A Tiny Specification Metalanguage

Walter Wilson, Yu Lei
Dept. of Computer Science and Engineering,
The University of Texas at Arlington
Arlington, Texas 76019, USA
wwwilson1@sbcglobal.net, ylei@cse.uta.edu

***Abstract*— A logic programming language with potential software engineering benefit is described. The language is intended as a specification language where the user specifies software functionality while ignoring efficiency. The goals of the language are: (1) a pure specification language – "what, not how", (2) small size, and (3) a metalanguage – able to imitate and thus subsume other languages. The language, called "axiomatic language", is based on the idea that any function or program can be defined by an infinite set of symbolic expressions that enumerates all possible inputs and the corresponding outputs. The language is just a formal system for generating these symbolic expressions. Axiomatic language can be described as pure, definite Prolog with Lisp syntax, HiLog higher-order generalization, and "string variables", which match a string of expressions in a sequence.**

Keywords- specification; metalanguage; logic programming; Prolog; HiLog; Lisp; program transformation

I. INTRODUCTION

Programming languages affect programmer productivity. Prechelt [1] found that the scripting languages Perl, Python, Rexx, and Tcl gave twice the productivity over conventional languages C, C++, and Java. A factor of 4-10 productivity gain has been reported for the declarative language Erlang [2]. Both studies showed that lines of source code per hour were roughly the same, regardless of language.

With current programming languages programmers generally keep efficiency in mind. Even in the logic programming field, which is supposed to be declarative, the programmer must understand the execution process in order to write clauses that terminate and are efficient. Program transformation potentially offers a way of achieving the goal of declarative programming where one can write specifications without considering their implementation and then have them transformed into efficient algorithms. One may thus ask what kind of programming language would we want if we could ignore efficiency and assume that a smart translator could transform our specifications?

This paper describes a logic programming language called "axiomatic language" [3] that is intended to be such a programming/specification language. The goals of the language are as follows:

1) a pure specification language – There are no language features to control program execution. The programmer writes specifications while ignoring efficiency and implementation. Program transformation is thus required. We assume that a translator can be built that can transform the specifications into efficient programs.
2) small, but extensible – The language should be as small as possible with semantics as simple as possible. Nothing is built-in that can be defined. This means that arithmetic would not be built-in. (We assume the smart translator can "recognize" the definitions of arithmetic and replace them with the corresponding hardware operations.) Of course, any small language must also be highly extensible, so that those features which are not built-in can be easily defined and used for specification as if they had been built-in.
3) a metalanguage – There are many ways in which one might want to specify software: first-order logic, set operations, domain specific languages, even a procedural language, etc. An ideal language would be able to define within itself other language features and paradigms, which the user could then use for software specification. With such a metalanguage capability, one could not only define functions, but also define new ways of defining functions.

We also have the goal of beauty and elegance for the language.

Axiomatic language is based on the idea that the external behavior of a program – even an interactive program – can be specified by a static infinite set of symbolic expressions that enumerates all possible inputs – or sequences of inputs – along with the corresponding outputs. Axiomatic language is just a formal system for defining these symbolic expressions.

Section II defines the language and section III gives examples. Section IV shows how symbolic expressions can be interpreted as the input and output of real programs. Section V discusses the novel aspects of the language and section VI gives conclusions. This paper extends an earlier publication [4] with syntax extensions and higher-order examples.

II. THE LANGUAGE

This section defines axiomatic language, which is intended to fulfill the objectives identified in the introduction. Section A introduces the language informally with examples. The semantics and syntax of the core language are defined in section B, and section C gives some syntax extensions.

A. An Overview

Axiomatic language can be described as pure definite Prolog with Lisp syntax, HiLog [5] higher-order generalization,

and “string variables”, which match a string of expressions in a sequence. A typical Prolog predicate is represented in axiomatic language as follows:

```
father(sue,X)    ->    (father Sue %x)
```

Predicate and function names are moved inside the parentheses, commas are replaced with blanks, “expression” variables start with %, and both upper and lowercase letters, digits, and most special characters can be used for symbols.

Clauses, or “axioms” as they are called here, are essentially the same as in traditional logic programming, as shown by the following definitions of natural number operations in successor notation:

```
(number 0).          ! set of natural numbers
(number (s %n))< (number %n).

(plus 0 % %)< (number %).        ! addition
(plus (s %1) %2 (s %3))< (plus %1 %2 %3).

(times 0 % 0)< (number %).    ! multiplication
(times (s %1) %2 %3)< (times %1 %2 %x),
                      (plus %x %2 %3).
```

The symbol < replaces the Prolog symbol :- and comments start after !. These axioms generate “valid expressions” such as (number (s (s (s 0)))) and (plus (s 0) (s (s 0)) (s (s (s 0)))), which are interpreted as the statements “3 is a number” and “1 + 2 = 3”, respectively.

