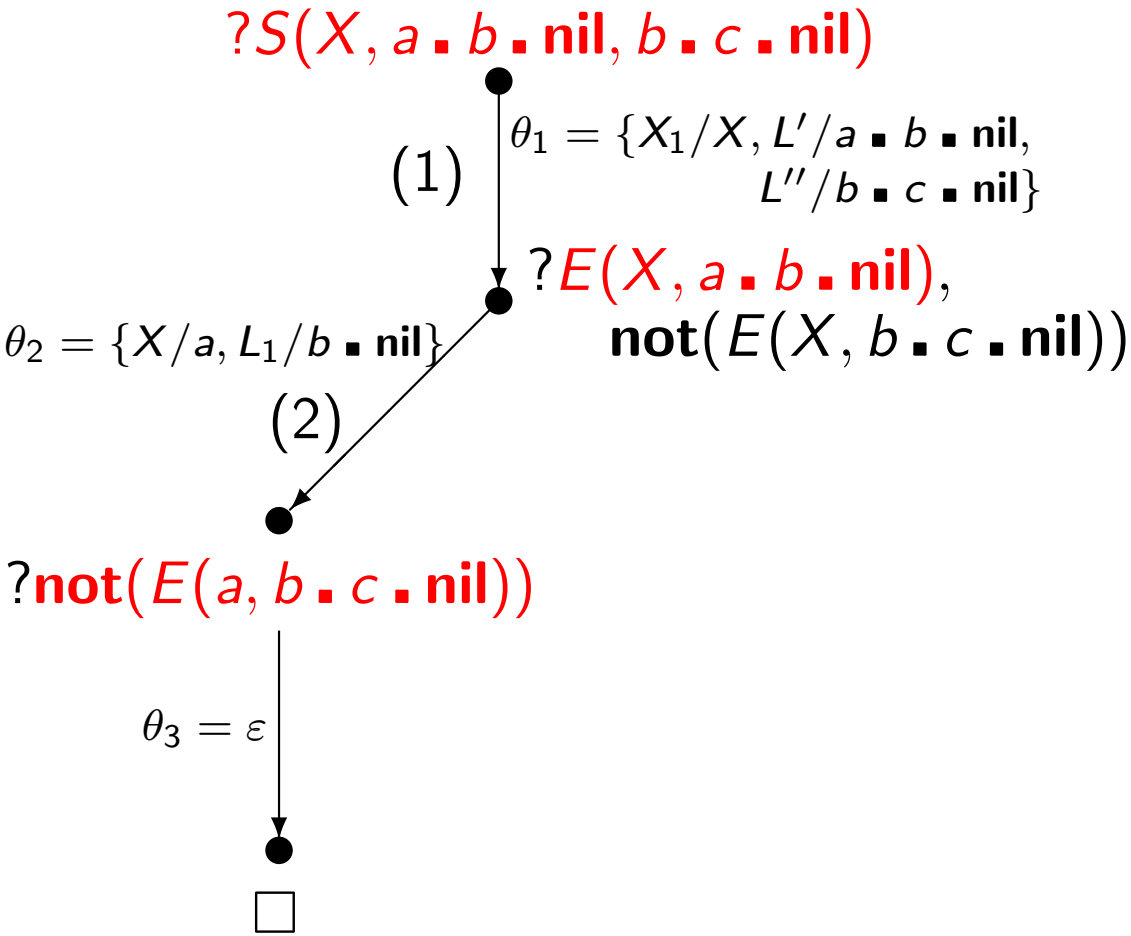


Успех



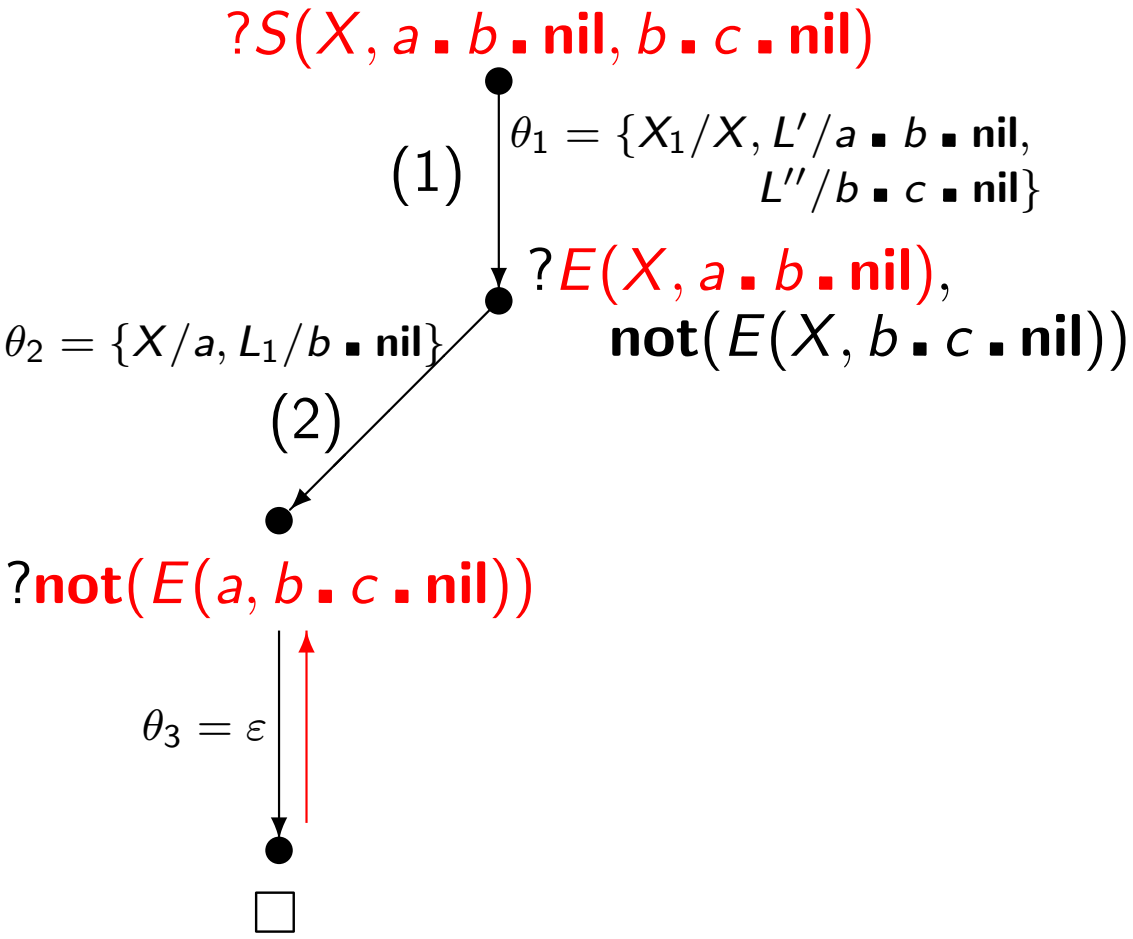
failure

$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''))$; (1)
$E(X, X \blacksquare L) \leftarrow$; (2)
$E(X, Y \blacksquare L) \leftarrow E(X, L)$; (3)



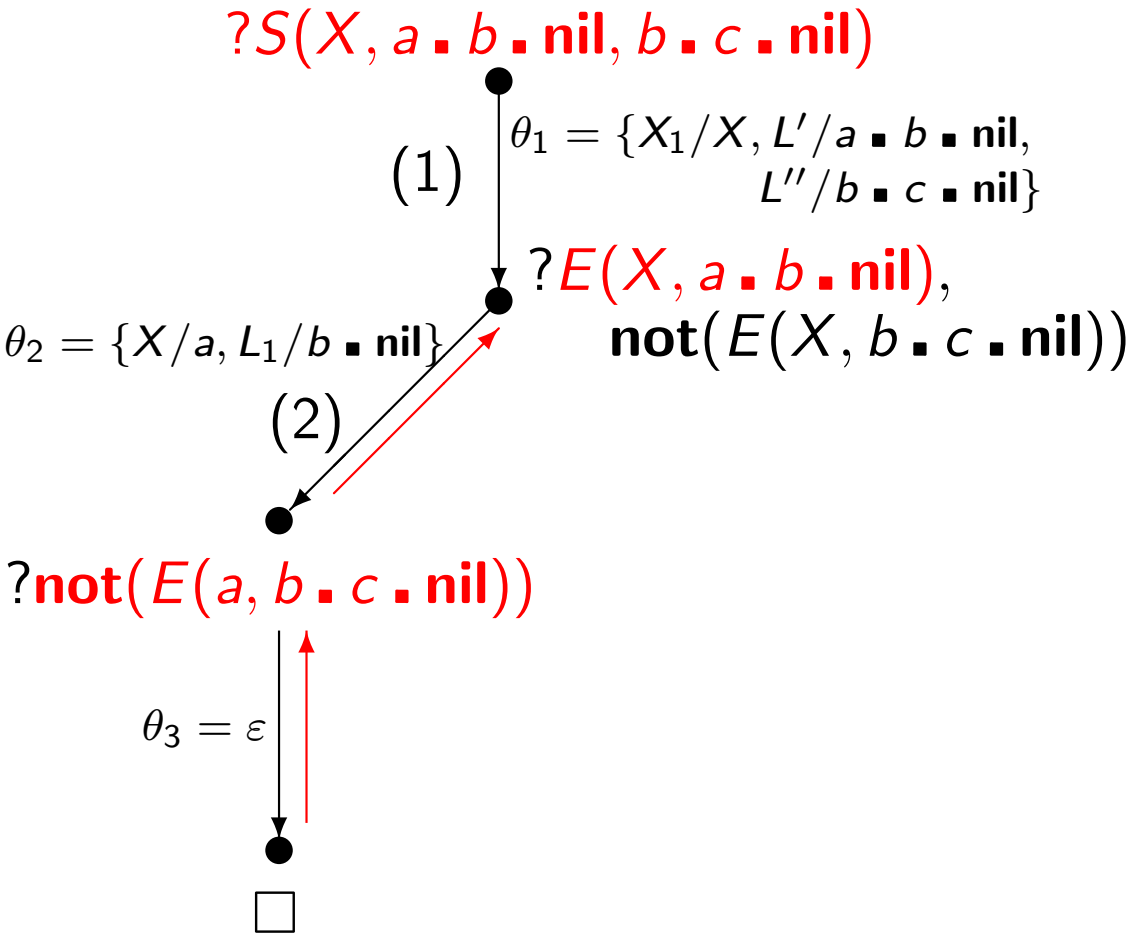
Ответ: $\{X/a\}$

$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''))$; (1)
$E(X, X \blacksquare L) \leftarrow$; (2)
$E(X, Y \blacksquare L) \leftarrow E(X, L)$; (3)



