Использование SMT-решателей для анализа систем ограничений на .NET-типы

А.В. Мисонижник
Санкт-Петербургский государственный университет
Кафедра системного программирования
Email: misonijnik@gmail.com

Д.А. Мордвинов
JetBrains Research
Санкт-Петербургский государственный университет
Кафедра системного программирования
Email: dmitry.mordvinov@se.math.spbu.ru

CEUR-WS.org/Vol-2135/SEIM_2018_paper_36.pdf

***Аннотация*—Точный анализ поведения объектно-ориентированного кода в изоляции от точек его вызова может потребовать рассуждений о взаимоотношениях открытых типов. В работах Кеннеди и Пирса подробно изучены номинальные системы типов с вариантностью, показана неразрешимость подтипирования закрытых типов и предложены ограничения на систему типов, делающие задачу разрешимой. Открытые типы, тем не менее, остаются неизученными, и все еще не существует разрешающей процедуры для задачи выполнимости системы ограничений на открытые типы. В данной работе представлен алгоритм, сводящий задачу выполнимости системы ограничений на .NET-типы к задаче выполнимости предложений логики первого порядка. Исследована его завершаемость и показано, что результирующие логические кодировки принадлежат к разрешимому фрагменту логики первого порядка.**

I. Введение

При статическом анализе функций, написанных на объектно-ориентированных языках программирования, нередко возникает необходимость проверки выполнимости некоторой системы ограничений на типы и типовые переменные. К примеру, если в некоторой точке исполнения программы известно, что типовая переменная T — подтип класса A, а тип B — брат A в иерархии наследования, анализатор может определить недостижимость из нее веток, защищенных условием $T\ is\ B$; для этого достаточно обнаружить противоречивость системы ограничений $T\ is\ A \wedge T\ is\ B$.

Для современных объектно-ориентированных языков (таких как C# и Java) задача определения выполнимости системы ограничений на типы является особенно нетривиальной. Во-первых, во время анализа больших программ могут возникать объемные системы ограничений, являющиеся достаточно хитрой комбинацией элементарных ограничений и их отрицаний, соединенных логическими связками. Во-вторых, ситуация довольно сильно усугубляется сложностью самих систем типов: как известно, в номинальных системах типов с вариантностью даже задача определения, является ли *один* закрытый тип подтипом другого, неразрешима [10], а обобщения в Java тьюринг-полны [7]. Следует также заметить, что в случае платформы .NET известно разрешимое сужение системы типов. Хороший алгоритм решения систем ограничений также должен гарантированно останавливаться и выдавать корректный ответ, если все типы в системе ограничений относятся к этому сужению.

Наконец, во время анализа библиотечной функции в изоляции от проекта, ее использующего, может быть известна только часть таблицы классов. К примеру, предположим, что класс *Base*, как и его потомок *Derived* и все классы в иерархии между *Base* и *Derived*, *не* реализуют интерфейс *IComparable*; предположим также, что в известной анализатору части таблицы классов не существует типов, наследующих класс *Base* и реализующих интерфейс *IComparable* одновременно. Рассмотрим теперь систему ограничений $T\ is\ Base \wedge T\ is\ IComparable$. Очевидно, что несмотря на отсутствие классов-кандидатов на роль T в известной таблице типов, такая система ограничений все еще выполнима — достаточно ввести новый тип, наследующий *Base* и реализующий *IComparable*. Полезной в таком случае может служить возможность автоматического конструирования объявлений типов-кандидатов на роль T: к примеру, верификатор может использовать такие объявления для автоматической генерации исполняемого файла, воспроизводящего ошибочную трассу в библиотеке. Тем не менее, при добавлении ограничения $Derived\ is\ T$, система становится противоречивой, но единственный способ доказать это — перебрать все базовые классы класса *Derived*.

Необходимость учета всех подобных деталей вместе со сложностью системы типов анализируемого языка делает задачу анализа ограничений на типы крайне громоздкой и сложной. К тому же удобно было бы иметь возможность комбинировать ограничения на типы с другими условиями (например, арифметическими). К счастью, существуют SMT-решатели — инструменты эффективного поиска моделей формул логики первого порядка, в которой некоторым символам приписана особая интерпретация [1, 4]. Современные SMT-решатели поддерживают большое количество теорий, включая линейную арифметику, алгебраические типы данных, неинтерпретируемые функции и т.д.

В данной работе представлен алгоритм решения систем ограничений на типы в платформе .NET. Ал-

горитм кодирует систему ограничений в предложение логики первого порядка и использует SMT-решатель для поиска ее модели. Показано, что для разрешимого фрагмента системы типов .NET алгоритм кодирования всегда останавливается и выдает предложение из разрешимого фрагмента логики первого порядка. Модели, получаемые от SMT-решателя, содержат логическую интерпретацию для отношения подтипирования и могут быть использованы для конструирования объявлений типов, явно не присутствующих в части таблицы классов, отправленной в решатель. Насколько известно авторам, это первая работа, обсуждающая применение SMT-решателей для анализа системы типов объектно-ориентированного языка во всей ее полноте. Несмотря на заточенность данной работы под систему типов .NET, подход может быть адаптирован для других объектно-ориентированных языков со сложной системой типов (таких как Java и Scala).

II. Обзор существующих работ

Существует большое количество работ, посвященных статическому анализу различных систем типов. Среди них есть и такие, в которых анализ осуществляется путём кодирования типов в формулы языка логики первого порядка и анализа их выполнимости при помощи SMT-решателей [2, 3, 6, 8, 11, 12, 15, 16].

Например, в работе [11] подробно описывается кодирование инвариантных обобщенных типов в логику первого порядка с использованием логического представления алгебраических типов данных. Но в ней никак не затрагивается кодирование отношения подтипирования. Напротив, в работе [16] предложен способ аксиоматизации логики первого порядка с номинальной типизацией, в которой есть аналоги типовых операторов (`is`, `as`, оператор преобразования типа). Однако её недостаточно, чтобы закодировать систему типов .NET с её структурными элементами.

Важной вехой является статья Кеннеди и Пирса, которая исследует подтипирование в номинальных системах типов с вариантностью и систему типов .NET в частности [10]. В работе показано, что отношение подтипирования в таких системах неразрешимо даже для закрытых типов и предъявлены несколько разрешимых фрагментов. Однако задача подтипирования для открытых типов является более общей, и к ней нельзя напрямую применить подход, описанный в [10].