The two main features of axiomatic language in comparison to Prolog are its higher-order property and string variables. The higher-order property comes from the fact that predicate and function names can be arbitrary expressions, including variables, and that variables can represent entire predicates in axioms. Section III.A gives some higher-order examples.

The other major feature of axiomatic language is its string variables. A string variable, beginning with $, matches zero or more expressions in a sequence, while an expression variable %... matches exactly one. String variables enable more concise definitions of predicates on lists:

```
(append ($1) ($2) ($1 $2))    ! concatenation

(member % ($1 % $2)).    ! member of sequence

(reverse () ()).       ! reversing a sequence
(reverse (% $) ($rev %))<
  (reverse ($) ($rev)).
```

Some example valid expressions are (append (a b) (c d) (a b c d)) and (member c (a b c)). String variables can be considered a generalization of Prolog's list tail variables. For example, the Prolog list [a, X | Z] would be represented in axiomatic language as the sequence (a %X $Z). But string variables can occur anywhere in a sequence, not just at the end. Note that sequences in axiomatic language are used both for data lists and term arguments. This single representation is convenient for representing code as data.

B. The Core Language

This section gives the definitions and rules of axiomatic language, while the next section gives some syntax extensions. In axiomatic language, functions and programs are specified by a finite set of “axioms”, which generate a (usually) infinite set of “valid expressions”, analogous to the way productions in a grammar generate strings. An **expression** is

> an **atom** – a primitive, indivisible element,
> an **expression variable**,
> or a **sequence** of zero or more expressions and **string variables**.

The hierarchical structure of expressions is inspired by the functional programming language FP [6].

Atoms are represented syntactically by symbols starting with the backquote: `` `abc ``, `` `+ ``. (The non-variable symbols seen previously are not atoms, as section C explains.) Expression and string variables are represented by symbols beginning with % and $, respectively: `%expr`, `%`, and `$str`, `$1`. A sequence is represented by a string of expressions and string variables separated by blanks and enclosed in parentheses: `` (`xyz %n), (`M1 $ ()). ``

An **axiom** consists of a **conclusion** expression and zero or more **condition** expressions, in one of the following formats:

```
conclu < cond1, …, condn.     ! n>0
conclu.          ! an unconditional axiom
```

Axioms may be written in free format over multiple lines and a comment may appear on the right side of a line, following an exclamation point. Note that the definition of expressions allows atoms and expression variables to be conclusion and condition expressions in axioms.

An axiom generates an **axiom instance** by the substitution of values for the expression and string variables. An expression variable can be replaced by an arbitrary expression, the same value replacing the same variable throughout the axiom. A string variable can be replaced by a string of zero or more expressions and string variables. For example, the axiom,

```
(`a (%x %y) $1)< (`b %x $w), (`c $w %y).
```

has an instance,

```
(`a ((` %) `y))< (`b (` %) () `v),
                 (`c () `v `y).
```

by the substitution of `` (` %) `` for `%x`, `` `y `` for `%y`, the string ‘`() `v`’ for `$w` and the empty string for `$1`.

The conclusion expression of an axiom instance is a **valid expression** if all the condition expressions of the axiom instance are valid expressions. By default, the conclusion of an unconditional axiom instance is a valid expression. For example, the two axioms,

```
(`a `b).
((%) $ $)< (% $).
```

generate the valid expressions (`a `b), ((`a) `b `b), (((`a)) `b `b `b `b), etc. Note that the semantics of axiomatic language is based on definitions that enumerate a set of hierarchical expressions and not on the operation of a resolution algorithm. Note also that valid expressions are just abstract symbolic expressions without any inherent meaning. Their association with real computation and inputs and outputs is by interpretation, as described in section IV.

C. Syntax Extensions

The expressiveness of axiomatic language is enhanced by adding syntax extensions to the core language. A single printable character in single quotes is syntactic shorthand for an expression that gives the binary code of the character:

```
'A'  =  (`char (`0 `1 `0 `0  `0 `0 `0 `1))
```

This underlying representation, which would normally be hidden, provides for easy definition of character functions and relations, which are not built-in.

A character string in single quotes within a sequence is equivalent to writing the single characters separately:

```
(… 'abc' …)  =  (… 'a' 'b' 'c' …)
```

A character string in double quotes represents a sequence of those characters:

```
"abc"  =  ('abc')  =  ('a' 'b' 'c')
```

A single or double quote character is repeated when it occurs in a character string enclosed by the same quote character: `"'""" = ('''"')=(''''  '"')`.

