Proof, Implementation, and CAD-Application Challenges of Axiomatic Language

Walter W. Wilson
Lockheed Martin
Fort Worth, Texas, USA
wwwilson@acm.org

A type of logic programming called "axiomatic language" is described and some research challenges are discussed: (1) proof in the language, (2) the automatic transformation of specifications to efficient programs, and (3) application of the language to the representation of computer-aided design data.

1 Introduction

This paper describes some research challenges for a type of logic programming called "axiomatic language" [3] [4] [http://www.axiomaticlanguage.org]. Axiomatic language is a pure specification language so its implementation requires automatically transforming specifications to efficient programs – a grand challenge of computer science. Proof is needed to guarantee correctness of this transformation and to prove properties that help validate a specification. Axiomatic language is also proposed as an underlying representation for computer-aided design data. Section 2 defines the language and sections 3-5 discuss the proof, implementation, and CAD-application research challenges.

2 Axiomatic Language

Axiomatic language can be summarized as pure, definite Prolog with Lisp syntax, HiLog-like higher-order generality [1], plus "string variables", which can match any substring in a sequence. The language has the following goals:

1. Pure specification – you tell the computer what to do without telling it how to do it
2. Minimal, but extensible – as small as possible without impacting expressiveness
3. Metalanguage – one can define new language features and paradigms within the language

Axiomatic language is based on the idea that the external behavior of a program – even an interactive program – can be represented by a static, infinite set of symbolic expressions. These expressions enumerate program inputs – or sequences of inputs – along with the corresponding outputs. For an interactive program each expression would represent the inputs/outputs of a particular execution history as seen by an external observer. The set of expressions would enumerate all possible execution histories. The language is just a formal system for defining these infinite sets. [http://csl.stanford.edu/~christos/pldi2010.fit/wilson.specio.pdf]

2.1 The Core Language

This section gives the definitions and rules for axiomatic language – its complete semantics. In axiomatic language a finite set of axioms generates a (usually) infinite set of valid expressions. An **expression** is

Submitted to:
ICLP 2020

an **atom** – a primitive, indivisible element,

an **expression variable**,

or a **sequence** of zero or more expressions and **string variables**.

Syntactically, atoms are represented by symbols that begin with a backquote: `‘abc`, `‘+`. Expression and string variables begin with `%` and `$`, respectively. Sequences have their elements separated by blanks and enclosed in parentheses: `(‘M () (% $1))`.

An **axiom** consists of a **conclusion** expression and zero or more **condition** expressions:

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

Comments start with an exclamation point.

Axioms generate **axiom instances** 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 $w)< (‘B ($ %y %x)), (‘C $w).
```

has an instance

```
(‘A ‘x ‘u ‘v)< (‘B (() ‘x)), (‘C ‘u ‘v).
```

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

Axiom instances generate **valid expressions** by the rule that if all the conditions of an axiom instance are valid expressions, the conclusion 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)`, ...

2.2 Syntax Extensions

The expressiveness of axiomatic language is enhanced with some syntax extensions. A single character in single quotes is equivalent to writing an expression that gives the binary code of the character using bit atoms:

```
’A’  ==  (‘char (‘0 ‘1 ‘0 ‘0  ‘0 ‘0 ‘0 ‘1))
```

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

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

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

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

A symbol that does not begin with `‘ % $ ( ) ’ "` is syntactic shorthand for this expression,

```
ABC  ==  (‘ "ABC")
```

which gives the symbol as a character string and uses the atom represented by just the backquote.

2.3 Examples

Here are axioms for natural numbers in successor notation and their addition:

```
(num 0).                    ! n0: zero is a natural number
(num (s %n))< (num %n).     ! n1: the successor of a number is a number

(plus % 0 %)< (num %).       ! p0: n + 0 = n
(plus %1 (s %2) (s %3))<     ! p1: n1 + (n2+1) = (n3+1)
  (plus %1 %2 %3).           !   if n1 + n2 = n3
```

These axioms generate valid expressions such as `(num (s (s (s 0))))` and `(plus (s (s 0)) (s 0) (s (s (s 0))))`, representing the statements "3 is a natural number" and "2 + 1 = 3", respectively.

3 Proof in Axiomatic Language

Axiomatic language is minimal and pure, which should make it well-suited to proof. A set of axioms generates a set of valid expressions. If an axiom is added to a set of axioms, additional valid expressions may or may not be generated. For example, consider this candidate axiom n2:

```
(num (s (s %n)))< (num %n).     ! n2: 2+n is a number, if n is a number
```

If added to the natural number axioms n0,n1, no new natural number valid expressions are generated.

A **clause** is defined the same as an axiom – a **conclusion** and zero or more **conditions**. (Axioms are just specially designated clauses.) A clause is a **valid clause** with respect to a set of axioms if no new valid expressions are generated if the clause is added to the set of axioms. Unfold/fold proofs can be used to prove that a clause is valid. For example, unfolding axiom n1 with itself gives valid clause n2.

Valid clauses can be used to assert interesting properties. Let us add this "identical expressions" axiom to the axioms for natural numbers and their addition:

```
(== % %).       ! ==: argument expressions are identical
```

The following clause asserts that addition is commutative:

```
(== %12 %21)< (plus %1 %2 %12), (plus %2 %1 %21).
```

If addition is commutative the addition sums `%12` and `%21` will be identical, so no new `==` valid expressions are generated and thus the clause is valid. A valid clause can be considered a "true statement" about a set of axioms – a statement that is "implied" by the set of axioms.

Proof rules and example proofs can be found here: `http://axiomaticlanguage.org/proof.htm`

4 Implementation Challenge

The implementation of axiomatic language requires a system that can "understand" the meaning of a user's input specification and then automatically generate an equivalent efficient program from that meaning. The system would need comprehensive built-in knowledge of programming concepts like arithmetic, sorting, higher-order forms, etc. Once a specification is "understood", the system would generate an equivalent efficient program using pre-stored algorithm knowledge such as binary arithmetic

using machine hardware operations. Proof would be embedded in the transformation knowledge to guarantee the equivalence of the generated efficient program with the input specification.

One can argue that no finite amount of programming knowledge can transform all possible specifications that a user might write. It will always be possible, for example, to come up with an arbitrarily complicated representation for, say, natural numbers and their addition, which the transformation system would not be able to "understand". Our goal is a system with sufficient programming knowledge to automatically transform straightforward specifications for most typical problems. If an input specification is not so straightforward or the problem not so typical, an expert would need to add knowledge to the system so that successful automatic transformation can be achieved. Here are some "baby steps" toward this grand challenge: `http://axiomaticlanguage.org/LOPSTR18_LM_released.pdf`

5 Application to Computer-Aided Design

The metalanguage capability of axiomatic language would make it a good host for an embedded domain-specific language for computer-aided design data, instead of saving it in a CAD vendor's proprietary file format [`http://axiomaticlanguage.org/A_Vision_for_CAD_released`]. A declarative foundation for engineering design would provide accessible mathematics, powerful scripting for design optimization, and would be a good standard for long-term data preservation [`http://axiomaticlanguage.org/LOTAR_Thoughts`]. It may be possible to prove correctness for geometric algorithms that use symbolic approximate arithmetic, which could help guarantee, say, the safety of an airliner.

6 Conclusion

An idealistic, pure specification language has been presented. Unlike conventional logic programming, axiomatic language does not have operational semantics based on resolution. It does not have non-logical operations, built-in functions, state changes, or input/output operations. It also does not have built-in negation (but see "extended axiomatic language" [`http://axiomaticlanguage.org/EAL.html`]).

Axiomatic language should support greater programmer productivity and software correctness. Specifications should be smaller, more readable, more reusable, and more likely to be correct than implementation code. The metalanguage capability of the language would support language-oriented programming [2]. Proof would support the formal verification of high-assurrance software. Use of the language for engineering design would be a billion-dollar application. In summary, axiomatic language is ambitious in its goals, intriguing in its potential, and formidable in its realization, and is a rich source of research challenges.

References

[1] Weidong Chen, Michael Kifer & David S. Warren (1993): *HILOG: A Foundation for Higher-order Logic Programming*. *J. Log. Program.* 15(3), pp. 187–230, doi:10.1016/0743-1066(93)90039-J. Available at `http://dx.doi.org/10.1016/0743-1066(93)90039-J`.

[2] M. P. Ward (1995): *Language Oriented Programming*. *Software—Concepts and Tools* 15, pp. 147–161.

[3] Walter W. Wilson (1982): *Beyond PROLOG: Software Specification by Grammar*. *SIGPLAN Not.* 17(9), pp. 34–43, doi:10.1145/947955.947959. Available at `http://doi.acm.org/10.1145/947955.947959`.

[4] Walter W. Wilson & Yu Lei (2012): *A Tiny Specification Metalanguage*. In: *Proc. of the 24th Intl. Conf. on Software Engineering & Knowledge Engineering (SEKE'2012), Redwood City, CA, July, 2012*, pp. 486–490.