III. Система типов .NET[1]

В своей основе система типов .NET является *номинальной* (см. ниже), однако в ней имеются и элементы *структурной* системы типов: к примеру, обобщения интерфейсов и делегатов могут явно определять *вариантность* своих типовых параметров, а массивы ковариантны по типу своего элемента.

[1]Часть обозначений и правила вывода для отношения $<:$, используемые в данной главе, взяты из работы [10].

Типы (обозначим их как T, U, V и W) можно разделить на два вида: типовые переменные X, Y и Z и сконструированные типы $C\texttt{<}\overline{T}\texttt{>}$, где C — это конструктор типа, а $\overline{T}$ — упорядоченный список аргументов типов. *Закрытыми* будем называть типы, которые в листьях дерева конструкторов не содержат типовые переменные. *Открытыми* назовём все типы, которые не являются закрытыми. *Высотой* типа будем называть высоту дерева его конструкторов. Обозначим высоту функцией $height$.

В рамках данной работы можно считать, что все типы являются подтипами `System.Object`: типы указателей здесь рассматриваться не будут из-за тривиальности их подтипирования.

Номинальность системы типов означает, что отношение подтипирования явно определяется в коде программы и может быть представлено в виде конечной *таблицы классов*. Каждая запись в таблице классов для системы типов .NET имеет следующий вид:

$$C^{*}\texttt{<}\overline{X}\texttt{>} <:: T_1, \ldots, T_n$$

Для каждого типового параметра имеется запись, которая ограничивает его:

$$C\#i^{*} <: U_1, \ldots, U_k \tag{1}$$

Без потери общности можно считать, что в таблице классов нет двух типовых переменных с одним названием (иначе переименуем их).

Левая часть каждой записи в таблице классов содержит аннотацию типа, помещаемую вместо *. Аннотация указывает, является ли тип интерфейсом (в таком случае будет записана буква I), типом значения (V), ссылочным типом (R), классом, массивом или делегатом (C), запечатанным типом (аннотация взята в квадратную рамку, например, $\boxed{C}$) и имеется ли у типа конструктор без аргументов (в таком случае около буквы записана пара скобок).

Например, для `System.Object`, `System.IComparable` и структуры `S` (не реализующей ни одного интерфейса), записи будут выглядеть следующим образом:

$$\begin{aligned}
&\texttt{System.Object}^{\,C()} && <:: \\
&\texttt{System.IComparable}^{\,I} && <:: \texttt{System.Object} \\
&\texttt{S}\,\boxed{V()} && <:: \texttt{System.ValueType}
\end{aligned}$$

Символ $<::$ обозначает отношение номинального подтипирования. Доопределим также $C\texttt{<}\overline{U}\texttt{>} <:: [\overline{U}/\overline{X}]T_i$. Через $<::^{+}$ обозначим его транзитивное замыкание. Отношение $<::^{+}$ отражает номинальность системы типов, но его не достаточно, чтобы выразить структурные элементы. Для отражения вариантности, потребуем от записей таблицы классов иметь вид

$$C^{*}\texttt{<}v_1X_1, \ldots, v_mX_m\texttt{>} <:: T_1, \ldots, T_n,$$

где v_i определяет вариантность по i-тому типовому параметру и может принимать следующие значения:

$\circ$ (инвариантность), $+$ (ковариантность), $-$ (контравариантность). Положим $C\#i \stackrel{\text{def}}{=} X_i$ и $var(C\#i) \stackrel{\text{def}}{=} v_i$.

Заметим, что в записях вида (1) содержится символ подтипирования $<:$ вместо символа номинального подтипирования. Отношение подтипирования в .NET будет строго определено ниже.

Примеры 1, 2 и 3 демонстрируют таблицы классов для трех фрагментов кода на языке C#.

Пример 1.

```
interface IBuilder<out Z> where Z: IComponent {}
interface IComponent {}
sealed class Symbol:
    IComponent, IBuilder<Symbol> {}
abstract class Line:
    IComponent, IBuilder<Line> {}
class Combo<X, Y>
    where X: IComponent, IBuilder<X>
    where Y: IComponent, IBuilder<Y> {}
```

System.Object	$<::$
IBuilder ^{I}<$+$ Z>	$<::$ System.Object
Z	$<:$ IComponent
IComponent ^{I}	$<::$ System.Object
Symbol $^{\boxed{C()}}$	$<::$ IComponent, IBuilder<Symbol>
Line ^{C}	$<::$ IComponent, IBuilder<Line>
Combo $^{C()}$<X, Y>	$<::$ System.Object
X	$<:$ IComponent, IBuilder<X>
Y	$<:$ IComponent, IBuilder<Y>

Пример 2.

```
interface IBox<in Z> {}
class Thing: IBox<IBox<Thing>> {}
```

System.Object	$<::$
IBox ^{I}<$-$ Z>	$<::$ System.Object
Z	$<:$ System.Object
Thing $^{C()}$	$<::$ IBox<IBox<Thing>>

Пример 3.

```
class Rec<Z> {}
class MutualRec<X, Y> where X: Rec<Y>
    where Y: Rec<X> {}
```

System.Object	$<::$
Rec $^{C()}$<Z>	$<::$ System.Object
Z	$<:$ System.Object
MutualRec $^{C()}$<X, Y>	$<::$ System.Object
X ^{C}	$<:$ Rec<Y>
Y ^{C}	$<:$ Rec<X>

Наложим ограничения на таблицу классов:

1) отношение $<::^{+}$ должно быть ациклическим;
2) записи в таблице должны быть корректными относительно вариантности: например, запрещено

$B\ ^{I}\!<-X>$	$<::\ \ldots$
$A\ ^{I}\!<+X>$	$<::\ B<X>$

3) запечатанному типу (в т.ч. делегату или массиву) разрешено появляться в правой части записи только в качестве типового аргумента;
4) в правой части любой записи может стоять максимум один тип с аннотацией C (множественное наследование запрещено);
5) аннотации не должны противоречить естественным ограничениям .NET (интерфейс не может быть запечатанным или иметь конструктор без аргументов, типы значений обязательно запечатаны и имеют конструктор без аргументов и т.д.).

Определение 1. *Расширением* таблицы классов CT называется таблица классов, в которой содержатся все записи из CT, а все левые части прочих записей содержат лишь конструкторы и их типовые переменные, не входящие ни в одну из левых частей CT.

Наконец, можно определить отношение подтипирования с учетом вариантности.

