Baby Steps Toward an Implementation of Axiomatic Language

Extended Abstract

Walter W. Wilson

Lockheed Martin, P.O. Box 748, Fort Worth TX 76101, USA
wwwilson@acm.org

Abstract. This paper describes an initial crude attempt at implementing a type of logic programming called "axiomatic language" [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. The proposed approach is for a transformation system to have comprehensive knowledge of programming concepts such as arithmetic, utility functions and their composition, data aggregations, specification patterns, etc. The system would then "understand" the meaning of an input specification against this knowledge. From this understanding the system would generate an equivalent efficient program using built-in algorithm knowledge. An initial transformation infrastructure has been developed with the built-in knowledge to handle a few tiny examples.

Keywords: specification language, program transformation, program synthesis, axiomatic language.

1 Introduction

This paper describes an initial attempt at implementing a type of logic programming called “axiomatic language” [AL, Wilson 12, 00, 82]. Axiomatic language is a pure specification language so its implementation requires automatically transforming specifications to efficient programs – a grand challenge of computer science. An approach to do this is proposed and a primitive implementation is described that can handle some tiny examples.

Section 2 reviews axiomatic language. Section 3 describes a query system for mapping inputs to outputs. In section 4 the framework of the transformation system is described and examples are shown. Conclusions are found in section 5.

2 Axiomatic Language

This section reviews axiomatic language. Axiomatic language has the following goals:

1. Pure specification – you tell the computer what to do, not how to do it.
2. Minimal, but extensible – as small as possible without sacrificing expressiveness.
3. Metalanguage – able to define new language features within itself.

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 all possible inputs – or sequences of inputs – along with the corresponding outputs. The language is just a formal system for defining these infinite sets.

2.1 The Core Language

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

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

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

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 the 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 simple rule that if all the conditions of an axiom instance are valid expressions, then the conclusion is a valid expression. By default, the conclusion of an unconditional axiom instance is immediately a valid expression. For example, the following two axioms,

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

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

2.2 Syntax Extensions

The expressiveness of axiomatic language is enhanced by defining 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')
```

Finally, a symbol that does not begin with one of the special characters seen so far 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 character.

2.3 Examples

As an example of axiomatic language, here are the familiar definitions of natural numbers and their addition:

```
(num 0).                   ! zero is a number
(num (s %n))< (num %n).  ! if n is num, so is successor
```

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

These axioms generate valid expressions such as (num (s (s (s 0)))) and (plus (s 0) (s 0) (s (s 0))), which can be interpreted to mean "3 is a natural number" and "1 + 1 = 2", respectively.

Axiomatic language string variables can be considered a generalization of Prolog list tail variables: `[X, Y | W]` -> `(%x %y $w)`, but string variables can occur anywhere in a sequence. They enable more concise definitions of some list predicates:

```
(member % ($1 % $2)).    ! element is member of sequence
(concat ($1) ($2) ($1 $2)).  !concatenation of two seqs
(`append1 ($) % ($ %)).    ! append 1 elem to end of seq
(`reverse () ()).            ! reverse a sequence
(`reverse (% $) ($rev %))< (`reverse ($) ($rev)).
```

These axioms generate valid expressions such as (member b (a b c d)) and (`reverse (x y z) (z y x)). (We use atoms for the append1 and reverse predicate names for conciseness in the examples below.)

2.4 GAL – SAL – PAL

String variables in axiomatic language have the property that their unification can have multiple most-general unifiers – even an infinite number. For example, the unification of (a $) and ($ a) is obtained with any string of a's assigned to $. (Exercise: What are the most-general unifiers of ($1 $) and ($ $2)?)

For implementation purposes it is useful to consider restricted forms of axiomatic language. We call the unrestricted version defined above "general axiomatic language" (GAL). A version where string variables are only allowed at the ends of sequences is called "simplified axiomatic language" (SAL). An even simpler version called "primitive axiomatic language" (PAL) has no string variables and sequences always have length two. Expressions for both SAL and PAL have unique most-general unifiers.

3 Queries on a Set of Axioms

Axiomatic language is just a formal system for defining recursively enumerable sets of hierarchical symbolic expressions. To produce outputs from this language we define "queries" on the set of axioms and their valid expressions. A query is like an axiom – a conclusion expression and zero or more condition expressions. If all the conditions of a query instance are valid expressions, the conclusion of that instance is generated as an "output expression" or a "solution" to the query. Given the above axioms, the query

