Министерство образования и науки Российской Федерации
Московский физико-технический институт (государственный университет)

Физтех-школа прикладной математики и информатики
Кафедра дискретной математики

Выпускная квалификационная работа бакалавра по направлению 010900
«Прикладные математика и информатика»

Формальная верификация программ и их асимптотик

Студент 7910б группы
Григорянц С. А.

Научный руководитель
Дашков Е. В.

Долгопрудный
2021

Содержание

Глава 1

Введение

Сегодня, все больше отраслей компьютерных технологий нуждаются в формальной верификации. В первую очередь, в этот список входят программы, связанные с транспортом, коммуникацией, медициной, компьютерной безопасностью, криптографией и банковским делом. В этих критических сферах, последствия отсутствия формальной верификации в некоторых проектах привели к очень плачевным последствиям [6]. Фундаментальным отличием формальной верификации от классического тестирования ПО заключается в том, что, в то время как классическое тестирование проверяет систему лишь на каком-то подмножестве возможных входов, формальная верификация ПО позволяет утверждать о корректности системы на всех возможных входах. Такого вида гарантии конечно же дают несравнимо большую уверенность в корректности работы данного ПО, даже учитвая то, что остальные части системы, такие как компилятор, аппаратное обеспечение, и, в конце концов, само ПО, используемое для верификации, могут дать сбои. Ибо мы таким образом убеждаемся в том, что сам код, который является частью ПО, ошибок не содержит. Можно привести в пример много проектов, использующих формальную верификацию для довольно сложных систем. Например, в верификации компиляторов – проект Jinja [10, 11], проект Verisoft [12, 17], а также проект CompCert [1, 13]. В блокчейне – проект Scilla [16]. Таким образом, мы можем убедиться в том, что формальная верификация ПО и вправду очень востребованна.

Но наряду с верификацией программ, также встает вопрос верификации асимптотики применяемого алгоритма. Действительно, довольно часто мы хотим убедиться не только в том, что алгоритм корректен, но также и в том, что его асимптотика соответствует нашим ожиданиям. На самом деле, баги, связанные с асимтотикой алгоритма, могут возникать довольно часто только для конкретных входов, что делает классический подход тестирования неприемлемым для их отыскания. например, рассмотрим следующую реализацию бинарного поиска (на языке Python).

Рис. 1. Проблемный бинарный поиск

```
# Requires t to be a sorted array of integers.
# Returns k such that i <= k < j and t[k] = v
# or -1 if there is no such k.
def bsearch(t, v, i, j):
    if j <= i:
        return -1
    k = i + (j-i) // 2
    if v == t[k]:
        return k
    elif v < t[k]:
        return bsearch(t, v, i, k)
    return bsearch(t, v, i+1, j)
```

Проблема этого кода состоит в том, что при попадании в правую часть списка, вместо того, чтобы рассматривать интервал $[k+1; j)$, мы рассматриваем интервал $[i+1; j)$. Конечно, сам алгоритм корректно реализует бинарный поиск, но асимптотика при вводе, скажем, последнего элемента массива, превращается из логарифмической в линейную. На этом примере хорошо видно, что даже формальной верификации алгоритма и классического тестирования его на проверку асимптотики работы не всегда гарантирует нам корректность этой самой асимптотики. В связи с этим, возникает потребность в том, чтобы иметь также возможность формально верифицировать не только корректность алгоритма, но и его асимптотику.

В данной работе мы покажем, как формальная верификация алгоритмов и их асимптотик может быть реализована с помощью Сепарационной Логики с временными кредитами [8]. С помощью этой теории, мы формально верифицируем алгоритм нахождения наибольшей общей подпоследовательности.

Глава 2

Теоретические сведения

В этой главе мы дадим все формальные определения и формулировки, чтобы объяснить, на чем основана теория, разработанная в [8], а также интерактивное программное средство доказательства теорем Coq [19], на котором эта теория построена. Мы начнем с лямбда-исчисления [15].

