Experiments in Software Verification using SMT Solvers

VS Experiments 2008 – Toronto, Canada

Leonardo de Moura
Microsoft Research

Agenda

- What is SMT?
- Experiments:
  - Windows kernel verification.
    - Extending SMT solvers.
  - Garbage collector (Singularity) verification
    - Supporting decidable fragments.

Satisfiability Modulo Theories (SMT)

- Arithmetic
- Bit-vectors
- Arrays
- …

Satisfiability Modulo Theories (SMT)

$$x+2=y \Rightarrow f(read(write(a,x,3),y-2)) = f(y-x+1)$$

Arithmetic

Satisfiability Modulo Theories (SMT)

$$x + 2 = y \Rightarrow f(read(write(a, x, 3), y - 2)) = f(y - x + 1)$$

Array Theory

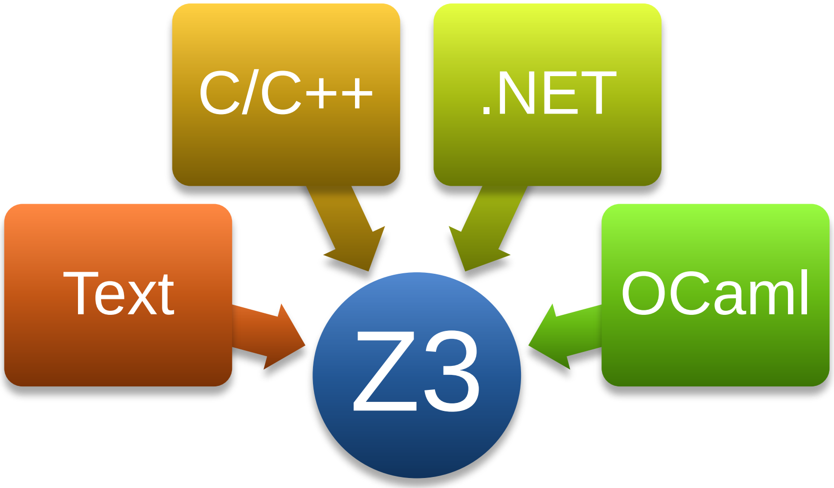
Satisfiability Modulo Theories (SMT)

$$x + 2 = y \Rightarrow f(read(write(a, x, 3), y - 2)) = f(y - x + 1)$$

Uninterpreted Functions

Z3

- Z3 is a new solver developed at Microsoft Research.
- Development/Research driven by internal customers.
- Free for academic research.
- Interfaces:



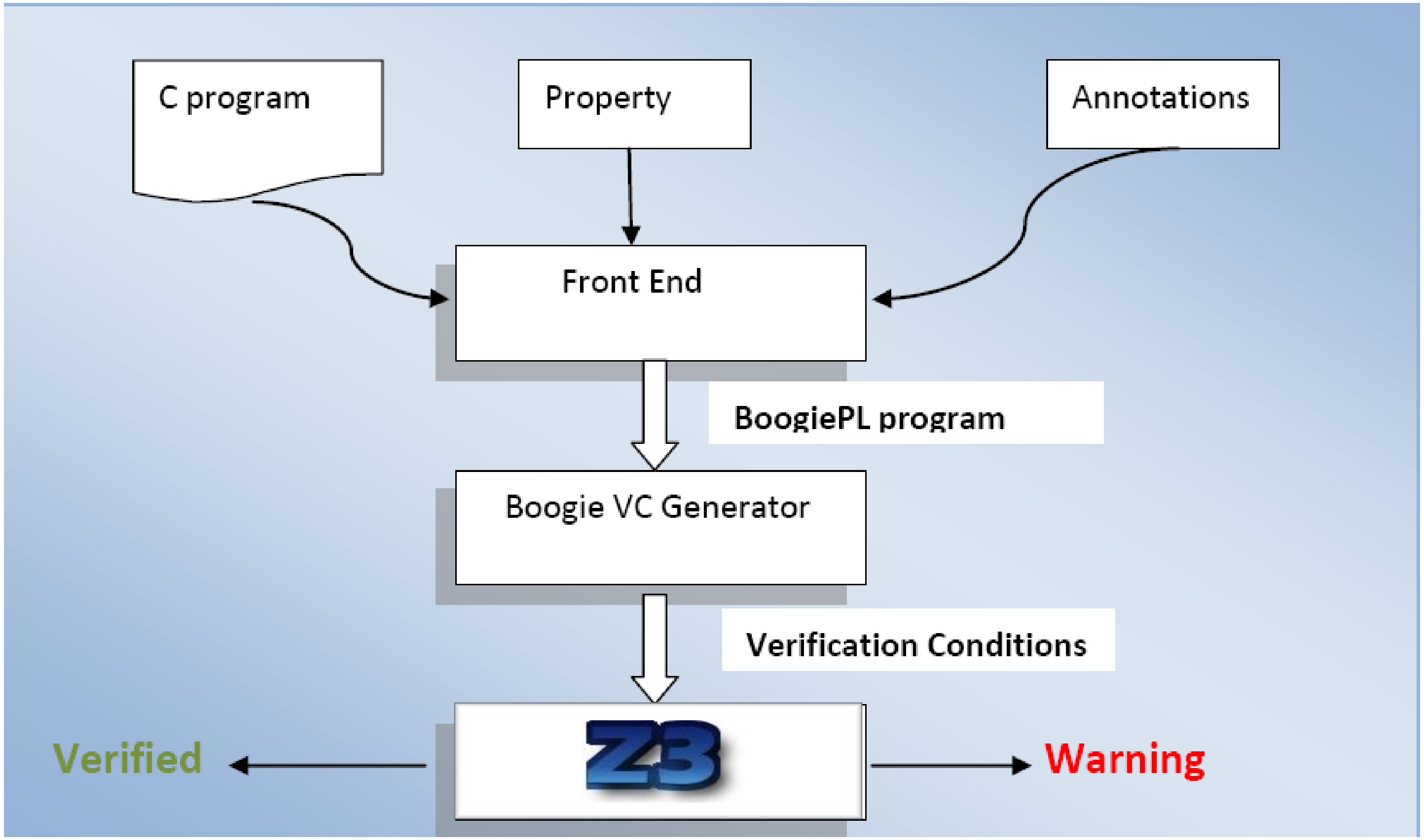
- http://research.microsoft.com/projects/z3

Microsoft Research

HAVOC
Verifying Windows Components

Lahiri & Qadeer, POPL'08,
Also: Ball, Hackett, Lahiri, Qadeer, MSR-TR-08-82.

