Phonecode in Axiomatic Language

Walter W. Wilson
retired
Fort Worth, Texas, USA
wwwilson3210@gmail.com

The phonecode project of Lutz Prechelt [4] measured program size and programming time for multiple implementations of the same problem in different languages with different programmers. The results support the maxim that programming time per line of noncomment code is roughly constant, regardless of language. An 8-line axiomatic language specification for the phonecode problem is presented, suggesting a productivity benefit.

1 Introduction

This paper gives a specification in axiomatic language [8] for the "phonecode" problem of Lutz Prechelt [5] [4] [6]. The problem is to encode phone numbers as sequences of words from a dictionary, given a mapping of letters to digits.

The software engineering assumption for this project is that programming time per line of (noncomment) code is roughly the same, regardless of language level. So, if a C language program for a problem is typically 1/3 the size of an assembly language program for the same problem (ignoring comment lines), we would expect programmer productivity in C to be roughly 3 times greater.

Another assumption for this project is that the automatic transformation of axiomatic language specifications to efficient programs for problems like this can eventually be achieved. Thus, this currently-nonexecutable specification can be considered an actual solution to the phonecode problem.

Section 2 discusses claims and evidence for the idea that a higher-level programming language that enables shorter programs leads to shorter programming time and thus greater programmer productivity. Section 3 defines the phonecode problem and section 4 gives an axiomatic language specification for that problem. Section 5 gives some final comments and discusses future work.

2 Roughly Constant LOC/hr?

The idea that programming time per line of noncomment code (loc) is roughly constant regardless of language level was an early observation in software engineering. Fred Brooks writes "Productivity seems constant in terms of elementary statements, a conclusion that is reasonable in terms of the thought a statement requires and the errors it may include" [1] (p.94). Prechelt [5] (p.25) refers to "Common software engineering wisdom, which says that 'the number of lines written per hour is independent of the language'".

Prechelt's phonecode results support this claim, where lines-of-code are "anything that contributes to the semantics of the program in each of the program source files, e.g. a statement, a declaration, or at least a delimiter such as a closing brace (end-of-block marker)" [6] (p.31). Based on median program lengths and programming times, "non-scripts [C, C++, Java] are typically two to three times as long as scripts [Perl, Python, Rexx, Tcl]" (p.31) and "scripts (total median 3.1 hours) take only about one third

Submitted to:
ICLP 2024

of the time required for the non-scripts (total median 10.0 hours)" (p.37), with some caveats, such as the script times being self-reported.

The median loc/hr values for the seven languages are shown in Fig. 24 (p.41) and are all between 20-40. We therefore accept the assumption of roughly constant loc/hr regardless of language. Paul Graham [3] argues this and makes the logical inference that "the main point of high-level languages is to make source code smaller". For very-high-level, declarative, functional, and logic languages, there may be unrealized potential to make our programs smaller and thus achieve greater productivity (without sacrificing readability). Exploring that possibility is the motivation for this project. (But some may be skeptical of success. [2])

3 The Phonecode Problem

The phonecode program first reads a text dictionary file of zero-or-more words, one per line and a text file of zero-or-more phone numbers, one per line. It then "encodes" the phone numbers as word strings, using a map of letters to digits, with single phone number digits inserted where no word is possible. A phone number is a string of zero or more digits, dashes (-), and slashes (/). A dictionary word is a string of (upper- and lowercase) letters, dashes, and double quotes ("), starting with a letter.

The following sequence of letter groups defines the mapping of letters to the digits 0-9:

```
    (e jnq rwx dsy ft am civ bku lop ghz)
->  0  1   2   3  4  5   6   7   8   9
```

(E.g., 'd' maps to '3'; 'H' to '9'.) From this we get a map of dictionary words to digit strings.

The encoding of a phone number to words is done left-to-right in incremental steps, with special characters ignored. At any point a prefix of a phone number will have been encoded to a word string with possible included digits. An encoding step attempts to match the starting digits of the not-yet-encoded phone number suffix with digit strings for dictionary words. For each match the dictionary word is appended to the encoding and the matching digits are removed from the phone number suffix. If there are no matches, a single phone number digit is moved to the encoding, but only if the previous encoding step did not also move a digit. (An encoding thus cannot have consecutive digits.) When a sequence of encoding steps has "consumed" the phone number, we have found an encoding for that number. A phone number may have zero or more encodings.

The phone number - encoding pairs found are listed in an output file, one pair per line, in the following format:

```
_phonenum: {[_digit] _word} [_digit]
```

The special characters in _phonenum and _word are included in the output. See the Prechelt papers [5] [4] [6] for his natural language description of the phonecode requirements, along with examples.