2.1 Лямбда-исчисление

Лямбда исчисления возникло в работе Алонзо Черча по основаниям математики [3]. В ней он пытался предложить формальную систему, в которой можно было бы избежать стандартных парадоксов наивной теории множеств [14]. Позднее было обнаружено, что предложенная система также является противоречивой [9]. После этого, Черч опубликовал часть этой системы, отвечающую за вычислимость [4], а позднее и исправленную полноценную формальную систему, известную как "просто типизированное лямбда-исчисление"[2]. В этой главе мы рассмотрим само лямбда-исчисление, то есть систему, рассмотренную в [4].

2.1.1 Формальное определение

Лямбда-исчисление, есть не что иное, как простая семантика вычислимости. Оно позволяет довольно просто оперировать вычислимыми функциями. Для описания лямбда исчисления будет использован следующий язык:

- переменные обозначаются буквами латинского алфавита – $x, y, z, \ldots$. Множество всех переменных обозначим за $\mathcal{V}$.
- символ абстракции λ и точка ".".
- Скобки "(" и ")".

Тогда, мы можем индуктивно построить множество лямбда-выражений Λ по слндующим правилам:

- $x \in \mathcal{V} \implies x \in \Lambda$.
- $x \in \mathcal{V} \wedge M \in \Lambda \implies (\lambda x.M) \in \Lambda$.
- $M \in \Lambda \wedge N \in \Lambda \implies (MN) \in \Lambda$.

Второе правило обычно называется лямбда-абстракцией, а третье – аппликацией. Лямбда-выражения также обычно называют лямбда-термами или просто термами.
Далее, зададим индуктивно множество свободных переменных $FV(N) \subset \mathcal{V}$ данного лямбда-выражения $N \in \Lambda$ следующим образом:

- $x \in \mathcal{V} \implies FV(x) = \{x\}$.
- $x \in \mathcal{V} \wedge M \in \Lambda \implies FV(\lambda x.M) = FV(M) \setminus \{x\}$.
- $M \in \Lambda \wedge N \in \Lambda \implies FV(MN) = FV(M) \cup FV(N)$.

Переменные, не являющиеся свободными, называются связанными.
Теперь мы готовы определить замену переменной $x \in \mathcal{V}$ на терм $N \in \Lambda$ в терме $M \in \Lambda$, получая при этом новый терм $M[x := N] \in \Lambda$ следующим образом:

- $x[x := N] = N$.
- $y \in \mathcal{V} \wedge x \neq y \implies y[x := N] = y$.
- $(M_1 M_2)[x := N] = (M_1[x := N] M_2[x := N])$.
- $(\lambda x.M)[x := N] = \lambda x.M$.
- $y \neq x \wedge y \notin FV(N) \implies (\lambda y.M)[x := N] = \lambda y.(M[x := N])$.

Стоит отметить, что замена связанных переменных не производится (правило 4). Действительно, ведь лямбда-абстракция работает только с переменными, а не с произвольными термами. Также стоит отметить то, что замена не может производиться, если в текущем терме присутствует переменная, которая также имеет свободное вхождение в предлагаемую замену (условие $y \notin FV(N)$ в правиле 5). Действительно, иначе $(\lambda x.y)[x := y] = \lambda y.y$, но совершенно ясно, что после простого переименования переменной x в y(это называется α-конверсия) смысл терма полностью поменялся.
После этих определений мы готовы ввести основные операции над лямбда-термами. Выделяют три операции:

- α-конверсия – это просто переименование переменной. То есть, применяя α-конверсию к терму $M \in \Lambda$, заменяя x на y, мы получаем $M[x := y]$.
- β-редукция – это преобразование, которое реализует применение функции в обычном понимании. Формально β-редукция применима к терму вида $(\lambda x.M)N$, и выдает терм $M[x := N]$.

- η-редукция – это преобразование, которое реализует идею так называемой functional extensionality. Смысл этого утверждения в том, что функцию полностью определяют ее значения на всех входах. Формально, η-редукция конвертирует $\lambda x.fx$ в f, если $x \notin FV(f)$.