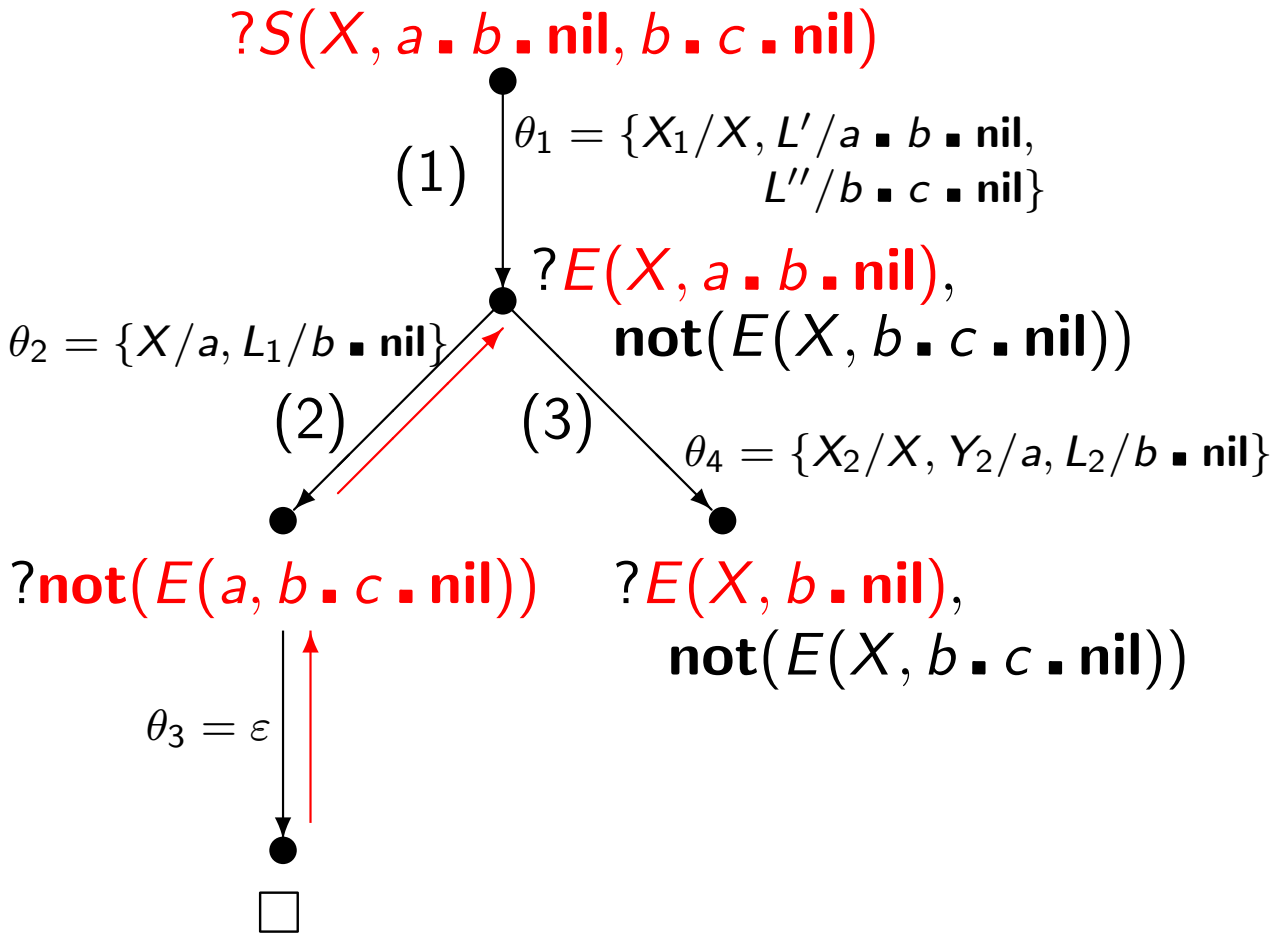
Ответ: $\{X/a\}$

$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''))$; (1)
$E(X, X \blacksquare L) \leftarrow$; (2)
$E(X, Y \blacksquare L) \leftarrow E(X, L)$; (3)



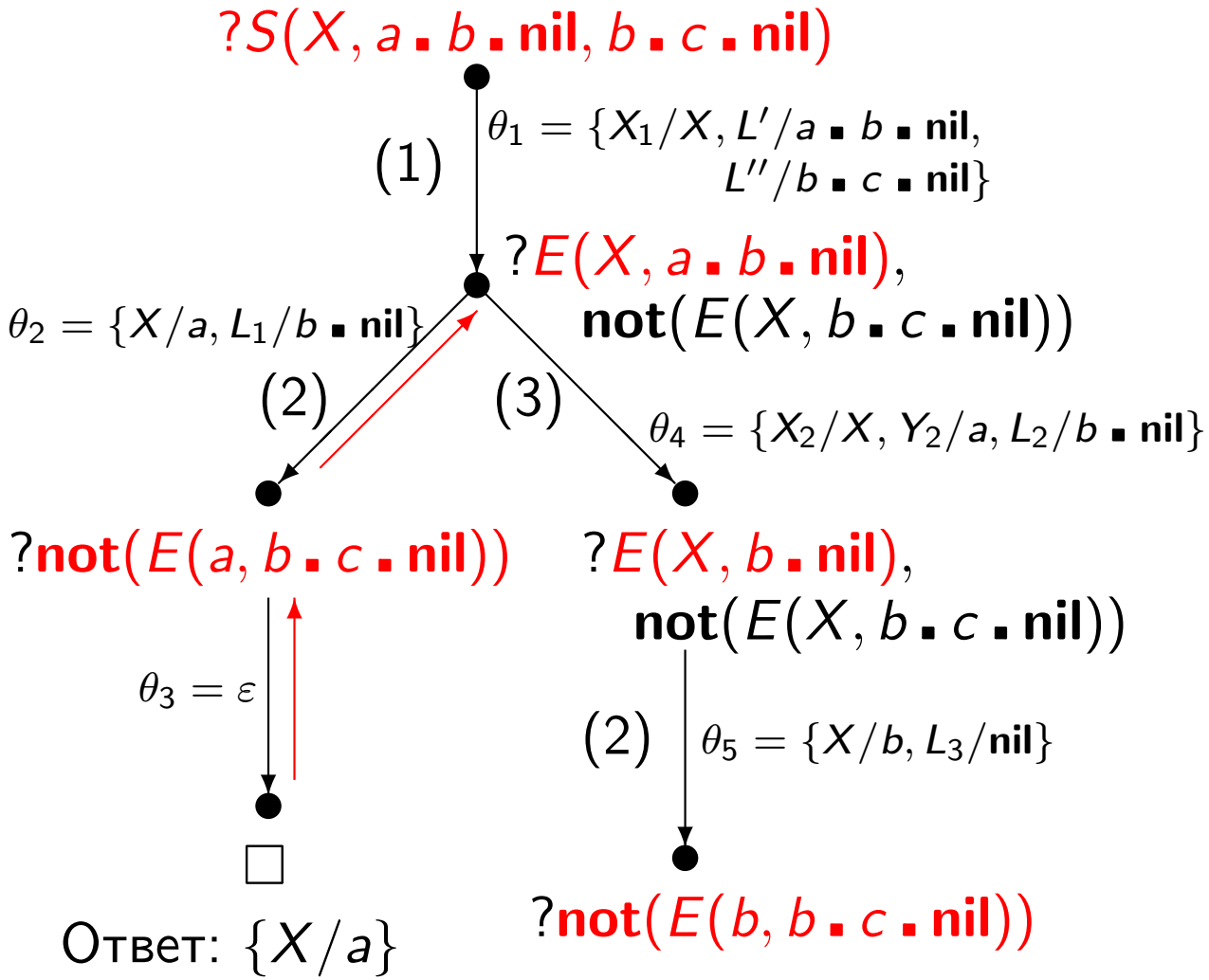
Ответ: $\{X/a\}$

$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''))$; (1)
$E(X, X \blacksquare L) \leftarrow$; (2)
$E(X, Y \blacksquare L) \leftarrow E(X, L)$; (3)



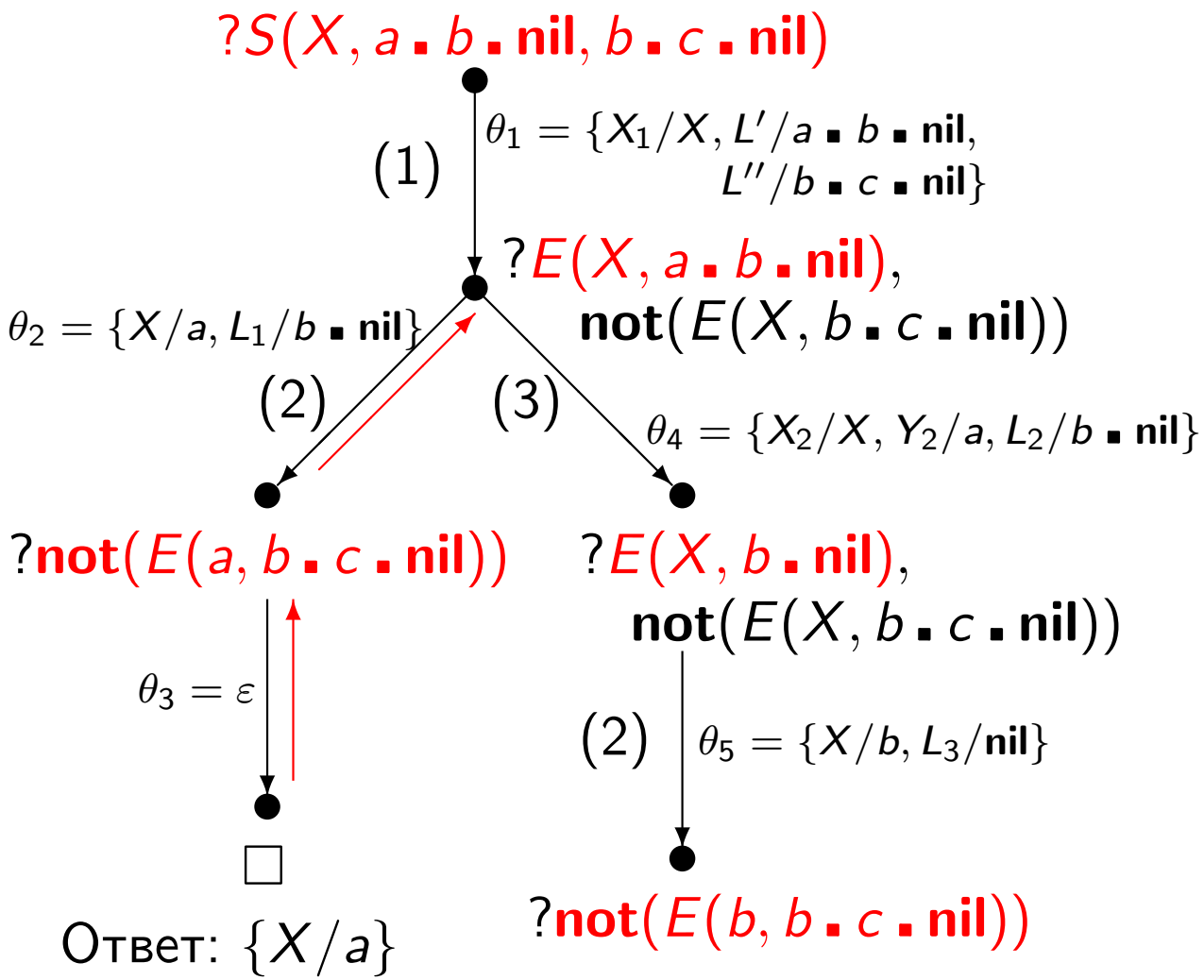
Ответ: $\{X/a\}$

$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''))$; (1)
$E(X, X \blacksquare L) \leftarrow$; (2)
$E(X, Y \blacksquare L) \leftarrow E(X, L)$; (3)