HAVOC's Architecture



Heaps and Shapes

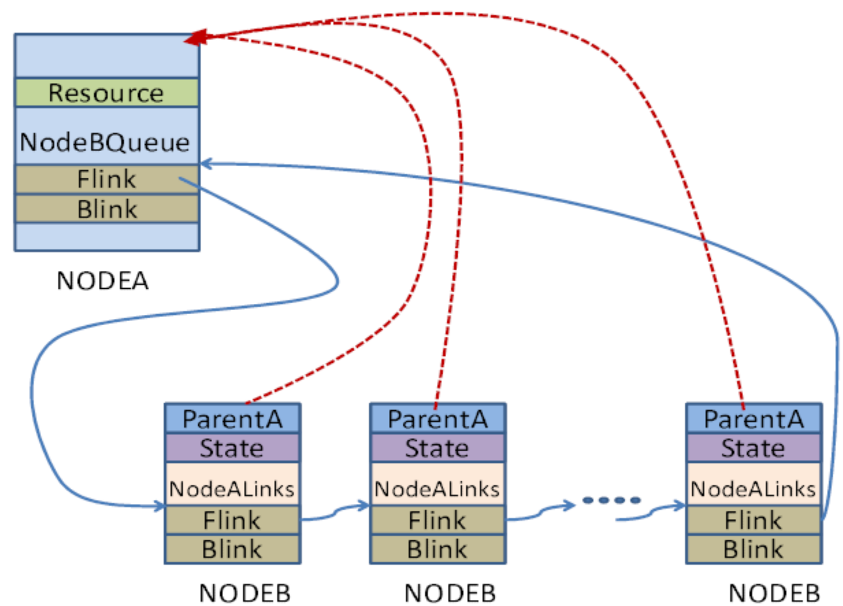
```
typedef struct _LIST_ENTRY{
  struct _LIST_ENTRY *Flink, *Blink;
} LIST_ENTRY, *PLIST_ENTRY;

typedef struct _NODEA{
  PERESOURCE Resource;
  LIST_ENTRY NodeBQueue;
  ...
} NODEA, *PNODEA;

typedef struct _NODEB{
  PNODEA     ParentA;
  ULONG State;
  LIST_ENTRY NodeALinks;
  ...
} NODEB, *PNODEB;

#define CONTAINING_RECORD(addr, type, field)\
   ((type *)((PCHAR)(addr) -\
             (PCHAR)(&((type *)0)->field)))
```

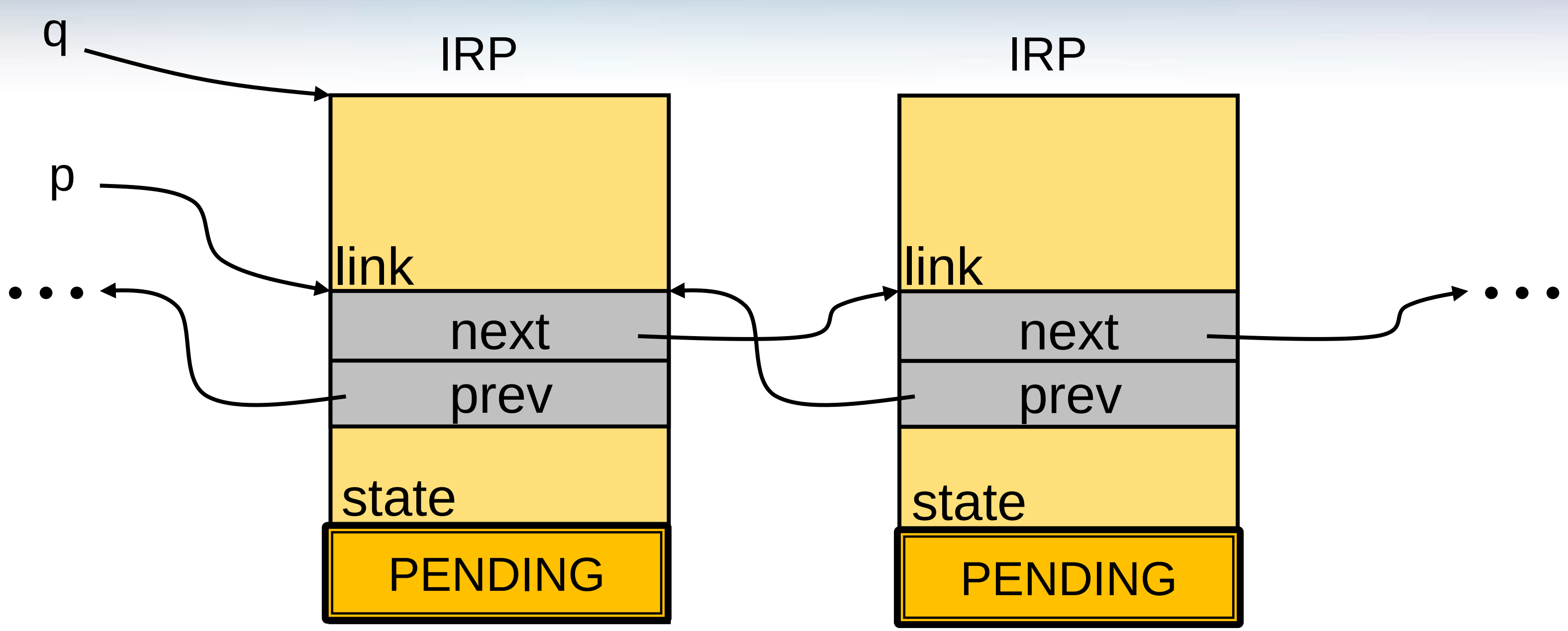
Doubly linked lists in Windows Kernel code



