# Toward A Constraint-Satisfaction Framework for Evaluating Program-Understanding Algorithms

*Alex Quilici*
Department of Electrical Engineering
University of Hawaii at Manoa
2540 Dole St, Holmes 483
Honolulu, Hawaii, 96822
alex@wiliki.eng.hawaii.edu

*Steven Woods*
Department of Computer Science
University of Waterloo
Davis Centre for Computer Research
Waterloo, ON N2L 3G1
sgwoods@logos.uwaterloo.ca

This work was partially supported by the Rome Labs KBSA project under Air Force contract #F30602-93-C-0257, the Natural Sciences and Engineering Research Council of Canada, and the Information Technology Research Centre.

# The Problem

- Many plan-based program understanding algorithms.
- Currently difficult to compare.
  - Performance.
  - Understanding capabilities.
- Major underlying differences.
  - Different representations.
  - Heuristic tricks.
- Need descriptive framework that supports comparative analysis.

# Our Solution

- View program understanding as a constraint-satisfaction problem (CSP).
- Attempt to map existing algorithms into CSP framework.
- Benefits:
  - Allow us to compare resulting CSPs.
  - Possibly augment algorithms with general heuristic mechanisms for solving CSPs.

# An Existing Algorithm: Concept Recognizer

- Kozaczynski and Ning, 1994.
- Representation: Components and Constraints.

```
define TRAVERSE-STRING(String) isa TRAVERSE-PLAN

plan TRAVERSE-STRING(String: ?a)
  components
    decl:     DECL-ARRAY(Name: ?s, Items: ?max, Type: char)
    init:     ZERO(Dest: ?i)
    loop:     LOOP(Test-Result: ?r, Body: ?body)
    access1:  ACCESS(Op1: ?s, Op2: ?i, Res: ?val1)
    test:     BIN-OP(Op1: ?i, Op2: ?val1, Op: !=, Res: ?r)
    access2:  ACCESS(Op1: ?s, Op2: ?i, Res: ?val2)
    update:   INCREMENT(Op: ?i)
  constraints
    declbef:  ControlPath(decl, loop)
    initbef:  DataDep(test, init, ?i)
    acc1bef:  DataDep(test, access1, ?val1)
    testin:   DataDep(loop, test, ?r)
    acc2in:   ControlDep(access2, ?body)
    updaft:   DataDep(access2, update, ?i)
```

- Algorithm: Top-Down matching of plan library to code.

