Symbolic Pointer Analysis

Jianwen Zhu

Electrical and Computer Engineering
University of Toronto

November 11th, 2002

jzhu@eecg.toronto.edu
http://www.eecg.toronto.edu/˜jzhu

Outline

- Background
- New Formalism
- New Efficiency
- Engineering and Results
- Conclusion

Synthesis from C-like Languages

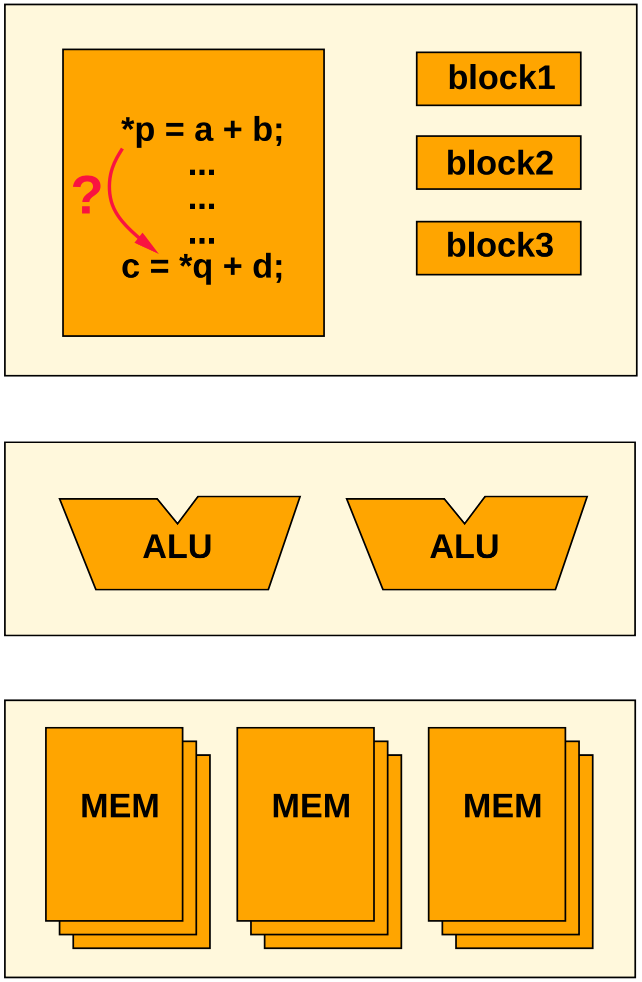
- Traditional high-level synthesis
  - Regretfully not embraced by design community
  - Complexity
  - Quality
- Commercial “C Synthesis” tools today
  - RTL in C flavor
- SystemC/SpecC
  - Primarily used for system-level modeling

The Pervasive Pointers

- All of them are pointers!
  - C: addresses of global, local and heap block
  - C++: plus addresses of class objects
  - Java: reference to class objects
  - Function pointers
  - Virtual methods, Interfaces
- Complex data structures
- Candidates for hardware synthesis
  - Multimedia
  - Networking
  - 3-D Graphics

The Evasive Pointers

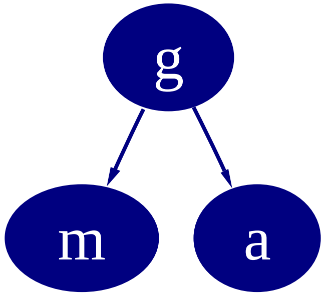
- Runtime values unknown at compile time
- Pointers maybe aliases to each other
- Evil for optimization
  - Dependency test for parallelization
  - Memory bank partition
  - Memory sharing



```
*p = a + b;
...
...
...
c = *q + d;
```

State of the Art

- Pointer Analysis Problem
  - Determine program state at every program point
  - Cares only about pointer values
  - Undecidable problem



- Context and Flow sensitivity

- Hind 2001: “75 papers, 9 PhD thesis”
- Flow and Context-insensitive
  - Steensgaard’96
  - Andersen’94
- Flow and Context-sensitive
  - Wilson and Lam’95
  - Liang and Harrold’01
- Applications in CAD
  - Semeria and De Micheli’01
  - Panda et. al.’01
  - Zhu’01