```
  % < (`append1 (a b c) d %).
```

generates the solution expression (a b c d). A query can produce multiple most-general solutions, even an infinite number. For example, the query

```
  % < (`reverse % %).
```

enumerates the set of palindromes:

```
  ()
  (%0)
  (%0 %0)
  (%0 %1 %0)
  (%0 %1 %1 %0)
   …
```

3.1 An SAL Query System

A simple query system for SAL axioms has been implemented. This top-down SLD solver selects query conditions (goals) in left-to-right order and resolves them against axioms (clauses) in axiom order. But, of course, this implementation often suffers from inefficiency and non-termination when given high-level specification axioms such as higher-order forms and metalanguage examples. Thus it does not qualify as an implementation of axiomatic language. Also, it only implements SAL instead of GAL to avoid the complexities of string variable unification. But this implementation can be usable for small examples that are carefully written.

3.2 Query Formats

In order to define programs that map inputs to outputs, we use "query formats". A query format is a query where some variables are “Inputs”, with an uppercase I after the % or $. (Non-Input variables have an underscore after % or $.) These Input variables will be replaced by ground expressions to form a query to be executed. (Replacement order matches variable symbol order.) A query format thus defines a program that maps the Input expressions to the query solution output expressions. For example, two query formats and some of their input/output mappings are as follows:

```
  %_result < (`append1 %I0list %I1x %_result).
    (), x  =>  (x)
    (a b c), d  =>  (a b c d)
```

```
`yes < (`reverse %Ilist %Ilist).
  (a b a)  =>  `yes
  (a b)  =>  <no output>
```

The second query format tests whether a list is a palindrome and writes out the atom `yes if so, or nothing if not.

4 A Transformation System

4.1 The Problem

Our problem is to automatically transform a user's GAL specification axioms and query format into an equivalent efficient program. We want this transformation process to be completely automatic since, otherwise, it negates the benefits of writing specifications. We would also like a guarantee of equivalence between the generated efficient program and the input specification, but providing this guarantee with proof is left to future work.

For now we will let an SAL axiom set and query format be our target "efficient" program, which will run on the current SAL interpreter. A future processing step will transform this SAL program into a procedural program in, say, a C subset, which will provide a significant constant-factor speed-up. Keeping the near-term implementation target as axioms will provide a better framework for future proof.

Note that unlike some program synthesis, we do not have to deal with natural language requirements. Nor do we derive a specification from input/output examples [Gulwani et al 17]. For this problem we have a specification that is complete and completely formal.

This problem has been addressed by the logic programming transformation work of Pettorossi, Proietti, and colleagues [MAP]. They start with a specification-like logic program and then apply incremental transformation steps to produce a final, equivalent logic program that can execute with greater efficiency. Each step has a proven guarantee of semantic equivalence. But their process is not completely automatic since some interactive guidance may be needed.

4.2 The Approach

The approach of this paper is to first "understand" the user’s specification and then generate an equivalent efficient program to solve it. We need to understand a specification in terms of the high-level concepts that a human programmer would understand it. We decompose this understanding process into two phases. First we understand the set of valid expressions defined by the set of GAL axioms. Then we understand the function defined by a query format against those valid expressions. Once the specification is understood, the final step is to use built-in algorithm knowledge to generate the executable SAL axioms and query format.

A set of axioms defines a recursively enumerable set of abstract symbolic expressions without any inherent meaning. But to the human that wrote them, those expressions represent "concepts" like arithmetic, list predicates, sorting, etc. We want to "recognize" these concepts when valid expressions represent them. We want to extract the high-level meaning of the set of valid expressions and then encode this understanding into a "Valid Expression Definition" (VED) expression that represents this high-level meaning along with its syntactic representation. Given that axiomatic language encourages the use of powerful higher-order forms and embedded language features for specification expressiveness, we must be able to recognize and understand sophisticated specification patterns as well.

Once we have a high-level description of the set of valid expressions embodied in a VED expression, a query format defines a function that maps input expressions to the set of query output expressions. This function also needs to be recognized and understood in terms of high-level programming concepts. This understanding is encoded as a "Function Definition" FD expression.

To completely understand the meaning of a typical axiomatic language specification will require a transformation system with comprehensive human programmer knowledge. One can argue that no finite amount of knowledge can successfully transform all possible specifications that a user might write. There will always be new concepts that the knowledge base doesn't cover or one might specify a familiar concept in an arbitrarily complicated way. When that happens an expert would need to add transformation knowledge to the system to enable it to handle the new specification. The goal will be to have sufficient built-in knowledge to automatically transform straightforward specifications for most typical programs.

4.3 A Working Example

This section gives some results computed by an initial transformation system written in SAL. Given the GAL axioms for the `append1 and `reverse predicates in section 2.3, the first step is to encode these axioms as a ground data expression suitable for processing. An encoded axiom set is just a sequence of encoded axioms, each an encoded conclusion expression followed by the encoded conditions. An encoded sequence is just a sequence of encoded elements. We let encoded expression and string variables be represented by symbols beginning with # and *, respectively. It is also helpful to have atoms be represented by non-atom symbols beginning with . (period). Our `append1 and `reverse axiom set is thus encoded as follows:

```
(((.append1 (*) # (* #)))
 ((.reverse () ()))
 ((.reverse (# *) (*rev #)) (.reverse (*) (*rev)))
)
```

The transformation system “recognizes” the pattern of these definitions and thus “understands” the predicate concepts they represent. A Valid Expression Definition expression is then formed that represents the meaning of the generated valid expressions. For this axiom set we have the following VED expression that defines the append1 and reverse predicate valid expressions:

```
((Append1 .append1 seq (l x lx))
 (Reverse .reverse seq))
```

Each predicate expression gives the predicate concept, the encoded predicate name, and some arguments. The “seq” symbol says that lists are represented as ordinary sequences, the “(l x lx)” expression gives the argument order for the append1 predicate (list, expression, list with expression appended). These argument symbols represent built-in knowledge concepts that the transformation system “knows about”. One can view a VED expression as a terse form of documentation for the human reader in addition to being a formal definition of the set of valid expressions.

Given the set of valid expressions defined by the VED expression, a query format clause defines a function between input expressions and a set of output expressions. A Function Definition expression is produced that represents this function using built-in concepts. For the example query formats of section 3.2 we have the following FD expressions:

```
(Append1 seq (l x) #_lx)
(Palindrome_test seq .yes)
```

The Append1 function has implied ordered Input variable names of `%I0l` and `%I1x`, given argument (l x), or `%I0x` and `%I1l`, given argument (x l). The output variable name is `%_lx`. The last argument of the FD expressions is the encoded output expression which includes the output variables and, optionally, the Input variables.

Our final transformation step is to generate an executable SAL axiom set and query format from the above Function Definition expressions using built-in algorithm knowledge. From the Append1 FD expression, we get the following SAL axiom set and query format:

```
(`app1 () %x (%x)).
(`app1 (% $) %x (% $')) < (`app1 ($) %x ($')).
qf:  %_lx < (`app1 %I0l %I1x %_lx).
```

Note that having different predicate names for the SAL axioms is fine, so long as the query format outputs are identical. For the Palindrome_test FD expression we have:

```
(`rev %seq %rev) < (`rev_acc %seq () %rev).
(`rev_acc () %rev %rev).
(`rev_acc (% $seqsuf) ($revpre) %rev) <
  (`rev_acc ($seqsuf) (% $revpre) %rev).
qf:  `yes < (`rev %Il %Il).
```

Note that the “reverse_accumulator” predicate enables computation of the reverse function in O(n) time instead of O(n^2) time for naïve reverse.

5 Conclusions

Axiomatic language is proposed as a tool with potential software engineering benefits. Specifications should be smaller, more readable, more reusable, and more correct than algorithmic code. The small size and purity of the language should make it well-suited to proof.

But the implementation of axiomatic language requires encoding much of human progamming knowledge into a transformation system. A crude, initial attempt has been presented that can handle a few tiny examples. Future work will grow this system to determine if this approach can possibly scale to real-world-sized problems.

References

[AL] Axiomatic Language homepage, http://axiomaticlanguage.org/.

[Gulwani et al 17] Gulwani, S., Polozov, O., Singh, R., Program Synthesis. In: Foundations and Trends in Programming Languages, 4(1-2), 1-119, http://dx.doi.org/10.1561/2500000010 (2017).

[MAP] MAP Transformation System, http://www.iasi.cnr.it/~proietti/system.html.

[Wilson 12] Wilson, W., Lei, Y., A tiny specification metalanguage, 24th Intl. Conf. on Software Engineering and Knowledge Engineering (SEKE), pp. 486-490 (2012).

[Wilson 00] Wilson, W., A minimal specification language, LOPSTR Pre-Proceedings (2000).

[Wilson 82] Wilson, W., Beyond Prolog: software specification by grammar, ACM SIGPLAN Notices, 17(9), 34-43, Sept. (1982).