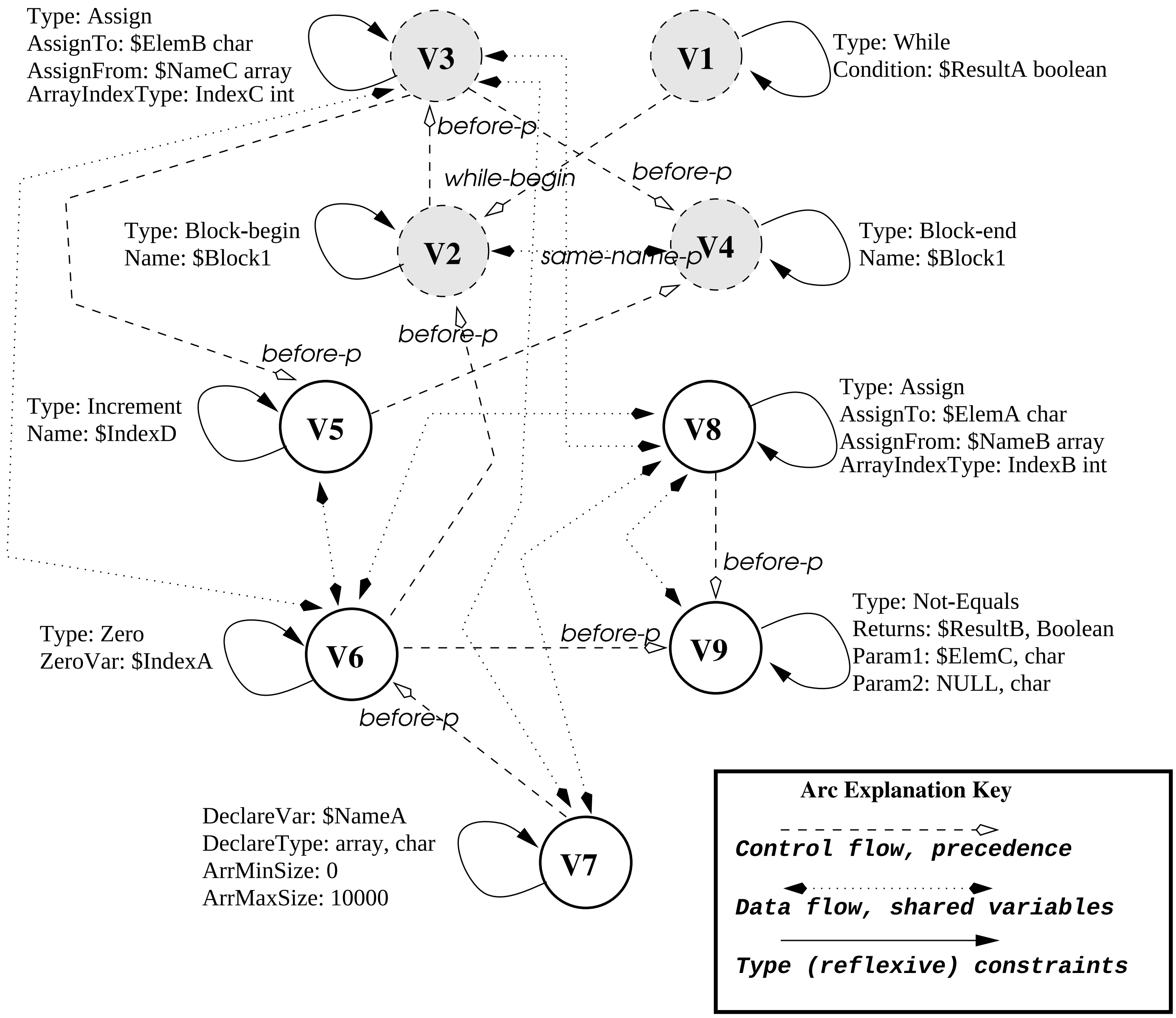
# Constraint Satisfaction Problems

- Major components:
  - Set of variables.
  - Set of finite domain values for each variable.
  - Set of inter-variable constraints (restrict domain assignments).
- Problem solution:
  - Assignment of domain value to variables.
  - Assignment satisfies all inter-variable constraints.

# Concept Recognizer as CSP

- MAP-CSP: Locate all instances of a given plan.
  - Plan components become variables.
    - Reflexive “type” constraint based on AST/subplan entries.
    - Domain values are instances of ”type” in source.
  - Constraints between plan components become inter-variable constraints.
- Layered MAP-CSP: Locate all plans in the source.
  - Layer plan library based on dependencies on other plans.
  - Apply MAP-CSP bottom-up, a layer at a time.

# MAP-CSP Plan Representation



# MAP-CSP vs. Concept Recognizer

- Representation: Exact Mapping.
- Control strategy:
  - Imposes a particular layered ordering in how plans are tried.
  - No ordering originally specified in Concept Recognizer.

# Another Algorithm: DECODE

- Chin and Quilici, 1996.
- Attemp to extend Concept Recognizer:
  - Speed-up recognition.
  - Have recognition strategy match user studies.
- Key differences:
  - Code-driven rather than library-driven.
  - Careful indexing/organization of plan library.

# DECODE's Programming Plans

```
define TRAVERSE-STRING(String) isa TRAVERSE-PLAN
define PRINT-STRING(String) isa PRINT-PLAN
define PRINT-CHAR(Char) isa PRINT-PLAN
define ZERO(Dest) isa ASSIGN-PLAN

plan TRAVERSE-STRING(String: ?a)
  components
    decl:     DECL-ARRAY(Name: ?s, Items: ?max, Type: char)
    init:     ZERO(Dest: ?i)
    loop:     LOOP(Test-Result: ?r, Body: ?body)
    access1:  ACCESS(Op1: ?s, Op2: ?i, Res: ?val1)
    test:     BIN-OP(Op1: ?i, Op2: ?val1, Op: !=, Res: ?r)
    access2:  ACCESS(Op1: ?s, Op2: ?i, Res: ?val2)
    update:   INCREMENT(Op: ?i)
  constraints
    declbef:  ControlPath(decl, loop)
    initbef:  DataDep(test, init, ?i)
    acc1bef:  DataDep(test, access1, ?val1)
    testin:   DataDep(loop, test, ?r)
    acc2in:   ControlDep(access2, ?body)
    updaft:   DataDep(access2, update, ?i)
  index
    access2 WHEN accin

  implies PRINT-STRING(Srting: ?a)
    with
      dump:     PRINT-CHAR(Source: ?value)
    when
      dumpaft:  DataDep(dump, access2, ?v)

plan PRINT-CHAR(Char: ?c)
  specializes Call-Function(Name: putchar, Args: ?c)

plan ZERO(Item: ?i)
  specializes Assign(Dest: ?i, Value: 0)
```

