A More General Specification Language

Walter W. Wilson
Lockheed Martin

Abstract: This position paper proposes a type of logic programming called “axiomatic language” as a more general specification language for constraint programming problems. Axiomatic language is defined and then used for the SEND+MORE=MONEY problem. Its extensibility eliminates the need for the usual built-in features of CP and CLP languages. But its idealistic emphasis on pure specification makes its implementation a greater challenge.

1. Introduction

The Holy Grail of programming can be described as pure specification – you tell the computer *what* to do, not *how* to do it. But what specification language should we use? This position paper proposes “axiomatic language” [http://www.axiomaticlanguage.org/, Wilson & Lei 12, Wilson 82] as a more general specification language than constraint programming. Axiomatic language has the following goals:

(1) pure specification – what, not how. Inputs/outputs are defined without defining the internal processing.

(2) minimal, but extensible – as small as possible without sacrificing expressiveness. Nothing is built-in that can be defined.

(3) metalanguage – able to define other language features within itself. The intent is to be able to subsume other languages, including constraint programming.

In addition, we have the goal of beauty and elegance for the language.

Section 2 defines axiomatic language. A solution to the canonical contraint programming problem SEND+MORE=MONEY is given in section 3. Section 4 gives conclusions.

2. Axiomatic Language

Axiomatic language is based on the idea that the external behavior of a function or program ─ even an interactive program – can be represented by a static infinite set of symbolic expressions that enumerate 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** – a primitive indivisible element,
an **expression variable**,
or a **sequence** of zero or more expressions and **string variables**.

Syntactically, atoms, expression variables, and string variables are represented by symbols beginning with `, %, $, respectively. A sequence has its elements separated by blanks and enclosed in parentheses.

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 with an arbitrary expression, the same value replacing the same variable throughout the axiom. Similarly, a string variable can be replaced by a string of zero or more expressions and string variables. For example, the axiom

```
(`A %x $)< (`B %x %y), (`C $).
```

has an instance

```
(`A `x `u %)< (`B `x ()), (`C `u %).
```

by the substitution of `x for %x, () for %y, and the string `u % 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 a valid expression.) For example, the 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 the addition of 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 those characters separately within 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 `` ` % $ ' " ( ) `` is syntactic shorthand for the following expression that gives the symbol as a character string:

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

Here `` ` `` is the atom represented by just the backquote. This convention allows us to define decimal number representation, which is not built-in.

2.3. Example – Natural Number Arithmetic

Symbolic natural numbers in successor notation can be defined as follows:

```
  (num (0)).                    ! zero is a natural number
  (num (s $n))< (num ($n)).     ! successor of a number is a number
```

These axioms generate valid expressions such as (num (s s s 0)), which represents the statement “3 is a natural number”.

Here are axioms for the addition and multiplication of these natural numbers:

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

  (times %n (0) (0))< (num %n).   ! n * 0 = 0
  (times %1 (s $2) %4)<           ! n1 * n2+1 = n4 if
    (times %1 ($2) %3),           !   n1 * n2 = n3 and
    (plus %1 %3 %4).              !   n1 + n3 = n4
```

These axioms generate valid expressions such as (plus (s 0) (s s 0) (s s s 0)) meaning “1 + 2 = 3” and (times (s s 0) (s 0) (s s 0)) meaning “2 * 1 = 2”.

3. SEND+MORE=MONEY

This section gives axioms that solve the constraint problem

```
     S E N D
   + M O R E
   ---------
   M O N E Y
```

where each letter represents a different digit and letters S and M are not zero. The problem is to find an assignment of digits to letters that makes the addition operation true. The following "top-level" axiom generates valid expressions for all solutions to this problem:

```
  (solution: $eqn)<
    (== ($eqn) (%S %E %N %D + %M %O %R %E = %M %O %N %E %Y)),
    (equation: $eqn),                        ! equation to solve
    (all digit (%S %E %N %D %M %O %R %Y)),  ! letters are digits
    (different (%S %E %N %D %M %O %R %Y)),  ! digits are distinct
    (/= %S 0), (/= %M 0).                    ! S and M are not 0
```

This axiom combined with the axioms below generate the unique solution valid expression (solution: 9 5 6 7 + 1 0 8 5 = 1 0 6 5 2).

Let us first define the set of decimal digit symbols and their numeric values:

```
  (digit (` (%dc)) %n)<          ! single-digit symbols are
    (digit_char %dc %n).         ! defined from character digits

  (finite_set digit_char "0123456789").  ! set of digit characters

  (%set %elem %n)<     ! finite set element and its ordinal value
    (finite_set %set ($1 %elem $2)),
    (length ($1) %n).

  (length () (0)).            ! length of a sequence
  (length (% $) (s $n))<
    (length ($) ($n)).
```

These axioms generate valid expressions such as (digit_sym 3 (s s s 0)).

Now we will give axioms for the equations to be solved. For this problem the grammar below represents numeric constants as a string of single-digit symbols:

```
! eval - evaluate a natural number arithmetic expression
! (eval ( <E> ) <value>)
!   E = E + T | T       -- Expression
!   T = T * P | P       -- Term
!   P = ( E ) | C       -- Primary
!   C = string of >0 decimal digit symbols  -- Constant

  (eval ($E) %val)<
    (eval_E ($E) %val).
```

```
! eval_E - evaluate an Expression:  E = E + T | T

  (eval_E ($E + $T) %valE+T)<
    (eval_E ($E) %valE),
    (eval_T ($T) %valT),
    (plus %valE %valT %valE+T).
  (eval_E ($T) %val)<
    (eval_T ($T) %val).