Representative shape graph
in Windows Kernel component

Microsoft Research

Precise and expressive heap reasoning



- Pointer Arithmetic
  q = CONTAINING_RECORD(p, IRP, link)
  = (IRP *) ((char*)p – (char*)(&(((IRP *)0)→link)))
- Transitive Closure
  Reach(next, u) ≡ {u, u->next, u->next->next, …}
  forall (x, Reach(next,p), CONTAINING_RECORD(x, IRP, link)->state == PENDING)

Annotation Language & Logic

- Procedure contracts
  - requires, ensures, modifies
- Arbitrary C expressions
  - program variables, resources
  - Boolean connectives
  - quantifiers
- Can express a rich set of contracts
  - API usage (e.g. lock acquire/release)
  - Synchronization protocols
  - Memory safety
  - Data structure invariants (linked list)
- Challenge:
  - Retain efficiency
  - Decidable fragments

```
__requires (NodeA != NULL)
..
__ensures  ((*PNodeB)->ParentA == NodeA)
__modifies (PNodeB)
void CompCreateNodeB
   (PNODEA NodeA,  PNODEB *PNodeB);
```

```
__requires (__forall(_H_x, __list1, __dataPtr(_H_
__requires (__setin(__head, __list1))
__ensures (__forall(_H_x, __list2, __initializedD
__modifies (__dataPtrSet(__list1))

void InitializeList() {
  LIST_ENTRY *iter;

  iter = pdata->list.Flink;

  __loop_invariant(
        __loop_assert (__setin(iter, __list1))
        __loop_assert (__forall(_H_x, __listBtwn(
        __loop_modifies (__old(__dataPtrSet(__lis
                       )
    while (iter != &pdata->list) {
  PDATA elem = CONTAINING_RECORD(iter, DATA, li
```

$$\frac{\ldots t_1)\quad t_1 \xrightarrow{f} t_2}{t_1 \xrightarrow{f} f(t_1) \xrightarrow{f} t_2}$$

[...WICH]

$$\frac{\xrightarrow{f} t_2 \xrightarrow{f} t_1}{t_1 = t_2}$$

[ORDER1]

$$\frac{t_1 \xrightarrow{f} t_2 \quad t_1 \xrightarrow{f} t_3}{t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3 \quad t_1 \xrightarrow{f} t_3 \xrightarrow{f} t_2}$$

[ORDER2]

$$\frac{t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3}{t_1 \xrightarrow{f} t_2,\ t_2 \xrightarrow{f} t_3}$$

[TRANSITIVE1]

$$\frac{t_1 \xrightarrow{f} t_2 \quad t_2 \xrightarrow{f} t_3}{t_1 \xrightarrow{f} t_3}$$

[TRANSITIVE2]

$$\frac{t_0 \xrightarrow{f} t_1 \xrightarrow{f} t_2 \quad t_1 \xrightarrow{f} t \xrightarrow{f} t_2}{t_0 \xrightarrow{f} t_1 \xrightarrow{f} t,\ t_0 \xrightarrow{f} t \xrightarrow{f} t_2}$$

[TRANSITIVE3]

$$\frac{t_0 \xrightarrow{f} t_1 \xrightarrow{f} t_2 \quad t_0 \xrightarrow{f} t \xrightarrow{f} t_1}{t_0 \xrightarrow{f} t \xrightarrow{f} t_2,\ t \xrightarrow{f} t_1 \xrightarrow{f} t_2}$$

Efficient logic for program verification

[REFLEXIVE] $$\frac{}{t \xrightarrow{f} t}$$

[STEP] $$\frac{f(t)}{t \xrightarrow{f} f(t)}$$

[REACH] $$\frac{f(t_1) \quad t_1 \xrightarrow{f} t_2}{t_1 = t_2 \quad t_1 \xrightarrow{f} f(t_1) \xrightarrow{f} t_2}$$

[CYCLE] $$\frac{f(t_1) = t_1 \quad t_1 \xrightarrow{f} t_2}{t_1 = t_2}$$

[SANDWICH] $$\frac{t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_1}{t_1 = t_2}$$

[ORDER1] $$\frac{t_1 \xrightarrow{f} t_2 \quad t_1 \xrightarrow{f} t_3}{t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3 \quad t_1 \xrightarrow{f} t_3 \xrightarrow{f} t_2}$$

[ORDER2] $$\frac{t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3}{t_1 \xrightarrow{f} t_2,\ t_2 \xrightarrow{f} t_3}$$

[TRANSITIVE1] $$\frac{t_1 \xrightarrow{f} t_2 \quad t_2 \xrightarrow{f} t_3}{t_1 \xrightarrow{f} t_3}$$

[TRANSITIVE2] $$\frac{t_0 \xrightarrow{f} t_1 \xrightarrow{f} t_2 \quad t_1 \xrightarrow{f} t \xrightarrow{f} t_2}{t_0 \xrightarrow{f} t_1 \xrightarrow{f} t,\ t_0 \xrightarrow{f} t \xrightarrow{f} t_2}$$

