The Why/Krakatoa/Caduceus Platform for Deductive Program Verification

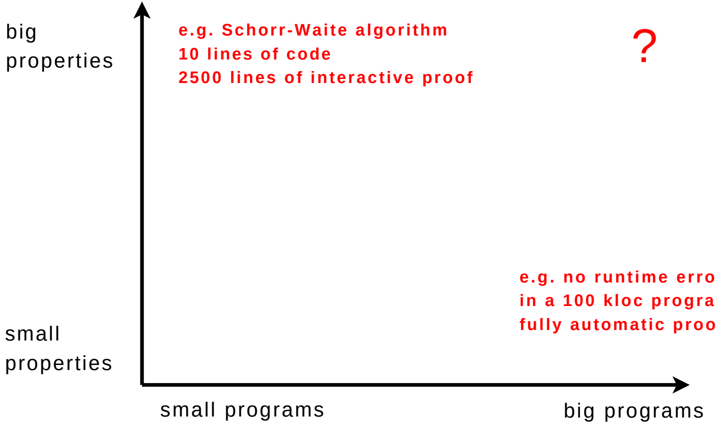
Jean-Christophe Filliâtre

CNRS
Orsay, France

The ProVal project — `http://proval.lri.fr/`

- general **goal**: prove behavioral properties of **pointer programs**
- pointer program = program manipulating data structures with **in-place mutable fields**
- we currently focus on **C** and **Java** programs

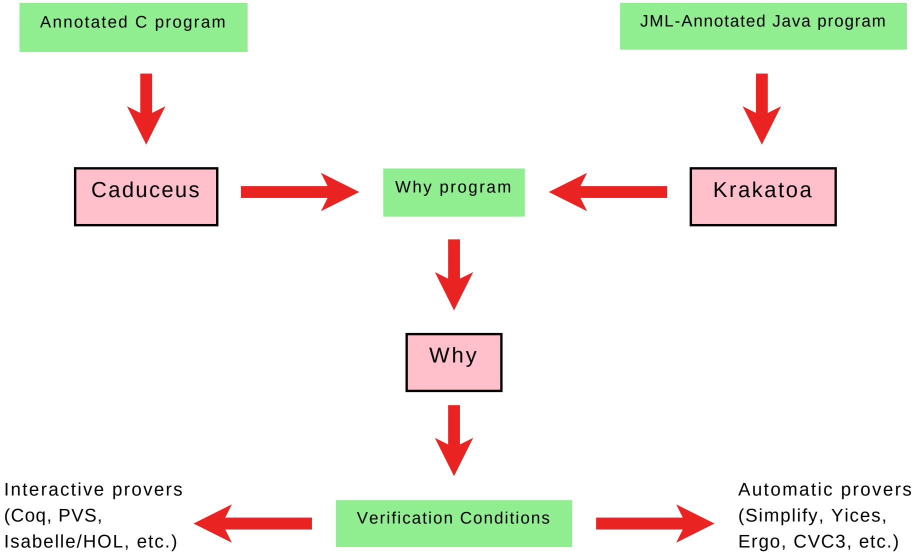
What kind of properties



Principles

- specification as **annotations** at the source code level
  - JML (Java Modeling Language) for Java
  - our own language for C (mostly JML-inspired)
- generation of **verification conditions** (VCs)
  - using Hoare logic / weakest preconditions
  - other similar approaches: static verification (ESC/Java, SPEC#), B method, etc.
- **multi-prover** approach
  - off-the-shelf provers, as many as possible
  - automatic provers (Simplify, Yices, Ergo, etc.)
  - proof assistants (Coq, PVS, Isabelle/HOL, etc.)

Platform Overview



Outline

1. specification languages
   - how to formally specify behaviors
2. generation of VCs
   - models of program execution
3. discharging VCs
   - multi-prover approach

part I

Formally Specifying Pointer Programs

Which language to specify behaviors?

Java already has a specification language: **JML** (Java Modeling Language) used in runtime assertion checking tools, ESC/Java, JACK, LOOP, CHASE

JML allows to specify

- precondition, postcondition and side-effects for methods
- invariant and variant for loops
- class invariants
- model fields (~ ghost code)

Which language to specify behaviors?

we designed a similar language for **C programs**, largely inspired by JML

additional features:

- pointer arithmetic
- algebraic models
  - any axiomatized theory can be used in specifications
  - no runtime assertion checking
- floating-point arithmetic
  - round errors can be specified

A First Example: Binary Search

binary search: search a sorted array of integers for a given value

famous example; see J. Bentley's *Programming Pearls*:
most programmers are wrong on their first attempt to write binary search

