# Implementation of Axiomatic Language

Walter W. Wilson
wwwilson@acm.org
http://www.axiomaticlanguage.org

Univ. of Texas at Arlington
Advisor: Dr. Jeff Lei

ICLP 2011 Doctoral Consortium
July 6, 2011
Lexington, Kentucky

# Some Questions

- Is axiomatic language interesting? Novel? Significant?
- Is this “automatic programming” problem solvable?
- Can the meaning of programs be recognized given pre-stored knowledge?
- Can a manageable quantity of knowledge successfully transform most programs?
- Are large-scale unfold/fold proofs possible?

# Axiomatic Language

- Goals
  - Pure specification language
  - Minimal, but extensible
  - Meta-language – imitate other languages
- Idea: Program specified by static infinite set of symbolic expressions that enumerate inputs and corresponding outputs.
- Description:
  - pure, definite Prolog with Lisp syntax
  - HiLog higher-order generalization
  - "string variables"

# Informal Overview

Prolog predicate in axiomatic language:

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

"Axioms" for natural numbers and their addition:

```
(number 0).
(number (s %n))< (number %n).
```

→ generated "valid expressions": `(number 0), (number (s (s 0)))`

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

→ `(plus (s 0) (s 0) (s (s 0)))`

"String variables":

```
(member % ($1 % $2)).        → (member c (a b c d))
(concat ($1) ($2) ($1 $2)) → (concat (a b) (c) (a b c))
(reverse () ()).
(reverse (% $) ($rev %))< (reverse ($) ($rev)).
```

→ `(reverse (u v) (v u))`

# The Core Language

Finite set of axioms generates infinite set of valid expressions.

an **expression:**

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

syntax:

- atoms: `` `abc ``, `` `+ ``
- expression variables: `%1`, `%n`
- string variables: `$xyz`, `$`
- sequences: `()`, ``(`M (%x $2))``

# The Core Language (cont.)

**axiom** – a **conclusion** expression and zero or more **condition** exprs:

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

**axiom instance -** substitute values for expression and string variables

– arbitrary expression for an expression variable

– string of expressions and string variables for a string variable

```
(`a %x $1)< (`b $1 %x).
→ (`a `c ($) `d)< (`b ($) `d `c).
```

**valid expression** – conclusion of axiom instance is valid expression

if all conditions are valid expressions

```
(`a `b).
((%) $ $)< (% $).
→ (`a `b), ((`a) `b `b), (((`a)) `b `b `b `b), …
```

# Syntax Extensions

characters & strings:

```
'A'  =  (`char (`0 `1 `0 `0 `0 `0 `0 `1))
(… 'abc' …)  =  (… 'a' 'b' 'c' …)
"abc"  =  ('abc')  =  ('a' 'b' 'c')
```

symbols:

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

# Specification by Enumeration

Axioms generate expressions for all inputs and corresponding outputs:

```
(Program <input> <output>)
```

Sorting example:

```
(Program ("horse" "dog" "cat")  ! input file
         ("cat" "dog" "horse"))  ! output
```

Interactive program:

```
(Program <out> <in> <out> … <in> <out>)
```

-- generate for each possible execution history

# Beauty of Axiomatic Language

- Pure specification – what declarative programming should be
- Smaller code size, more readable, more reusable
- Minimal to the extreme
- Simple, clear semantics
- No ugly non-logical features
- No awkward non-declarative input/output
- Higher-order power + Lisp syntax
- Able to subsume other languages
- Has the beauty of Lisp (and FP in general)
- String variables
- Explicit approximate arithmetic
- Long-term stability

# A Transformation System

- Large knowledge base of functions and specification patterns
- System attempts to “understand” user’s specification
- Pre-defined algorithm for problem is then output
- Steps are supported by pre-defined unfold/fold proofs

# Transformation Example

```
(T () H).
(B (L %a) (L %b))< (B %a %b).
(C (L %a))< (C %a).
(J () () ()).
(B H %a)< (C %a).
(C H).
(U %a H %a)< (C %a).
(R %a $a)< (W () ($a)).

(P A "0123456789").
(F (%a)).
(W %a ()).
(F ()).
(M %a H H)< (C %a).
(K () ()).
(E (%a) %b)< (Q A %a %b).

(J ($a ($b)) ($c %a) ($d ($b %a)))< (J ($a) ($c) ($d)).
(U %a (L %b) (L %c))< (U %a %b %c).
(W %a (%a $a))< (W %a ($a)).
(Program %a %b)< (K %a %b), (R E %b %c), (F %c).
(T (%a $a) (L %b))< (T ($a) %b).
(F (%a %b $a))< (B %a %b), (F (%b $a)).
(Q %a %b %c)< (P %a ($a %b $b)), (T ($a) %c).
(M %a (L %b) %c)< (M %a %b %d), (U %a %c %d).
(E ($a %a) %b)< (E ($a) %c), (Q A %a %d),
  (M %c (L (L (L (L (L  (L (L (L (L (L H))))) ))))) %e),
  (U %e %d %b).
(R %a $a)< (R %a $b), (%a $c), (J ($b) ($c) ($a)).
(K ($a %a $b) ($c %b $d))< (K ($a $b) ($c $d)).
```