[TRANSITIVE3] $$\frac{t_0 \xrightarrow{f} t_1 \xrightarrow{f} t_2 \quad t_0 \xrightarrow{f} t \xrightarrow{f} t_1}{t_0 \xrightarrow{f} t \xrightarrow{f} t_2,\ t \xrightarrow{f} t_1 \xrightarrow{f} t_2}$$

- Logic with Reach, Quantifiers, Arithmetic
  - Expressive
  - Careful use of quantifiers

Encoding using quantifiers and triggers

```
// transitive2
axiom(forall f: [int]int, x: int, y: int, z: int, w: int :: {ReachBe
} ReachBetween(f, x, y, z) && ReachBetween(f, y, w, z) ==> ReachBetw
);

// transitive3
axiom(forall f: [int]int, x: int, y: int, z: int, w: int ::
      {ReachBetween(f, x, y, z), ReachBetween(f, x, w, y)}
         ReachBetween(f, x, y, z) && ReachBetween(f, x, w, y) ==>
           ReachBetween(f, x, w, z) && ReachBetween(f, w, y, z));
```

Success Story

- Used to check Windows Kernel code.
- Found 50 bugs, most confirmed.
  - 250 lines required to specify properties.
  - 600 lines of manual annotations.
  - 3000 lines of inferred annotations.

Houdini-like algorithm (Flanagan, Leino)

Extending Z3

- Axioms
- Inference rules (not supported yet)
- Very lazy loop
- New Z3 theory (too complicated for users)

Axioms

- Easy if theory can be encoded in first-order logic.
- Example: partial orders.
  - $\forall$ x: p(x,x)
  - $\forall$ x,y,z: p(x,y), p(y,z) $\Rightarrow$ p(x,z)
  - $\forall$ x,y: p(x,y), p(y,x) $\Rightarrow$ x = y
- Problems:
  - Is E-matching or SP a decision procedure for this theory?
  - Model extraction
  - Efficiency

Inference rules

- Some users (e.g., HAVOC) want to provide inference rules to Z3.
- More flexibility (e.g., side conditions)
- High level language for implementing custom decision procedures.

Very lazy loop

- Adding a theory T:
  1. Replace T symbols with uninterpreted symbols.
  2. Invoke Z3.
  3. If unsatisfiable, then return UNSAT.
  4. Inspect the model + implied equalities (i.e., assigned literals and equalities).
  5. Check if the assigned theory literals + equalities are satisfiable.
  6. If they are, then return SAT.
  7. Otherwise, add a new lemma and/or implied equalities go back to step 2.
- Model Based Theory Combination [SMT'08]

Very lazy loop (example)

¬p(a,f(c)) or ¬p(a, f(d))
p(a,b)
p(b,f(c))
c = d

z3

Model:
¬p(a,f(c)), p(a,b), p(b,f(c)),
b = c, f(c) = f(d)

unsat

T-Lemma:
¬ p(a,b) or ¬ p(b,f(c)) or p(a,f(c))

¬p(a,f(c)) or ¬p(a, f(d))
p(a,b)
p(b,f(c))
c = d
¬ p(a,b) or ¬ p(b,f(c)) or p(a,f(c)

z3

unsat

Verifying Garbage Collectors
- *Automatically and fast*

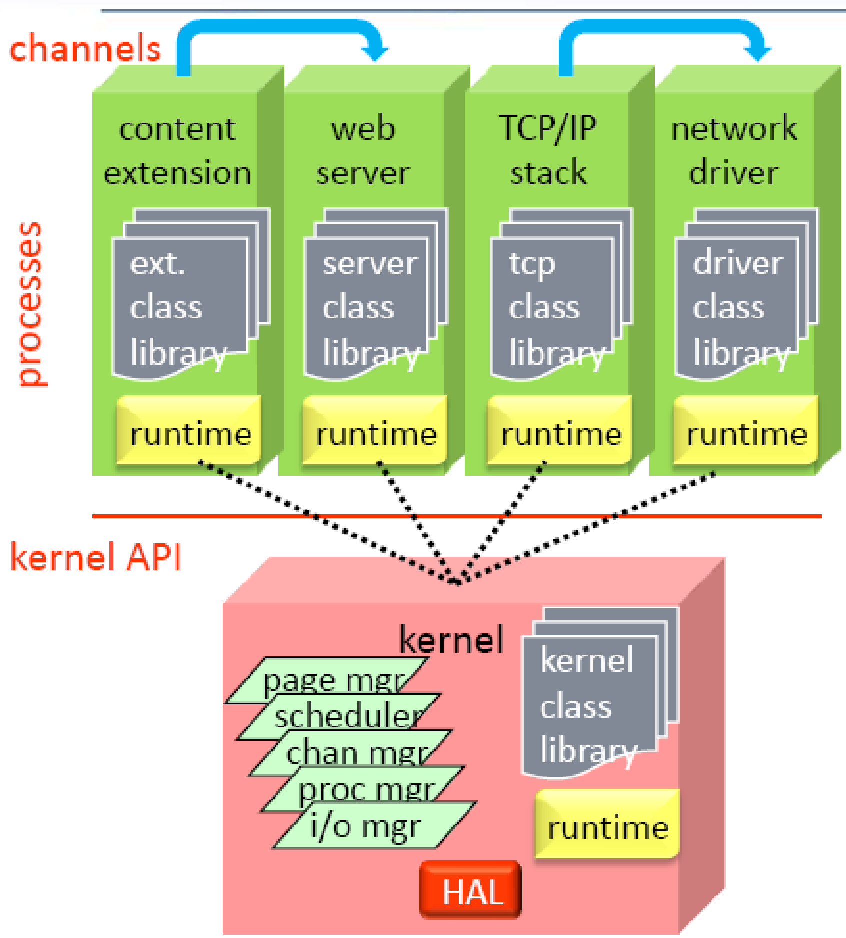
Chris Hawblitzel

http://www.codeplex.com/singularity/SourceControl/DirectoryView.aspx?SourcePath=%24%2fsingularity%2fbase%2fKernel%2fBartok%2fVerifiedGCs&changeSetId=14518