Ответ: $\{X/a\}$

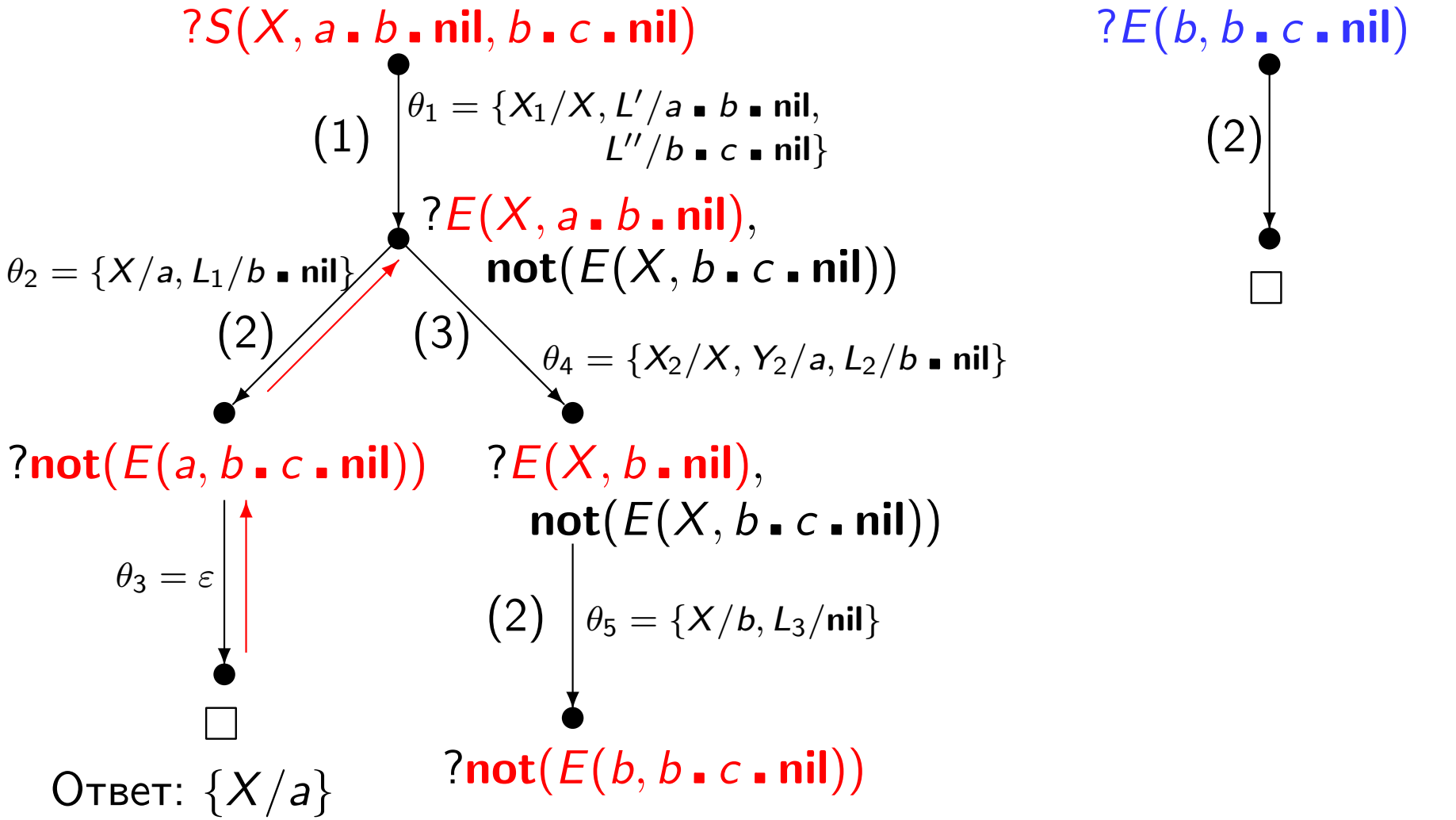
$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''))$; (1)
$E(X, X \centerdot L) \leftarrow$; (2)
$E(X, Y \centerdot L) \leftarrow E(X, L)$; (3)



?E(b, b . c . nil)

Ответ: $\{X/a\}$

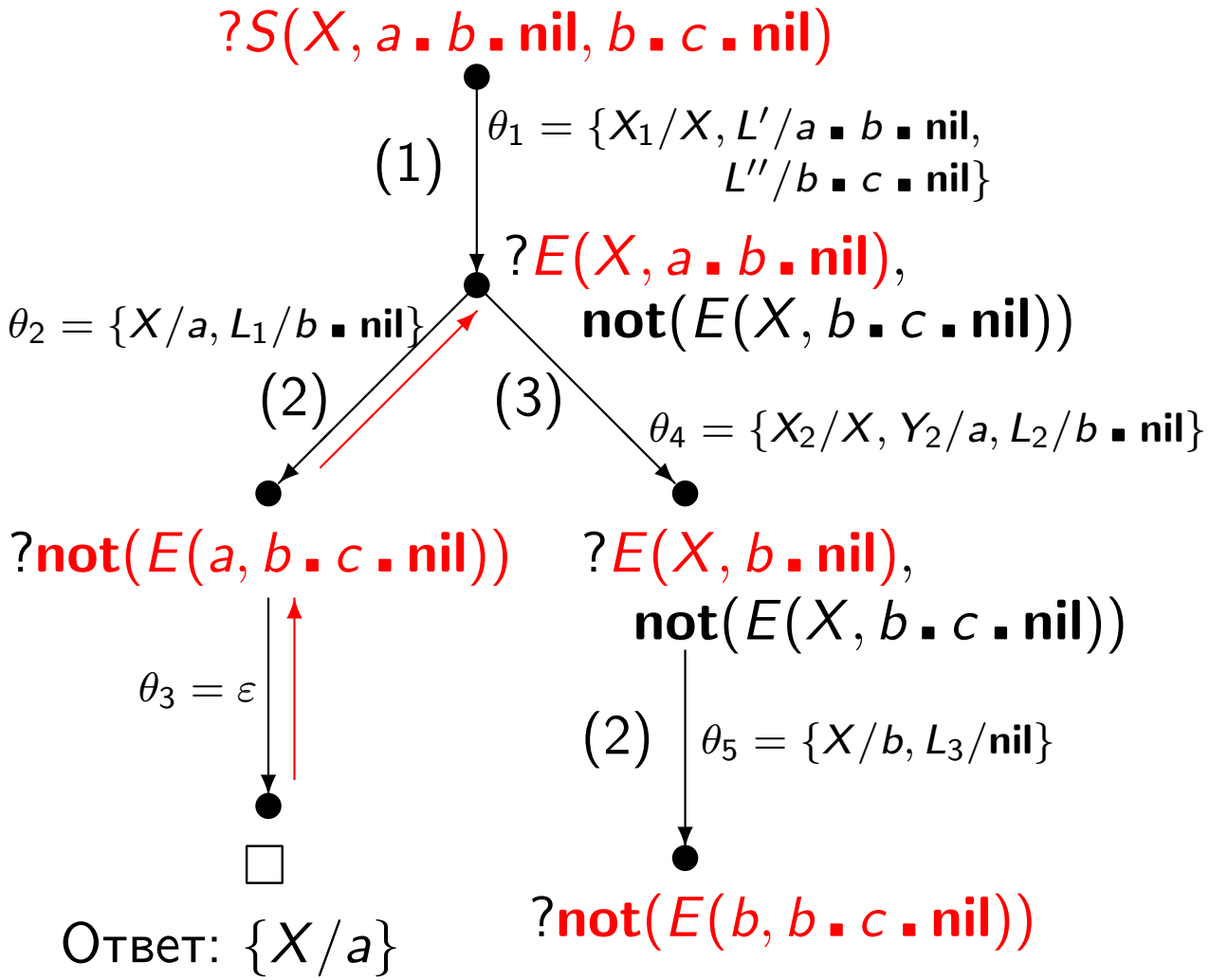
$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''))$; (1)
$E(X, X \centerdot L) \leftarrow$; (2)
$E(X, Y \centerdot L) \leftarrow E(X, L)$; (3)



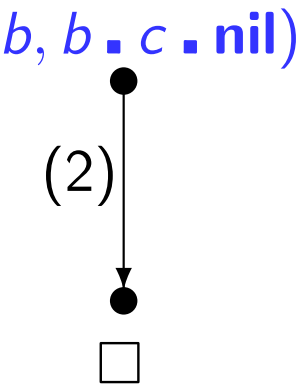
Ответ: $\{X/a\}$

$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''));$ (1)
$E(X, X \blacksquare L) \leftarrow;$ (2)
$E(X, Y \blacksquare L) \leftarrow E(X, L);$ (3)