A symbol that does not begin with one of the special characters ` % $ ( ) ' " ! is equivalent to an expression consisting of the atom ` and the symbol as a character sequence:

```
abc  =  (` "abc")
```

This is useful for higher-order definitions, decimal number representation, and for defining inequality between symbols, which is not built-in.

Other syntax extensions might be useful, such as in-line text, macros, or indentation-based syntax. Any syntax extension that has a clear mapping to the core language could be considered as an addition to the language. We view the definitions and rules of the core language as fixed and permanent, but its syntactic realization and any syntax extensions are open to refinement and enhancement.

III. Examples

This section gives some examples of axiomatic language and discusses its features. Section A gives examples of higher-order definitions and section B shows the metalanguage capability of the language.

A. Higher-Order Definitions

In axiomatic language, variables can be used for predicate names and for entire predicates. These higher-order constructs can be powerful tools for defining sets and relations. For example, a predicate for a finite set can be defined in a single expression, as follows:

```
(%set %elem)<(finite_set %set ($1 %elem $2)).
    ! used to define finite sets:
(finite_set day
    (Sun Mon Tue Wed Thu Fri Sat)).
```

These axioms generate valid expressions such as (day Tue). Similarly, instead of defining facts in separate axioms, as would be done in Prolog, we can generate them from a single expression, as follows:

```
% < (valid $1 % $2).
  ! specify multiple facts in one expression:
(valid (father Sue Tom)
       (father Bill Tom)
       (father Jane Bill)).
```

From this we get valid expressions such as (father Jane Bill).

The higher-order capability allows valid expressions to be combined into a single expression. The following axioms form lists of expressions that are all valid:

```
(all_valid).
(all_valid % $)< % , (all_valid $).
    ! all expressions in list are valid:
(grandparent %x %z)<
   (all_valid (parent %x %y) (parent %y %z)).
```

This expression represents the conjunction of valid expressions. We can also have a condition that asserts that at least one expression in a list is valid:

```
(one_valid $1 % $2)< % .
  ! at least one expression in list is valid:
(parent %x %y)<
   (one_valid (mother %x %y) (father %x %y)).
```

This represents valid expression disjunction.

Axioms themselves can be represented as expressions. We can represent a single axiom in an expression as follows:

```
%conclu < (axiom %conclu $conds),
          (all_valid $conds).
   ! specify axiom in a single expression:
(axiom (sort %1 %2)
       (permutation %1 %2) (ordered %2)).
```

This is easily extended to represent a set of axioms in a single expression:

```
(axiom $axiom)< (axiom_set $1 ($axiom) $2).
   ! a set of axioms in a single expression:
(axiom_set ((length () 0))  ! sequence length
    ((length (% $) (s %n)) (length ($) %n))).
```

This last axiom generates valid expressions such as (length (a b c) (s (s (s 0)))).

The mapping of a relation or function to lists of arguments is easily defined. First we need a utility that generates sequences of zero-or-more copies of an expression:

```
(zero_or_more % ()).
(zero_or_more % (% $))< (zero_or_more % ($)).
```

We also need a utility that distributes a sequence of elements over the fronts of sequences:

```
(distr () () ()).   ! distr elems over seqs
(distr (%el $els) (($seq) $seqs)
       ((%el $seq) $seqsx))<
   (distr ($els) ($seqs) ($seqsx)).
```

Now we make use of symbol representation to map previously-defined functions and relations to sequences of arguments:

```
((` ($rel '*')) $nulls)<   ! empty arg seqs
  (zero_or_more () ($nulls)).
```

```
((` ($rel '*')) $argseqsx)<  !non-empty seqs
  ((` ($rel '*')) $argseqs),
  ((` ($rel)) $args),      ! relation to map
  (distr ($args) ($argseqs) ($argseqsx)).
```

The expression (` ($rel)) matches the symbol for the relation name and (` ($rel '*')) represents that symbol with an asterisk appended. These axioms generate valid expressions such as (day* (Sat Tue Tue)) and (append* ((a b) ()) ((c) (u v)) ((a b c) (u v))). Note that our mapping definition automatically applies to predicates of any arity. The higher-order property of axiomatic language is essentially the same as that of HiLog. That is, the language has higher-order syntax but the semantics are really first-order.