Условие $x \notin FV(f)$ в η-редукции существенно, иначе мы могли бы свести $\lambda x.xx$ к x.

2.1.2 Нормальная форма

После того, как мы определили лямбда-термы и преобразования над ними, сразу встает вопрос – существует ли некая каноническая форма для каждого терма, к которой можно свести этот терм данными преобразованиями? Обычно, каноническим термом считается такой терм, к которому больше нельзя применить никаких преобразований. Такие термы называются нормальными. Ясно, что если мы будем рассматривать α-конверсию, то нормальных термов не существует, так что мы будем рассматривать только $\beta\eta$-конверсии. Формальные системы обладают свойством сильной нормализации, если применение любых доступных преобразований к любому терму за конечное число шагов выдает нормальный терм. Если же для каждого терма просто существует последовательность преобразований, после которой терм становится нормальным, то такая система обладает свойством слабой нормализации. На самом деле, лямбда-исчисление не обладает даже свойством слабой нормализации. Действительно, рассмотрим терм $(\lambda x.xxx)(\lambda x.xxx)$. Заметим, что мы здесь можем применить лишь β-редукцию, получая $(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)$. Здесь, опять, можно применить только β-редукцию, получив $(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)$, и т.д. Очевидно, что у этого терма просто нет никакой нормальной формы. Тем не менее, верно следующее утверждение, называемое теоремой Черча-Россера:

Теорема 2.1.1 (Черча-Россера, [5])
$\forall M, N_1, N_2 \in \Lambda, M \to_{\beta\eta} N_1 \wedge M \to_{\beta\eta} N_2 \implies \exists X \in \Lambda, N_1 \to_{\beta\eta} X \wedge N_2 \to_{\beta\eta} X.$

Здесь $\to_{\beta\eta}$ – рефлексивное, транзитивное замыкание β и η редукции. Из этой теоремы следует, в частности, что нормальная форма единственна (по модулю α-конверсии), если она существует. На самом деле тот факт, что лямбда-исчисление не обладает свойствами нормальности, как раз позволяем выразить в этом исчислении все вычислимые функции. Т.к. любая система, обладающая этими свойствами, Тьюринг-полной уже являться не может [2]. Далее мы будем рассматривать лямбда-термы с точностью до эквивалентности по преобразованием. Формально, мы рассматриваем фактор-множество Λ по отношению эквивалентности – транзитивному, симметричному и рефликсивному замыканию α, β, η-конверсий. Далее, приступим к выражению примитивов вычислимости с помощью λ-термов. Мы будем выражать их с помощью т.н. комбинаторов – замкнутых λ-термов.

2.1.3 Арифметика

Числа в лямбда-исчислении моделируются с помощью так называемых нумералов Черча. Формально, определим:

- $\underline{0} = \lambda fx.x$
- $\underline{1} = \lambda fx.fx$
- $\underline{2} = \lambda fx.ffx$
- $\underline{n} = \lambda fx.f^n x$

То есть, число n моделируется аппликацией функции n раз. Выразим, например функцию следующего элемента **SUCC**:

$$\mathbf{SUCC} = \lambda nfx.f(nfx)$$

Ясно, что $\forall n \in \mathbb{N}\ \mathbf{SUCC}\ \underline{n} = \underline{n+1}$. В этом смысле λ-терм **SUCC** выражает функцию следующего элемента. Подобным образом мы можем выразить и другие арифметические операции:

- **PLUS** $= \lambda mnfx.m(nfx)$
- **MULT** $= \lambda mnf.m(nf)$
- **EXP** $= \lambda mn.nm$

2.1.4 Логика

Определим, что мы будем называть истиной и ложью в лямбда-исчислении:

- **TRUE** $= \lambda xy.x$
- **FALSE** $= \lambda xy.y$

В этих терминах несложно выразить основные логические функции:

- **AND** $= \lambda pq.pqp$
- **OR** $= \lambda pq.ppq$
- **NOT** $= \lambda p.p$ **FALSE TRUE**

Это логические функции в том смысле, что их поведение на термах **TRUE** и **FALSE** соответствуют таблице истинности. Т.е. **AND TRUE FALSE** = **FALSE**, **AND TRUE TRUE** = **TRUE** и т.д.