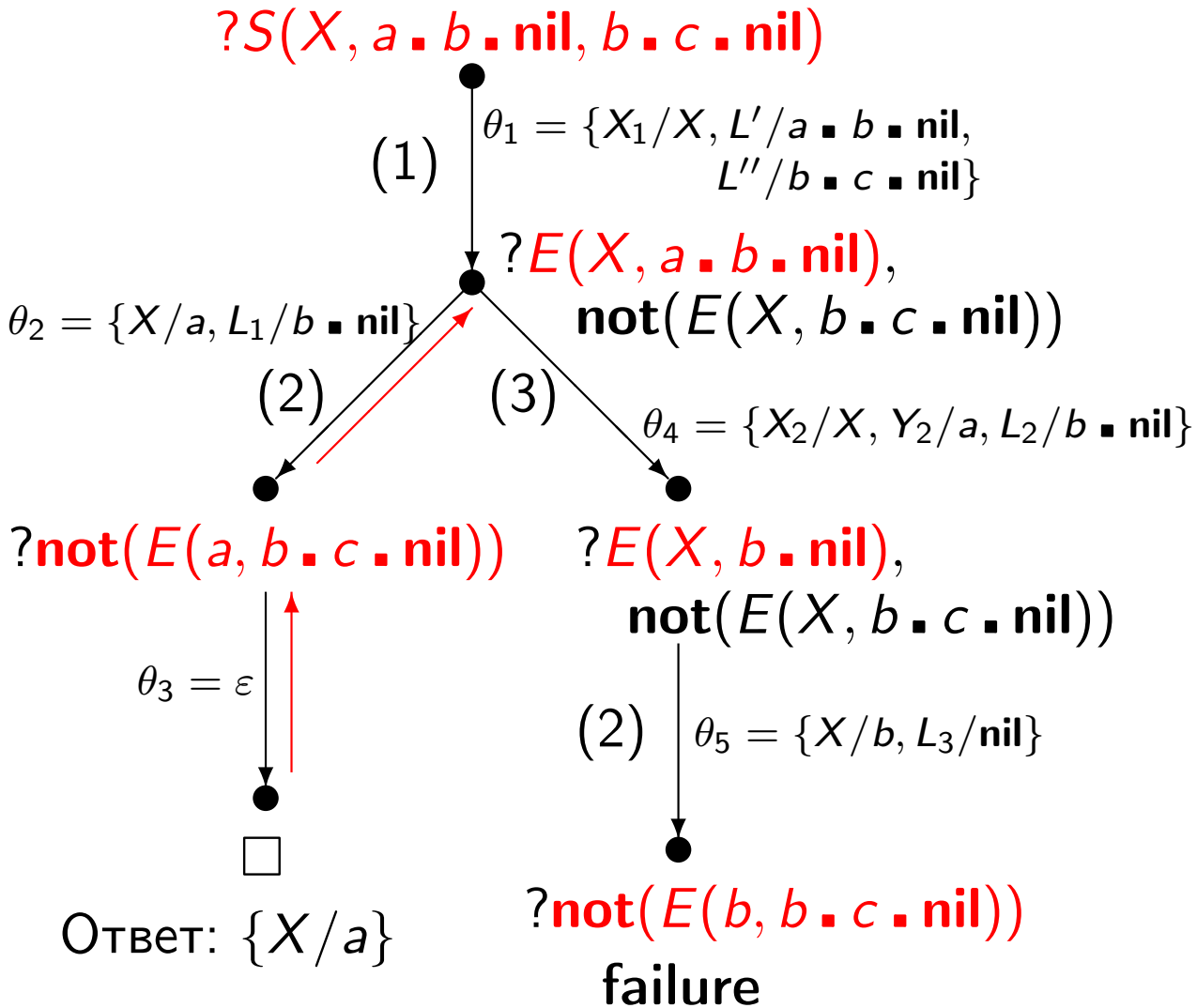
Неудача



Ответ: $\{X/a\}$

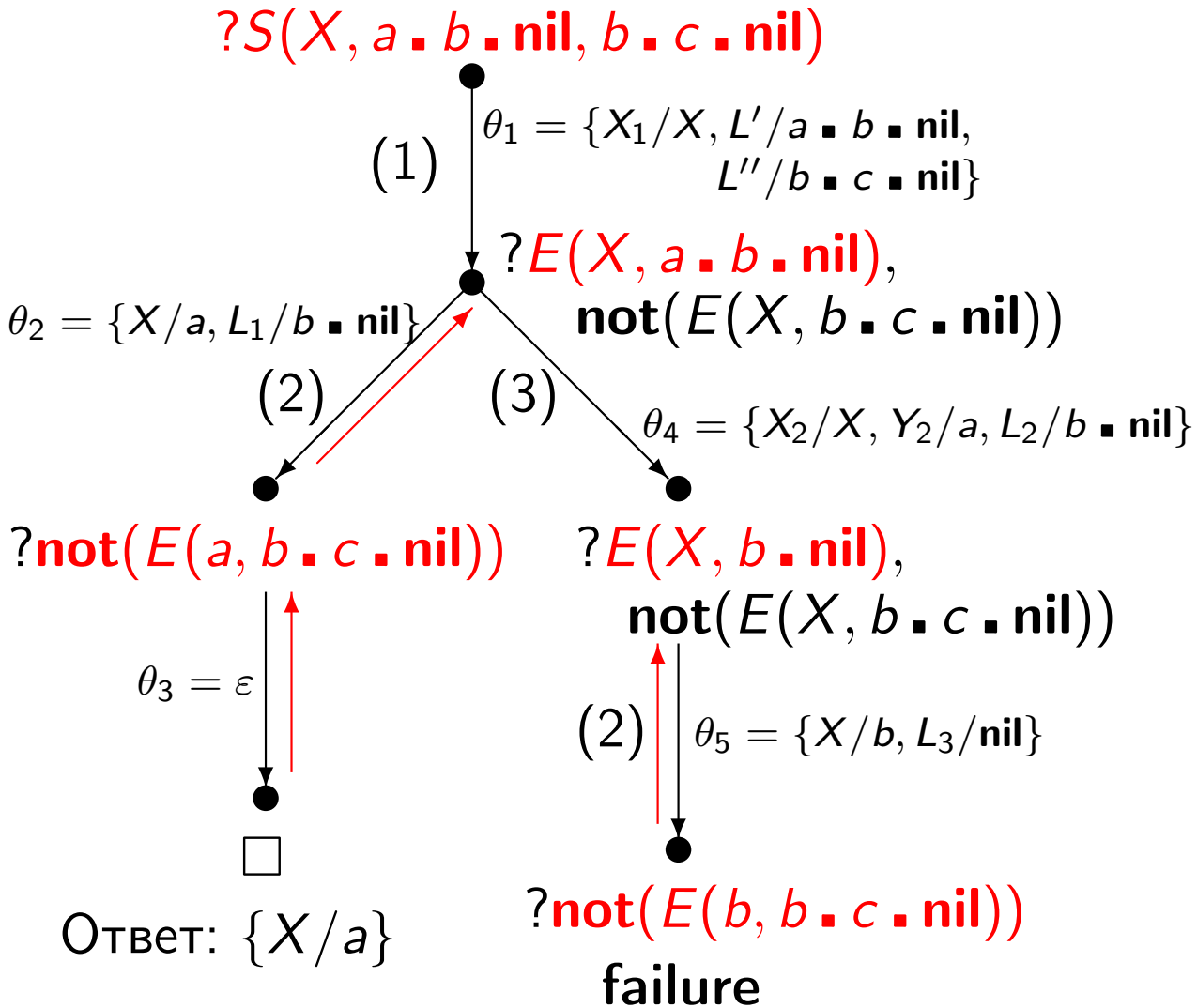


$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''))$; (1)
$E(X, X \blacksquare L) \leftarrow$; (2)
$E(X, Y \blacksquare L) \leftarrow E(X, L)$; (3)



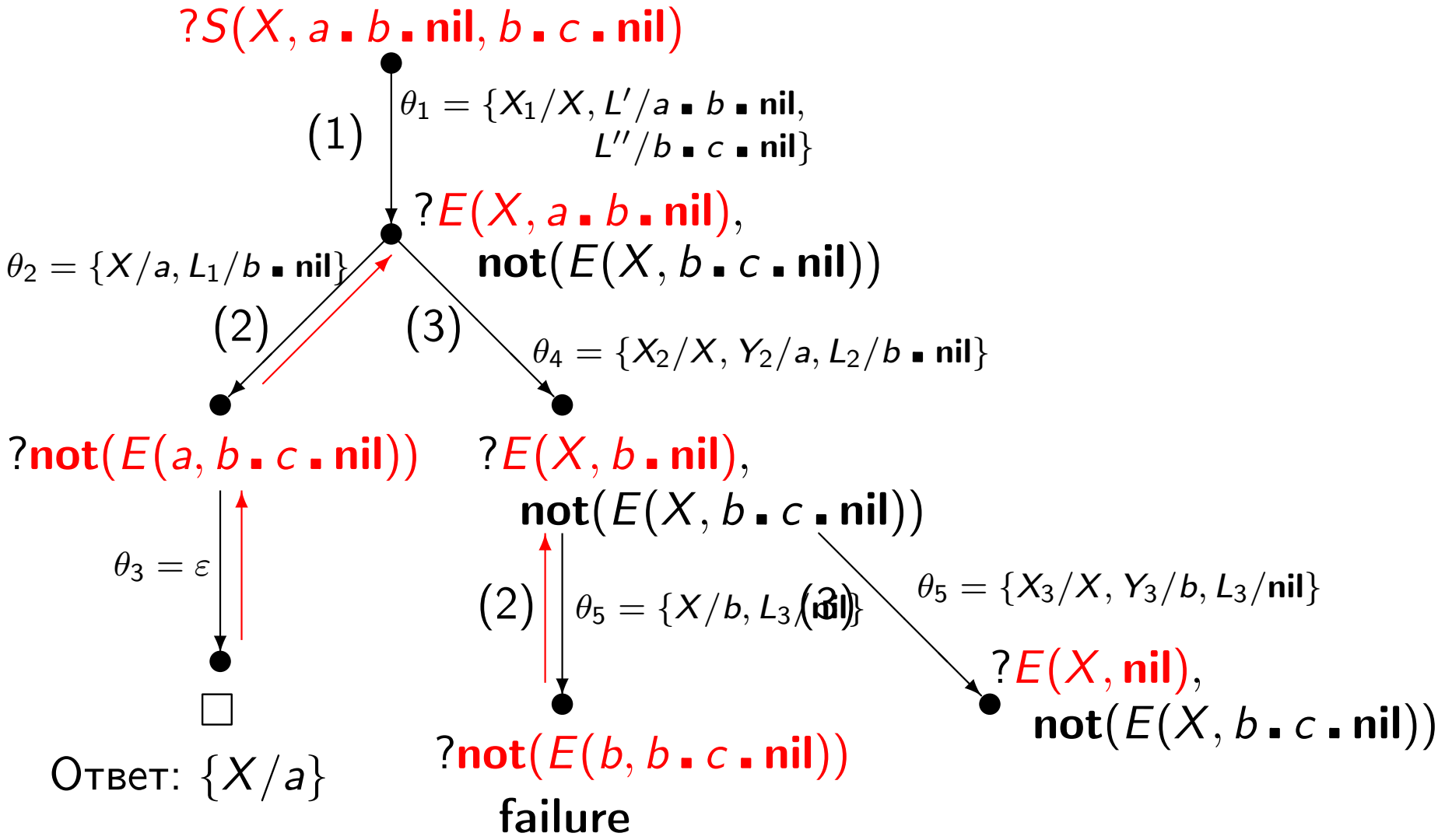
Ответ: $\{X/a\}$

$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''))$; (1)
$E(X, X \blacksquare L) \leftarrow$; (2)
$E(X, Y \blacksquare L) \leftarrow E(X, L)$; (3)