Определение 2. Отношение подтипирования для закрытых типов $<:$ задается следующими правилами:

$$(\text{Var})\ \frac{\text{for each } i \quad T_i <:_{var(C\#i)} U_i}{C\overline{<T>} <: C\overline{<U>}}$$

$$(\text{Super})\ \frac{C\overline{<X>} <:: V \quad [\overline{T}/\overline{X}]V <: D\overline{<U>}}{C\overline{<T>} <: D\overline{<U>}} C \neq D$$

$$\frac{T <: U}{T <:_{+} U} \qquad \frac{}{T <:_{\circ} T} \qquad \frac{U <: T}{T <:_{-} U}$$

Предложение 1. Отношение подтипирования $<:$ разрешимо для закрытых типов, при условии, что таблица классов будет *нерасширяющийся*.

За определением нерасширяемости таблицы классов и доказательством Предложения 1 читатель отсылается к работе [10]. Из этого следует разрешимость подтипирования для закрытых типов .NET, т.к. по спецификации любая расширяемая таблица классов отклоняется средой выполнения [5].

Другими словами, работа [10] доказывает существование алгоритма, принимающего на вход любую таблицу классов (прошедшую валидацию среды выполнения .NET) и утверждение вида $T <: U$, где T и U — *закрытые* типы. Данная работа решает аналогичную задачу, но для *открытых* типов.

Очевидно, вопрос $T <: U$ в случае открытых типов сам по себе не имеет смысла: уже для двух типовых переменных на вопрос $X <: Y$ можно дать как утвердительный, так и отрицательный ответ; такая постановка имеет смысл лишь при связывании типовых переменных кванторами. Естественной, поэтому, будет формулировка задачи в терминах логики первого порядка.

IV. Алгоритм

На протяжении всей главы предполагается, что имеется и зафиксирована таблица классов CT.

Рассмотрим язык первого порядка $\mathcal{L}_t$ над сигнатурой $(\mathcal{F}_t, \mathcal{P}_t)$. Здесь

- $\mathcal{F}_t$ — множество функциональных символов, отождествляемое с множеством конструкторов типов. Для удобства применение функционального символа C к аргументам $\overline{T}$ будет по-прежнему записываться как $C\texttt{<}\overline{T}\texttt{>}$.
- $\mathcal{P}_t = \{<:\}$, где $<:$ является двухместным предикатным символом и будет записываться в инфиксном стиле. Для удобства, формулы вида $\neg(T <: U)$ и $\neg(T = U)$ будут записываться как $T \not<: U$ и $T \neq U$.

Данная глава описывает алгоритм решения следующей задачи. По натуральному числу h и данной формуле $\phi_t \in \mathcal{L}_t$ без кванторов необходимо построить такую модель $\mathcal{M}_t = (\mathcal{D}_t, \sigma_t)$, что $\mathcal{M}_t \models cl_\exists(\phi_t)$, либо доказать, что такой модели не существует. Однако на $\mathcal{M}_t$ должен накладываться набор ограничений;

- носитель $\mathcal{D}_t$ должен быть подмножеством множества замкнутых типов;
- для каждого $C \in \mathcal{F}_t$, $\sigma_t(C)(\overline{T})$ должен конструировать замкнутый тип $C\texttt{<}\overline{T}\texttt{>}$;
- $\sigma_t(<:) \subseteq \mathcal{D}_t \times \mathcal{D}_t$ — отношение подтипирования из Определения 2, заданное некоторым расширением таблицы классов CT, а $cl_\exists(\phi)$ — экзистенциальное замыкание формулы ϕ_t (т.е. формула $\exists x_1, \ldots, x_n.\phi$, где $x_1, \ldots, x_n$ — свободные переменные в ϕ);
- val_t — оценка переменных $fv(\phi_t)$ и

$$\max_{T \in val_t(fv(\phi_t))} height(T) \leqslant h$$

- семантика равенства стандартна.

Пример 4. Рассмотрим системы ограничений (формулы ϕ_t) для таблиц классов из примеров 1—3:

1) $\mathrm{X} <: \texttt{IBuilder<Symbol>} \wedge \mathrm{X} <: \texttt{Line} \wedge \texttt{Line} <: \mathrm{Y}$ (выполнимо, если X — новый тип, а Y — `Line`)
2) $\texttt{Thing} <: \texttt{IBox<Thing>}$ (невыполнимо из-за зацикливания правил вывода)
3) $\mathrm{X} \not<: \mathrm{Y} \wedge \mathrm{Y} \not<: \mathrm{X}$ (выполнимо, если X и Y— новые типы)

Основным препятствием для применения решателей логики первого порядка к данной задаче является наличие ограничения (и притом нетривиального) на искомую модель. Алгоритм решает эту проблему, переводя формулу ϕ_t на другой язык первого порядка $\mathcal{L}_m$, для которого уже может быть применен решатель. Перевод осуществляется в несколько шагов: (1) замыкание множества типов, (2) расширение отношения подтипирования, (3) формирование аксиом частичного порядка, (4) запрет множественного наследования, (5) формирование ограничений на запечатанные типы, (6) формирование ограничений на ссылочность, (7) формирование ограничений на конструкторы без параметров, (8) формирование ограничений на типовые параметры и (9) формирование окончательной формулы. Шаг 2 предполагает наличие предиката-оракула $groundSubtype$, который для пары замкнутых типов определяет, является ли первый подтипом второго, используя таблицу классов CT.

Далее будет дано общее описание языка $\mathcal{L}_m$ и детальное описание каждого шага алгоритма, проанализирована разрешимость выполнимости формул языка $\mathcal{L}_m$, выдаваемых алгоритмом, а также представлен способ преобразования модели $\mathcal{M}_m$ обратно в модель $\mathcal{M}_t$. Доказательство корректности сведения, хоть и представляет интерес, опущено из-за недостатка места.

A. Язык $\mathcal{L}_m$

Язык $\mathcal{L}_m$ описывается сигнатурой $(\varnothing, \mathcal{P}_m)$, где $\mathcal{P}_m = \{<:, subclass, con, interface, hplc\}$. Здесь $<:$, $subclass$, con — двухместные предикатные символы, а $interface$ и $hplc$ — одноместные. За $\mathcal{V}$ обозначим счетное множество предметных переменных, в идентификаторах которых разрешим использовать «<» и «>».

B. Замыкание множества типов

Определение 3. Множество типов S будем называть замкнутым относительно декомпозиции, наследования и ограничений (далее просто *замкнутым*), если (a) $C\texttt{<}\overline{T}\texttt{>} \in S \Rightarrow \forall i, T_i \in S$, (b) $T \in S \wedge T <:: V_1, \ldots, V_m \Rightarrow \forall i, V_i \in S$, и (c) $X \in S \Rightarrow \forall i, U_i \in S$, где U_i — элементы правой части в записи таблицы классов $X <: U_1, \ldots, U_n$. *Замыканием* множества S (обозначается $cl(S)$) будем называть минимальное замкнутое множество, содержащее S.