Context

Singularity

- Safe micro-kernel
  - 95% written in C#
  - all services and drivers in processes
- Software isolated processes (SIPs)
  - all user code is verifiably safe
  - some unsafe code in trusted runtime
  - processes and kernel sealed at execution
  - static verification replaces hardware protection
  - all SIPs run in ring 0
- Communication via channels
  - channel behavior is specified and checked
  - fast and efficient communication
- Working research prototype
  - not Windows replacement
  - shared source download



Context

Bartok

- MSIL → X86 Compiler

BoogiePL

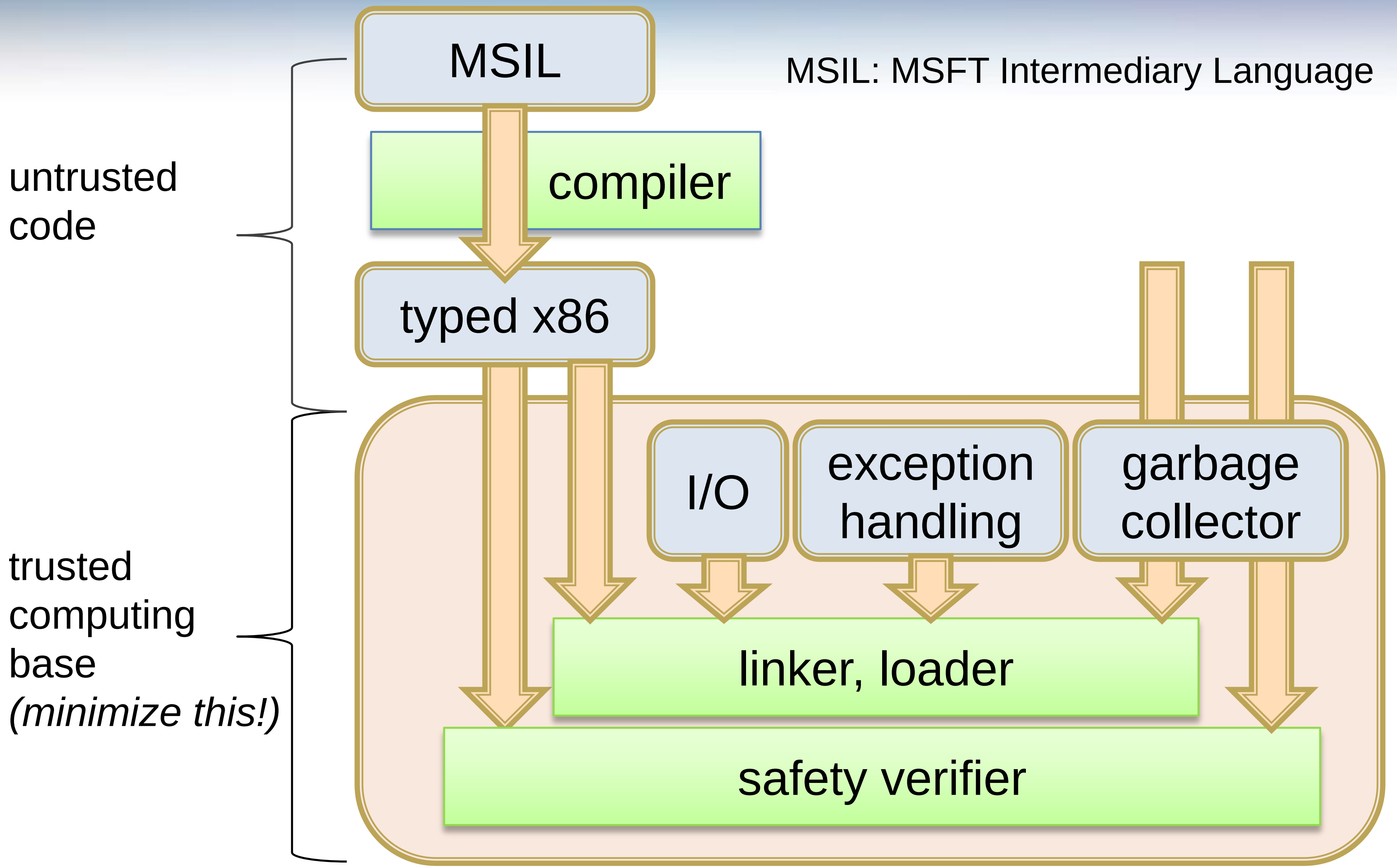
- Procedural low-level language
- Contracts
- Verification condition generator

