Implementation of Axiomatic Language

Walter W. Wilson[1]

1 **Dept. of Computer Science & Engineering**
The University of Texas at Arlington
Arlington, Texas 76019, USA
wwwilson@acm.org

Abstract

This report summarizes a PhD research effort to implement a type of logic programming language called "axiomatic language". Axiomatic language is intended as a specification language, so its implementation involves the transformation of specifications to efficient algorithms. The language is described and the implementation task is discussed.

1998 ACM Subject Classification D.1.6 Logic Programming, I.2.2 Automatic Programming

Keywords and phrases axiomatic language, specification, program transformation, unfold/fold

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.290

1 Introduction and Problem Description

This research project investigates a type of logic programming language called "axiomatic language" [1,2]. Axiomatic language is intended as a specification language where the user defines the external behavior of a program without giving an algorithm. The language implementation has the task of transforming this input specification into an equivalent efficient algorithm. This research project will attempt to make progress on this difficult problem. A secondary goal will be to make a software engineering case for axiomatic language as a specification language through example applications. Section 2 describes axiomatic language and its attributes and sections 3-6 discuss the transformation problem.

2 Background and Existing Literature

This section defines axiomatic language and discusses its novel aspects in comparison to other languages. Axiomatic language has three goals:

1. a pure specification language - what, not how
2. minimal, but extensible - as small and simple as possible
3. a meta-language - able to imitate and thus subsume other languages

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 function or program – even an interactive program – can be specified by a static infinite set of symbolic expressions that enumerate all possible inputs – or sequences of inputs – along with the corresponding outputs. The language is just a formal system for defining this symbolic expression set.

Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 290–295

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2.1 An Informal Overview

Axiomatic language can be described as pure definite Prolog with Lisp syntax, HiLog [3] higher-order generalization, and "string variables", which match a substring in a sequence. A typical Prolog predicate is represented in axiomatic language as follows,

```
father(bob,X)   ->   (father Bob %x)
```

where (expression) variables start with % and both upper and lowercase letters (as well as many special characters) can be used for symbols.

Natural numbers and their addition can be defined by the following "axioms":

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

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

These axioms generate "valid expressions" such as `(plus (s 0) (s 0) (s (s 0)))`, which is interpreted as the statement "1 + 1 = 2". Comments start after !.

String variables, which start with $, match a string of elements within a sequence. They enable more concise definitions of predicates on lists:

```
(concat ($1) ($2) ($1 $2))       ! list concatenation

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

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

Some example valid expressions are `(concat (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.

2.2 The Core Language

In axiomatic language a finite set of axioms generates a (usually) infinite set of valid expressions. An **expression** is:

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

Atoms are represented syntactically by symbols starting with the backquote ` (so the symbols seen previously are not atoms). A sequence is represented by a string of expressions and string variables separated by blanks and enclosed in parentheses: `(`abc %n)`, `(`M $ ())`.

An **axiom** consists of a **conclusion** expression and zero or more **condition** expressions, represented as follows:

<conclu> < *<cond1>*, . . . , *<condn>*.
<conclu>. ! an unconditional axiom

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 varable can be replaced by a string of expressions and string variables. For example, the axiom,

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

has the instance,

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

by the substitution of `(` %)` for %x, `` `y `` for %y, '`() `v`' for $w, and '' (the null 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.

2.3 Syntax Extensions

The expressiveness of axiomatic language is enhanced by adding certain syntax extensions to the core language. We let a single character in single quotes be syntactic shorthand for an expression that gives the 8-bit character code:

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

A string of characters in single quotes within a sequence is equivalent to writing those single characters separately:

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

A string of characters in double quotes is equivalent to the sequence of those characters:

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

(A 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 that contains the symbol as a character string:

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

2.4 Specification by Enumeration [4]

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 program that sorts the lines of a text file could be represented by valid expressions such as the following:

```
(Program ("dog" "horse" "cow")     ! 3-line input file
         ("cow" "dog" "horse"))    ! sorted output file
```

Axioms would define these valid expressions for all possible input 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.

2.5 Novelty and Existing Work

This section lists some of the novel aspects of axiomatic language:

1. goal of pure specification – Axiomatic language has the ambitious and idealistic goal of completely freeing the user from thinking about implementation and efficiency.
2. specification by enumeration – We specify the external behavior of programs by enumerating all possible inputs/outputs represented as static symbolic expressions. This is a completely pure approach to the awkward problem of i/o in declarative languages [5].
3. 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.
4. minimal language – We see elegance in having minimal size with maximum expressiveness. Axiomatic language shares this goal with minimal Lisp systems.
5. Lisp syntax – Axiomatic language, like some other LP languages (MicroProlog [6], Allegro [7]) uses Lisp syntax instead of Edinburgh syntax. Predicate and function names are moved inside the parentheses and commas are replaced with blanks. Data lists also use this notation, which supports representing code as data.
6. higher-order syntax – 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.
7. non-atomic characters – Axiomatic language separates the definitions and rules of its semantics, which are fixed, from its syntax and syntax extensions such as character sets and their representation, which are likely to evolve. Non-atomic characters make character representations more explicit, but should be hideable with a well-designed library.
8. non-atomic symbols – Non-atomic symbols eliminate the need for built-in decimal numbers, since they can be easily defined through library utilities.
9. string variables – These provide pattern matching and meta-language support.
10. meta-language – Axiomatic language has the goal of being a single language that can provide the user with the features and expressiveness of any other language. Its flexible syntax and higher order capability should make it well-suited to metaprogramming, language-oriented programming [8], and embedded domain-specific languages [9].

11. 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 can be formally treated like regular code.
12. 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 that symbolically defined numerical results would always be identical down to the last bit, regardless of future floating point hardware.
13. no built-in inequality between distinct symbols – Symbol inequality is easily defined from the non-atomic nature of symbols and characters.
14. no built-in negation – Axiomatic language defines recursively enumerable sets but not their complement. One can, however, define negation-as-failure on encoded axioms.
15. no non-logical operations such as cut – This follows from there being no procedural interpretation in axiomatic language.
16. no meta-logical operations such as var, set_of, find_all – These would have to be defined on encoded axioms.
17. no assert/retract - A set of axioms is static. Modifying this set must be done "outside" the language.

3 Goal of this Research

The ultimate goal of this research is the efficient implementation of axiomatic language. This means the automatic transformation of specifications to efficient algorithms. Unlike Prolog, where the user is careful to write clauses that will execute efficiently, we want the axiomatic language user to write specifications without concern about efficiency. A further goal is that this transformation be proven correct — the implementation algorithm is guaranteed to be equivalent to the specification.

One can argue that no finite system can successfully transform all possible user specifications. Just knowing whether or not a specification defines no output is, of course, undecidable. The best we can hope for is a system that can successfully transform the specifications of most "typical" programs.

In addition to their purpose of specification, axioms should be able to define an implementation algorithm, either by executing them in a Prolog-like manner or by modeling, say, a C subset. Thus, this transformation problem can be reduced to transforming one set of axioms to an equivalent set. Unfold/fold transformations [10] can be used here and will provide the guarantee of correctness for the resulting implementation.

4 Current Status and Preliminary Results

My initial work on proof involves proving "valid clauses". We define a "clause" to be the same as an axiom – a conclusion expression and zero or more condition expressions. A clause is "valid" with respect to a set of axioms if no additional valid expressions are generated when the clause is added to the set of axioms. For example, consider the set of axioms consisting of the number and plus axioms along with the following equality axiom:

```
(== % %).            ! identical expressions
```

The following clause,

```
(== %3a %3b)< (plus %1 %2 %3a), (plus %2 %1 %3b).
```

would be valid since no new equality expressions would be generated if this clause were added to the set of axioms. This clause states the commutativity of addition. Unfold/fold rules for proving valid clauses are being worked out and should be a basis for proving equivalent sets of axioms. Proving valid clauses may also be useful for proving assertions about specifications.

The initial work on transformation has involved developing a framework for manipulating axioms as data.

5 Open Issues and Expected Achievements

The automatic transformation of specifications to efficient algorithms is widely considered unsolvable [11]. Other work, however, shows more positive results [12]. The axiomatic language implementation problem has the advantage that the specifications will be completely detailed and the specification language is an extremely simple formal system. Unlike an interactive transformation system [13], we don't expect the user to manually transform his program; instead the transformation system will have to do its job on its own. We acknowledge that the system may not be able to do a good job on a given input specification, in which case an expert will be called on to add new knowledge. We hope that as knowledge continues to be added and generalized, the need for an expert's intervention will decline.

I wish to thank my advisor, Dr. Jeff Lei, for our discussions.

References

1 Wilson, W.W., Beyond Prolog: Software Specification by Grammar, ACM SIGPLAN Notices, Vol. 17, #9, pp. 34–43, Sept. (1982)
2 http://www.axiomaticlanguage.org
3 Chen, W., Kifer, M., Warren, D.S., HiLog: A Foundation for Higher-Order Logic Programming, J. of Logic Programming, Vol. 15, #3, pp. 187–230 (1993)
4 Wilson, W.W., Lei, Y., Specifying Input/Output by Enumeration, http://csl.stanford.edu/~christos/pldi2010.fit/wilson.specio.pdf (2010)
5 Peyton Jones, S., Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell, in Engineering Theories of Software Construction, ed Tony Hoare, Manfred Broy, Ralf Steinbruggen, IOS Press, pp. 47–96 (2001)
6 Clark, K.L., Micro-Prolog: Programming in Logic, Prentice Hall (1984)
7 Allegro Prolog, http://www.franz.com/support/documentation/8.2/doc/prolog.html
8 Ward, M., Language Oriented Programming, Software Concepts and Tools, 15, pp 147–161 (1994)
9 Hudak, P., Building Domain-Specific Embedded Languages, ACM Computing Surveys, Vol. 28, #4es, Dec. (1996)
10 Pettorossi, A., Proietti, M., Giugno, R., Synthesis and Transformation of Logic Programs Using Unfold/Fold Proofs, J. Logic Programming, Vol. 41 (1997)
11 Rich, C., Waters W., Automatic Programming: Myths and Prospects, IEEE Computer, vol. 21, pp. 40–51 (1988)
12 Binoux, G., et al, DESCARTES: An Automatic Programming System for Algorithmically Simple Programs, IWSSD '98 Proceedings of the 9th International Workshop on Software Specification and Design, IEEE (1988)
13 Renault, S., A System for Transforming Logic Programs, RR-97.04, Dept. of Computer Science, University of Rome – Tor Vergata (1997)