Теорема 1. *Для нерасширяющийся таблицы классов замыкание конечного множества типов конечно.*

Доказательство. В доказательстве Теоремы 9 в работе [10] показан аналогичный факт для замыкания относительно декомпозиции и наследования (т.е. без учета п.(c) Определения 3). Обозначим за $cl^{a,b}(A)$ замыкание A относительно декомпозиции и наследования.

Пусть теперь дано конечное множество S. Заметим, что взятие замыкания $cl(S)$ аналогично выполнению двух шагов до достижения неподвижной точки:

1) вычисление $S' = cl^{a,b}(S)$;
2) вычисление нового $S = S' \cup U$, где U — множество всех элементов всех правых частей записей таблицы классов вида $X <: U_1, \ldots, U_n$, где $X \in S'$.

Итак, если S — конечное множество, то S' — также конечное множество, что влечет конечность нового множества S (к S' добавляется конечное множество записей U, т.к. количество типовых переменных в S' конечно). Наконец, заметим, что каждая итерация алгоритма добавляет к S лишь типовые переменные, явно содержащиеся в таблице классов. Но т.к. таблица

классов конечна, количество итераций также конечно, что и дает нам конечность $cl(S)$. □

Определим множество S', как множество всех термов, входящих в формулу языка $\mathcal{L}_t$.

Далее надо определить функцию $newVars$, зависящую от таблицы классов CT и натурального числа h. Для этого введём новые обозначения:

- CS — множество не нульарных конструкторов из таблицы классов;
- R — максимальная арность конструкторов из таблицы классов CT;
- $newVar(C, n)$ — функция, возвращающая множество типов, определенная следующим образом:

$$newVar(C, n) \stackrel{\text{def}}{=} \left\{C\texttt{<}\overline{X^i}\texttt{>}\right\}_{i=1}^{n},$$

  где все типовые переменные X_j^i различны;
- $count(h,\ R)$ — функция, возвращающая максимальное число раз, которое можно использовать один и тот же конструктор арности R из множества CS для построения типа высоты h:

$$count(h,\ R) \stackrel{\text{def}}{=} \begin{cases} 0, & h = 1 \text{ и } R \neq 0 \\ 1, & h = 1 \text{ и } R = 0 \\ \frac{R^h - 1}{R - 1}, & otherwise \end{cases}$$

Теперь можно определить функцию $newVars$ и результат её применения к выходным данным CT и h:

$$newVars(CT,\ h) \stackrel{\text{def}}{=} \bigcup_{C \in CS} newVar(C,\ count(h, R))$$
$$NV \stackrel{\text{def}}{=} newVars(CT,\ h)$$

Обозначим за S объединение множеств S' и NV. Так как множество S будет конечным, по Теореме 1 его замыкание $cl(S)$ также будет конечным. Выберем произвольное инъективное отображение $\tau : cl(S) \to \mathcal{V}$. Обозначим за FV образ $\tau(cl(S))$. Очевидно, τ задает взаимно-однозначное соответствие между $cl(S)$ и FV.

Пусть предикат $inCT(x)$ истинен, если главный конструктор аргумента принадлежит оригинальной таблице классов, и ложен иначе. Другими словами, для $v \in FV$, $inCT(x)$ истинен, если $x = v$ и $\tau^{-1}(v)$ является применением какого-либо функционального символа в $\mathcal{L}_t$, и ложен в ином случае[2]:

$$inCT(x) \stackrel{\text{def}}{=} \bigvee_{\substack{T \in cl(S),\\ T \text{ не переменная}}} x = \tau(T)$$

[2]Фактически, такое определение не гарантирует, что $inCT$ опишет *все* конструкторы из таблицы классов CT, а только те, что встретились в замыкании S. Однако, не умаляя общности, мы можем считать, что CT содержит только релевантные для ϕ_t записи

Пример 5. Для примера 4.1 и $h = 1$,

$$S = \{\text{X, Y, } \texttt{Line, IBuilder<Symbol>, Line}\}$$
$$cl(S) = \{\text{X, Y, } \texttt{Line, IBuilder<Symbol>,}$$
$$\texttt{IBuilder<Line>, Symbol, IComponent,}$$
$$\texttt{IBuilder<X>, IBuilder<Y>, System.Object}\}$$
$$inCT(x) = (x = \texttt{Line}) \vee (x = \texttt{Symbol}) \vee$$
$$(x = \texttt{IComponent}) \vee (x = \texttt{IBuilder<Symbol>}) \vee$$
$$(x = \texttt{IBuilder<Line>}) \vee (x = \texttt{IBuilder<X>}) \vee$$
$$(x = \texttt{IBuilder<Y>}) \vee (x = \texttt{System.Object})$$

Для примера 4.2 и h = 2,

$$S = \{\texttt{Thing, IBox<Thing>, IBox<Z>}\}$$
$$cl(S) = \{\texttt{Thing, IBox<Thing>, IBox<Z>,}$$
$$\texttt{IBox<IBox<Thing>>, System.Object, Z}\}$$
$$inCT(x) = (x = \texttt{Thing}) \vee (x = \texttt{IBox<IBox<Thing>>}) \vee$$
$$(x = \texttt{IBox<Thing>}) \vee (x = \texttt{IBox<Z>}) \vee$$
$$(x = \texttt{System.Object})$$

C. Расширение отношение подтипирования

После построения замыкания множества типов $cl(S)$ и сопоставления им переменных FV алгоритм распространяет отношение подтипирования на пары типов в $cl(S)$, вводя предикат

$$st \stackrel{\text{def}}{=} \bigwedge_{(A,B) \in cl(S) \times cl(S), A \neq B} st_{A,B},$$

где $st_{A,B}$ определяется одним из четырех способов.