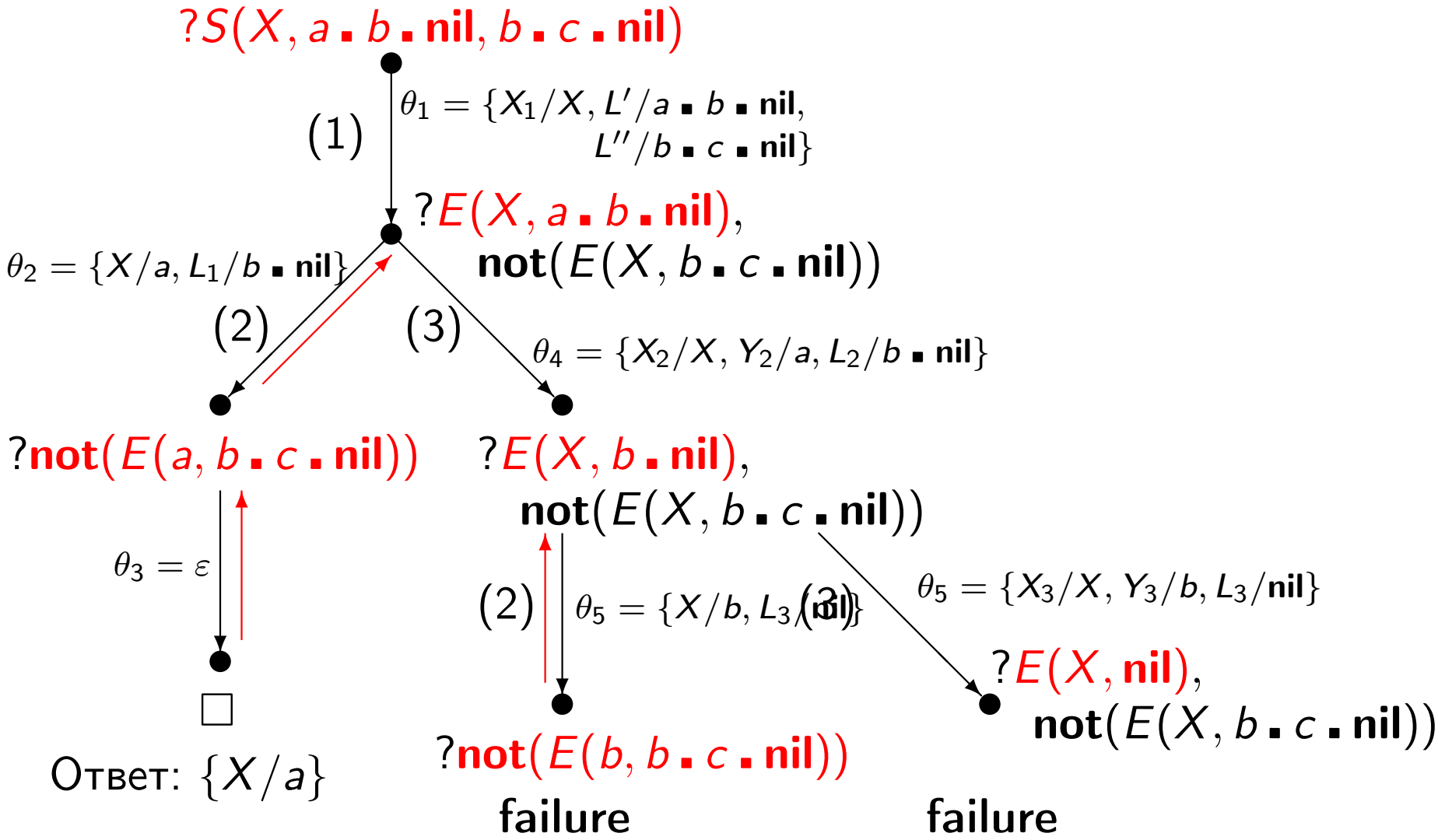


Ответ: $\{X/a\}$

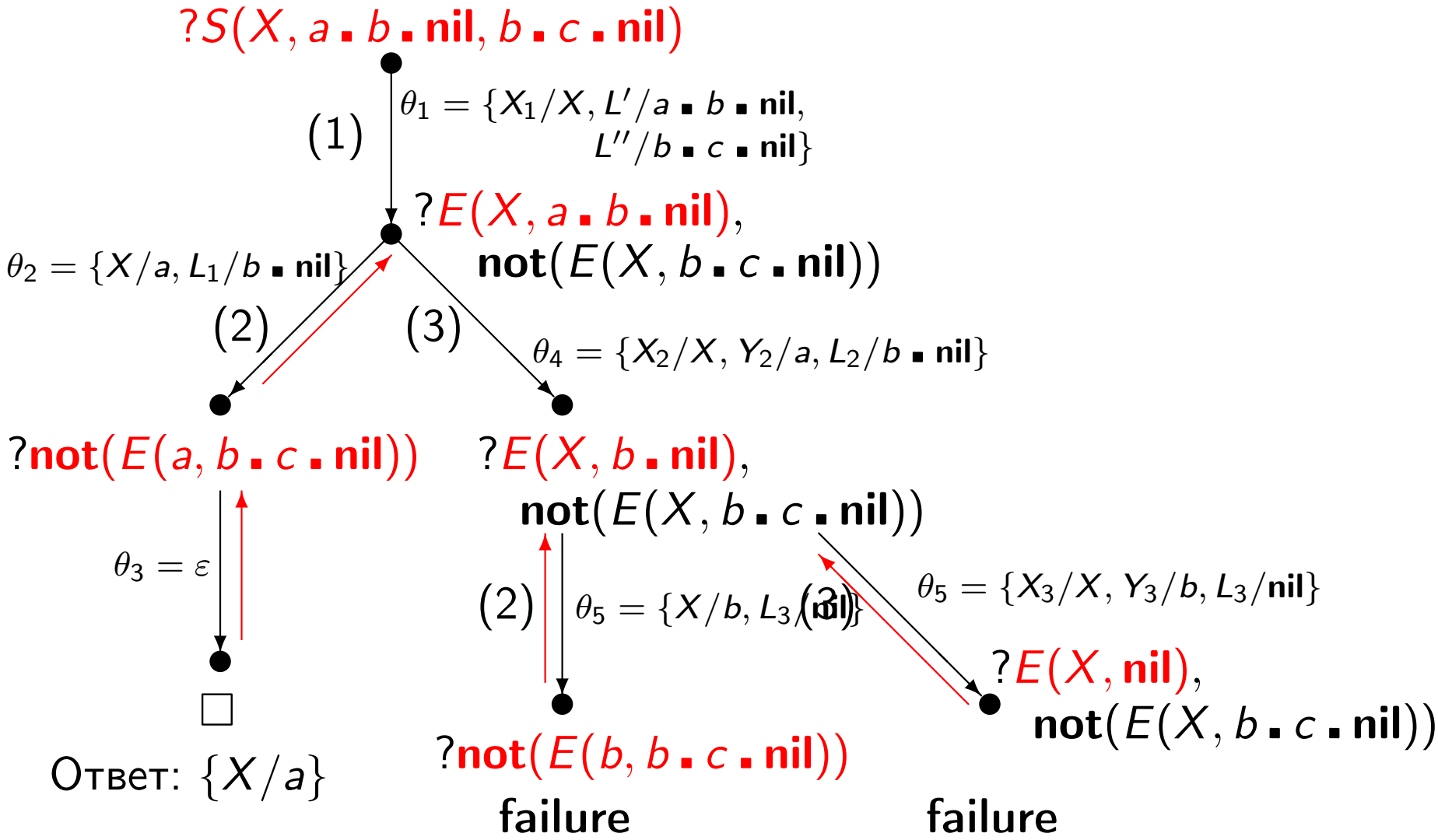
$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''))$; (1)
$E(X, X \centerdot L) \leftarrow$; (2)
$E(X, Y \centerdot L) \leftarrow E(X, L)$; (3)



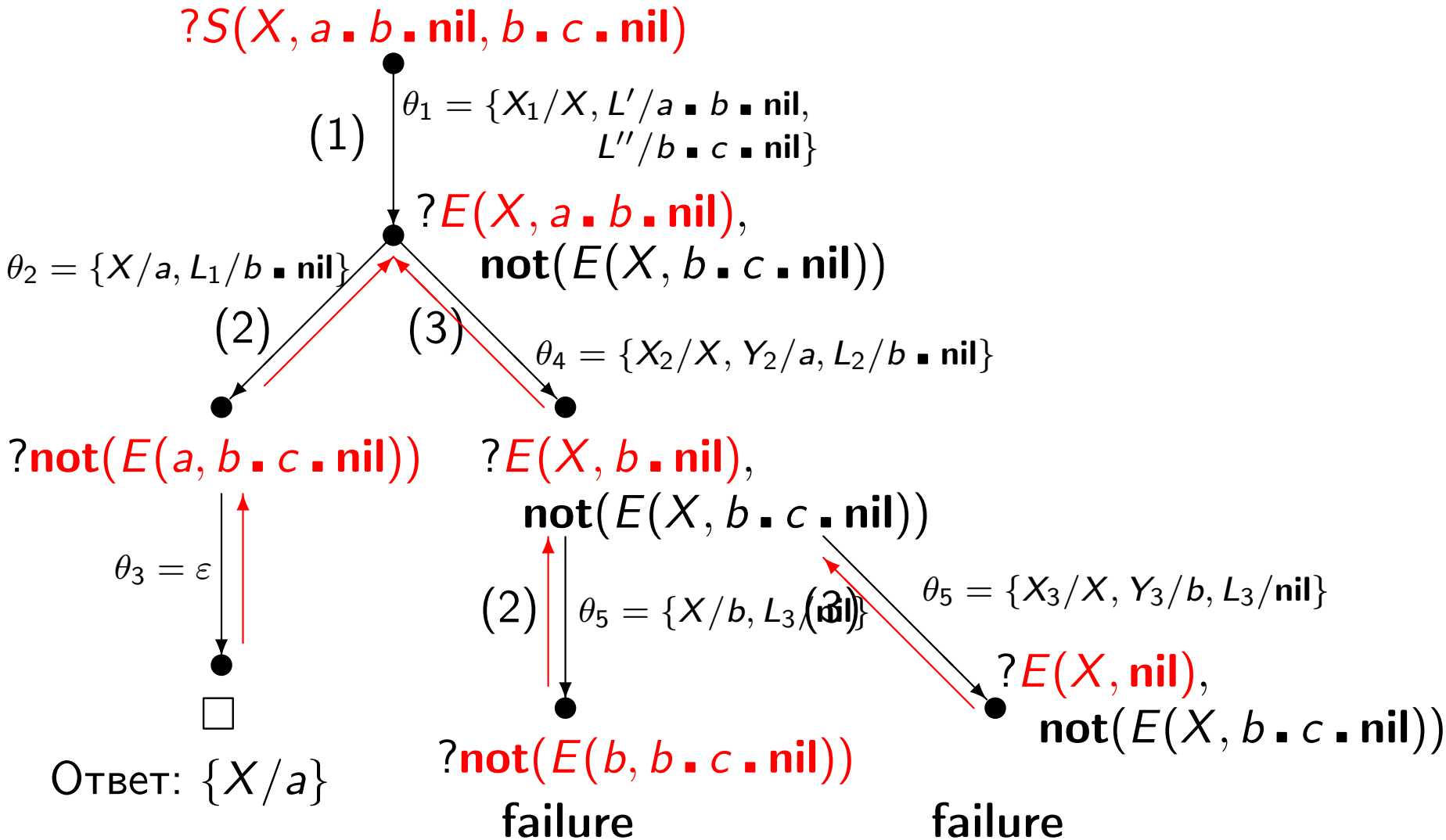
$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''));$ (1)
$E(X, X \centerdot L) \leftarrow;$ (2)
$E(X, Y \centerdot L) \leftarrow E(X, L);$ (3)