# DECODE's Understanding Algorithm

- Initialize the program tree ($PT$) to the set of elements in the program's abstract syntax tree
- For each plan library layer $L$:
  - For each element $E_i$ in $PT$:
    - $*$ For each plan implementation $P_j$ in $L$ indexed by $E_i$:
      - $\cdot$ Form the set of partial plan instances ($PPI$) that result from binding $E_i$ to each $P_j$.
      - $\cdot$ Replace $PPI$ with the set that results from processing the indexing constraints on the original $PPI$.
      - $\cdot$ If $PPI$ is non-null, set the recognized plan instances ($RPI$) to the result of processing the remaining constraints on each element in $PPI$.
      - $\cdot$ Add each element of $RPI$ to $PT$ and add each plan it implies to the set of potentially implied plans ($PIP$).
  - For each plan $P_j$ in $L$:
    - $*$ For any corresponding $PIP_k$ in $PIP$:
      - $\cdot$ Set the implied plan instances ($IPI$) to the result of processing implication constraints on $PIP_k$.
      - $\cdot$ Add $IPI$ to $PT$.

# DECODE's Understanding Algorithm (Part II)

Process-Constraints(CS, PPI)

- For each constraint $C_i$ in $CS$:
  - For each $PPI_i$ in $PPI$,
    * Form the set of new partial plan instances ($NPPI$) that result from binding the components in $PPI_i$ that are necessary to evaluate $C$ against elements of $PT$.
    * Form the set of remaining partial plan instances ($RPPI$) that result from evaluating $C_i$ on each item in $NPPI$.
  - Set $PPI$ to the concatenation of all the $RPPI$s.

# DECODE as MEMORY-CSP

- Capture indexing with hierarchical CSP:
  - MAP-CSP to recognize all instances of an index.
  - MAP-CSPs to recognize remaining plan for each found index.
  - Indexing seen as variable/constraint ordering mechanism.
- Capture implication as another form of indexing

# MAP-CSP Plan Index Representation

```
'( "quilici-t1-index"
   (
     (q1-c While      (ResultA (boolean)))
     (q1-d Begin      (Block1 (block)))
     (q1-g Assign     (NameC (array (char))) (IndexC (int))
                      (ElemB (char)))
     (q1-e End        (Block2 (block)))
   )
   (
     (before-p    (q1-c q1-d))
     (close-to-p  (q1-c q1-d) 10)
     (before-p    (q1-d q1-g))
     (same-name-p (q1-d q1-e) (Block1 Block2))
     (before-p    (q1-g q1-e))
   ))
```

# MAP-CSP Complete Plan Representation

```
'( "quilici-t1"
   (
     (q1-c While      (ResultA (boolean)))
     (q1-d Begin      (Block1 (block)))
     (q1-g Assign     (NameC (array (char))) (IndexC (int))
                           (ElemB (char)))
     (q1-e End        (Block2 (block)))
     (q1-i Increment  (IndexD (int)))
     (q1-a Decl       (NameA (array (char) (0 10000))))
     (q1-b Zero       (IndexA (int)))
     (q1-f Assign     (NameB (array (char))) (IndexB (int))
                             (ElemA (char)) )
     (q1-h Not-Equals (ElemC (char)) (NULL (char)) (ResultB (boolean)))
   )
   (
     (before-p    (q1-c q1-d))
     (close-to-p  (q1-c q1-d) 10)
     (before-p    (q1-d q1-g))
     (same-name-p (q1-d q1-e) (Block1 Block2))
     (before-p    (q1-g q1-e))
     (before-p    (q1-b q1-c))
     (before-p    (q1-a q1-b))
     (before-p    (q1-b q1-h))
     (before-p    (q1-d q1-e))
     (before-p    (q1-f q1-h))
     (before-p    (q1-g q1-i))
     (before-p    (q1-d q1-i))
     (before-p    (q1-i q1-e))
     (same-name-p (q1-c q1-h) (ResultA ResultB))
     (same-name-p (q1-f q1-h) (ElemA ElemC))
     (same-name-p (q1-a q1-f) (NameA NameB))
     (same-name-p (q1-a q1-g) (NameA NameC))
     (same-name-p (q1-b q1-f) (IndexA IndexB))
     (same-name-p (q1-b q1-g) (IndexA IndexC))
     (same-name-p (q1-b q1-i) (IndexA IndexD))
  ))
```