! eval_T - evaluate a Term:  T = T * P | P

  (eval_T ($T * $P) %valT*P)<
    (eval_T ($T) %valT),
    (eval_P ($P) %valP),
    (times %valT %valP %valT*P).
  (eval_T ($P) %val)<
    (eval_P ($P) %val).

! eval_P - evaluate a Primary:  P = (E) | C

  (eval_P (($E)) %val)<           ! P = ( E )
    (eval_E ($E) %val).
  (eval_P ($C) %val)<             ! P = C
    (eval_C ($C) %val).

! eval_C - evaluate Constant:   C = string of >0 single-digit symbols

  (eval_C (%d) %n)<         ! single digit
    (digit %d %n).
  (eval_C ($d %d) %n')<     ! multiple digits
    (eval_C ($d) %n),
    (digit %d %k),
    (times %n (s s s s s  s s s s s  0) %10n),
    (plus %10n %k %n').

! equation: - equality between two natural number expressions

  (equation: $E1 = $E2)<
    (eval ($E1) %val),
    (eval ($E2) %val).        ! both sides have the same value
```

These axioms generate valid expressions such as (equation: 1 2 * (1 + 2) = 3 2 + 4).

The following relation is used to assert that two expressions are identical:

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

A higher-order definition can be used to assert that all elements of a sequence are members of a set:

```
! all - a sequence with all its members from a set
! (all <setname> (..elems..))

  (all %set ()).
  (all %set (%elem $elems))<
    (%set %elem $),
    (all %set ($elems)).
```

Finally, we need to define inequality between expressions distinguishable by bit atoms and define a predicate that asserts that all members of a sequence are distinct:

```
! /= - inequality between expressions involving bit atoms

  (/= `0 `1).
  (/= `0 ($)).
  (/= `1 ($)).
  (/= () (% $)).
  (/= %1 %2)< (/= %2 %1).
  (/= (%1 $1) (%2 $2))< (/= %1 %2).
  (/= (%1 $1) (%2 $2))< (/= ($1) ($2)).
  ! -- inequality between distinct chars, char strings, symbols, etc.

! not_in - element is not in a sequence (given bit-inequality)

  (not_in %x ()).          ! element is not in empty sequence
  (not_in %x (% $))<       ! element not in non-empty sequence
    (/= %x %),
    (not_in %x ($)).

! different - elements of sequence are distinct

  (different ()).
  (different (%)).
  (different (% $))<
    (different ($)),
    (not_in % ($)).
```

An example valid expression is (different (3 2 1)).

4. Conclusions

The SEND+MORE=MONEY example shows that axiomatic language can express constraint problems without all the built-in features of CP and CLP languages. It is a more general specification language, intended for a broader class of problems than just CP. Axiomatic language is idealistic in providing a clean separation between problem specification and implementation algorithm. It is elegant and extreme in its minimality.

Specifications should be smaller, more readable, more reusable, and more correct than algorithmic code. Smaller code size should give greater programmer productivity [Prechelt 00]. The small size and purity of the language should make it well-suited to proof. Proof would guarantee equivalence between the user's specification and the generated efficient program [Pettorossi & Proietti 99]. One may also be able to prove assertions about a specification to validate it and this may be more powerful than just testing.

The implementation of axiomatic language requires automatically transforming specifications to equivalent efficient programs – a grand challenge of computer science [Rich & Waters 88, Goebl 00]. Since CP language features are not part of axiomatic language, the transformation system would need knowledge to recognize CP problems and then apply the appropriate implementation algorithm. The generality of axiomatic language makes its implementation a more difficult and ambitious Holy Grail problem.

References

http://www.axiomaticlanguage.org/

[Goebl 00] W. Goebl, A Survey and a Categorization Scheme of Automatic Programming Systems, GCSE'99, LNCS 1799, pp. 1-15, 2000.

[Pettorossi & Proietti 99] A. Pettorossi, M. Proietti, Synthesis and transformation of logic programs using unfold/fold proofs, J. of Logic Programming 41, pp. 197-230, 1999.

[Prechelt 00] L. Prechelt, An Empirical Comparison of Seven Programming Languages, IEEE Computer, Vol. 33, No. 10, pp. 23-29, 2000.

[Rich & Waters 88] C. Rich, R. Waters, Automatic Programming: Myths and Prospects, IEEE Computer, Vol. 21, pp. 40-51, 1988.

[Wilson 82] W. Wilson, Beyond Prolog: software specification by grammar, SIGPLAN Notices, Vol. 17, No. 9, pp. 34-43, 1982.

[Wilson & Lei 12] W. Wilson, Y. Lei, A Tiny Specification Metalanguage, 24th Intl. Conf. on Software Engineering and Knowledge Engineering, pp. 486-490, 2012.