Garbage Collectors

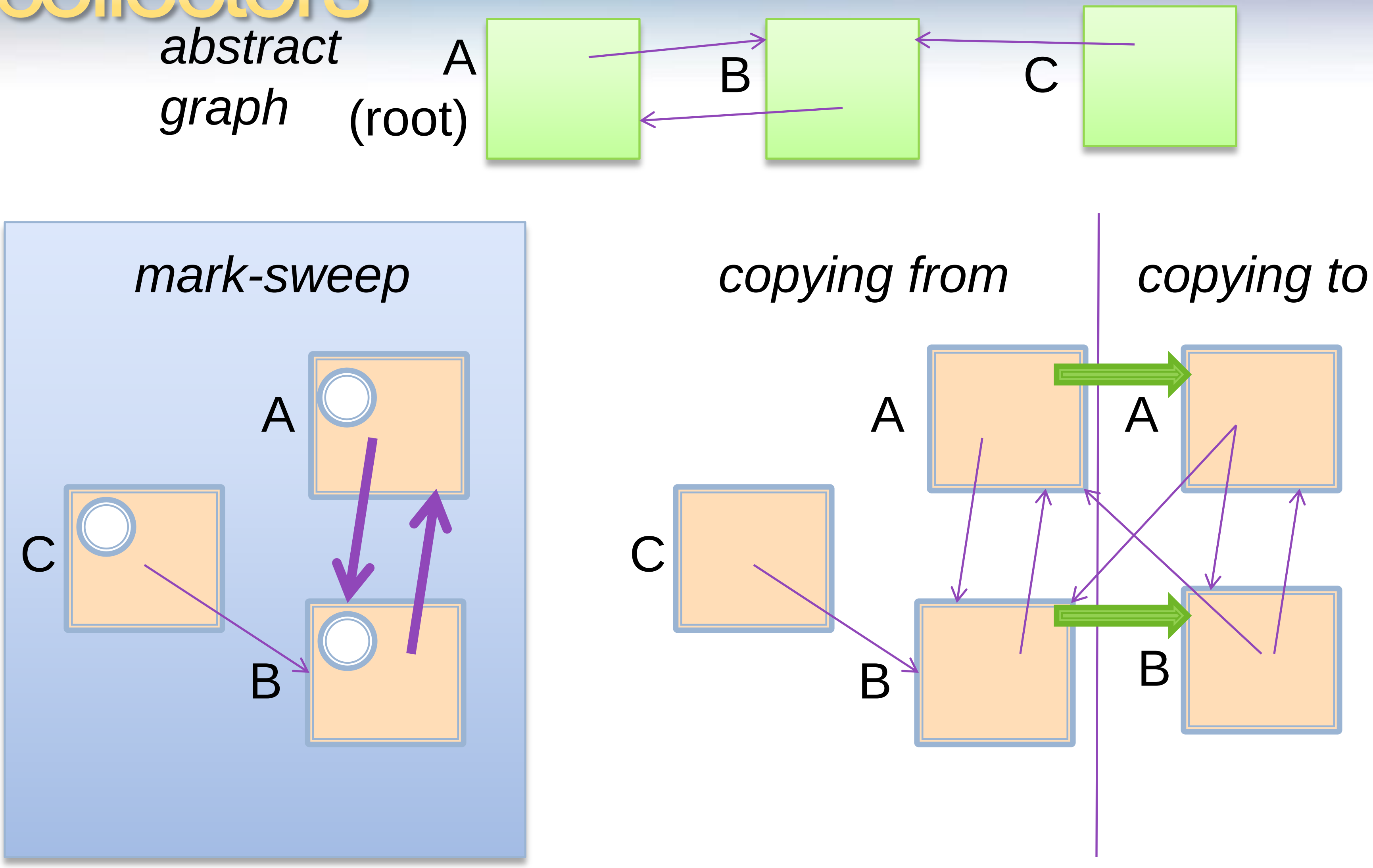
- Mark&Sweep
- Copying GC
- Verify small garbage collectors
  - more automated than interactive provers
  - borrow ideas from type systems for regions

Microsoft Research

Goal: safely run untrusted code



Mark-sweep and copying collectors



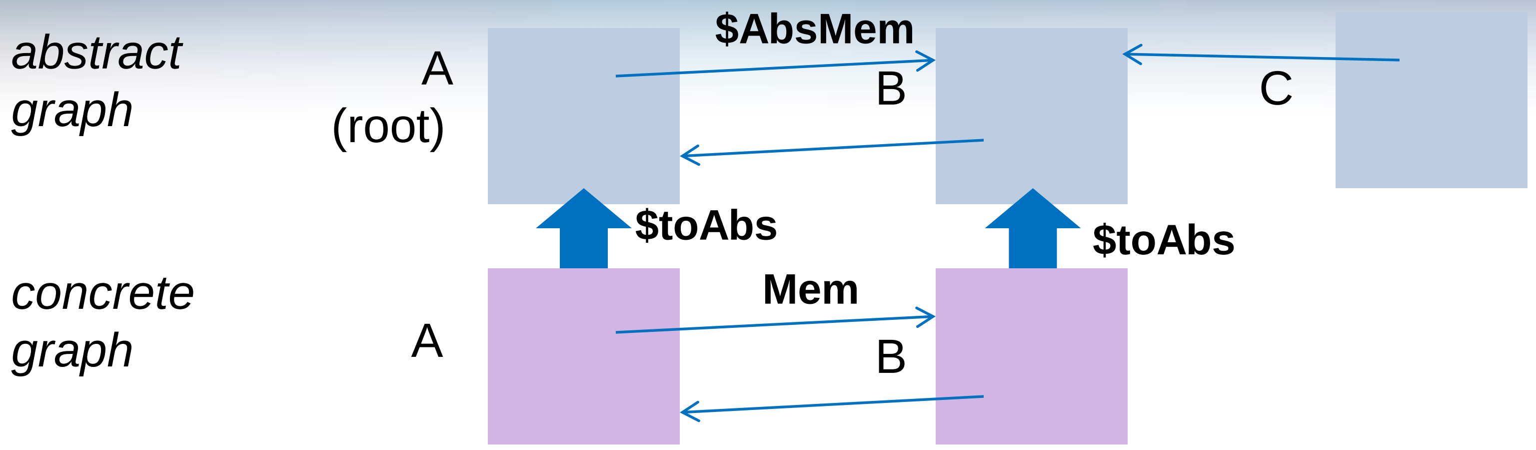
Garbage collector properties

- safety: gc does no harm
  - type safety
    - gc turns well-typed heap into well-typed heap
  - graph isomorphism
    - concrete graph represents abstract graph
- effectiveness
  - after gc, unreachable objects reclaimed
- termination
- efficiency

verified (safety, effectiveness)

not verified (termination, efficiency)