# Transformation Example (cont.)

```
(Program %a %b)< (K %a %b), (R E %b %c), (F %c).

(E ($a %a) %b)< (E ($a) %c), (Q A %a %d),
  (M %c (L (L (L (L (L  (L (L (L (L (L H))))) ))))) %e),
  (U %e %d %b).
(E (%a) %b)< (Q A %a %b).

(F (%a %b $a))< (B %a %b), (F (%b $a)).
(F (%a)).
(F ()).

(B (L %a) (L %b))< (B %a %b).
(B H %a)< (C %a).

(M %a (L %b) %c)< (M %a %b %d),
  (U %a %c %d).
(M %a H H)< (C %a).

(U %a (L %b) (L %c))< (U %a %b %c).
(U %a H %a)< (C %a).

(T (%a $a) (L %b))< (T ($a) %b).
(T () H).

(C (L %a))< (C %a).
(C H).
```

```
(Q %a %b %c)< (P %a ($a %b $b)),
  (T ($a) %c).

(R %a $a)< (R %a $b), (%a $c),
  (J ($b) ($c) ($a)).
(R %a $a)< (W () ($a)).

(K ($a %a $b) ($c %b $d))<
  (K ($a $b) ($c $d)).
(K () ()).

(J ($a ($b)) ($c %a) ($d ($b %a)))<
  (J ($a) ($c) ($d)).
(J () () ()).

(W %a (%a $a))< (W %a ($a)).
(W %a ()).

(P A "0123456789").
```

*Next:* C

# Transformation Example (cont.)

```
(Program %a %b)< (K %a %b), (R E %b %c), (F %c).

(E ($a %a) %b)< (E ($a) %c), (Q A %a %d),
  (M %c (s (s (s (s (s  (s (s (s (s (s 0)))))) ))))) %e),
  (U %e %d %b).
(E (%a) %b)< (Q A %a %b).

(F (%a %b $a))< (B %a %b), (F (%b $a)).
(F (%a)).
(F ()).

(B (s %a) (s %b))< (B %a %b).
(B 0 n)< (number n)  .

(M %a (s %b) %c)< (M %a %b %d),
  (U %a %c %d).
(M %n 0 0)< (number %n).

(U %a (s %b) (s %c))< (U %a %b %c).
(U %n 0 %n)< (number %n).

(T (%a $a) (s %b))< (T ($a) %b).
(T () 0).

(number n)
H = 0, L = s

(P A "0123456789").
```

```
(Q %a %b %c)< (P %a ($a %b $b)),
  (T ($a) %c).

(R %a $a)< (R %a $b), (%a $c),
  (J ($b) ($c) ($a)).
(R %a $a)< (W () ($a)).

(K ($a %a $b) ($c %b $d))<
  (K ($a $b) ($c $d)).
(K () ()).

(J ($a ($b)) ($c %a) ($d ($b %a)))<
  (J ($a) ($c) ($d)).
(J () () ()).

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

*Next:* T

# Transformation Example (cont.)

```
(Program %a %b)< (K %a %b), (R E %b %c), (F %c).

(E ($a %a) %b)< (E ($a) %c), (Q A %a %d),
  (M %c <10> %e),
  (U %e %d %b).
(E (%a) %b)< (Q A %a %b).

(F (%a %b $a))< (B %a %b), (F (%b $a)).
(F (%a)).
(F ()).

(B (s %a) (s %b))< (B %a %b).
(B 0 %n)< (number %n).

(M %a (s %b) %c)< (M %a %b %d),
  (U %a %c %d).
(M %n 0 0)< (number %n).

(U %a (s %b) (s %c))< (U %a %b %c).
(U %n 0 %n)< (number %n).

(length seq len)

(number n)

(P A "0123456789").
```

```
(Q %a %b %c)< (P %a ($a %b $b)),
  (length ($a) %c).

(R %a $a)< (R %a $b), (%a $c),
  (J ($b) ($c) ($a)).
(R %a $a)< (W () ($a)).

(K ($a %a $b) ($c %b $d))<
  (K ($a $b) ($c $d)).
(K () ()).

(J ($a ($b)) ($c %a) ($d ($b %a)))<
  (J ($a) ($c) ($d)).
(J () () ()).

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

*Next:* U

# Transformation Example (cont.)

(Program <file_of_dec_nums> <sorted_file>)

(digs->num *decstr num*)

(elem->num digit *dig num*)

(ordered_num *num_seq*)

(map rel ..argseqs..)

(<= a b)

(perm *seq seq'*)

(times *a b a*b*)

(plus *a b a+b*)

(distr *seqs seq seqs+seq*)

(length *seq len*)

(number *n*)

(seq_of *expr seq-of-expr*)

(set digit "0123456789")

*Done!*

# Transformation Summary

- My assumptions:
  - Recognition of functions possible given pre-stored knowledge
  - Efficient algorithm exists to recognize these input axiom patterns
  - Pre-stored efficient implementation algorithm can be provided once specification is "understood"
  - Unfold/fold proofs can guarantee equivalence of generated implementation
  - When input axioms cannot be "understood", expert can add knowledge and proofs
    - No productivity benefit – faster for user to write efficient program
    - Some software engineering benefit – separation of specification & implementation and proof of correctness
- Long-term optimism:
  - Specification pattern knowledge can be generalized
  - Less expert intervention needed over time
  - Eventually system will handle typical programs automatically