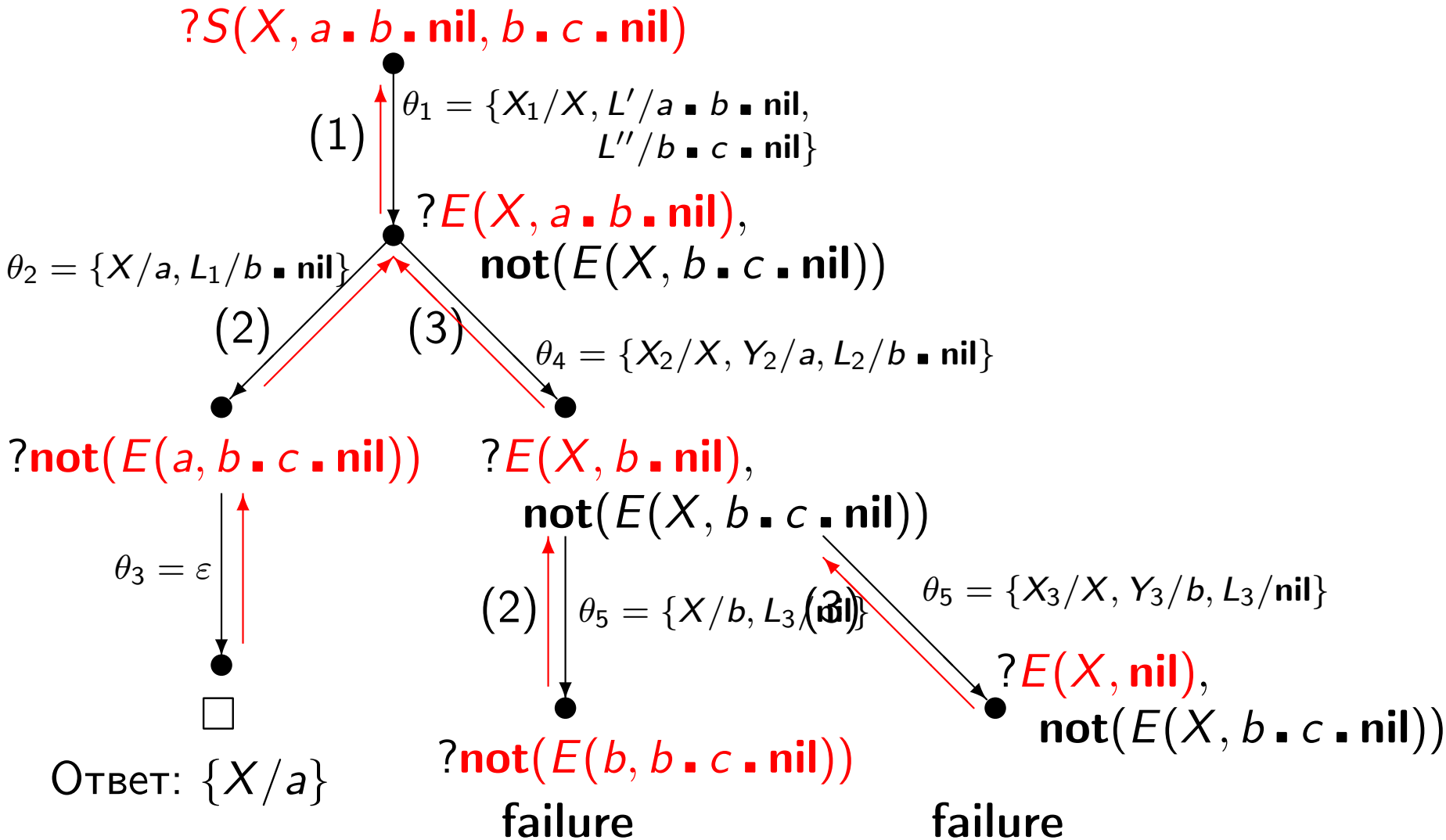
$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''));$ (1)
$E(X, X \centerdot L) \leftarrow;$ (2)
$E(X, Y \centerdot L) \leftarrow E(X, L);$ (3)



$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''));$ (1)
$E(X, X \centerdot L) \leftarrow;$ (2)
$E(X, Y \centerdot L) \leftarrow E(X, L);$ (3)



$S(X, L', L'') \leftarrow E(X, L'), \mathbf{not}(E(X, L''));$ (1)
$E(X, X \cdot L) \leftarrow;$ (2)
$E(X, Y \cdot L) \leftarrow E(X, L);$ (3)



ОПЕРАТОР not

В том случае, когда построение дерева SLD-резолютивных вычислений проводится по стандартной стратегии, оператор **not** можно выразить через оператор отсечения **!**.

Для этого введем логическую константу **fail** — 0-местный предикат, имеющий постоянное значение *false*. С точки зрения операционной семантики, **fail** — это задача, которая не имеет решений. Тогда...

для обработки запроса ? **not**(C_0) в рамках SLD-резолютивного вывода с отсечением достаточно добавить к программе $\mathcal{P}$ два дополнительных программных утверждения

$$\begin{aligned} \textbf{not}(C_0) &\leftarrow C_0, \textbf{!}, \textbf{fail}; \\ \textbf{not}(C_0) &\leftarrow ; \end{aligned}$$

ВСТРОЕННЫЕ ФУНКЦИИ И ПРЕДИКАТЫ

Хорновские логические программы — это универсальная модель вычислений, в которой можно реализовать любой алгоритм. Однако для удобства пользования логическими программами к ним можно добавлять специальные встроенные функции и предикаты, операционная семантика которых определяется **вне рамок** SLD- резолютивного вывода.

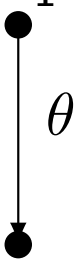
Рассмотрим некоторые наиболее широко используемые встроенные средства логического программирования.

ВСТРОЕННЫЕ ФУНКЦИИ И ПРЕДИКАТЫ

Предикат равенства $=$

Предикат равенства $t_1 = t_2$ — это 2-х местный предикат, предназначенный для унификации термов. Его операционная семантика задается следующими правилами:

$? \; t_1 = t_2$



$\theta \quad \Leftrightarrow \quad \begin{cases} HOY(t_1, t_2) \neq \emptyset \\ \theta \in HOY(t_1, t_2) \end{cases}$

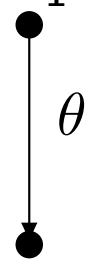
$\square$

ВСТРОЕННЫЕ ФУНКЦИИ И ПРЕДИКАТЫ

Предикат равенства $=$

Предикат равенства $t_1 = t_2$ — это 2-х местный предикат, предназначенный для унификации термов. Его операционная семантика задается следующими правилами:

$? \; t_1 = t_2$



$\theta \quad \Leftrightarrow \quad \begin{cases} HOY(t_1, t_2) \neq \emptyset \\ \theta \in HOY(t_1, t_2) \end{cases}$

$? \; t_1 = t_2$

failure $\quad \Leftrightarrow \quad HOY(t_1, t_2) = \emptyset$

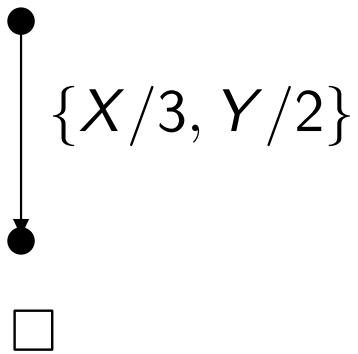
$\square$

ВСТРОЕННЫЕ ФУНКЦИИ И ПРЕДИКАТЫ

Предикат равенства $=$

Пример

$? \; X + 2 = 3 + Y$

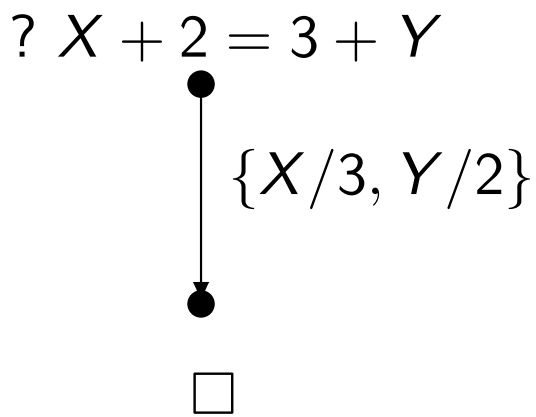


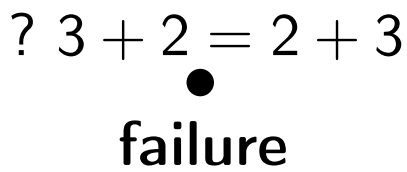
$\square$

ВСТРОЕННЫЕ ФУНКЦИИ И ПРЕДИКАТЫ

Предикат равенства $=$

Пример



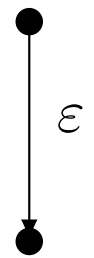


ВСТРОЕННЫЕ ФУНКЦИИ И ПРЕДИКАТЫ

Предикат тождества $==$

Предикат тождества $t_1 == t_2$ — это 2-х местный предикат, предназначенный для проверки тождественности (синтаксической идентичности) термов. Его операционная семантика задается следующими правилами:

$? \; t == t$



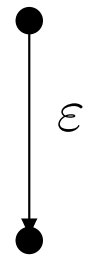
$\square$

ВСТРОЕННЫЕ ФУНКЦИИ И ПРЕДИКАТЫ

Предикат тождества $==$

Предикат тождества $t_1 == t_2$ — это 2-х местный предикат, предназначенный для проверки тождественности (синтаксической идентичности) термов. Его операционная семантика задается следующими правилами:

$? \; t == t$



$\square$

$? \; t_1 == t_2$

failure

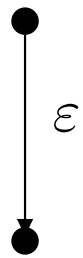
$\Leftrightarrow$ t_1, t_2 — разные термы

ВСТРОЕННЫЕ ФУНКЦИИ И ПРЕДИКАТЫ

Предикат тождества $==$

Пример

$? \; X + 2 == X + 2$

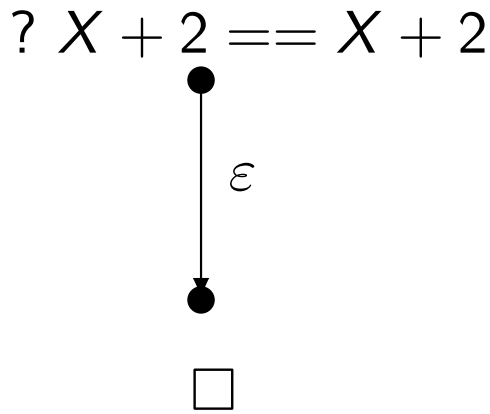


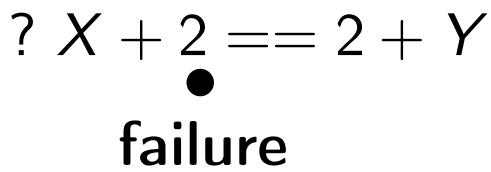
$\square$

ВСТРОЕННЫЕ ФУНКЦИИ И ПРЕДИКАТЫ

Предикат тождества ==

Пример





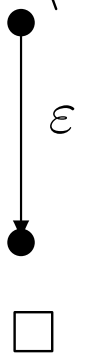
ВСТРОЕННЫЕ ФУНКЦИИ И ПРЕДИКАТЫ

Предикаты неравенства $\backslash =$, $= \backslash =$

Предикаты неравенства $t_1 \backslash = t_2$, $t_1 = \backslash = t_2$ определяются при помощи оператора отрицания выражениями $\mathbf{not}(t_1 = t_2)$ и $\mathbf{not}(t_1 == t_2)$.

