вычисляющий dle_π на всём множестве входов. Алгоритм $\mathcal{B}$ на входе (A, B, E, p) работает следующим образом:

1) Сгенерировать случайно и равномерно $y \in \{0, \ldots, p-1\}$ и вычислить $A' = A + yB$.
2) Запустить алгоритм $\mathcal{A}$ на (A', B, E, p).
3) Если $\mathcal{A}(A', B, E, p) = z \in \mathbb{N}$, то $A' = zB = A + yB$, откуда $x = z - y$ — дискретный логарифм для исходной задачи (A, B, E, p).
4) Если $\mathcal{A}(A', B, E, p) =?$, то выдать 0.

Заметим, что алгоритм $\mathcal{B}$ может выдать неправильный ответ только на шаге 4. Докажем, что вероятность этого меньше $1/2$. Действительно, $A' = A + yB$ при $y = 0, \ldots, p-1$ пробегает все элементы $\langle B \rangle$, поэтому множество $\{(A', B, E, p) : y \in \{0, \ldots, p-1\}\}$ совпадает с множеством всех входов размера n. Но алгоритм $\mathcal{A}$ генерический, поэтому доля тех входов (A', B, E, p), на которых он выдаёт неопределённый ответ, стремится к 0 с ростом n и с некоторого момента становится меньше $1/2$. ■

Непосредственным следствием теоремы 1 является следующая

Теорема 2. Если для вычисления функции dle не существует полиномиального вероятностного алгоритма, то существует экспоненциальная последовательность π, такая, что для вычисления функции dle_π не существует генерического полиномиального алгоритма.

ЛИТЕРАТУРА

1. *Kapovich I., Miasnikov A., Schupp P., and Shpilrain V.* Generic-case complexity, decision problems in group theory and random walks // J. Algebra. 2003. V. 264. No. 2. P. 665–694.
2. *Impagliazzo R. and Wigderson A.* P=BPP unless E has subexponential circuits: Derandomizing the XOR lemma // Proc. 29th STOC. El Paso: ACM, 1997. P. 220–229.
3. *Романьков В. А.* Введение в криптографию. 2-е изд., испр. М.: ФОРУМ, 2012. 240 с.

УДК 004.431.4 DOI 10.17223/2226308X/11/42

СОЗДАНИЕ СИСТЕМЫ ТИПОВ ДЛЯ СЕМЕЙСТВА ЯЗЫКОВ АССЕМБЛЕРА

Н. В. Сороковиков

Строится система типов для семейства языков ассемблера, в том числе формально определяются команды, программы и термы языка. Показывается разрешимость задач населённости и проверки типа для ассемблеров с командами `mov` и `jz`.

Ключевые слова: *система типов, ассемблер, статический анализ, бинарные приложения.*

При анализе бинарных приложений исследователи полагаются на различные средства автоматизации. Одной из таких автоматизаций является извлечение типа данных участков памяти процесса. Под типом данных можно понимать стратегию использования участка памяти или, эквивалентно, взаимное расположение разных видов информации относительно друг друга в памяти. Уже существуют и используются средства для определения типов переменных [1], поэтому имеет смысл вопрос, каковы границы применимости статических способов нахождения типов в бинарной программе. Для ответа на этот вопрос можно воспользоваться аппаратом теории типов, а для этого необходимо построить систему типов для языков ассемблера.

При построении системы типов строится модель, в которой существуют несколько родов объектов: типы, термы, суждения о термах и типах, правила появления суждений, и все эти объекты появляются из необходимостей предметной области [2]. Определим сначала термы — объекты, о типах которых идёт речь, то есть переменные в программах. Программы на языках ассемблера состоят из команд, у которых есть ноль, один или несколько операндов, явных или неявных. Операндами выступают регистры и участки памяти, различные на разных этапах выполнения программы, поэтому в терме также необходимо указать, на какой стадии выполнения находится программа в момент типизации.

Определение 1. Назовём символы регистров (например, `eax`, `r15`, `ss`), числовые константы, операции разыменования адресов в формате $[\mathrm{const} + reg_1 + reg_2 \cdot \mathrm{const}]_{\mathrm{size}}$ *символьными операндами*. Здесь const — числовая константа; reg_1 и reg_2 — символы регистров; size — размер в битах.

Назовём символы команд (например, `mov`, `call`) с соответствующими им арностями, включающими также неявные операнды, *символьными операциями*; строку из символьной операции и соответствующего ей по количеству набора символьных операндов — *символьной командой*; упорядоченный набор символьных команд — *символьной программой*.

Символьную программу вместе с натуральным числом, называемым *этапом выполнения*, назовём *контекстом терма*. Контекст терма вместе с символьным операндом, присутствующим в этом контексте, назовём *простым термом*.