2.1.5 Пары и трюк Клини

Введем формализм для кодирования пар элементов:

- **PAIR** $= \lambda xyf.fxy$
- **FIRST** $= \lambda p.p$ **TRUE**
- **SECOND** $= \lambda p.p$ **FALSE**

Трюк Клини заключается в построении функционала **K** на парах, который работает следующим образом $(x, y) \overset{\mathbf{K}}{\mapsto} (f(x), x)$ для каких-то фиксированных f, x, y. Теперь ясно как его выразить:

$$\mathbf{K} = \lambda fp.\mathbf{PAIR}\ (f(\mathbf{FIRST}\ p))\ (\mathbf{FIRST}\ p)$$

Трюк Клини является очень полезным приемом для построения комбинаторов. Выразим через него например комбинатор $\mathrm{pre} : \mathbb{N} \to \mathbb{N}$,

$$\mathrm{pre}(n) = \begin{cases} n-1, & n > 0 \\ 0, & n = 0 \end{cases}$$

Обозначим искомый комбинатор за **PRE**. Заметим, что (**PRE** n) есть не что иное, как применение (**K SUCC**) n раз к паре $(0, 0)$ и взятия второго элемента пары:

$$\mathbf{PRE} = \lambda n.\mathbf{SECOND}\ (n\ (\mathbf{K}\ \mathbf{SUCC})\ (\mathbf{PAIR}\ \underline{0}\ \underline{0}))$$

.

2.1.6 Рекурсия

Очень важным приемом в λ-исчислении является реализация рекурсии. Разберем ее на примере реализации факториала. Для начала научимся сравнивать с нулем:

$$\mathbf{ISZERO} = \lambda n.n\ (\lambda x.\mathbf{FALSE})\ \mathbf{TRUE}$$

Здесь мы используем числа как аппликацию функции. Вспомним теперь рекурсивное определение факториала:

$$n! = \begin{cases} n \times (n-1)!, & n > 0 \\ 1, & n = 0 \end{cases}$$

Тогда, комбинатор **FACT** можно выразить просто следующим образом:

$$\mathbf{FACT} = \lambda n.(\mathbf{ISZERO}\ n)\ \underline{1}\ (\mathbf{MULT}\ n\ (\mathbf{FACT}(\mathbf{PRE}\ n)))$$

Проблема этого определения в том, что **FACT** находится в обоих частях равенства, и это формально некорректнеое определение в λ-исчислении. Но здесь нам приходит на помощь еще один трюк. Рассмотрим следующий функционал:

$$\textbf{FACT'} = \lambda gn.(\textbf{ISZERO}\ n)\ \underline{1}\ (\textbf{MULT}\ n\ (g(\textbf{PRE}\ n)))$$

Тогда заметим следующий (чисто синтаксический, на первый взгляд) факт:

$$\textbf{FACT} = \textbf{FACT' FACT}$$

Получается, что искомый комбинатор **FACT** суть неподвижная точка **FACT'**. Тогда, нам осталось предъявить комбинатор, выражающий неподвижную точку данного. То есть такой комбинатор **Y**, что для любого другого комбинатора **F**, верно

$$\textbf{Y F} = \textbf{F}(\textbf{Y F})$$

Мы можем предъявить один из таких комбинаторов [7]:

$$\textbf{Y} = \lambda g.(\lambda x.g(x\ x))(\lambda x.g(x\ x))$$

Докажем это:

$$\begin{aligned} & YF \\ \equiv\ & (\lambda g.(\lambda x.g(x\ x))(\lambda x.g(x\ x)))F \\ \to_\beta\ & (\lambda x.F(x\ x))(\lambda x.F(x\ x)) \\ \to_\beta\ & F((\lambda x.F(x\ x))(\lambda x.F(x\ x))) \\ \equiv\ & F(YF) \end{aligned}$$

Итого, используя **Y**, получаем

$$\textbf{FACT} = \textbf{Y FACT'}$$

Описанная выше схема позволяет нам выражать любые рекурсивные функции.