Sources of Inefficiency

- Aggressive optimization needs accurate analysis
  - Context-sensitive + Flow-sensitive + more!
- Best available algorithms have exponential complexity
  - Cost for summarize procedure call
    - Wilson’s partial transfer function
    - Liang’s parameterized summary
  - Cost for propagating call effect at call site
  - Redundant program state representation

Outline

- Background
- New Formalism
- New Efficiency
- Engineering and Results
- Conclusion

Program Modeling

- Representing instructions
  - Only two types interesting

    store dest, src

    call dest, [src$_1$ … src$_n$]
  - dest, src$_i$ = $\langle block, level \rangle$
  - Example:

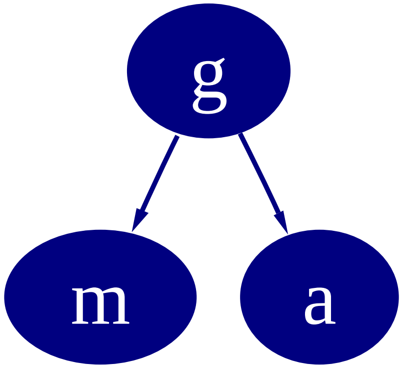
store t &g;	store $\langle t,0\rangle$, $\langle g,-1\rangle$
store *r, *t;	store $\langle r,1\rangle$, $\langle t,1\rangle$
call getg, [q, g];	call $\langle getg,0\rangle$, [$\langle q,0\rangle$, $\langle g,0\rangle$]
store *f, &m;	store $\langle f,1\rangle$, $\langle m,-1\rangle$

```
char *g, a;
void main() {
    call alloc, [p];
    call getg,  [q, g];
    store g, &a;
    }
getg( [r1, g1] ) {
    store t, &g;
    others
    call alloc [*t];
    store *r, *t;
    }
alloc( [f1] ) {
    store *f, &m;
    }
```

Point-to Graph

- Captures program state $\langle V, E \rangle$
- Vertices V
  - Global block
  - Local block
  - Heap block
  - Procedure block
  - Initial block (λ *state@callsite*)
- Edges E
  - $\langle u, v \rangle \in E \Rightarrow$ the content of block u may be the address of location v

- Basic algorithms
  - state query
  - evaluating store
  - evaluating call

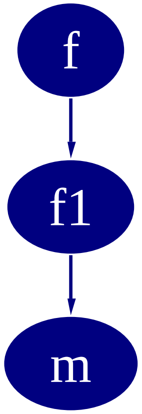


A Symbolic Alternative

- Key observation: edge set of a graph captures a Relation
- Big idea: represent relation using Boolean function
- Define Boolean space: domain and range space
- Encoding memory locations
  - initials ↦Boolean variable
  - Others ↦minterms

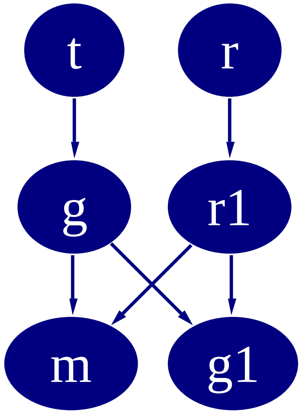
Block	domain	range
a	$\overline{x_1^*}\,\overline{x_2^*}\,\overline{x_3^*}$	$\overline{x_1}\,\overline{x_2}\,\overline{x_3}$
g	$\overline{x_1^*}\,\overline{x_2^*}\,x_3^*$	$\overline{x_1}\,\overline{x_2}\,x_3$
p	$\overline{x_1^*}\,x_2^*\,\overline{x_3^*}$	$\overline{x_1}\,x_2\,\overline{x_3}$
q	$\overline{x_1^*}\,x_2^*\,x_3^*$	$\overline{x_1}\,x_2\,x_3$
t	$x_1^*\,\overline{x_2^*}\,\overline{x_3^*}$	$x_1\,\overline{x_2}\,\overline{x_3}$
r	$x_1^*\,\overline{x_2^*}\,x_3^*$	$x_1\,\overline{x_2}\,x_3$
f	$x_1^*\,x_2^*\,\overline{x_3^*}$	$x_1\,x_2\,\overline{x_3}$
m	$x_1^*\,x_2^*\,x_3^*$	$x_1\,x_2\,x_3$
f1	y_1^*	y_1
g1	y_2^*	y_2
r1	y_3^*	y_3

