Proof in Axiomatic Language

Walter W. Wilson[0009−0009−2169−8763]

(retired), Fort Worth TX, USA
wwwilson@acm.org

Abstract. This paper describes a system of proof for a type of logic programming called axiomatic language [www.axiomaticlanguage.org]. The language is defined and its relation to traditional logic programming is discussed. Axiomatic language is intended for specification, and proof would be used to prove assertions about the specifications. The proof system is not based on logic but on the inference rules of axiomatic language. The integration between proof and the specification language should help support formal verification. A proof checker has been implemented in a Prolog-like restricted form of axiomatic language.

Keywords: proof · specification · formal verification.

1 Introduction

Formal verification can benefit from a tight integration with the programming language being used. This paper presents a proof system for a type of logic programming called "axiomatic language" [5, 6]. Axiomatic language is intended as a pure specification language, so its implementation requires automatically synthesizing efficient programs from user specifications – a grand challenge of computer science. If this ambitious goal can be achieved, the higher level of specifications relative to implementation code should yield a programming productivity benefit [3].

Axiomatic language is minimal and pure, which makes it well-suited to proof. The proof system would be used to prove assertions about a user's specification to validate it. This would be more powerful than just testing. (A test case checks a single input. A proven assertion helps check a class of inputs.) Proof would also be incorporated into an eventual axiomatic language implementation to guarantee equivalence between the user's specification and the generated efficient program.

This proof system is not based on logic but instead on the inference rules of axiomatic language itself. This should make proofs more understandable and modifiable by future axiomatic language programmers and should help support the formal validation and verification of axiomatic language software. (Here, validation means showing that the axiomatic language specification represents the user's intent, and verification means the generated efficient program is proven equivalent.) A proof checker has been implemented in a restricted form of axiomatic language that executes like pure Prolog.

Section 2 reviews axiomatic language. Proof in axiomatic language is defined in section 3. Section 4 gives some final comments.

2 Axiomatic Language

This section defines axiomatic language. Axiomatic language has the following goals: (1) pure specification, (2) extreme minimality, (3) metalanguage extensibility, and (4) beauty. Section 2.1 gives the main idea of the language. Section 2.2 gives the definitions and rules of the core language, and section 2.3 gives syntax extensions. Section 2.4 has some examples and some summary comments.

2.1 Main Idea – Specification by Enumeration

The main idea of axiomatic language is that the external behavior of a program – even an interactive program – can be represented by a static, infinite set of symbolic expressions. Each expression encodes the program input – or sequence of inputs – along with the corresponding output for an execution of the program, as seen by an external observer. The set of such expressions would enumerate all possible executions, one for each possible program input. Our claim is that this infinite set of symbolic expressions idealistically specifies the external behavior of a program, without having anything to say about the internal processing. Note that in axiomatic language the expressions have no inherent meaning; they are interpreted by the human user and the implementation system to represent bits, characters, lines of text, etc. of the external real-world environment.

For a program that reads an input text file and writes an output text file, each 'Program' expression can represent the input and output files with a sequence of symbolic expressions representing lines, each a sequence of symbolic expressions representing characters. For a program that sorts the lines of an input file, the infinite set of expressions would enumerate each possible input text file along with the sorted output file:

```
(Program () ())        - empty input file -> empty sorted output
      ...
(Program ("C" "A") ("A" "C"))    - two single-char lines sorted
      ...
(Program ("cat" "dog" "ant") ("ant" "cat" "dog"))  - 3-line file
      ...
```

These symbolic expressions show character strings, but, as section 2.3 explains, these character strings are just a syntax extension for underlying abstract symbolic expressions, which are then interpreted as character strings. The infinite set of these Program expressions specifies this sorting program without specifying a sorting algorithm.

For an interactive program that, say, reads and writes lines of text in a text window, a typical program might write blocks of zero or more output lines interspersed with single input lines typed by the user. Each Program expression would contain the sequence of inputs and outputs for an execution history. The infinite set of Program expressions would give all possible execution histories based on all possible inputs the user might type at any point.

Consider a program that accepts arbitrary input strings and checks whether any parentheses present are balanced, giving an error message if not. An empty input string halts the program. An example Program expression (with annotations) might be as follows:

```
(Program
 ("Enter character strings with balanced parentheses."
  "An empty string ends the program.")   - initial output lines
  "((xyz)a b)()"              - correct user input line
 ()                             (no output lines)
  "(W) (W))12)"               - incorrect input line
 ("       ^ unmatched right parenthesis")   - error msg output
  "((( )"
 ("     ^ unmatched left parentheses")
  ""                          - empty input line ends program
 ("End-of-program.")        - final output line
)
```

This symbolic expression represents a possible execution of this interactive program as seen by an external observer. An infinite set of these expressions, for all possible execution histories, can be considered a static specification of this interactive program.

