Programming Languages for Feasible Programs

Datalogisk institut, Copenhagen University (DIKU)
Master's Thesis

Oleksandr Shturmov
oleks@oleks.info

December 11, 2014.

Chapter 0

Preface

0.1 Audience

The audience of this thesis is anyone interested in the connection of computability and complexity to the theory of programming languages. In particular, what the admittance of particular programming language constructs implies for the complexity of the programs that you can write.

The thesis is directed towards the level of a Computer Science graduate student at the time of writing: The reader is assumed to be familiar with the basics of discrete mathematics, as in [Graham et al. (1998)]. The analysis of time and space complexity of algorithms, as in [Cormen et al. (2009)], §§1–17 and §§21–24. Regular and context-free languages, as in [Sipser (2013)] §§1–2, and their use for programming language design, as in [Mogensen (2010)]. The reader should also be familiar with Logic in Computer Science, as in [Huth & Ryan (2004)], §§1–4.

0.2 Formal Systems

A formal system is a system of symbols and rules for manipulating them.

A formal system $\mathcal{F}$ can be reasonably realised by a physical system $\mathcal{P}$, provided that (a) there is a reasonable correspondence between the symbols of $\mathcal{F}$ and the physical symbols of $\mathcal{P}$, and (b) $\mathcal{P}$ can manipulate the physical symbols in reasonable accordance with the rules of $\mathcal{F}$.

Physical systems realising formal systems are typically referred to as "computers".

Formal systems bear little intrinsic purpose, but serve as a means to specify the desired behaviour of physical systems.

"Reasonability" therefore is a matter of how accurate

0.3 Preliminaries

Definition 1. *Let $\mathbb{N}$ denote the type of natural numbers, including 0.*

Definition 2. *Given a type Σ, let Σ^* be the type of* ***symbolic strings*** *over Σ, formed using either the* ***empty string*** *or the* ***string concatenation*** *operator:*

$$\frac{}{\varepsilon_\Sigma : \Sigma^*} \qquad \frac{s_0 : \Sigma \quad s : \Sigma^*}{s_0 \cdot s : \Sigma^*}$$

We refer to Σ as an ***alphabet****, and to the terms of Σ as* ***symbols****.*

We omit Σ in ε_Σ, when it is clear from context, e.g. $\varepsilon : \Sigma^*$.

Definition 3. *We say that a term $s \equiv s_0 \cdot s_1 \cdots s_{n-1} \cdot \varepsilon : \Sigma^*$ is a string of length $n : \mathbb{N}$ over the alphabet Σ.*

Definition 4. *Let $|\Sigma|$ be the number of symbols in alphabet Σ, called it's* ***cardinality****.*

If the cardinality of an alphabet is finite, strings over the alphabet are **recursively enumerable**, i.e. there exists a bijection $f : \Sigma^* \to \mathbb{N}$.

Theorem 1. *If $|\Sigma| = n$ for some $n \in \mathbb{N}$, then $|\Sigma^*| = |\mathbb{N}|$.*

Proof. We have $|\Sigma| = n$. The terms of Σ can be arranged in a sequence such that for each $s : \Sigma$ we assign a unique natural number $g(s)$, such that $1 \le g(s) \le |\Sigma|$. We now inductively define the function $f : \Sigma^* \to \mathbb{N}$.

$$f(\varepsilon) = 0 \tag{1}$$
$$f(s_0 \cdot s) = g(s_0) + f(s) \cdot |\Sigma| \tag{2}$$

To show that the function is a bijection, we also define its inverse by induction:

$$f^{-1}(0) = \varepsilon \tag{3}$$
$$f^{-1}(n) = g^{-1}(n \mod |\Sigma|) \cdot f^{-1}(\lfloor n / |\Sigma| \rfloor) \qquad \text{for } n > 0 \tag{4}$$

□

This means that strings over any finite alphabet can be used to represent strings over any other finite alphabet. One such basic alphabet that has proven useful in practice is the binary alphabet, consisting of e.g. the symbols `0` and `1`.