Пример

$? \; 3 + 2 \backslash = 2 + 3$



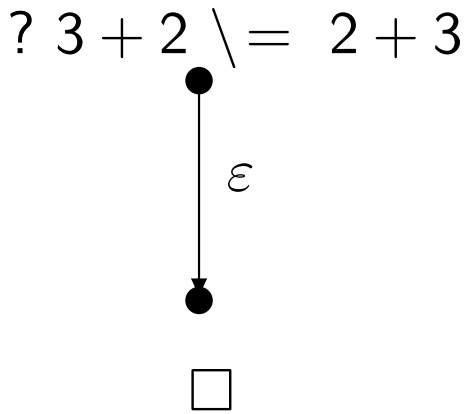
$\square$

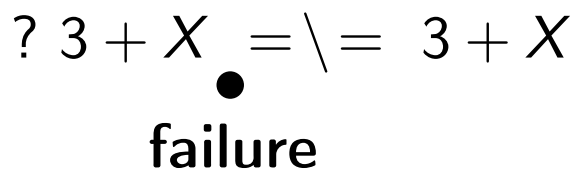
ВСТРОЕННЫЕ ФУНКЦИИ И ПРЕДИКАТЫ

Предикаты неравенства $\backslash =$, $= \backslash =$

Предикаты неравенства $t_1 \backslash = \ t_2$, $t_1 \ = \backslash = t_2$ определяются при помощи оператора отрицания выражениями $\mathbf{not}(t_1 = t_2)$ и $\mathbf{not}(t_1 == t_2)$.

Пример





ВСТРОЕННЫЕ ФУНКЦИИ И ПРЕДИКАТЫ

Типы данных

Для определения более сложных встроенных предикатов вводятся типы данных

$$\textbf{integer}, \textbf{real}, \textbf{boolean}, \textbf{char}, \text{ и т. д.}$$

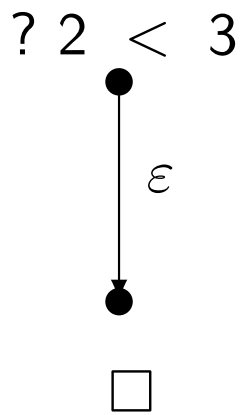
Каждый тип данных определяется множеством констант, обозначающих элементы этого типа данных. Например, **boolean** $= \{true, false\}$.

Тогда в каждом типе данных вводятся необходимые отношения, присущие элементам этого типа. Например, в типе данных **integer** можно ввести отношения сравнения $<$, $<=$, $>=$, $>$, и др.

ВСТРОЕННЫЕ ФУНКЦИИ И ПРЕДИКАТЫ

Типы данных

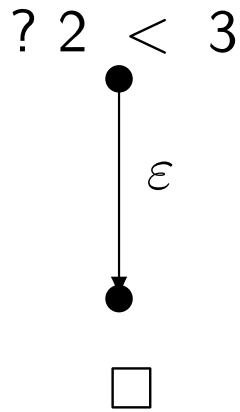
Пример

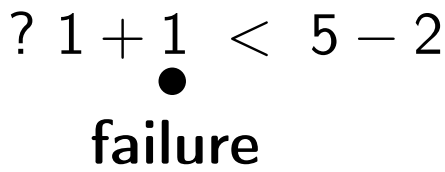


ВСТРОЕННЫЕ ФУНКЦИИ И ПРЕДИКАТЫ

Типы данных

Пример



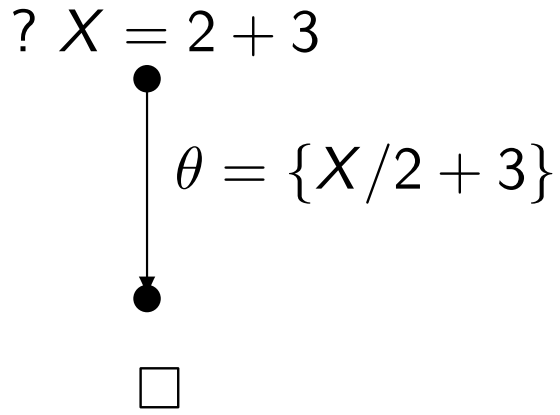


ВСТРОЕННЫЕ ФУНКЦИИ И ПРЕДИКАТЫ

Типы данных

Для введенных типов данных можно ввести встроенные функции (операции), присущие элементам этих типов
Например, в типе данных **integer** можно ввести двухместные функции (операции) $+$, $-$, $\times$, **div**, и др.

Однако чтобы вычисленные значения можно было передавать переменным, необходимо специальное средство. Предикат равенства $=$ для этой цели не годится, поскольку он занимается лишь унификацией термов.



ПРЕДИКАТ ВЫЧИСЛЕНИЯ ЗНАЧЕНИЙ

Предикат вычисления значений **is** — это встроенный предикат, предназначенный для вычисления значений встроенных функций. Операционная семантика этого предиката задается следующими правилами.

Условимся использовать запись $val(t)$ для обозначения значение терма t, составленного из встроенных функций и констант. Тогда...

? t_1 **is** t_2

$$\theta \quad \Leftrightarrow \quad \begin{cases} t_1 \in Var, \\ \text{определено } val(t_2) \end{cases}$$

$\square$ $\qquad \theta = \{t_1/val(t_2)\}$

ПРЕДИКАТ ВЫЧИСЛЕНИЯ ЗНАЧЕНИЙ

Предикат вычисления значений **is** — это встроенный предикат, предназначенный для вычисления значений встроенных функций. Операционная семантика этого предиката задается следующими правилами.

Условимся использовать запись $val(t)$ для обозначения значение терма t, составленного из встроенных функций и констант. Тогда...

? t_1 **is** t_2

$$\theta \quad \Leftrightarrow \quad \begin{cases} t_1 \in Var, \\ \text{определено } val(t_2) \end{cases}$$

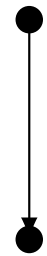
$\square$ $\qquad \theta = \{t_1/val(t_2)\}$

ПРЕДИКАТ ВЫЧИСЛЕНИЯ ЗНАЧЕНИЙ

Предикат вычисления значений **is** — это встроенный предикат, предназначенный для вычисления значений встроенных функций. Операционная семантика этого предиката задается следующими правилами.

Условимся использовать запись $val(t)$ для обозначения значение терма t, составленного из встроенных функций и констант. Тогда...

? t_1 **is** t_2



$\theta \quad \Leftrightarrow \quad \begin{cases} t_1 \in Var, \\ \text{определено } val(t_2) \end{cases}$

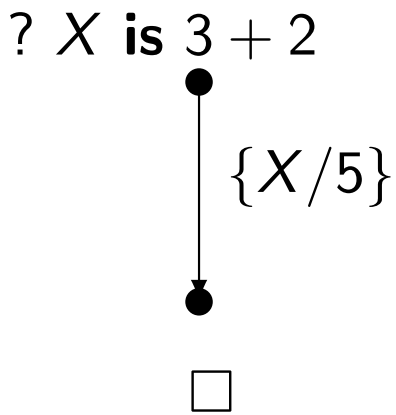
? t_1 **is** t_2 $\Leftrightarrow$ в остальных случаях