- Если и A, и B — закрытые типы, то решение об их подтипировании принимает оракул $groundSubtype$:

$$st_{A,B} \stackrel{\text{def}}{=} \tau(A) <: \tau(B) \Leftrightarrow groundSubtype(A, B)$$

  Пример 6. Для примера 4.1,

  $\texttt{Symbol} <: \texttt{IComponent} \Leftrightarrow \top$
  $\texttt{IBuilder<Symbol>} <: \texttt{IBuilder<Line>} \Leftrightarrow \bot$

- A или B открыт, но ни один из них не является типовой переменной. В таком случае можно записать ограничения в виде эквивалентных преобразований, которые задаются правилами вывода Var и Super. Введем вспомогательное множество $supertype(C\texttt{<}\overline{T}\texttt{>},\ D\texttt{<}\overline{U}\texttt{>}) \stackrel{\text{def}}{=} \{D\texttt{<}\overline{W}\texttt{>} \mid C\texttt{<}\overline{T}\texttt{>} <::^{+} D\texttt{<}\overline{W}\texttt{>}\}$. Тогда

$$\tau(C\texttt{<}\overline{T}\texttt{>}) <: \tau(D\texttt{<}\overline{U}\texttt{>}) \Leftrightarrow \bigvee_{V \in supertype(C\texttt{<}\overline{T}\texttt{>}, D\texttt{<}\overline{U}\texttt{>})} \tau(V) <: \tau(D\texttt{<}\overline{U}\texttt{>}) \quad (2)$$

  отображает применение детерминированной версии правила Super, а

$$\tau(C\texttt{<}\overline{T}\texttt{>}) <: \tau(C\texttt{<}\overline{U}\texttt{>}) \Leftrightarrow \bigwedge_{i} \tau(T_i) <:_{var(C\#i)} \tau(U_i) \quad (3)$$

отображает применение правила Var. $st_{A,B}$ в этом случае является конъюнкцией формул (2) и (3).

Пример 7. Для примера 4.1,

$$\texttt{IBuilder<}\overline{\texttt{X}}\texttt{>} <: \texttt{IComponent} \Leftrightarrow \texttt{IComponent} <: \texttt{IComponent}$$

$$\texttt{IBuilder<X>} <: \texttt{IBuilder<Y>} \Leftrightarrow \texttt{X} <: \texttt{Y}$$

В процессе применения правил вывода может произойти зацикливание (*occurs check*). В тексте спецификации CLI [5] отсутствует явное указание на поведение среды в ситуации occurs check, однако в разделе `I.8.7.1` сказано, что отношение подтипирования есть *наименьшее* отношение, замкнутое относительно набора некоторых правил. Из этого следует, что ситуация occurs check отвергает подтипирование одного типа другим. Итак, в случае зацикливания вывода добавляется правило

$$st_{A,B} \stackrel{\text{def}}{=} C\texttt{<}\overline{T}\texttt{>} \not<: D\texttt{<}\overline{U}\texttt{>}$$

Пример 8. Для примера 4.2,

$$\texttt{Thing} <: \texttt{IBox<Thing>} \Leftrightarrow \texttt{IBox<IBox<Thing>>} <: \texttt{IBox<Thing>}$$

$$\texttt{IBox<IBox<Thing>>} <: \texttt{IBox<Thing>} \Leftrightarrow \texttt{Thing} <: \texttt{IBox<Thing>}$$

$$\texttt{Thing} \not<: \texttt{IBox<Thing>}$$

- Если $A = C\texttt{<}\overline{T}\texttt{>}$ — сконструированный тип, а $B = X$ — типовая переменная, то необходимо ввести ограничение, показывающее, что типовая переменная X должна принадлежать к изначальной таблице классов CT (в противном случае, решатель может «вклинить» новый класс в иерархию $C\texttt{<}\overline{T}\texttt{>}$). Один из способов сделать это — добавить правило следующего вида:

$$st_{A,B} \stackrel{\text{def}}{=} C\texttt{<}\overline{T}\texttt{>} <: X \Rightarrow inCT(X)$$

- Наконец, случай, когда A — типовая переменная, будет покрыт на последующих шагах. На данном шаге все такие пары игнорируются: $st_{A,B} \stackrel{\text{def}}{=} \top$.

D. Аксиомы частичного порядка

Исходя из того, что отношение подтипирования является рефлексивным и транзитивным [10], а таблица классов ацикличной, можно доказать, что отношение антисимметрично. Важно явным образом добавить аксиомы частичного порядка, так как это гарантирует, что отношение частичного порядка будет выполняться не только для закрытых типов, но и для открытых.

$$\begin{aligned} po \stackrel{\text{def}}{=} &(\forall x.\, x <: x) \wedge \\ &(\forall x, y.\, x <: y \wedge y <: x \Rightarrow x = y) \wedge \\ &(\forall x, y, z.\, x <: y \wedge y <: z \Rightarrow x <: z) \end{aligned}$$

E. Запрет множественного наследования

Множественное наследование в .NET разрешено только для интерфейсов. Если какой-то тип является подтипом не-интерфейса, то он не является интерфейсом, с одним исключением: тип `System.Object` является классом, но также является подтипом для интерфейсов. Для отражения этих свойств используются предикатные символы $interface$ и $subclass$. Предикатный символ $interface$ истинен для всех интерфейсов (т.е. типов, аннотированых в CT буквой I) и ложен для не-интерфейсов (т.е. типов, аннотированых в CT буквой C или V).

$$\begin{aligned} iface \stackrel{\text{def}}{=} &\bigwedge_{\substack{T \in cl(S), \\ T \text{ аннотирован «}I\text{»}}} interface\big(\tau(T)\big) \;\wedge \bigwedge_{\substack{T \in cl(S), \\ T \text{ аннотирован «}C\text{» или «}V\text{»}}} \neg interface\big(\tau(T)\big) \\ mi \stackrel{\text{def}}{=} &\, iface \wedge (\forall x, y, z.\, subclass(x, y) \wedge subclass(x, z) \Rightarrow \\ &\, subclass(z, y) \vee subclass(y, z)) \wedge \\ &(\forall x, y.\, \neg interface(x) \wedge \neg interface(y) \Rightarrow \\ &\, (x <: y \Leftrightarrow subclass(x, y))) \wedge \\ &(\forall x, y.\, \neg interface(y) \wedge x <: y \wedge y \neq \texttt{System.Object} \Rightarrow \\ &\, \neg interface(x)) \end{aligned}$$

Пример 9. Для примера 4.2,

$$\begin{aligned} intr = &\neg interface(\texttt{Thing}) \wedge interface(\texttt{IBox<Thing>}) \wedge \\ &interface(\texttt{IBox<IBox<Thing>>}) \wedge \\ &\neg interface(\texttt{System.Object}) \end{aligned}$$

F. Запечатанные типы

На следующем шаге вводится ограничение $seal$ для запрета наследования запечатанного типа:

$$seal \stackrel{\text{def}}{=} \bigwedge_{\substack{T \in cl(S), \\ T \text{ запечатанный}}} \forall x.\, x <: T \Rightarrow x = T \vee inCT(x)$$