Definition 5. *Let $\mathbb{B}$ denote the type of booleans.*

Define the set $\mathbb{B}$, and the usual boolean connectives.

Definition 6. *An infix function $\le : A \times A \to \mathbb{B}$ defines a* ***total order*** *on the type A iff for all $x, y, z : A$:*

$$x \le y \wedge y \le x \Rightarrow y = x \quad \textit{(antisymmetry)} \tag{5}$$
$$x \le y \wedge y \le z \Rightarrow x \le z \quad \textit{(transitivity)} \tag{6}$$
$$x \le y \vee y \le x \quad \textit{(totality)} \tag{7}$$

Definition 7. *Given a total order on Σ, we define the **lexicographic order** on Σ^* as follows:*

$$\frac{}{\varepsilon \leq s_0 \cdot s} \quad \frac{s_0 = t_0 \quad s \leq t}{s_0 \cdot s \leq t_0 \cdot t} \quad \frac{s_0 \neq t_0 \quad s_0 \leq t_0}{s_0 \cdot s \leq t_0 \cdot t} \tag{8}$$

Theorem 2. *A lexicographic order is a total order.*

Proof. ☐

Definition 8. *An infix function $<: A \times A \to \mathbb{B}$ defines a **strict total order** on the type A, iff*

$$\text{F-LESS:}\frac{\neg\left(y \leq x\right)}{x < y} \tag{9}$$

We say that y has a higher **value** than x, whenever $x < y$, and equal in value whenever $x = y$.

Definition 9. *A function f is defined by **primitive recursion** from the functions $h, g_1, g_2, \ldots, g_n$ for $n : \mathbb{N}$, iff*

$$f\left(\varepsilon, \overline{y}\right) = h\left(\overline{y}\right) \tag{10}$$

$$f\left(s_i\left(x\right), \overline{y}\right) = g_i\left(x, \overline{y}, f\left(x, \overline{y}\right)\right) \tag{11}$$

$$x < s_i\left(x\right) \tag{12}$$

That is, a function is defined by primitive recursion, if on every invocation of the function, we recurse on at most one formal parameter, and only recurse after the actual parameter has been decreased in value. It follows that primitive recursion demands a strict total order on the value type in question.

Definition 10. *A function is **primitive recursive** if it is non-recursive, or defined by primitive recursion from non-recursive, or primitive recursive functions.*

Primitive recursive functions are not necessarily polytime functions.

Example 1. *Unary addition over binary notation is primitive recursive, but not polytime.*

Let $\leq$ be the lexicographic order on binary notation.

We now define unary addition over binary notation using primitive recursion:

$$add'\left(0,0,0\right) = \left(0,0\right) \tag{13}$$

$$add'\left(1,0,0\right) = \left(1,0\right) \tag{14}$$

$$add'\left(0,1,0\right) = \left(1,0\right) \tag{15}$$

$$add'\left(0,0,1\right) = \left(1,0\right) \tag{16}$$

$$add'\left(0,0,1\right) = \left(1,0\right) \tag{17}$$

$$add'\left(1,1,0\right) = \left(0,1\right) \tag{18}$$

$$add'\left(1,1,1\right) = \left(1,1\right) \tag{19}$$

Part I

Background

Chapter 1

Computability

Notion 1. *A problem is "computable" if it can be solved by transforming a mathematical object over a finite amount of time, without ingenuity.*

Any attempt at a more definite notion of computability seems to arrive at a philosophical impasse, where the notions of "transformation", "mathematical object", and "ingenuity" form a philosophical conundrum. The indefinite notion however, is sufficient to state the following theorem:

Theorem 3. *The class of computable problems is closed under concatenation.*

That is, if a problem P can be solved by solving a computable problem Q, followed by solving a computable problem R, then P itself is computable.

Proof. Since both Q and R are computable, and no transformations are performed, other than to solve the problems Q and R, P itself is computable. $\square$

Thus we arrive at the folklore notion of an algorithm:

Notion 2. *An "algorithm" is a specification of how a problem can be solved by solving a finite sequence of computable problems.*

Such indefinite notions are useful for little else. We are left to take a philosophical leap of faith and define some notion of mathematical object, and transformation without ingenuity.