B. Metalanguage Examples

The higher-order property and string variables along with the absence of commas give axiomatic language its metalanguage property – its ability to imitate other languages. In this section we define the evaluation of nested functions in Lisp format. First we need decimal number representation:

```
(finite_set digit "0123456789").   ! digits
(index %set %elem %index)<  ! index for elems
  (finite_set %set ($1 %elem $2)),
  (length ($1) %index).
     ! -> (index digit '2' (s (s 0)))
(dec_sym (` (%digit)) %value)<
  (index digit %digit %value).
     ! -> (dec_sym 1 (s 0))  -- single digit
(dec_sym (` ($digits %digit)) %valuex)<
  (dec_sym (` ($digits)) %value),
  (index digit %digit %n),
  (length (* * * * *  * * * * *) %10),
  (times %value %10 %10val),
  (plus %10val %n %valuex).  ! multi digits
```

These axioms generate valid expressions, such as `(dec_sym 325 (s (s … (s 0)…)))`, which give the natural number value for a symbol of decimal digits. We use the length function to hide the representation for the natural number 10. Now we define the evaluation of nested expressions:

```
(eval (quote %expr) %expr).     ! identity fn
(eval %dec_sym %value)<         ! decimal num
  (dec_sym %dec_sym %value).
(eval (%fn $args) %result)<     ! eval func
  (eval* ($args) ($results)),   ! eval args
  (%fn $results %result).       ! func result
```

These axioms turn some of the previously-defined predicates into functions which can be applied in a Lisp-like manner:

```
(eval (times (length (append (quote (a b c))
                     (reverse (quote (e d)))))
             (plus 3 17)
      ) %value)
```

When this expression is used as a condition, the variable %value will get instantiated to the natural number representation for 100. Of course, these nested expressions only make sense when formed from predicates that are functions that yield a single result as the last argument.

A contrasting procedural-language example can be found in the original paper [4].

IV. Specification by Interpretation

We want to specify the external behavior of a program using a set of valid expressions. A program that maps an input file to an output file can be specified by an infinite set of symbolic expressions of the form

```
(Program <input> <output>)
```

where <*input*> is a symbolic expression for a possible input file and <*output*> is the corresponding output file. For example, a text file could be represented by a sequence of lines, each of which is a sequence of characters. A program that sorts the lines of a text file could be defined by valid expressions such as the following:

```
(Program ("dog" "horse" "cow")   ! input
         ("cow" "dog" "horse"))  ! output
```

Axioms would generate these valid expressions for all possible input text files.

An interactive program where the user types lines of text and the computer types lines in response could be represented by valid expressions such as

```
(Program <out> <in> <out> <in> …
                   <out> <in> <out>)
```

where <*out*> is a sequence of zero or more output lines typed by the computer and <*in*> is a single input line typed by the user. Each Program expression gives a possible execution history. Valid expressions would be generated for all possible execution histories. This static set of symbolic expressions is interpreted to represent real inputs and outputs over time. This is a completely pure approach to the awkward problem of input/output in declarative languages [7] and avoids Prolog's ugly, non-logical read/write operations. Example programs can be found at the language website [3].

V. Novelty and Related Work

This section discusses some of the more novel aspects of axiomatic language in comparison to Prolog and other languages:

(1) specification by interpretation – We specify the external behavior of programs by interpreting a static set of symbolic expressions. Even languages with "declarative" input/output [9] have special language features, but axiomatic language has no input/output features at all – just interpretation of the generated expressions.

(2) definition vs. computation semantics – Axiomatic language is just a formal system for defining infinite sets of symbolic expressions, which are then interpreted. Prolog semantics, in contrast, are based on a model of computation.

(3) Lisp syntax – Axiomatic language, like some other logic programming languages (MicroProlog [10], Allegro [11]), uses Lisp syntax. This unified representation for code and data supports metaprogramming.

(4) higher-order – Predicate and function names can be arbitrary expressions, including variables, and entire predicates can be represented by variables. This is the same as in HiLog, but with Lisp syntax. The XSB [12] implementation of HiLog,

however, does not allow variables for head predicates in clauses [13]. Higher-order programming can be done in standard Prolog, but requires special features, such as the 'call' predicate. [14]

(5) non-atomic characters – Character representation is not part of the core language, but defined as a syntax extension. There are no built-in character functions, but instead these would be defined in a library.

(6) non-atomic symbols – Non-atomic symbols eliminate the need for built-in decimal numbers, since they can be easily defined through library utilities.

(7) flat sequences – Sequences in axiomatic language are completely "flat" compared with the underlying head/tail "dot" representation of Prolog and Lisp.