Пусть существует интерпретация символьных операндов в множество A и символьных операций в отображения на A. Поставим некоторым бинарным и унарным отображениям в соответствие некоторые символы (например, сложению — символ «+», побитовому отрицанию — символ «$\sim$» и т. п.). Предтермы определим индуктивно: простые термы являются предтермами; предтермы, соединённые символами операций, а также разыменования предтермов в формате $[\mathrm{const} + t_1 + t_2 \cdot \mathrm{const}]_{\mathrm{size}}$ — предтермы. Здесь t_1 и t_2 — предтермы; const и size, как и в предыдущий раз, числовая константа и размер в битах.

Если в предтерме все простые термы имеют одинаковую символьную программу, то его можно назвать *термом*.

Далее зададим типы системы. Типы должны показывать расположение и роль объектов в памяти. Для их задания понадобятся следующие правила: если Size — размер участка памяти в битах, то Int Size, Float Size, UPtr, FPtr — типы. Если A, A_1, …, A_n — типы, то Ptr A, Struct$(A_1, \ldots, A_n)$ — тоже типы. Поскольку типы дают информацию об использовании памяти и эта информация должна иметь смысл для исследователя, то смысл, вкладываемый в UPtr, — указатель на ещё не определённую область; FPtr — указатель на функцию. Int, Float, Ptr и Struct несут то же значение, что и в языке Си. Для удобства вводится структурная эквивалентность [2] типов Struct в формате ассоциативности: $\mathrm{Struct}(\mathrm{Struct}(A_1, \ldots, A_n), B_1, \ldots, B_m) = \mathrm{Struct}(A_1, \ldots, A_n, \mathrm{Struct}(B_1, \ldots, B_m)) = \mathrm{Struct}(A_1, \ldots, A_n, B_1, \ldots, B_m)$.

В языках ассемблера команды могут изменять значения некоторых своих аргументов. Для каждой символьной операции можно определить, какие её аргументы являются изменяемыми, и с помощью этого ввести несколько правил вычисления, связывающих между собой различные термы и их типы:

1) если простой терм t находится на этапе выполнения N, его символьный операнд не является изменяемым параметром командного символа на строке N

его символьной программы и этап выполнения M является следующим для команды на строке N, то простой терм s, отличающийся от t только заменой этапа выполнения на M, эквивалентен t;

2) если терм t эквивалентен s, а s эквивалентен r, то t эквивалентен r;
3) если терм t эквивалентен s, то s эквивалентен t;
4) если терм t имеет тип A и эквивалентен s, то s имеет тип A.

Можно заметить, что правила 2, 3 и 4 повторяют свойства отношений эквивалентности. Это формальная необходимость систем типов, поскольку введённая эквивалентность является не отношением, а формой записи, которая может присутствовать в одной из части правила [3].

Помимо правил вычисления, зададим два правила для введения типов указателей:

1) если простой терм t находится на этапе выполнения N и на строке N его выражение *expr* присутствует как операнд вида $[expr]_{\text{size}}$, то t имеет тип UPtr;
2) если терм t имеет тип UPtr, а терм $[t]_{\text{size}}$ — тип A, то t имеет тип Ptr A;

и правило введения структуры:

1) если терм t имеет тип UPtr, $[t]_S$ имеет тип A и терм $[t + S]_{\text{size}}$ — тип B, то t имеет тип Ptr Struct(A, B).

Все остальные правила введения типов определяются интерпретацией командных символов, то есть интерпретация дополняет систему типов правилами, говорящими о том, какие типы можно присвоить операндам команд. В этой связи интересно рассмотреть два символа операций: `mov`, правило которого говорит, что левый операнд эквивалентен правому, и `jz`, для которого существует два следующих этапа выполнения (следующая строка и адрес перехода, записанный как аргумент команды). С ними можно доказать две теоремы теории типов: населённости и возможности проверки.

Теорема 1 (о населённости типов). В описанной системе, какой бы ни была интерпретация, для каждого типа существует терм, который можно им типизировать.

Теорема 2 (о проверке типов). В описанной системе, какой бы ни была интерпретация, существует алгоритм, проверяющий, можно ли данному терму приписать данный тип.

ЛИТЕРАТУРА

1. *Caballero J. and Zhiqiang L.* Type inference on executables // ACM Computing Surveys. 2016. V. 48. No. 4. C. 65:1–65:35.
2. *Cardelli L.* Type Systems. N.Y.: CRC Press, 2004.
3. *Pierce B. C.* Types and Programming Languages. Cambridge: MIT Press, 2002.