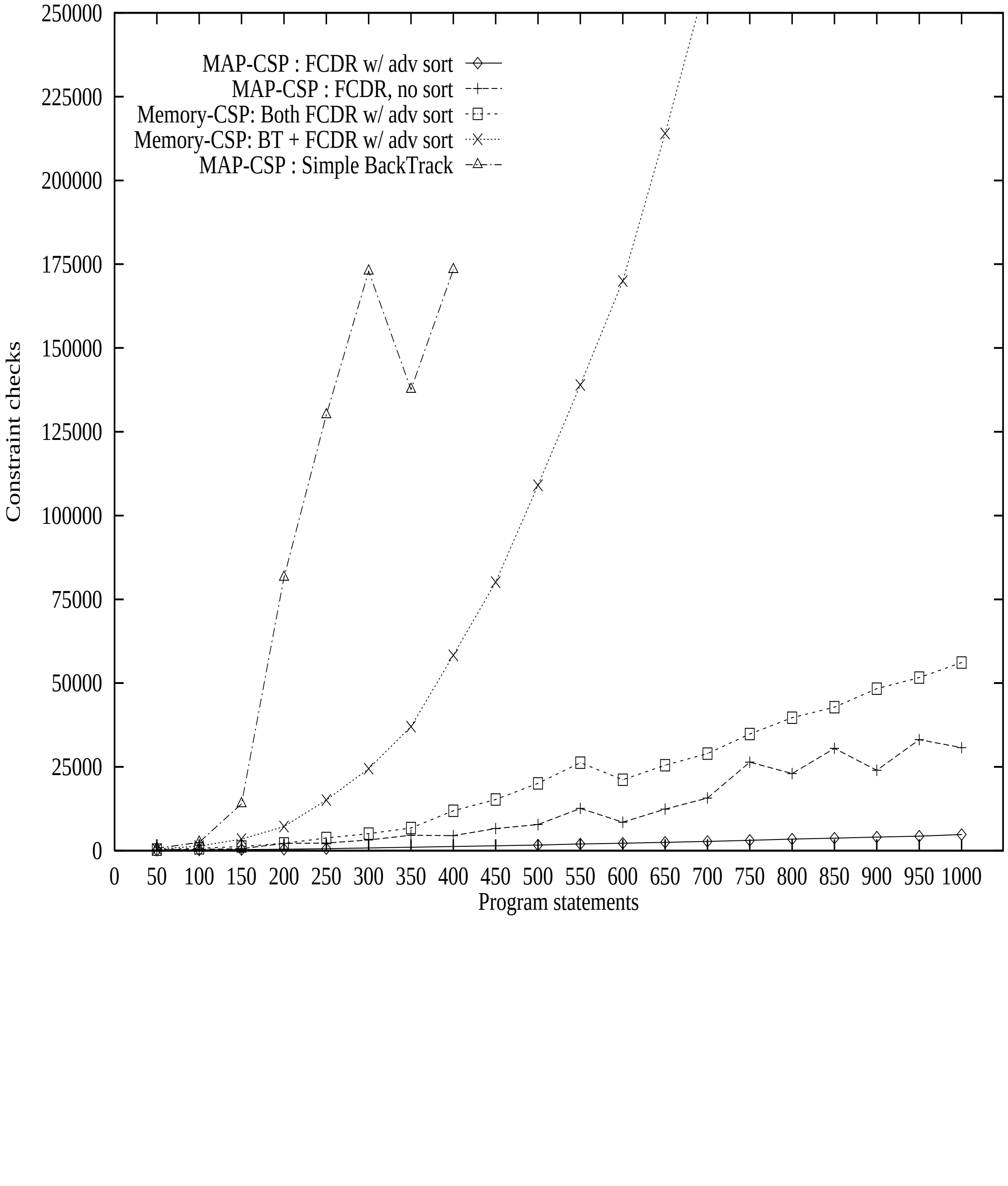
# Empirically Comparing CSPs

- Three variants of MAP-CSP:
  1. Simple Backtracking.
  2. Forward Checking with Dynamic Rearrangement (FCDR).
     - Without advance variable-ordering.
     - With advance variable ordering.
- Two variants of Memory-CSP:
  1. Backtracking, FCDR with advance variable sorting
  2. Backtracking, FCDR without advance variable sorting

# Methodology

- Generate artificial programs of varying sizes.
  - Grown around instance of plan we want to recognize.
  - Distribution of statement types matching student C programs.
- Run various CSPs on each program.
- Measure constraints evaluated.

# Experimental Results



# Experimental Conclusion

- Heuristic CSP methods (other than backtracking) win over DECODE's indexing.
- Why?
  - $Static$ indexing vs. $dynamic$ contraint evaluation.
  - Differences in structural properties between artificial and real-world programs.

## Summary

- Accomplished:
  - Mapped a pair of existing algorithms into CSP framework.
  - General CSP solutions better than specific heuristic tricks.
- Still to do:
  - Better understanding of our CSP understanders.
  - Map other algorithms into CSP framework.
  - Verify results not experimental artifact.
- Benefits:
  - Framework allows us to compare understanding algorithms.
  - Deeper understanding of program understanding problem.