Symbolic Replacement of Point-to Graph



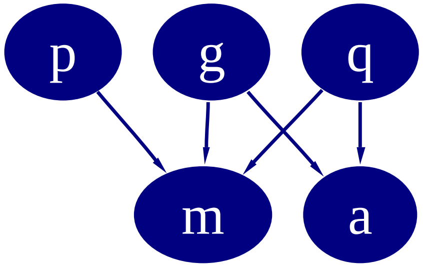
$$\begin{aligned} & x_1^* x_2^* \overline{x_3^*} y_1 \\ + \quad & y_1^* x_1 x_2 x_3 \end{aligned}$$

(a) alloc



$$\begin{aligned} & x_1^* \overline{x_2^*}\, \overline{x_3^*}\, \overline{x_1}\, \overline{x_2} x_3 \\ + \quad & x_1^* \overline{x_2^*} x_3^* y_3 \\ + \quad & \overline{x_1^*}\, \overline{x_2^*} x_3^* x_1 x_2 x_3 \\ + \quad & \overline{x_1^*}\, \overline{x_2^*} x_3^* y_2 \\ + \quad & y_3^* x_1 x_2 x_3 \\ + \quad & y_3^* y_2 \end{aligned}$$

(b) getg

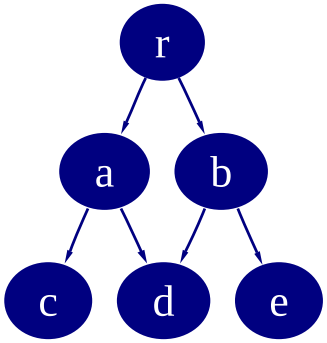


$$\begin{aligned} & \overline{x_1^*} x_2^* \overline{x_3^*} x_1 x_2 x_3 \\ + \quad & \overline{x_1^*}\, \overline{x_2^*} x_3^* x_1 x_2 x_3 \\ + \quad & \overline{x_1^*}\, \overline{x_2^*} x_3^* \overline{x_1}\, \overline{x_2}\, \overline{x_3} \\ + \quad & \overline{x_1^*} x_2^* x_3^* x_1 x_2 x_3 \\ + \quad & \overline{x_1^*} x_2^* x_3^* \overline{x_1}\, \overline{x_2}\, \overline{x_3} \end{aligned}$$

(c) main

Symbolic State Query

- Graph query: **r



- Symbolic query:

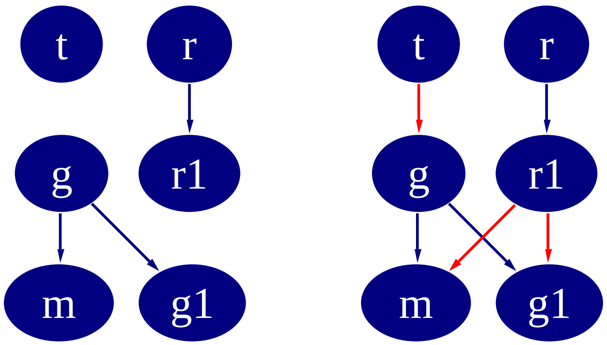
$$\begin{aligned} S &= r^*a + r^*b + a^*c \\ &+ a^*d + b^*d + b^*e \\ L1 &= a + b \\ L2 &= S \cdot mirror(L1) \\ &= c + d + e \end{aligned}$$

Algorithm 1 State query.

```
spaQueryState = func(
  spa, state, from, level
  ) : SpaDD {
  if( level == 0 ) return from ;
  return bddAndAbstract(
    spa, state, bddMirror(
      spa, spaQueryState(
       spa, state, from, level-1
       ))) ;
  }
```

Symbolic Evaluation of Stores

- Examples
  - $S_0 = r^* r1 + g^* m + g^* g1$
  - $t = \&g : \Delta = t^* g$
  - $*r = *t : \Delta = r1^* m + r1^* g1$

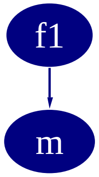


Algorithm 2 State update.