1.1 Function Algebras

A formal system is a system of mathematical symbols and rules for employing them. We'll refer to formal systems as algebras.

An algebra is a set of symbols and rules for manipulating them. In this sense, and algebra

Definition 11. *A **function algebra** $\mathcal{A}$ is a set of functions, including a set of basic functions $\mathcal{A}_{\mathrm{B}}$ and a set of operations $\mathcal{A}_{\mathrm{O}}$.*

1.2 Machines

TODO:

- The above definition is useful for little else - provide some historical perspective on defining computability beyond this notion, and provide a characterization using function algebras (similar to Church) and Turing machines. Draw parallels to type theory an constructivism.
- The classical result that primitive recursive functions are computable. To argue for this, we probably need to argue for a type theoretic approach, in that, what we can construct, we can compute. Function algebras should also be introduced here.
- General recursion (due to Kleene wrt. definition) and it's undecidability (due to Church).
- A different approach to computability: Post and Turing machines. Prove their equivalence to general recursion above.

Chapter 2

Complexity

This may be a good point to mention that, although I have so far been tacitly equating computational difficulty with time and storage requirements, I don't mean to commit myself to either of these measures. It may turn out that some measure related to the physical notion of work will lead to the most satisfactory analysis; or we may ultimately find that no single measure adequately reflects our intuitive concept of difficulty.

— ALAN COBHAM, *Logic, Methodology and Philosophy of Science* (1964)

In practice, the length of computer computations must be restricted, otherwise the cost in time and money would be prohibitive.

— H. E. ROSE, *Subrecursion: functions and hierarchies* (1984)

Definition 12. *The computational complexity of a function* f*, wrt. a particular resource, quantifies the use of that resource as a function of the length of the input string.*

2.1 Time

2.1.1 Polynomial Time

- Recursive characterization of polytime functions in [Rose (1984)], proving certain claims by [Cobham (1965)]. Both question the relation to the Grzegorczyk hierarchy [Grzegorczyk (1953)].
- Leivant's paper - A Foundational Delineation of Computational Feasibility.
- Bellantoni and Cook paper - A NEW RECURSION-THEORETIC CHARACTERIZATION OF THE POLYTIME FUNCTIONS
- Niel Jones paper.
- Caporaso
- Upper bounds (algorithms) can be produced by expressing the property of interest in one of our languages. Lower bounds proven elsewhere can be used as a proof that the language is expressive enough.

2.1.2 Subpolynomial Time

In what follows, we delineate a hierarchy of complexity classes, strictly under polynomial time. That is, we present a sequence $C_1(n)$, $t(n) = o\left(n^{O(1)}\right)$. For each complexity class C, we present a representative problem P_C. We aim to find problems which are known to be $t(n) = \Theta(C(n))$.

- Some problems, although computable in polynomial time, are still hard to compute in practice (ICALP'2014, Amir Abboud).
- Remind of the definitions of O, Ω, etc.
- For each of the below show that every subsequent class is distinct from the proceeding, and exhibit some "complete" problems for these classes.

$O(1)$ **— Constant**

$O(\alpha(n))$ **— Inverse Ackermann**

$O(\log^*(n))$ **— Log star**

$O(\log\log(n))$ **— Log-log**

$O(\log(n))$ **— Log**

$O\left(\log(n)^{O(1)}\right)$ **— Polylog**

$O(n^c)$**, for** $0 < c < 1$ **— Fractional power**

$O(n)$ **— Linear time**

$O(n\log^*(n))$ **— n log star**

$O(n\log\log(n))$ **— n log-log**

$O(n\log(n))$ **— n log n** Comparison-based sorting $o(n\log n)$.

$O\left(n^2\right)$ **— quadratic**

$O\left(n^3\right)$ **— cubic**

$O\left(n^O(1)\right)$ **— polynomial**

2.1.3 Space

Chapter 3

Implicit Characterizations of P

3.1 Primitive Recursion on Notation

When dealing with the manipulation of symbolic strings, there is a natural total order on the values of the formal parameters — the length of their representing string. Primitive recursion on notation utilizes this order, requiring that the length of the input string be decreased before a recursive call.