Пример 10. Для примера 4.1:

$$seal = \forall x.\, x <: \texttt{Symbol} \Rightarrow x = \texttt{Symbol} \vee inCT(x)$$

G. Ссылочные типы и типы значений

Для каждого типового параметра можно указать, будет ли он типом значения или ссылочным типом. Это можно сделать, добавив аннотацию V или R соответственно. Наличие такой аннотации в таблице классов гарантирует, что в ней будет запись с конструктором `System.ValueType`. Эти ограничения будут представлены в виде формулы

$$\begin{aligned} vl(x) = &\, x <: \tau(\texttt{System.ValueType}) \wedge \\ &\, x \neq \tau(\texttt{System.ValueType}) \\ vrt \stackrel{\text{def}}{=} &\bigwedge_{\substack{T \in cl(S), \\ T \text{ аннотирован «}R\text{»}}} \big(\neg vl(\tau(T))\big) \;\wedge \bigwedge_{\substack{T \in cl(S), \\ T \text{ аннотирован «}V\text{»}}} \big(vl(\tau(T))\big) \end{aligned}$$

H. Конструкторы без параметров

В некоторых случаях существенны ограничения на переменные вида `where X : new()`. К примеру, если для такой переменной известно, что $C\langle\overline{T}\rangle <: X$, а в иерархии $C\langle\overline{T}\rangle$ лишь `System.Object` имеет конструктор без параметров, можно заключить, что X является типом `System.Object`. Ограничения на конструктор без параметров кодируются символом *hplc*.

$$plcs \stackrel{\text{def}}{=} \bigwedge_{\substack{T\in cl(S),\\ T \text{ аннотирован «()»}}} hplc\big(\tau(T)\big) \;\wedge\; \bigwedge_{\substack{T\in cl(S),\\ T \text{ не аннотирован «()»},\\ T \text{ не типовая переменная}}} \neg hplc\big(\tau(T)\big)$$

I. Ограничения на типовые параметры

Для каждого формального типового параметра X в таблице есть запись $X <: T_1 \dots T_n$, которая ограничивает его; добавим такую запись для каждой типовой переменной, которая является аргументом для конструктора типа.

$$constr \stackrel{\text{def}}{=} \bigwedge_{\substack{C\langle\overline{U}\rangle\in cl(S),\\ X\in cl(S),\, U_i=X}} \big(\tau(X) <: \tau(T_1)\big) \wedge \ldots \wedge \big(\tau(X) <: \tau(T_n)\big)$$

J. Запрет бесконечных типов

Так как в системе типов .NET у всех типов конечная высота, а текущие аксиомы не запрещают иметь бесконечную высоту, надо явно указать это свойство:

$$ct \stackrel{\text{def}}{=} \bigwedge_{C\langle\overline{U}\rangle\in cl(S)} \forall x.con(x,\tau(C\langle\overline{U}\rangle)) \Leftrightarrow \bigvee_i x = \tau(U_i) \vee con(x,\tau(U_i))$$

$$rl \stackrel{\text{def}}{=} \forall x.\neg con(x,x) \wedge \forall x,y.\neg(con(x,y) \wedge con(y,x))$$

$$ft \stackrel{\text{def}}{=} ct \wedge rl$$

K. Формирование окончательной формулы

Наконец, алгоритм создает окончательную кодировку в $\mathcal{L}_m$. Пусть ϕ_m — формула, полученная из ϕ_t заменой каждого атома $T \leqq U$ на атом $\tau(T) \leqq \tau(U)$. Тогда результатом является

$$\psi_m \stackrel{\text{def}}{=} po \wedge mi \wedge st \wedge seal \wedge plcs \wedge vrt \wedge constr \wedge ft \wedge \phi_m$$

L. Разрешимость

Полученный список формул принадлежит разрешимому фрагменту логики первого порядка, который называется *effectively propositional logic* (EPR) [14]. Алгоритм эффективного решения EPR с равенством, использующий подстановочные множества, реализован, к примеру, в SMT-решателе Z3 [13].

Важно, что если у EPR-формулы есть модель, то у нее есть модель с конечным носителем. Другое важное замечание состоит в том, что алгоритмы решения EPR первым шагом обычно сколемизируют переменные под кванторами существования в функциональные константы. Таким образом, в случае выполнимости формулы ϕ_t, от решателя можно получить оценку переменных под кванторами существования.

Интуитивно, разрешимость системы типовых ограничений возможна из-за ограничений на таблицу классов (она должна быть нерасширяемой) и из-за требования на отсутствие кванторов в ϕ_t.

M. Преобразование модели

Пусть $cl_\exists(\psi_m)$ имеет модель $\mathcal{M}_m = (\mathcal{D}_m, \sigma_m)$ с конечным носителем $(\mathcal{D}_m)$. Более того, как сказано в конце предыдущей секции, вместе с моделью можно получить оценку $val_m : FV \to \mathcal{D}_m$ переменных под кванторами существования. Данная секция обсуждает, как перейти от $\mathcal{M}_m$ к модели $\mathcal{M}_t = (\mathcal{D}_t, \sigma_t)$ формулы $cl_\exists(\phi_t)$ в $\mathcal{L}_t$.

Лемма 1. *Если для некоторого $a \in \mathcal{D}_m$, $\tau(C\langle\overline{T}\rangle) \in val_m^{-1}(\{a\})$ и $\tau(D\langle\overline{U}\rangle) \in val_m^{-1}(\{a\})$, то конструкторы C и D совпадают и $\forall i.val_m(\tau(T_i)) = val_m(\tau(U_i))$.*

Утверждение Леммы 1 можно вывести из антисимметричности подтипирования путем применения правил (2) и (3) к условиям леммы.

Лемма 2. *Если существуют сконструированный тип $C\langle\overline{T}\rangle$ и тип V такие, что $\exists i.val_m(\tau(T_i)) = val_m(\tau(V))$, тогда $val_m(\tau(C\langle\overline{T}\rangle)) \neq val_m(\tau(V))$*

Утверждение Леммы 2 можно вывести из набора аксиом 4.

Лемма 1 и Лемма 2 дают способ конструирования отображения $d : \mathcal{D}_m \to \mathcal{D}_t$, сопоставляющее элементы носителя $\mathcal{M}_m$ замкнутым типам.