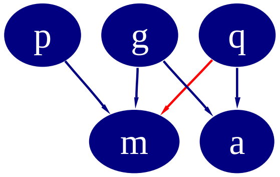
```
spaUpdateState = func(
  spa, state, dst, src
  ) : SpaDD {
  return bddOr( state, bddAnd(
      bddMirror( spa,
           spaQueryState( spa, state,
            dst.blk.range, dst.level )),
       spaQueryState( spa, state,
            src.blk.range, src.level+1 ))) ;
  }
```

Symbolic Evaluation of Calls

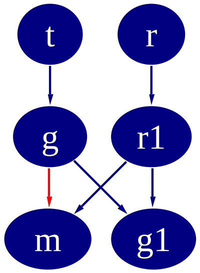
- Examples
  - Callee alloc with tf = $f1^{*}m$
  - At callsite of getg:
    $f1^{*}m|_{f1/p} = p^{*}m$
  - At callsite of main:
    $f1^{*}m|_{f1/g} = g^{*}m$



alloc



getg



main

Algorithm 3 Evaluate call.

```
spaApply = func(
  spa, state, srcs, proc, tf
  ) : SpaDD {
  var proj : SpaDD ↦ SpaDD;

  build projection;
  return bddOr(
    state, bddCompose( spa, tf, proj )
    ) ;
}
```

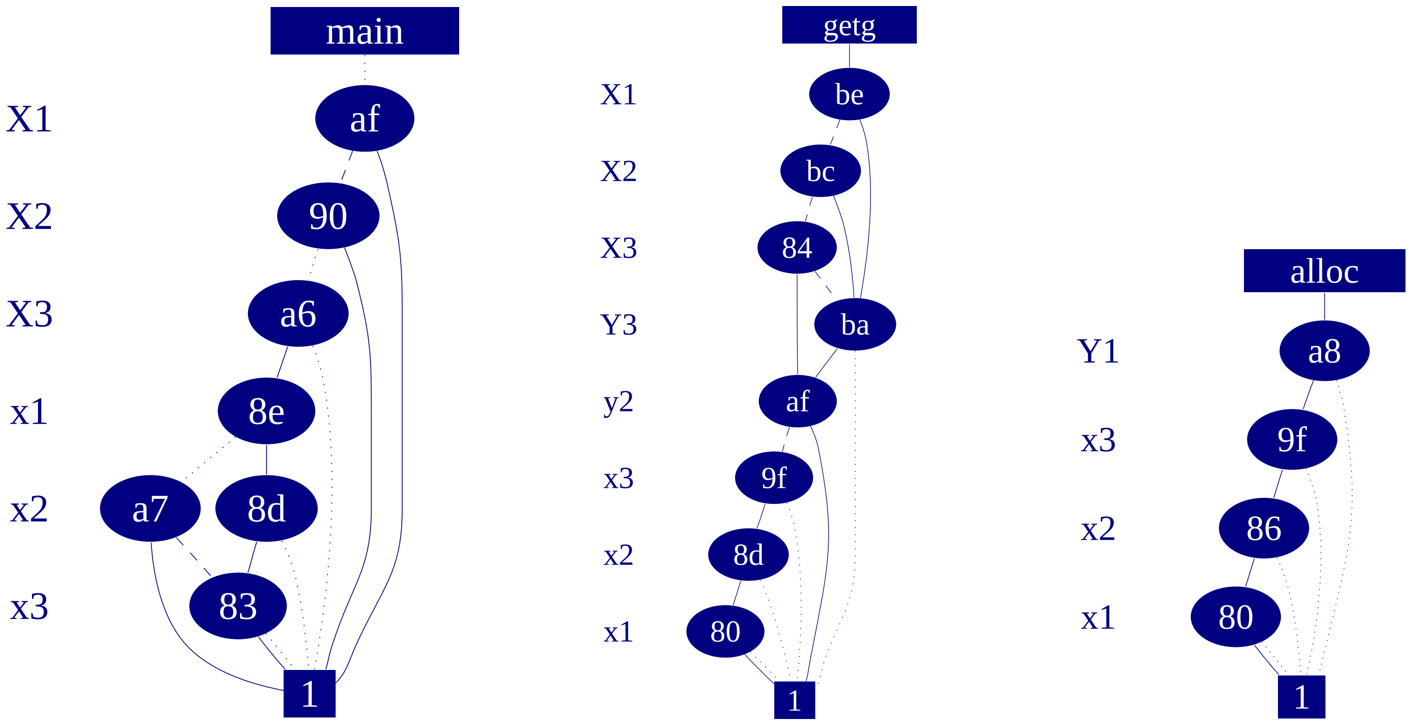
Outline

- Background
- New Formalism
- New Efficiency
- Engineering and Results
- Conclusion

Binary Decision Diagram (BDD)

- Summary
  - Established a Boolean formalism for the manipulation of Point-to relation
  - Sounds elegant, how efficient?
- Efficiency derive from Bryant's ROBDD
  - Rooted directed graph based on Shannon expansion
  - Small size for large amount of functions
  - Canonical
  - State query = Image computation?