Definition 13. *A function* f *is defined by primitive recursion on notation from functions* $g, h_1, h_2, \ldots, h_{|\Sigma|}$ *iff*

$$f\left(\varepsilon, \bar{t}\right) = g\left(\bar{t}\right) \tag{3.1}$$

$$f\left(s_i \cdot s, \bar{t}\right) = h_i\left(s, \bar{t}, f\left(s, \bar{t}\right)\right) \qquad \forall s_i : \Sigma \tag{3.2}$$

We say that a function is primitive recursive on notation (PRN), if it is defined by primitive recursion on notation from non-recursive or PRN functions.

Unfortunately, not all PRN functions take polynomial time.

Theorem 4. *There exist PRN functions which do a superpolynomial amount of work.*

Proof. Consider a function g, which duplicates every symbol in the input string:

$$g\left(\varepsilon\right) = \varepsilon \tag{3.3}$$

$$g\left(0 \cdot s\right) = 0 \cdot 0 \cdot g\left(s\right) \tag{3.4}$$

Consider furthermore a function h, which calls g iteratively, the same number of times as the length of its input string:

$$h\left(\varepsilon\right) = 0 \tag{3.5}$$

$$h\left(0 \cdot s\right) = g\left(h\left(s\right)\right) \tag{3.6}$$

Calling g with a string of length n, we obtain a string of length 2^n due to iterated duplication. It follows that g does a superpolynomial amount of work. □

Example 2. *We illustrate the above proof with an example:*

$$
\begin{aligned}
h(0\cdot 0) &\rightsquigarrow g(h(0)) && (3.7)\\
&\rightsquigarrow g(g(h(\varepsilon))) && (3.8)\\
&\rightsquigarrow g(g(0)) && (3.9)\\
&\rightsquigarrow g(0\cdot 0\cdot g(\varepsilon)) && (3.10)\\
&\rightsquigarrow g(0\cdot 0) && (3.11)\\
&\rightsquigarrow 0\cdot 0\cdot g(0) && (3.12)\\
&\rightsquigarrow 0\cdot 0\cdot 0\cdot 0\cdot g(\varepsilon) && (3.13)\\
&\rightsquigarrow 0\cdot 0\cdot 0\cdot 0 && (3.14)
\end{aligned}
$$

3.2 Bounded Primitive Recursion on Notation

With reference to the "extended rudimentary functions" of [Bennett (1962)], [Cobham (1965)] defined bounded primitive recursion on integers in decimal notation. For a similar definition on binary notation, see [Rose (1984), p. 127].

We generalize this to an arbitrary alphabet Σ, building upon PRN above:

Definition 14. *A function f is defined by bounded primitive recursion on notation from functions $g, h_1, h_2, \ldots, h_{|\Sigma|}$, and k iff*

$$
\begin{aligned}
f(\varepsilon, \overline{y}) &= g(\overline{y}) && && (3.15)\\
f(s_i \cdot x, \overline{y}) &= h_i(x, \overline{y}, f(x, \overline{y})) && \forall s_i : \Sigma && (3.16)\\
f(x, \overline{y}) &\leq k(x, \overline{y}) && && (3.17)
\end{aligned}
$$

We say that a function is bounded primitive recursive on notation (BPRN) if it is defined by bounded primitive recursion on notation from non-recursive, or BPRN functions. The scheme is also known as limited primitive recursion on notation.

The addition that we make to PRN is that the function f must be bounded from above by function i. Letting i characterize polynomial time functions, we obtain perhaps the earliest implicit characterization of P due to [Cobham (1965)].