2.1.7 Тьюринг-полнота

Для доказательства полноты λ-исчисления нужно реализовать машину Тьюринга на этом языке. Классическим способом это сделать является сведения λ-исчисления к μ-рекурсивным функциям, и доказательства уже их полноты. Формальное доказательство можно найти у самого Тьюринга [20].

2.2 Просто типизированное λ-исчисление

Данная формальная система также была предложена Алонзо Черчем [2]. В ней Черч пытался избежать парадоксальных конструкций нетипизированного λ-исчисления. Эта система уже обладает свойством сильной нормализации [18], и даже является в каком-то смысле полноценной логической теорией (то есть имеет место Соответствие Карри — Ховарда, о котором речь пойдет дальше).

2.2.1 Формальное определение

- Типами τ являются базовые типы из некоторого фиксированного множества B, типовые переменные с идентификаторами из множества переменных V ($B \cap V = \emptyset$), а также типы, полученные при помощи применения конструктора типов $\to$:

  $$\tau ::= T \mid v \mid \tau \to \tau$$

  здесь T ∈ B — типы из зафиксированного множества базовых типов, v ∈ V — типовые переменные, идентификаторы которых берутся из некоторого множества V.

- Выражениями e являются константы из некоторого зафиксированного множества C, переменные, абстракции и аппликации (применения):

  $$e ::= c \mid x \mid \lambda x : \tau.e \mid (e\ e)$$

  здесь $c \in C$ — константы, имеющие базовые типы. Константами, являются конкретные значения из базовых типов. Например, $True, False \in Bool$, $0, 1 \in Int$ и т. д.
  Запись $\lambda x : \tau.e$ обозначает типизированную абстракцию — связанная переменная x имеет тип τ.

- Контексты типизации представляют собой множества предположений о типизации вида $x : \tau$, то есть предположений, что «переменная x имеет тип τ»:

  $$\mathbf{\Gamma} = \{x_i : \tau_i\}_{i=1}^{n}$$

- Отношение типизации, обозначаемое как $\mathbf{\Gamma} \vdash e : \tau$ обозначает, что выражение e имеет тип τ в контексте $\mathbf{\Gamma}$ и, таким образом, корректно типизировано. Конкретизации отношений типизации называются суждениями о типизации.

Опишем теперь правила вывода для типов выражений.

Правила вывода типов

- Вывод типа для константы.

$$\frac{c : (T \in B) \in C}{\Gamma \vdash c : T} \vee \frac{c : (v \in V) \in C}{\Gamma \vdash c : v} \tag{1}$$

- Вывод типа для переменной.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \tag{2}$$

- Вывод типа для абстракции.

$$\frac{\Gamma \cup \{x : \tau\} \vdash e : \sigma}{\Gamma \vdash (\lambda x : \tau . e) : \tau \to \sigma} \tag{3}$$

- Вывод типа для аппликации.

$$\frac{\Gamma \vdash e_1 : \tau \to \sigma, e_2 : \tau}{\Gamma \vdash (e_1\ e_2) : \sigma} \tag{4}$$

Список литературы

[1] Xavier Leroy et al. *The CompCert verified compiler.* URL: http://compcert.inria.fr/.

[2] Alonzo Church. "A Formulation of the Simple Theory of Types". *The Journal of Symbolic Logic* **5** 2 (1940), с. 56—68. ISSN: 00224812. URL: http://www.jstor.org/stable/2266170.

[3] Alonzo Church. "A Set of Postulates for the Foundation of Logic". *Annals of Mathematics* **33** 2 (1932), с. 346—366. ISSN: 0003486X. URL: http://www.jstor.org/stable/1968337.