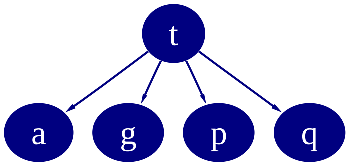
BDD Representation of Point-To Graph

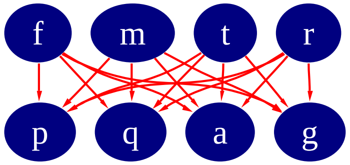


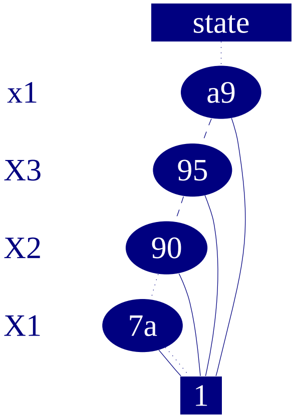
BDD Seems to be Larger, Why Bother?

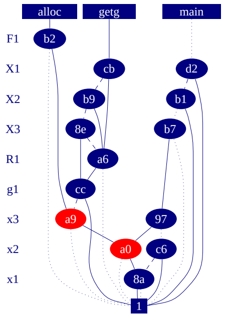
- Scale matters: the more #edges we have, the simpler the BDD!

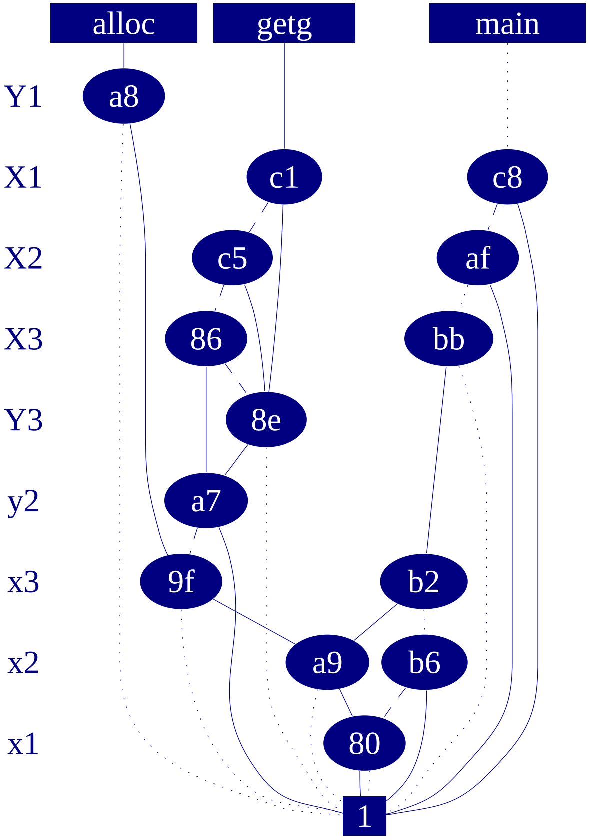
- Symbolic states can be shared among program points!











A Comment on Complexity

- Let G_1 and G_2 be two BDDs

operation	complexity
bddAnd	$O(\|G_1\|\|G_2\|)$
bddOr	$O(\|G_1\|\|G_2\|)$
bddCompose	$O(\|G_1\|^2\|G_2\|)$
practically	$O(\|G_1\|\|G_2\|)$
bddMirror	$O(\|G_1\|)$