This straight-forward approach has an obvious limitation: it requires defining an ordering relation on our functions — a problem that is undecidable in general. Furthermore

3.3 Finite Model Theory

Finite model theory

3.4 Ramification

3.4.1 Safe Recursion

3.4.2 Tiering

Bibliography

Published in Journals

[Grzegorczyk (1953)] Andrzej Grzegorczyk. Institut Matematiczny Polskiej Akademii Nauk. *Some classes of recursive functions*. 1953. Rozprawy Matematyczne: IV. Edited by Karol Borsuk, et al. Published by Polskie Towarzystwo Matematyczne Warszawa in Poland. Printed in Poland, by Wroclawska Drukarnia Naukowa.

Cited on page 9.

[Hofmann (2000)] Martin Hoffmann. Laboratory for Foundations of Computer Science, University of Edinburgh. *Programming languages capturing complexity classes*. 2000. ACM SIGACT News, Vol. 31, No. 1, pp. 31–42. Published by the ACM in New York, USA.

A concise survey of a range approaches to implcit complexity theory between 1965 and 2000.

No citations.

Books

[Cormen et al. (2009)] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Massachusetts Institute of Technology. *Introduction to Algorithms*, 3rd ed. 2009. Published by The MIT Press in Cambridge, Massachusetts, USA. ISBN: 978-0-262-03384-8. Printed in the USA.

Cited on page 2.

[Graham et al. (1998)] Ronald Lewis Graham, Donald Ervin Knuth, Oren Patashnik. *Concrete Mathematics*. Second Edition. 1998. Published by Addison-Wesley Publishing Company, Inc. ISBN: 978-0-201-55802-9. 24th printing. 2011. Printed in Westford, Massachusetts, USA.

Cited on page 2.

[Huth & Ryan (2004)] Michael Huth and Mark Ryan. Imperial College London and University of Birmingham, United Kingdom. *Logic in Computer*

Science: Modelling and Reasoning about Systems, Second Edition. 2004. Published by Cambridge University Press in New York, USA. ISBN: 978-0-521-54310-1. 7th printing. 2011. Printed in the United Kingdom by University Press, Cambridge.

Cited on page 2.

[Mogensen (2010)] Torben Ægidius Mogensen. Department of Computer Science, University of Copenhagen. *Basics of Compiler Design*. Anniversary Edition. 2010. Published through `lulu.com`. ISBN: 978-87-993154-0-6.

Cited on page 2.

[Rose (1984)] H. E. Rose, School of Mathematics, University of Bristol. *Subrecursion: functions and hierarchies*. 1984. Oxford Logix Guides: 9. Typeset by Joshua Associates, Oxford. Published by Clarendon Press, division of Oxford University Press, in New York, USA. ISBN 0-19-853189. Printed in Great Britain, by The Thetford Press Ltd.

Cited on pages 9 and 12.

[Sipser (2013)] Michael Sipser. *Introduction to the Theory of Computation*. Third Edition. 2013. Published by Cengage Learning. ISBN: 978-1-133-18779-0. Printed in the USA.

Cited on page 2.

In Proceedings

[Cobham (1965)] Alan Cobham. IBM Research Center, Yorktown Heights, New York, USA. *The intrinsic computational difficulty of functions*. 1965. In Proceedings of the 1964 International Congress for Logic, Methodology and Philosophy of Science, pp. 24–30. Edited by Yehoshua Bar-Hillel. Published by North-Holland Publishing Company in Amsterdam, Holland. Printed in Israel, by Jerusalem Academic Press Ltd.

Cited on pages 9, 12, and 16.

[Sazonov (1987)] V.Yu. Sazonov. Institute of Mathematics, Novosibirsk, USSR. *Bounded set theory and polynomial computability*. 1987. In Proceedings of the 1987 International Conference on Fundamentals of Computation Theory. Lecture Notes in Computer Science, Vol. 278, pp. 391–395. Editied by L. Budach, R.G. Bukharajev, and O.B. Lupanov. Published by Springer-Verlag in Berlin, Germany. Printed by Druckhaus Beltz.

No citations.

Other

[Bennett (1962)] J.H. Bennett. Ph.D.Thesis, Princeton University. *On Spectra*. 1962. Princeton, New Jersey, USA.

Referenced by [Cobham (1965)], not attained.
Cited on page 12.