Proving safety



procedure GarbageCollectMs()
requires MsMutatorInv(root, Color, $toAbs, $AbsMem, Mem);
modifies Mem, Color, $toAbs;
ensure
{
call M
call S
}

function MsMutatorInv(...) returns (bool) {
WellFormed($toAbs) && memAddr(root) && $toAbs[root] != NO_ABS
&& (forall i:int::{memAddr(i)} memAddr(i) ==> ObjInv(i, $toAbs, $AbsMem, Mem))
&& (forall i:int::{memAddr(i)} memAddr(i) ==> White(Color[i]))
&& (forall i:int::{memAddr(i)} memAddr(i) ==> ($toAbs[i]==NO_ABS <==> Unalloc(Color[i])))}

function ObjInv(...) returns (bool) { memAddr(i) && $toAbs[i] != NO_ABS ==>
... $toAbs[Mem[i, field1]] != NO_ABS ...
... $toAbs[Mem[i, field1]] == $AbsMem[$toAbs[i], field1] ... }

Controlling quantifier instantiation

- Idea: use marker

```
function{:expand false} T(i:int) returns (bool) { true }
```

- Relativize quantifiers using marker

```
function GcInv(Color:[int]int, $toAbs:[int]int, $AbsMem:[int,int]int,
Mem:[int,int]int) returns (bool) {
   WellFormed($toAbs)
 && (forall i:int::{T(i)} T(i) ==> memAddr(i) ==>
      ObjInv(i, $toAbs, $AbsMem, Mem)
    && 0 <= Color[i] && Color[i] < 4
    && (Black(Color[i]) ==> !White(Color[Mem[i,0]]) && !White(Color[Mem[i,1]]))
    && ($toAbs[i] == NO_ABS <==> Unalloc(Color[i])))
}
```

Controlling quantifier instantiation

- Insert markers to enable triggers

```
procedure Mark(ptr:int)
  requires GcInv(Color, $toAbs, $AbsMem, Mem);
  requires memAddr(ptr) && T(ptr);
  requires $toAbs[ptr] != NO_ABS;
  modifies Color;
  ensures  GcInv(Color, $toAbs, $AbsMem, Mem);
  ensures  (forall i:int::{T(i)} T(i) ==> !Black(Color[i]) ==> Color[i] == old(Color)[i]);
  ensures  !White(Color[ptr]);
{
  if (White(Color[ptr])) {
    Color[ptr] := 2; // make gray
    call Mark(Mem[ptr,0]);
    call Mark(Mem[ptr,1]);
    Color[ptr] := 3; // make black
  }
}
```

Can we do better?

Decidable Fragments

- EPR (Effectively Propositional)
  - Aka: **Bernays–Schönfinkel class**
- Stratified EPR
- Array Property Fragment
- Stratified Array Property Fragment

It can be used to verify the GC properties!

EPR

- Prefix $\exists^*\forall^*$ + no function symbols.
- Examples:
  - $\forall$x,y,z: $\neg$p(x,y) or $\neg$p(y,z) or p(x,z)
  - $\forall$x: $\neg$p(x,a) or $\neg$q(x,b)
- Why is it useful?
  - Model checking problems
  - QBF
  - Finite model finding
  - Useful theories: partial orders.

EPR: decidability

- Finite Herbrand Universe.

∀x: ¬p(x,a) or ¬q(x,b)
∀x: p(x,x)
p(c,a) or q(c, b)

Herbrand Universe
{a, b, c}

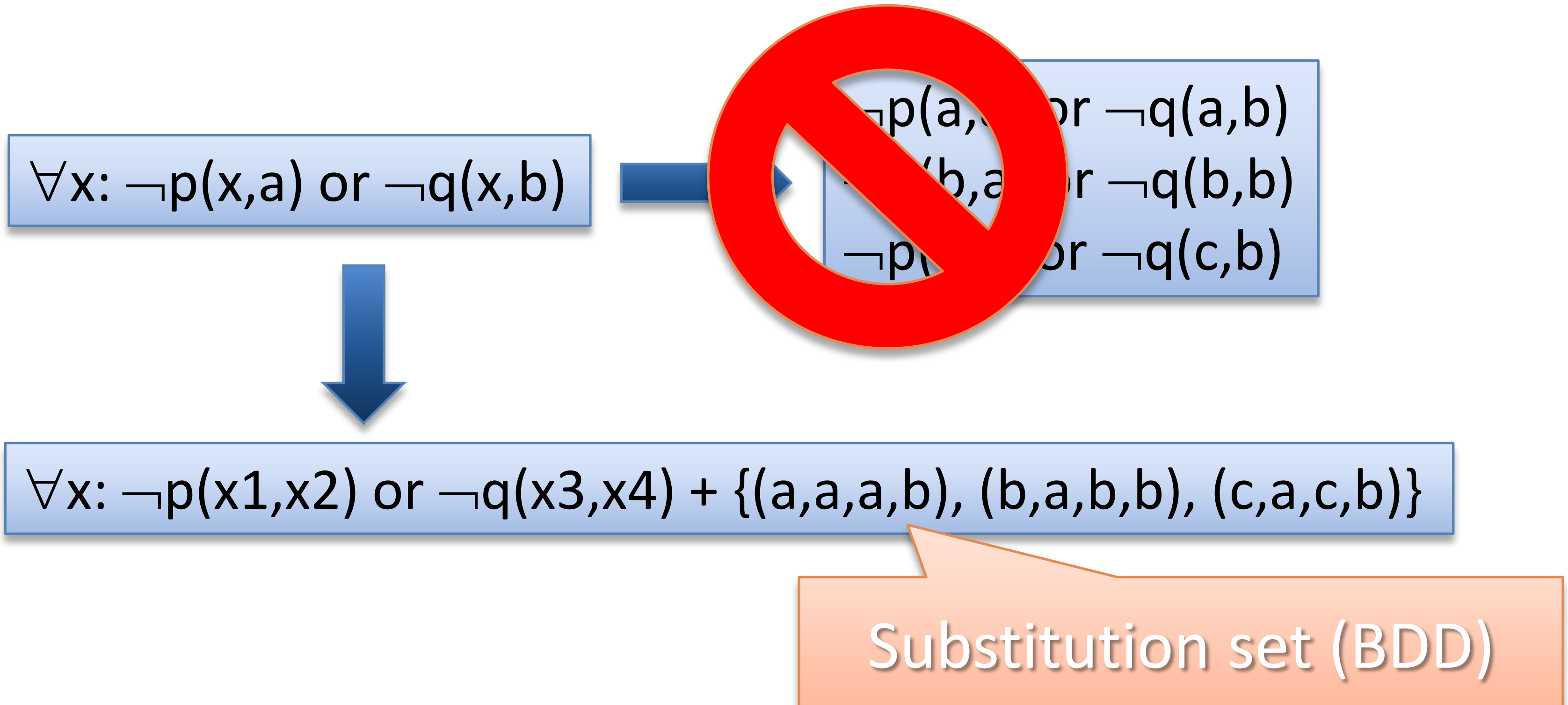
¬p(a,a) or ¬q(a,b)
¬p(b,a) or ¬q(b,b)
¬p(c,a) or ¬q(c,b)
p(a,a)
p(b,b)
p(c,c)
p(c,a) or q(c, b)