Возьмем отображение $fresh : \mathcal{D}_m \to \mathcal{D}_t$, сопоставляющее элементу из $\mathcal{D}_m$ произвольный нульарный (и потому замкнутый) конструктор типа, отсутствующий в таблице классов CT. Обозначим за $fv(\phi_t)$ множество свободных типовых переменных ϕ_t. Теперь можно индуктивно определить отображение d.

$$d(a) \stackrel{\text{def}}{=} \begin{cases} fresh(a), & \text{если } \tau^{-1}\big(val_m^{-1}(\{a\})\big) \subseteq fv(\phi_t) \\ C\langle\overline{d(T)}\rangle, & \text{если } \tau(C\langle\overline{T}\rangle) \in val_m^{-1}(\{a\}) \end{cases}$$

По Лемме 1 отображение d определено корректно, а по лемме 2 является определенным на всём $\mathcal{D}_m$.

Отображение $d \circ val_m \circ \tau$ является оценкой на $fv(\phi_t)$. Модель же $\mathcal{M}_t$ целиком определяется расширением таблицы классов CT'. Для построения CT' сначала зафиксируем множество новых типов, введенных отображением $fresh$ из носителя $\mathcal{D}_m$:

$$N \stackrel{\text{def}}{=} \big\{fresh(a) \mid \tau^{-1}\big(val_m^{-1}(\{a\})\big) \subseteq fv(\phi_t)\big\}$$

Теперь построим «каркасы» отношения подтипирования для элементов N, транзитивное замыкание которых даст всю их иерархию. Для этого для каждого $n \in N$ применим алгоритм алгоритм выделения транзитивного остова [9]. Обозначим за B_n множество вершин, соединенных в графе-каркасе с n ребром $(n, \cdot)$. Таблица классов CT' получается из CT добавлением записи $fresh(n)^* <:: d(b_n^1), \dots, d(b_n^k)$ для каждого $n \in N$. Здесь $b_n^1, \dots, b_n^k$ — элементы множества B_n. Аннотации для

$fresh(n)$ определяются очевидным образом из интерпретаций одноместных предикатных символов языка $\mathcal{L}_m$.

Теорема 2. *$\exists\ h \in \mathbb{N}$ и семейство формул в пренексной нормальной форме без кванторов существования $\{\psi_m^i\}_{i=0}^{\infty} \subset \mathcal{L}_m$ таких, что выполняются следующие утверждения*

- *Если $cl_\exists(\psi_m^h)$ имеет модель, тогда $cl_\exists(\phi_t)$ также имеет модель*
- *Если $cl_\exists(\phi_t)$ имеет модель, val_t — оценка переменных $fv(\phi_t)$ и $\max\limits_{t \in val_m(fv(\phi_t))} height(t) \leqslant h$, тогда $cl_\exists(\psi_m^h)$ имеет модель*

V. Эксперименты

Эксперименты проводились на SMT-решателе Z3. Всего было проведено 30 экспериментов, в таблицах классов было 10-20 записей. Во входных формулах было от 2 до 10 типов, количество типов в замыкании не превышало 50. Время работы решателя на всех запусках не превышало 0.1 секунды, все вердикты и модели были верными. Пример кодировки в формате SMT-LIB 2 представлен в Приложении А.

VI. Заключение

В работе было показано, как свести задачу определения выполнимости набора ограничений на открытые типы платформы .NET к разрешимому фрагменту логики первого порядка. Подход был реализован и показал свою работоспособность на практике.

Работа выполнена при финансовой поддержке компании JetBrains.

Список литературы

[1] Clark W Barrett и др. «Satisfiability modulo theories.» В: *Handbook of satisfiability* 185 (2009), с. 825—885.

[2] Gavin M Bierman и др. «Semantic subtyping with an SMT solver». В: *ACM Sigplan Notices*. Т. 45. 9. ACM. 2010, с. 105—116.

[3] Daniel de Carvalho и др. «Jolie Static Type Checker: a prototype». В: *Моделирование и анализ информационных систем* 24.6 (2017), с. 704—717.

[4] Leonardo De Moura и Nikolaj Bjørner. «Z3: An efficient SMT solver». В: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, с. 337—340.

[5] TC39 Ecma. *TG3. Common Language Infrastructure (CLI). Standard ECMA-335, June 2005*.

[6] Nils Gesbert, Pierre Genevès и Nabil Layaïda. «A logical approach to deciding semantic subtyping». В: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 38.1 (2015), с. 3.

[7] Radu Grigore. «Java generics are turing complete». В: *ACM SIGPLAN Notices*. Т. 52. 1. ACM. 2017, с. 73—85.

[8] Mostafa Hassan, Caterina Urban и Peter Muller. *SMT-based Static Type Inference for Python 3*. 2017.

[9] J van Leeuwen JA La Poutré. «Maintenance of transitive closures and transitive reductions of graphs». В: *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer. 1987, с. 106—120.

[10] Andrew J. Kennedy и Benjamin C. Pierce. *On Decidability of Nominal Subtyping with Variance*. 2006.

[11] K. Rustan M. Leino и Philipp Rümmer. *The Boogie 2 Type System: Design and Verification Condition Generation*.

[12] Zvonimir Pavlinovic, Tim King и Thomas Wies. «Practical SMT-based type error localization». В: *ACM SIGPLAN Notices* 50.9 (2015), с. 412—423.

[13] Ruzica Piskac, Leonardo de Moura и Nikolaj Bjørner. «Deciding effectively propositional logic using DPLL and substitution sets». В: *Journal of Automated Reasoning* 44.4 (2010), с. 401—424.

[14] FP Ramsey. «On a Problem of Formal Logic». В: *Proceedings of the London Mathematical Society* 2.1 (1930), с. 264—286.

[15] Georg Stefan Schmid и Viktor Kuncak. «SMT-based checking of predicate-qualified types for Scala». В: *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*. ACM. 2016, с. 31—40.

[16] Peter H Schmitt и Mattias Ulbrich. «Axiomatization of typed first-order logic». В: *International Symposium on Formal Methods*. Springer. 2015, с. 470—486.

SMT-Based Analysis of Constraints on .NET Types

Aleksandr V. Misonizhnik, Dmitry A. Mordvinov

A precise analysis of object-oriented code in isolation from its call points may require reasoning about the relationships of open types. A.Kennedy and B.Pierce extensively studied nominal type systems with variance, showed undecidability of subtyping relation between *ground* types and proposed the decidable fragments of the type system. *Open* types, however, remain unexplored, and still there is no decision procedure for the problem of satisfiability of constraints on the open types. In this paper we present an algorithm that reduces the problem of satisfiability of the constraints on the system of .NET-types to a satisfiability problem in first-order logic. Termination is investigated and the resulting logical encodings are shown to belong to a decidable fragment of the first-order logic.