(8) string variables – These provide pattern matching and metalanguage support. String variables can yield an infinite set of most-general unifications. For example, ($ a) unifies with (a $) with the assignments of $ = '', 'a', 'a a', ...

(9) metalanguage – The flexible syntax and higher-order capability makes axiomatic language well-suited to meta-programming, language-oriented programming [15], and embedded domain-specific languages [16]. This language extensibility is similar to that of Racket [17], but axiomatic language is smaller.

(10) no built-in arithmetic or other functions – The minimal nature and extensibility of axiomatic language means that basic arithmetic and other functions are provided through a library rather than built-in. But this also means that such functions have explicitly defined semantics and are more amenable to formal proof.

(11) explicit definition of approximate arithmetic – Since there is no built-in floating point arithmetic, approximate arithmetic must also be defined in a library. But this means symbolically defined numerical results would always be identical down to the last bit, regardless of future floating point hardware.

(12) negation – In axiomatic language a form of negation-as-failure could be defined on encoded axioms.

(13) no non-logical operations such as cut – This follows from there being no procedural interpretation in axiomatic language.

(14) no meta-logical operations such as var, setof, findall – These could be defined on encoded axioms.

(15) no assert/retract – A set of axioms is static. Modifying this set must be done "outside" the language.

VI. CONCLUSIONS

A tiny logic programming language intended as a pure specification language has been described. The language defines infinite sets of symbolic expressions which are interpreted to represent the external behavior of programs. The programmer is expected to write specifications without concern about efficiency.

Axiomatic language should provide increased programmer productivity since specifications (such as the earlier sort definition) should be smaller and more readable than the corresponding implementation algorithms. Furthermore, these definitions should be more general and reusable than executable code that is constrained by efficiency. Note that most of the axioms of this paper could be considered reusable definitions suitable for a library. Axiomatic language has fine-grained modularity, which encourages the abstraction of the general parts of a solution from specific problem details and minimizes boilerplate code. The metalanguage capability should enable programmers to define a rich set of specification tools.

The challenge, of course, is the efficient implementation of the programmer's specifications. [18] Higher-order definitions and the metalanguage capability are powerful tools for software specification, but make program transformation essential. The difficult problem of transformation should be helped, however, by the extreme simplicity and purity of the language, such as the absence of non-logical operations, built-in functions, state changes, and input/output. Future work will address the problem of transformation.

REFERENCES

[1] L. Prechelt, "An empirical comparison of seven programming languages," IEEE Computer, vol. 33, no. 10, pp. 23-29, October 2000.

[2] U. Wiger, "Four-fold increase in productivity and quality", Ericsson Telecom AB, 2001.

[3] http://www.axiomaticlanguage.org

[4] W. W. Wilson, "Beyond Prolog: software specification by grammar," ACM SIGPLAN Notices, vol. 17, #9, pp. 34-43, September 1982.

[5] W. Chen, M. Kifer, D. S. Warren, "HiLog: a foundation for higher-order logic programming," in J. of Logic Programming, vol. 15, #3, pp. 187-230, 1993.

[6] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," CACM, vol. 21, #8, pp. 613–641, August 1978.

[7] S. Peyton Jones, "Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell", Engineering Theories of Software Construction, ed. T. Hoare, M. Broy, R. Steinbruggen, IOS Press, pp. 47-96, 2001.

[8] A. Pettorossi, M. Proietti, R. Giugno, "Synthesis and transformation of logic programs using unfold/fold proofs," J. Logic Programming, vol. 41, 1997.

[9] P. Wadler, "How to declare an imperative," ACM Computing Surveys, vol. 29, #3, pp. 240-263, September, 1997.

[10] K. L. Clark, Micro-Prolog: Programming in Logic, Prentice Hall, 1984.

[11] Allegro Prolog, http://www.franz.com/support/documentation/8.2/doc/prolog.html.

[12] The XSB Research Group, http://www.cs.sunysb.edu/~sbprolog/index.html.

[13] D. S. Warren, K. Sagonas, private communication, 2000.

[14] L. Naish, "Higher-order logic programming in Prolog," Proc. Workshop on Multi-Paradigm Logic Programming, pp. 167-176, JICSLP'96, Bonn, 1996.

[15] M. Ward, "Language oriented programming", Software Concepts and Tools, vol. 15, pp. 147-161, 1994.

[16] P. Hudak, "Building domain-specific embedded languages," ACM Computing Surveys, vol. 28, #4es, December 1996.

[17] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, M. Felleisen, "Languages as Libraries", PLDI'11, 2011.

[18] P. Flener, Achievements and Prospects of Program Synthesis, LNAI 2407, pp. 310-346, 2002.