Exponential blowup

SAT-solver

EPR: efficient implementation

- DPLL(SX) calculus: DPLL + substitution sets (BDDs) [IJCAR’08]



Stratified EPR

- Many sorted first order logic.
- $S_1 < S_2$ if there is a function f : … S_1 … → S_2
- A formula is stratified if there is no sort S s.t. $S < S$
- A stratified formula has a finite Herbrand Universe.
- Example:

$\forall x\ S_1$: f(g(x)) = a
g(b) = c
where:
g : $S_1 \rightarrow S_2$
f : $S_2 \rightarrow S_3$
a : S_3
b : S_1
c : S_2

→

Herbrand Universe:
{ a, b, c,
g(b), f(g(b)), f(c)}

Stratified EPR and Unsorted Logic

- Sort inference + restrictions
- Problematic example:

$\forall$x, y: f(x) != f(y) or x = y
$\forall$x: f(x) != c
$\forall$x: x = a

$\forall$x S_1, y S_1: f(x) != f(y) or x = y
$\forall$x S_1: f(x) != c
$\forall$x S_3: x = a
f : $S_1 \rightarrow S_2$
c : S_2
a : S_3

Cardinality Constraint

Almost there…

```
(forall i:int::{T(i)} T(i) ==> memAddr(i) ==>
      ObjInv(i, $toAbs, $AbsMem, Mem)
    && 0 <= Color[i] && Color[i] < 4
    && (Black(Color[i]) ==> !White(Color[Mem[i,0]]) && !White(Color[Mem[i,1]]))
    && ($toAbs[i] == NO_ABS <==> Unalloc(Color[i])))
```

⬇

```
(forall i: Addr
      ObjInv(i, $toAbs, $AbsMem, Mem)
    && (color[i] = black or color[i] = white or color[i] = gray)
    && (Black(color[i]) ==> !White(color[Mem[i,f0]]) && !White(Color[Mem[i,f1]]))
    && ($toAbs[i] == NO_ABS <==> Unalloc(Color[i])))
```

Array Property Fragment (APF)

- $\forall i_1, \ldots, i_n: F[i_1, \ldots, i_n],$
- F is in NNF, then the following atoms can contain universal variables:
  - $i_k > t$ (t is ground)
  - $i_k > i_{k'}$
  - i_k != t (t is ground)
  - i_k != $i_{k'}$
  - $L[a[i_k]]$ (i_k only appears in $a[i_k]$)

Examples

- Array is sorted:
  - $\forall i,j$: $i <= j$ implies $a[i] <= a[j]$, or equivalently:
  - $\forall i,j$: $i > j$ or $a[i] <= a[j]$
- Array update $b = write(a, j, v)$
  - $b[j] = v$
  - $\forall x$: $x > j-1$ or $b[x] = a[x]$
  - $\forall x$: $x < j+1$ or $b[x] = a[x]$

Equivalent to:
$\forall x$: $x = j$ or $b[x] = a[x]$

Stratified APF

- Yeting Ge (Intern 2008)
- Nested (stratified) arrays in APF.
- Stratified EPR + some arithmetic.
- Example:
  - $\forall i,j$: i <= j implies $a[a'[i]] <= a[a'[j]]$
- It supports other extensions for pointer arithmetic.

Conclusion

- Users frequently need new theories.
  - Quantifiers.
  - Inference rules.
  - Very lazy loop.
- Decidable fragments are useful in practice.
- http://research.microsoft.com/projects/z3

Thank You!

Is Z3 available for commercial use?

- Not yet…
- However,
  - PEX (comes with Z3) and Chess will be available for commercial use for VS users.
    - http://research.microsoft.com/Pex/
    - http://research.microsoft.com/projects/chess/
  - SLAM/SDV 2.0 (comes with Z3) is part of DDK and will ship with the next version of Windows.
    - http://research.microsoft.com/slam/