4 An Axiomatic Language Solution

The axiomatic language specification below for the phonecode problem has just 8 noncomment lines of code. Since axiomatic language is an extremely minimal language, it must be built-up with assorted utility functions, such as numbers, higher-order forms, Boolean functions, etc. These utilities would be provided in a standard library, which would be shared by all applications, and thus this reusable

code would not be counted as part of the source code for any particular application. Axiomatic language encourages the abstraction of "general purpose functionality" from an application, thus making it shorter.

A "type" or expression syntactic category may be denoted by a name beginning with an underscore: _num, _ltr. A string of zero or more of the same type within a sequence may be denoted by an underscore before and after the type name: _dig_. See the phonecode webpage [10] for a list of abbreviations used.

A program that reads multiple input text files and writes a single output text file can be specified by a set of "Program" valid expressions of the following form:

```
(Program _infiles_ _outfile)  - >=0 input files, 1 output file
```

A text file is a sequence of lines, each a character sequence. A similar functional style is used for many predicates:

```
(_fn _iargs_ _oarg)   - func on >=0 input args yields single output arg
```

The axioms of the phonecode solution are given below. Indices in column 1 count the noncomment source lines.

```
1 (Program %dic %ph#s %outf)<      ! Program phonecode
    ! encode phone numbers with dictionary words using letter>digit map
2  ((ltr>dig digs>word) (e jnq rwx dsy ft am civ bku lop ghz) %dic %digs>word),
     ! given letter>digit map, get map of digits to dictionary words
3  (((* (filt dig?)) (* (eat (bite1 %digs>word)))
4    (* (filt (~consec? dig?)))) %ph#s %codes),
     ! "eat" ph# digits, outputting words (or digit when no "bite" possible)
     ! eliminate codes with consecutive digits
5  (((* (.- join)) cat* *fmt) %ph#s %codes %outf).
     ! format phone numbers and their codes for output file

! ltr>dig - get letter-to-digit mapping from seq of lc letter-group symbols
6 (def ltr>dig (*sel1 el>ix1 dup lc>uc cat)).

! digs>word - get map of digit strings to dictionary words
7 (def digs>word ((skip 1 (dup (* (filt ltr?)))) (.-- map) *join)).

! fmt - format phonenum-code pair for output line
8 (def fmt ((cur ins 1 ": ") (repl 2 (cur ins_btw ' ')) chxstr)).
```

The phonecode webpage [10] gives an expanded, tutorial presentation of these axioms with additional comments inserted to explain the details. Writing and reading this program requires comprehensive familiarity with the standard library utility functions (at `www.axiomaticlanguage.org/phcode/util/`). The utility source files and some of the predicates they define are as follows:

.../util/form.ax.txt - utilities involving the form of expressions:
`(join _a _b (_a _b))` - join two argument exprs into a 2-element sequence
`(cat (_str0_) (_str1_) (_str0_ _str1_))` - catenate two seqs into one
`(cat* ((_strA_) .. (_strZ_)) (_strA_ .. _strZ_))` - cat seq of seqs into one
`(ins_btw _e _seq _seq')` - insert expr _e between seq elements
`(map _map _ex _ex')` - apply _map, consisting of (..(_ex _ex')..) pairs, to an expr

.../util/ho.ax.txt - higher-order predicates and functions:
`(_f_)` - function composition on input arg stack, applied left-to-right
`def` - define a name for a function expression
`dup` - duplicate the first input argument in a function composition
`(* _pred)` - apply predicate to sequences of arguments
`(.- _binfn/rel)` - apply binary fn/rel to arg0 and all elems of arg1 seq
`(.-- _binfn/rel)` - apply binary fn/rel to arg0 and all elems of arg1 seq's seqs

.../util/natnum.ax.txt - natural number predicates and functions
- no utilities directly used

.../util/bit.ax.txt - predicates and functions that use bit representation
- no utilities directly used

.../util/char.ax.txt - predicates and functions involving characters:
`*pred` - apply predicate _pred to sequences of arguments
`lc>uc` - map lowercase letters to uppercase in nested char expr
`sel0, sel1, ..` - select elem from a sequence [0..]
`(el>ix1 _elgrps _el>dig)` - given seq of elem group seqs, get elem-to-1-dig-index map
`(skip _dsym _f)` - skip _dsym (decimal symbol) iargs, then apply fn
`(repl _dsym _uf)` - replace elem [_dsym] of iarg0 seq w unary fn _uf applied to it
`(ins _dsym _e _seq _seq')` - insert expr before elem [_dsym] of seq
`(chxstr _chexpr _chstr)` - flatten nested char expr into a char string