Приложение А: Кодировка примера 4.1 в синтаксисе SMT-LIB2

```
(declare-sort Type)
(declare-fun subtype (Type Type) Bool)
(declare-fun isinterface (Type) Bool)
(declare-fun hplc (Type) Bool)
(declare-fun sealed (Type) Bool)
(declare-fun subclass (Type Type) Bool)

(declare-const Object Type)
(declare-const X Type)
(declare-const Y Type)
(declare-const Line Type)
(declare-const Symbol Type)
(declare-const IComponent Type)
(declare-const IBuilder<X> Type)
(declare-const IBuilder<Y> Type)
(declare-const IBuilder<Line> Type)
(declare-const IBuilder<Symbol> Type)

(assert (forall ((x Type)) (subtype x x)))
(assert (forall ((x Type) (y Type)) (=> (and (subtype x y) (subtype y x)) (= x y))))
(assert (forall ((x Type) (y Type) (z Type)) (=> (and (subtype x y) (subtype y z)) (subtype x z))))
(assert (forall ((x Type) (y Type) (z Type))
    (=> (and (subclass x y) (subclass x z)) (or (subclass z y) (subclass y z)))))
(assert (forall ((x Type) (y Type))
    (=> (and (not (isinterface y)) (subtype x y) (not (= y Object))) (not (isinterface x)))))
(assert (forall ((x Type) (y Type))
    (=> (and (not (isinterface x)) (not (isinterface y))) (= (subclass x y) (subtype x y)))))

(assert (and (isinterface IBuilder<X>) (isinterface IBuilder<Y>) (isinterface IBuilder<Line>)
    (isinterface IBuilder<Symbol>) (isinterface IComponent) (not (isinterface Object))
    (not (isinterface Line)) (not (isinterface Symbol))
    (not (hplc IBuilder<X>)) (not (hplc IBuilder<Y>)) (not (hplc IBuilder<Line>))
    (not (hplc IBuilder<Symbol>)) (not (hplc IComponent)) (hplc Object) (hplc Line) (hplc Symbol)
    (not (sealed IBuilder<X>)) (not (sealed IBuilder<Y>)) (not (sealed IBuilder<Line>))
    (not (sealed IBuilder<Symbol>)) (not (sealed IComponent)) (not (sealed Object))
    (not (sealed Line)) (sealed Symbol)))

(assert (and
    (subtype Line Object) (subtype Symbol Object) (subtype IComponent Object) (subtype IBuilder<X> Object)
    (subtype IBuilder<Y> Object) (subtype IBuilder<Line> Object) (subtype IBuilder<Symbol> Object)
    (not (subtype Object Line)) (not (subtype Symbol Line)) (not (subtype IComponent Line))
    (not (subtype IBuilder<X> Line)) (not (subtype IBuilder<Y> Line))
    (not (subtype IBuilder<Line> Line)) (not (subtype IBuilder<Symbol> Line))
    (not (subtype Object Symbol)) (not (subtype Line Symbol)) (not (subtype IComponent Symbol))
    (not (subtype IBuilder<X> Symbol)) (not (subtype IBuilder<Y> Symbol))
    (not (subtype IBuilder<Line> Symbol)) (not (subtype IBuilder<Symbol> Symbol))
    (not (subtype Object IComponent)) (subtype Line IComponent) (subtype Symbol IComponent)
    (not (subtype IBuilder<X> IComponent)) (not (subtype IBuilder<Y> IComponent))
    (not (subtype IBuilder<Line> IComponent)) (not (subtype IBuilder<Symbol> IComponent))
    (not (subtype Object IBuilder<X>)) (not (subtype IComponent IBuilder<X>))
    (= (subtype Line IBuilder<X>) (subtype IBuilder<Line> IBuilder<X>))
    (= (subtype Symbol IBuilder<X>) (subtype IBuilder<Symbol> IBuilder<X>))
    (= (subtype IBuilder<Y> IBuilder<X>) (subtype Y X)) (= (subtype IBuilder<X> IBuilder<Y>) (subtype X Y))
    (= (subtype IBuilder<Line> IBuilder<X>) (subtype Line X))
    (= (subtype IBuilder<Symbol> IBuilder<X>) (subtype Symbol X))
    (not (subtype Object IBuilder<Y>)) (not (subtype IComponent IBuilder<Y>))
    (= (subtype Line IBuilder<Y>) (subtype IBuilder<Line> IBuilder<Y>))
    (= (subtype Symbol IBuilder<Y>) (subtype IBuilder<Symbol> IBuilder<Y>))
    (= (subtype IBuilder<Line> IBuilder<Y>) (subtype Line Y))
    (= (subtype IBuilder<Symbol> IBuilder<Y>) (subtype Symbol Y))
    (not (subtype Object IBuilder<Line>)) (subtype Line IBuilder<Line>)
    (not (subtype Symbol IBuilder<Line>)) (not (subtype IComponent IBuilder<Line>))
    (= (subtype IBuilder<X> IBuilder<Line>) (subtype X Line))
    (= (subtype IBuilder<Y> IBuilder<Line>) (subtype Y Line))
    (not (subtype IBuilder<Symbol> IBuilder<Line>)) (not (subtype Object IBuilder<Symbol>))
    (not (subtype Line IBuilder<Symbol>)) (subtype Symbol IBuilder<Symbol>)
    (not (subtype IComponent IBuilder<Symbol>)) (not (subtype IBuilder<Line> IBuilder<Symbol>))
    (= (subtype IBuilder<X> IBuilder<Symbol>) (subtype X Symbol))
    (= (subtype IBuilder<Y> IBuilder<Symbol>) (subtype Y Symbol))
    (subtype X IBuilder<X>) (subtype Y IBuilder<Y>) (subtype Y IComponent) (subtype X IComponent)))

(define-fun isconstructed ((x Type)) Bool
    (or (= x Line) (= x Symbol) (= x IComponent) (= x IBuilder<X>)
    (= x IBuilder<Y>) (= x IBuilder<Line>) (= x IBuilder<Symbol>) (= x Object)))
(assert (forall ((x Type) (y Type)) (=> (and (isconstructed x) (subtype x y)) (isconstructed y))))
(assert (forall ((x Type) (y Type)) (=> (and (sealed x) (subtype y x)) (or (= x y) (isconstructed y)))))

(assert (and (subtype X IBuilder<Symbol>) (subtype X Line) (subtype Line Y)))

(check-sat)
(get-model)
```