[4] Alonzo Church. "An Unsolvable Problem of Elementary Number Theory". *American Journal of Mathematics* **58** 2 (1936), с. 345—363. ISSN: 00029327, 10806377. URL: http://www.jstor.org/stable/2371045.

[5] Alonzo Church, J. B. Rosser. "Some properties of conversion". *Trans. Amer. Math. Soc.* **39** 3 (1936), с. 472—482. ISSN: 0002-9947. DOI: 10.2307/1989762. URL: https://doi.org/10.2307/1989762.

[6] N. Dershowitz. *SOFTWARE HORROR STORIES.* URL: https://www.cs.tau.ac.il/~nachumd/verify/horror.html.

[7] E. Engeler. "H. P. Barendregt. The lambda calculus. Its syntax and semantics. Studies in logic and foundations of mathematics, vol. 103. North-Holland Publishing Company, Amsterdam, New York, and Oxford, 1981, xiv 615 pp." *Journal of Symbolic Logic* **49** 1 (1984), с. 301—303. DOI: 10.2307/2274112.

[8] Armaël Guéneau, Arthur Charguéraud, François Pottier. "A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification". *Programming Languages and Systems.* под ред. Amal Ahmed. Cham: Springer International Publishing, 2018, с. 533—560. ISBN: 978-3-319-89884-1.

[9] S. C. Kleene, J. B. Rosser. "The Inconsistency of Certain Formal Logics". *Annals of Mathematics* **36** 3 (1935), с. 630—636. ISSN: 0003486X. URL: http://www.jstor.org/stable/1968646.

[10] Gerwin Klein, Tobias Nipkow. “A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler”. *ACM Trans. Program. Lang. Syst.* **28** 4 (июль 2006), с. 619—695. ISSN: 0164-0925. DOI: 10.1145/1146809.1146811. URL: https://doi.org/10.1145/1146809.1146811.

[11] Gerwin Klein, Tobias Nipkow. “Verified Bytecode Verifiers”. *TCS* **298** (2003), с. 583—626.

[12] Dirk Leinenbach, Wolfgang Paul, Elena Petrova. “Towards the Formal Verification of a C0 Compiler: Code Generation and Implementation Correctnes”. *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*. SEFM ’05. USA: IEEE Computer Society, 2005, с. 2—12. ISBN: 0769524354. DOI: 10.1109/SEFM.2005.51. URL: https://doi.org/10.1109/SEFM.2005.51.

[13] Xavier Leroy. “Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant”. *SIGPLAN Not.* **41** 1 (янв. 2006), с. 42—54. ISSN: 0362-1340. DOI: 10.1145/1111320.1111042. URL: https://doi.org/10.1145/1111320.1111042.

[14] Justin T Miller. *A Historical Account of Set-Theoretic Antinomies Caused by the Axiom of Abstraction*.

[15] Raul Rojas. *A Tutorial Introduction to the Lambda Calculus*. 2015. arXiv: 1503.09060 [cs.LO].

[16] Ilya Sergey, Amrit Kumar, Aquinas Hobor. *Scilla: a Smart Contract Intermediate-Level LAnguage*. 2018. arXiv: 1801.00687 [cs.PL].

[17] Martin Strecker. *Compiler Verification for C0 (intermediate report)*.

[18] W. W. Tait. “Intensional Interpretations of Functionals of Finite Type I”. *The Journal of Symbolic Logic* **32** 2 (1967), с. 198—212. ISSN: 00224812. URL: http://www.jstor.org/stable/2271658.

[19] The Coq Development Team. *The Coq Proof Assistant*. вер. 8.13. янв. 2021. DOI: 10.5281/zenodo.4501022. URL: https://doi.org/10.5281/zenodo.4501022.

[20] A. M. Turing. “Computability and λ-Definability”. *The Journal of Symbolic Logic* **2** 4 (1937), с. 153—163. ISSN: 00224812. URL: http://www.jstor.org/stable/2268280.

Благодарности

Благодарности идут тут.