.../util/bool.ax.txt - Boolean predicates and functions (- ending ? is a convention):
`_type?` - returns true/false if expr is of given type (e.g., `dig?`, `ltr?`)
`~bfn?` - returns complement of Boolean function _bfn?
`(consec? _type?)` - t/f if seq has consecutive elems of given type

.../util/stream.ax.txt - incremental stream processing functions:
`(bite1 _map)` - if possible, "bite" a stream prefix that matches map, outputting expr;
- else, copy 1 stream elem to output
`(eat _bite)` - "eat" a stream using all possible bite sequences, outputting expr seqs

5 Conclusions and Future Work

An axiomatic language solution to the phonecode problem has been defined in just 8 lines – much shorter than other phonecode results [4] (p.29). More example programs need to be written in axiomatic language to see if a small size is typical. If so, that would be strong quantitative evidence for a potential productivity benefit with this logic programming language.

The attributes of axiomatic language contribute to the small size of specifications. A conventional language program defines the internal algorithm, which implicitly defines the program external behavior. An axiomatic language specification defines just the external behavior – a proper subset of the conventional program information.

The minimal, purely-symbolic nature of axiomatic language means that basic functions can get ide-

alistic, elegant, foundational definitions. The language extensibility makes it easy to define powerful higher-order forms and to define new functions from existing ones with fine-grained modularity. The Lisp syntax may be the simplest, most-general form for representing data and code. The metalanguage capability of axiomatic language means that new language features, even new paradigms, can be defined and used without one leaving axiomatic language. Axiomatic language can thus subsume other languages for the purpose of specification.

The small size and purity of axiomatic language should make it well suited to proof [9]. Proof would be used to guarantee equivalence between a user's specification and the generated efficient program. Proof would also be used to prove assertions about a specification to validate it, which may be more powerful than just testing.

Additional syntax extensions could be defined to help reduce parentheses and improve readability. It would also be interesting to use byte codes x80 - xFF for new character symbols to represent commonly-used functions and higher-order forms. These enhancements would not modify the definitions and rules of the core of axiomatic language. There is also, however, "extended axiomatic language" [7], which uses the complement of the set of valid expressions as a form of negation, and this should have greater specification power.

The grand challenge of axiomatic language is to automatically transform specifications to efficient programs. A transformation system would need comprehensive "knowledge" of specification patterns and programming concepts in order to "understand" the input specification. Then it would use algorithm knowledge with embedded proof to generate an equivalent program from that understanding. One can argue that no system would be able to automatically transform all possible specifications that a user might write, but that may not be necessary. A satisfactory implementation would be one that could automatically transform straightforward specifications for typical problems. If a specification is not so straightforward or the problem not so typical, then an expert would need to add knowledge to the system so that automatic transformation could proceed.

References

[1] Frederick P. Brooks, Jr. (1975): *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley. Available at `https://web.eecs.umich.edu/~weimerw/2018-481/readings/mythical-man-month.pdf`.

[2] Jonathan Edwards (2010): *The Myth of the Super Programming Language*. Available at `https://alarmingdevelopment.org/?p=392`.

[3] Paul Graham (2002): *Succinctness is Power*. Available at `https://paulgraham.com/power.html`.

[4] Lutz Prechelt (2000): *An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program*. Technical Report, Dept. of Informatics, Univ. of Karlsruhe. Available at `https://page.mi.fu-berlin.de/prechelt/Biblio/jccpprtTR.pdf`.

[5] Lutz Prechelt (2000): *An empirical comparison of seven programming languages*. *Computer* 33(10), doi:10.1109/2.876288. Available at `https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=6bf1d8f08cfaf6db856813edbbb9241555f621ea`.

[6] Lutz Prechelt (2003): *Are Scripting Languages any Good? A Validation of Perl, Python, Rexx, and Tcl against C, C++, and Java*, pp. 205–270. *Advances in Computers* 57, Elsevier. Available at `https://page.mi.fu-berlin.de/prechelt/Biblio/jccpprt2_advances2003.pdf`.

[7] Walter W. Wilson (2013): *Extended Axiomatic Language*. Available at `http://www.axiomaticlanguage.org/EAL.html`.

[8] Walter W. Wilson (2020): *A Concise Definition of Axiomatic Language*. Available at `http://www.axiomaticlanguage.org/concise.htm`.

[9] Walter W. Wilson (2022): *Proof in Axiomatic Language*. Available at `http://www.axiomaticlanguage.org/proof.htm`.

[10] Walter W. Wilson (2024): *Phonecode in 8 Lines*. Available at `http://www.axiomaticlanguage.org/phcode/phonecode.htm`.