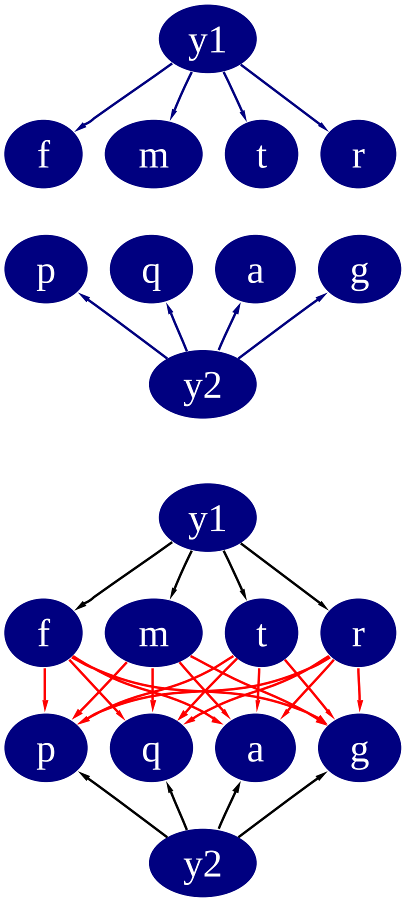
- Compound efficiency
  - Intra-procedural space sharing
  - Inter-procedural space sharing
  - Implicit batch processing
  - Dynamic programming
- Scalability

  gimp: 7M LOC, 131552 variables

  $\mapsto$18 Boolean variables

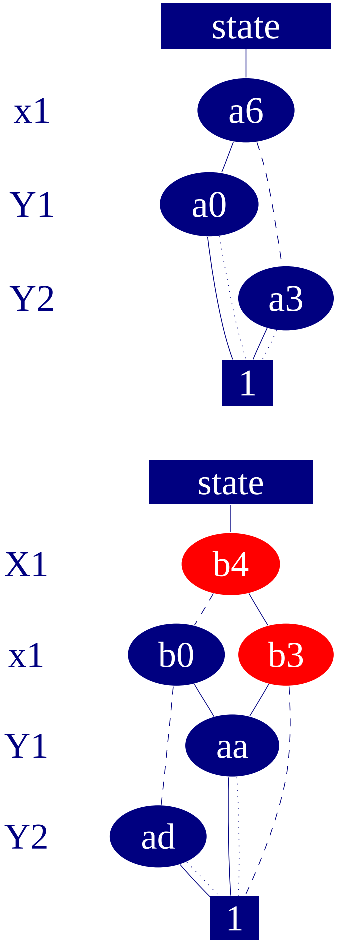
An Example of Batch Processing

- Explicit evaluation



*y1 = y2

- Implicit evaluation



*y1 = y2

Outline

- Background
- New Formalism
- New Efficiency
- Engineering and Results
- Conclusion

Engineering

- Relaxing simplifying assumptions
  - Records and classes
  - Arrays
  - Alias test of initial blocks
  - Strong and weak update
  - Recursive functions

- Engineering a fast algorithm
  - Partitioning of Boolean space
  - Fast bddMirror operation

    Consistent variable ordering between domain and range space
  - Fast bddCompose operation

    use single variable for initials and predicates
  - Caching of BDD operation

A Context-Sensitive Flow-Insensitive Validation

- Algorithm
  - Bottom-up evaluation of procedures
  - Use *spaUpdateState* for store instruction
  - Use *spaApply* for call instruction
  - Needs to iterate until fixed-point is reached
- Omissions
  - Field independent
  - No location set
  - Hardwired libraries
  - Ignore setjmp/longjmp

Experimental Results

- Standard benchmark from McGill and Landi
- While LOC is not large, invocation graph can be very large
- All finished in seconds

Name	Source	LOC	#procs	density	Run Time (s)
01.qbsort	McGill	325	8	24.1%	0.10
06.matx	McGill	350	7	13.5%	0.13
15.trie	McGill	358	13	23.4%	0.21
04.bisect	McGill	463	9	9.7%	0.10
17.bintr	McGill	496	17	8.8%	0.13
05.eks	McGill	1202	30	4.0%	0.20
08.main	McGill	1206	41	20.9%	1.33
09.vor	McGill	1406	52	28.6%	5.54
allroots	Landi	227	7	1.3%	3.02
football	Landi	2354	58	1.8%	2.38
compiler	Landi	2360	40	5.1%	5.3
assembler	Landi	3446	52	16.6%	10.63
simulator	Landi	4639	111	6.3%	4.03

Conclusion

- Pointer analysis is a crucial problem for C-based synthesis
- Contribution
  - Boolean algebra as new Formalism for pointer analysis
  - Efficient algorithms for fundamental symbolic pointer evaluation
  - Validation of new concept
- Future work
  - A scientific, comparative study of algorithm efficiency and scalability
  - Towards better precision
  - Towards faster speed
  - Towards application