For real interactive graphics it shouldn't be difficult to come up with a convention for symbolically representing screen pixels and mouse movements. The implementation system would need to "understand" this convention in order to synthesize a program with actual graphics operations.

The emphasis here is that these expressions are just abstract hierarchical symbolic expressions with no built-in meaning. They are just interpreted by the human user and a future implementation system to represent real-world entities. Note that in order to define Program expressions one will likely need to define predicates for supporting utilities like arithmetic, but these would also be symbolic expressions interpreted by the human user.

Using an infinite set of abstract symbolic expressions to specify the external behavior of a program provides a clean, idealistic separation between specification and implementation. It is a nice solution to the "awkward" problem of input/output in declarative languages. All that is required of the specification language is that it be a formal system for defining these infinite sets – and that is what axiomatic language is. Unlike the non-logical input/output predicates of Prolog, axiomatic language is completely pure.

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 atomic symbol,

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 and run to the end of the line.

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. By default, the conclusion of an "unconditional" axiom instance is a valid expression. For example, the two axioms

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

generate valid expressions `` (`a `b) ``, `` ((`a) `b `b) ``, `` (((`a)) `b `b `b `b) ``, ...

2.3 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 in that sequence:

```
(... '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 an expression that gives the symbol as a character string,

```
ABC  ==  (` "ABC")
```

and uses the atom represented by just the backquote.

2.4 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).       ! ns: successor of nat num is nat num

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

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

String variables enable more concise definitions of list predicates:

```
(member % ($1 % $2)).          ! expr is member of a sequence
(concat ($1) ($2) ($1 $2)).    ! concatenation of two sequences
```

In summary, axiomatic language can be roughly described as pure, definite Prolog with Lisp syntax, string variables, and HiLog-like higher-order generalization [1]. Lisp syntax provides a single uniform representation for data lists, terms, functions, and predicates. It provides syntactic flexibility for new language features, like infix operators, and natural support for higher-order forms, where code is treated as data. String variables complement expression variables. An expression variable represents a single expression; a string variable represents a string of zero or more expressions and string variables within a sequence.

Note that axiomatic language does not include built-in true/false values. However, this concept is easily defined and one can then define assorted Boolean functions and predicates. Axiomatic language is more like a type of formal language, except that instead of generating words as flat strings from a finite alphabet, axiomatic language generates recursively-enumerable hierarchical expressions formed from an infinite set of atom symbols and variables.

Axiomatic language also differs from Prolog in its goal of minimality and purity – no input/output operations, no state changes, no non-logical operations, no built-in predicates of any kind. For example, there is no built-in function for inequality between distinct atoms, but it is easy to define inequality between distinct syntax-extension symbols. The axiomatic language emphasis on pure specification means there is no execution model. Specification by enumeration defines program external behavior without defining internal computation steps. The only "semantics" for axiomatic language is the inference rules for generating axiom instances from an axiom and for generating valid expressions from axiom instances.

3 Proof in Axiomatic Language

This section proposes a system of proof for axiomatic language. Consider the following candidate axiom:

```
  (num (s (s %)))< (num %).       ! nss: 2+n is num if n is num
```

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

3.1 Valid Clauses

A **clause** is defined the same as an axiom – a conclusion and zero or more conditions. (Axioms are just specially designated clauses.) Assigning values to the clause variables gives a **clause instance**. If all the conditions of a clause instance are valid expressions for a set of axioms, then the conclusion is a **generated expression**. A clause is a **valid clause** with respect to a set of axioms if all its generated expressions are valid expressions for those axioms. Thus, adding a valid clause to a set of axioms does not add to the set of valid expressions. Clause nss above is thus a valid clause with respect to the natural number axioms. One can say that a valid clause is "implied" by the set of axioms. It can be considered a "true statement" about the axioms.

3.2 Rules for Proving Valid Clauses

Given a set of axioms, the following rules can be used to derive valid clauses:

R1 – An axiom is a valid clause.
R2 – An instance of a valid clause is a valid clause.
R3 – A permutation of valid clause conditions gives a valid clause.
R4 – Adding a condition to a valid clause gives a valid clause.
R5 – For any set of axioms we have this tautological valid clause:

```
    % < % .
```

R6 – For every valid expression 've', we have this valid clause:

```
    ve.
```

R7 – If no instance of expression ‘nve’ is a valid expression, its occurrence as a condition gives a valid clause (because no expressions can be generated):

```
% < ...,nve,....
```

R8 – Consider valid clauses A and B,

```
A: a0 < a1,...,an.
B: b0 < b1,...,bm.
```

where a0,b0 are conclusions and a1..an,b1..bm are conditions. If some condition ak is identical to conclusion b0, then we can construct valid clause C from clause A where condition ak is replaced by conditions b1..bm of clause B:

```
C: a0 < a1,..,ak-1,b1,..,bm,ak+1,..,an.
```

We call this an unfold of valid clause A condition k with valid clause B. Using the above rules we can now show that clause nss is valid:

```
a: (num (s %))< (num %).           R1 - axiom ns
b: (num (s (s %)))< (num (s %)).   R2 - instance of a
nss: (num (s (s %)))< (num %).     R8 - unfold b with a
```

R9 – Induction Rule. To show clause C valid,

```
C: c0 < c1,...,cn.
```

we need to show that all its generated expressions are valid. For each generated expression, a condition ci was matched with a valid expression, generated by some axiom. We can get all the ways that ci can be matched by unfolding it against the set of axioms. Each successful unfold of ci with an axiom Aj = a0<a1,..,an produces a clause C_Aj:

```
C_Aj: c0' < c1',..,ci-1',a1',..,an',ci+1',..,cn'.
```

The primes indicate that the substitution of the most-general unification between ci and a0 has been applied to the result clause. The set of C_Aj clauses covers all the ways that the original clause C can generate expressions. If all the C_Aj clauses can be proved valid, then C is valid. The induction hypothesis means we can unfold a valid clause condition with clause C in order to prove that a result clause C_Aj is valid.

Finally, note that for axiom sets A and B, if all the axioms of B can be shown to be valid clauses of axiom set A and all the axioms of A can be shown to be valid clauses of set B, then sets A and B generate the same set of valid expressions and are referred to as “equivalent axiom sets”. Every clause that can be proven valid with respect to one axiom set is valid with respect to the other. Typically, we will be proving that one subset of axioms is equivalent to another subset, given the rest of the axioms in the set. An induction proof may unfold against either subset.

3.3 Some Example Proofs

The natural number addition axioms p0,ps above do recursion on the first argument. Alternative axioms, pa0,pas, that do recursion on the second argument, can be proved valid with respect to p0,ps,n0,ns as follows:

```
ps0: (plus (s %) 0 (s %)< (plus % 0 %).   - instance of ps

pa0: (plus % 0 %)< (num %).          - induction on cond 1
  pa0_n0: (plus 0 0 0).            - = p0 unfolded with n0
  pa0_ns: (plus (s %) 0 (s %))< (num %).  =unf ps0 w ind hyp pa0

pas: (plus %1 (s %2) (s %3))< (plus %1 %1 %3).  -induc on cond 1
  pas_p0: (plus 0 (s %) (s %))< (num %).   - = unfold p0 with ns
  pas_ps: (plus (s %1) (s %2) (s (s %3)))< (plus %1 %2 %3).
                              - = unfold of ps with induc hypoth pas
```

Similar proofs can show that axioms p0,ps are valid clauses with respect to axiom set pa0,pas,n0,ns, thus showing that the two subsets are equivalent.

A proof checker has been implemented in a restricted form of axiomatic language where string variables can only occur at the ends of sequences. This enables definition of a Prolog-like query solver which can check small proofs, such as the commutativity of natural number addition. [4]

4 Final Comments

Axiomatic language proof is inspired by the logic programming transformations of Alberto Pettorossi, Maurizio Proietti, and colleagues (e.g., [2]). Their approach involves provably-correct incremental modifications to a set of program clauses to produce an equivalent program that is more efficient. The approach here is to prove clauses valid from a fixed set of axioms.

Future work will include proving the correctness of the proof inference rules. More example proofs will be defined, possibly with new proof rules, such as proving that a clause is not valid.

References

1. Chen, W., Kifer, M., Warren, D.S.: Hilog: A foundation for higher-order logic programming. The Journal of Logic Programming **15**(3), 187–230 (1993)
2. Pettorossi, A., Proietti, M.: Transformation of logic programs: Foundations and techniques. The Journal of Logic Programming **19**, 261–320 (1994)
3. Wilson, W.: Phonecode, http://www.axiomaticlanguage.org/phcode/paper.pdf,
4. Wilson, W.: Proof checker, http://www.axiomaticlanguage.org/proof/checker.htm
5. Wilson, W.: Beyond prolog: Software specification by grammar. SIGPLAN Not. **17**(9), 34–43 (Sep 1982). https://doi.org/10.1145/947955.947959
6. Wilson, W., Lei, Y.: A tiny specification metalanguage. In: Proc. of 24th Intl. Conf. on Software Engr. & Knowledge Engr. (SEKE'2012). pp. 486–490 (2012), http://ksiresearchorg.ipage.com/seke/Proceedings/seke/SEKE2012_Proceedings.pdf