failure

$\square \qquad \theta = \{t_1/val(t_2)\}$

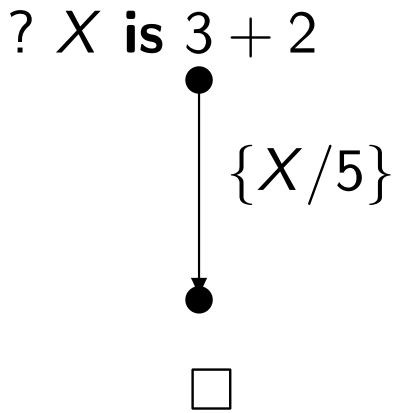
ПРЕДИКАТ ВЫЧИСЛЕНИЯ ЗНАЧЕНИЙ

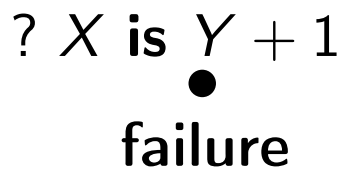
Пример



ПРЕДИКАТ ВЫЧИСЛЕНИЯ ЗНАЧЕНИЙ

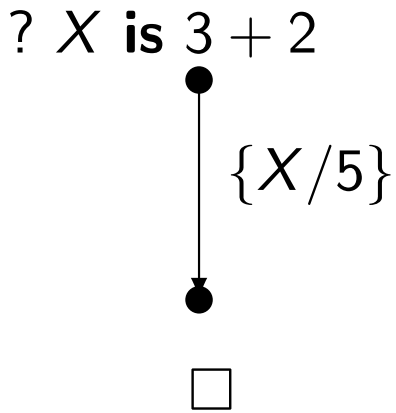
Пример

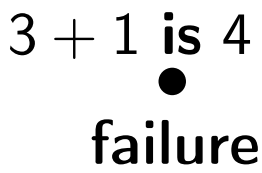




ПРЕДИКАТ ВЫЧИСЛЕНИЯ ЗНАЧЕНИЙ

Пример





МОДИФИКАЦИЯ БАЗ ДАННЫХ

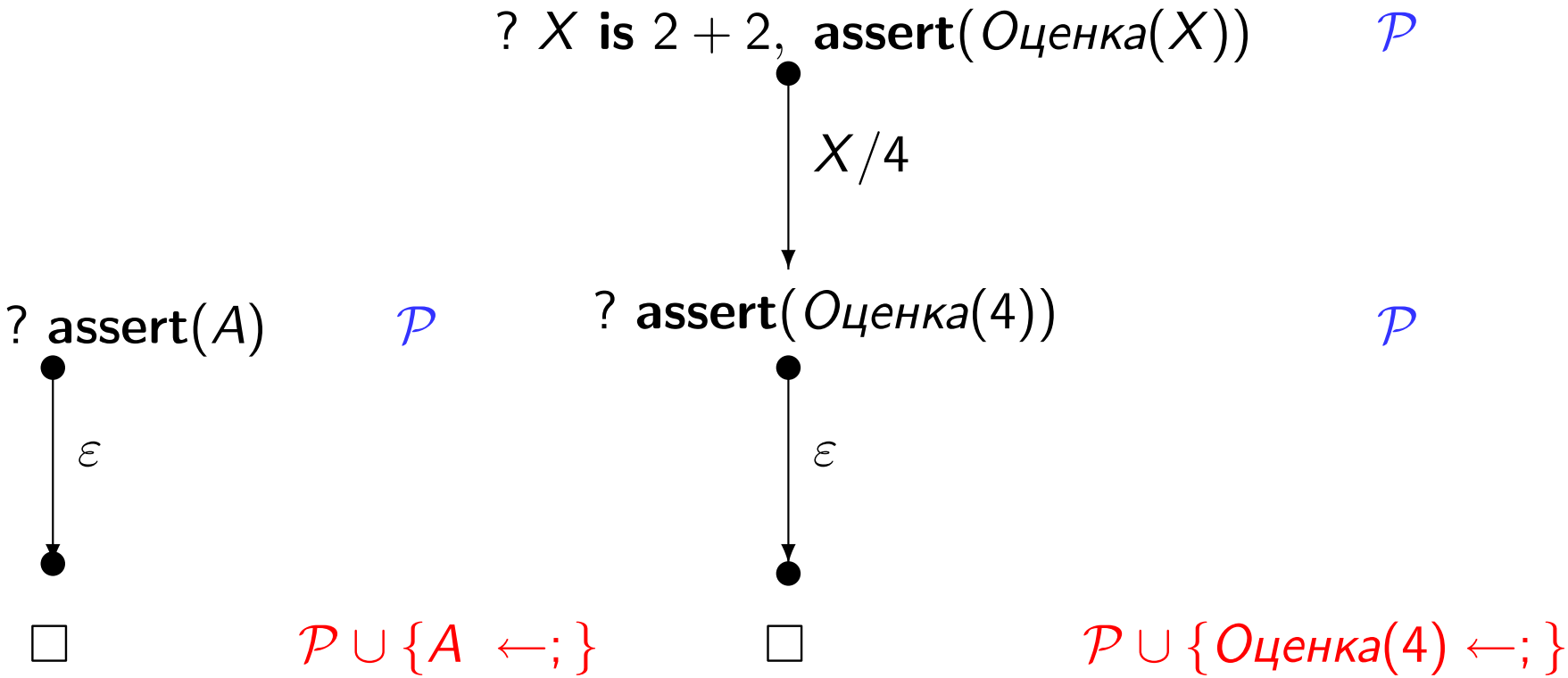
Обычно логическая программа $\mathcal{P}$ разделяется на две части — систему правил $\mathcal{P}_{clauses}$ и базу данных $\mathcal{P}_{database}$. База данных состоит из всех основных фактов программы.

Система правил $\mathcal{P}_{clauses}$ представляет, по сути дела, алгоритм решения задачи, и его неразумно изменять по ходу решения задачи.

А вот базу данных $\mathcal{P}_{database}$ по ходу вычисления можно модиифцировать. И для этой цели в системах логического программирования есть специальные встроенные операторы пополнения и сокращения базы данных.

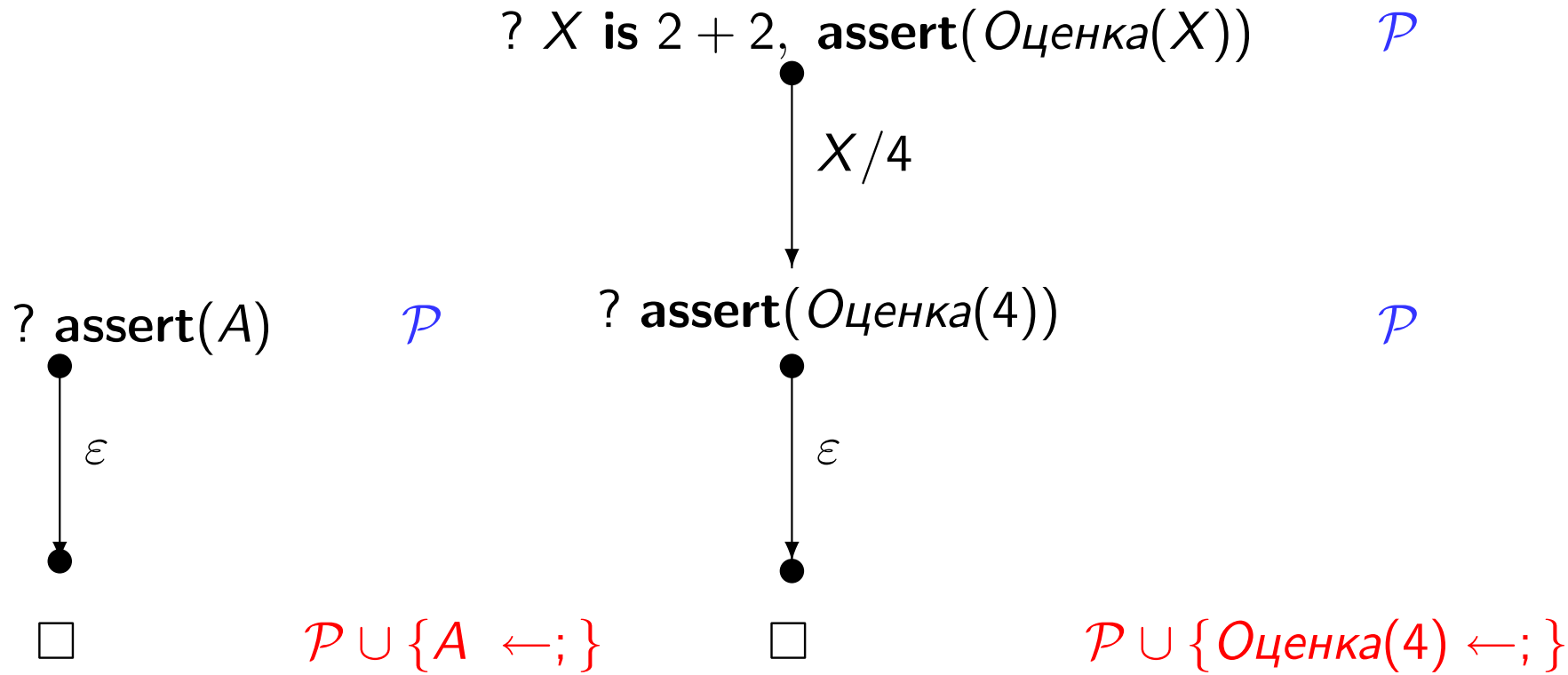
МОДИФИКАЦИЯ БАЗ ДАННЫХ

Оператор пополнения базы данных **assert**(A), где A — атом, добавляет к базе данных факт $A \leftarrow$;



МОДИФИКАЦИЯ БАЗ ДАННЫХ

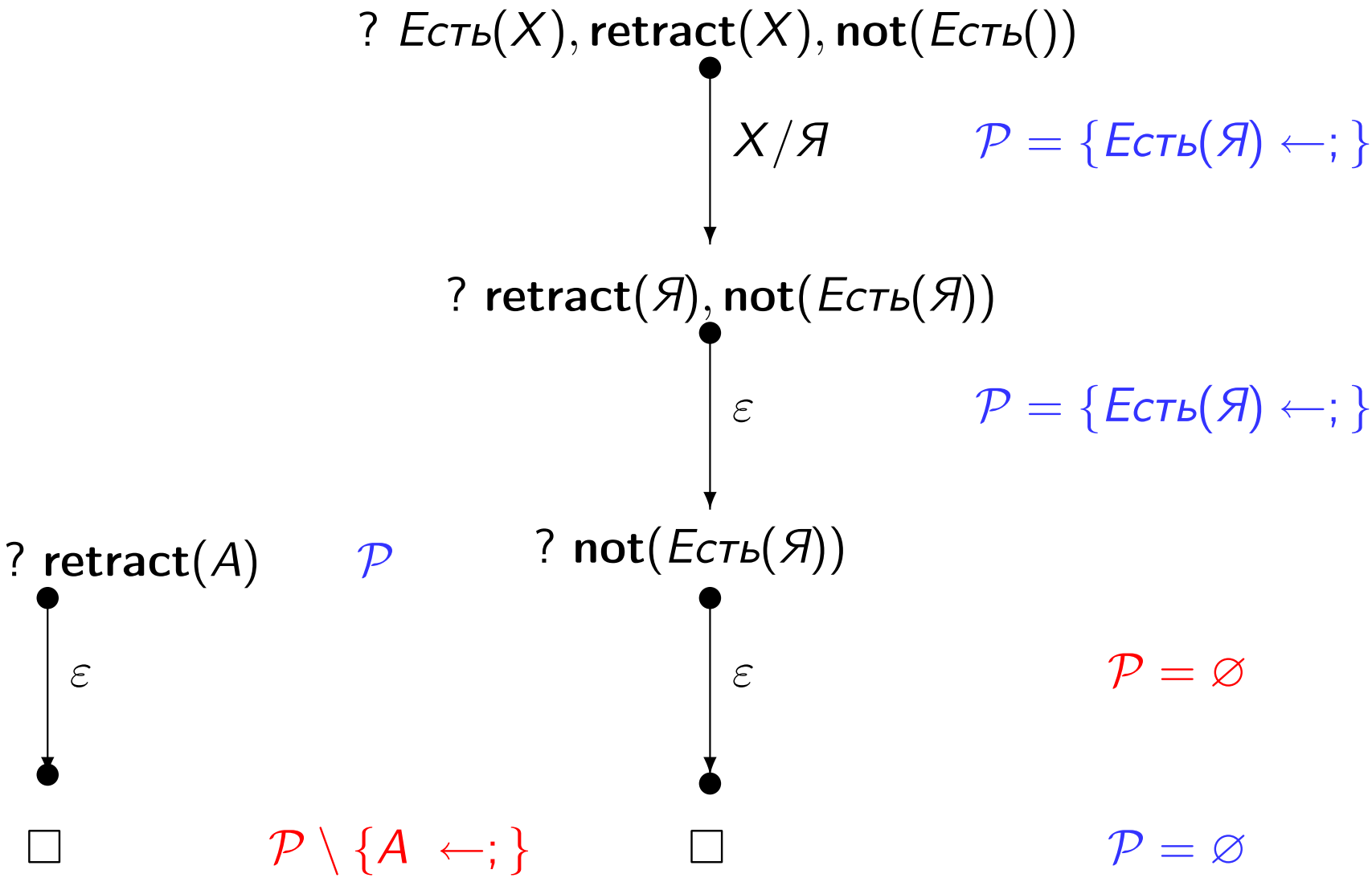
Оператор пополнения базы данных **assert**(A), где A — атом, добавляет к базе данных факт $A \leftarrow$;



Поскольку в логических программах правила и факты упорядочены, нужны два варианта оператора пополнения баз данных: **asserta**(A) и **assertz**(A).

МОДИФИКАЦИЯ БАЗ ДАННЫХ

Оператор сокращения базы данных **retract**(A), где A — атом, удаляет из базы данных факт $A \leftarrow$;



МОДИФИКАЦИЯ БАЗ ДАННЫХ

Творческая задача

А что если разрешить в теле правила использовать наряду с атомами также и программные утверждения? Как то, например,

$$A_0 \leftarrow A_1, \ldots, A_i, (B_0 \leftarrow B_1, \ldots, B_m), A_{i+1}, \ldots, A_n;$$

или $$A_0 \leftarrow A_1, \ldots, A_i, (B_0 \leftarrow), A_{i+1}, \ldots, A_n;$$

1. Как определить разумную декларативную семантику таких «составных» правил?
2. Как определить адекватную операционную семантику таких правил?
3. Можно ли определить адекватную операционную семантику «составных» правил на основе резолютивного вывода?

КОНЕЦ ЛЕКЦИИ 18.