Binary Search (code)

```
int binary_search(int* t, int n, int v) {
  int l = 0, u = n-1, p = -1;
  while (l <= u ) {
    int m = (l + u) / 2;
    if (t[m] < v)
      l = m + 1;
    else if (t[m] > v)
      u = m - 1;
    else {
      p = m; break;
    }
  }
  return p;
}
```

Binary Search (spec)

we want to prove:

1. absence of runtime error
2. termination
3. behavioral correctness

Binary Search (spec)

```
/*@ requires
  @   n >= 0 &&
  @   \valid_range(t,0,n-1) &&
  @   \forall int k1, int k2;
  @     0 <= k1 <= k2 <= n-1 => t[k1] <= t[k2]
  @*/
int binary_search(int* t, int n, int v) {
  ...
}
```

Binary Search (spec)

```
/*@ requires
  @   n >= 0 &&
  @   \valid_range(t,0,n-1) &&
  @   \forall int k1, int k2;
  @     0 <= k1 <= k2 <= n-1 => t[k1] <= t[k2]
  @ ensures
  @   (\result >= 0 && t[\result] == v) ||
  @   (\result == -1 && \forall int k;
  @                            0 <= k < n => t[k] != v)
  @*/
int binary_search(int* t, int n, int v) {
  ...
}
```

Binary Search (spec)

```
/*@ requires ...
  @ ensures  ...
  @*/
int binary_search(int* t, int n, int v) {
  int l = 0, u = n-1, p = -1;
  /*@ variant u-l
    @*/
  while (l <= u ) {
    ...
  }
}
```

Binary Search (spec)

```
/*@ requires ...
  @ ensures  ...
  @*/
int binary_search(int* t, int n, int v) {
  int l = 0, u = n-1, p = -1;
  /*@ invariant
    @   0 <= l && u <= n-1 && p == -1 &&
    @   \forall int k;
    @     0 <= k < n => t[k] == v => l <= k <= u
    @ variant u-l
    @*/
  while (l <= u ) {
    ...
  }
}
```

Binary Search (proof)

DEMO

Algebraic Models

in JML, annotations are written using **pure Java code**
this is mandatory to perform **runtime assertion checking**

but it is often convenient to introduce **axiomatized theories** in order to annotate programs, that is

- abstract types
- function symbols, w or w/o definitions
- predicates, w or w/o definitions
- axioms

Example: Priority Queues

static data structure for a **priority queue** containing integers

```
void clear();      // empties the queue
void push(int x); // inserts a new element
int  max();        // returns the maximal element
int  pop();        // removes and returns the maximal element
```

Bags

```
//@ type bag

//@ logic bag empty_bag()

//@ logic bag singleton_bag(int x)

//@ logic bag union_bag(bag b1, bag b2)

/*@ logic bag add_bag(int x, bag b)
  @   { union_bag(b, singleton_bag(x)) } */

//@ logic int occ_bag(int x, bag b)

/*@ predicate is_max_bag(bag b, int m) {
  @   occ_bag(m, b) >= 1 &&
  @   \forall int x; occ_bag(x,b) >= 1 => x <= m
  @ } */
```

Bags

```
//@ type bag

//@ logic bag empty_bag()

//@ logic bag singleton_bag(int x)

//@ logic bag union_bag(bag b1, bag b2)

/*@ logic bag add_bag(int x, bag b)
  @   { union_bag(b, singleton_bag(x)) } */

//@ logic int occ_bag(int x, bag b)

/*@ predicate is_max_bag(bag b, int m) {
  @   occ_bag(m, b) >= 1 &&
  @   \forall int x; occ_bag(x,b) >= 1 => x <= m
  @ } */
```

Bags

```
//@ type bag

//@ logic bag empty_bag()

//@ logic bag singleton_bag(int x)

//@ logic bag union_bag(bag b1, bag b2)

/*@ logic bag add_bag(int x, bag b)
  @   { union_bag(b, singleton_bag(x)) } */

//@ logic int occ_bag(int x, bag b)

/*@ predicate is_max_bag(bag b, int m) {
  @   occ_bag(m, b) >= 1 &&
  @   \forall int x; occ_bag(x,b) >= 1 => x <= m
  @ } */
```

Bags

```
//@ type bag

//@ logic bag empty_bag()

//@ logic bag singleton_bag(int x)

//@ logic bag union_bag(bag b1, bag b2)

/*@ logic bag add_bag(int x, bag b)
  @   { union_bag(b, singleton_bag(x)) } */

//@ logic int occ_bag(int x, bag b)

/*@ predicate is_max_bag(bag b, int m) {
  @   occ_bag(m, b) >= 1 &&
  @   \forall int x; occ_bag(x,b) >= 1 => x <= m
  @ } */
```

Bags

```
//@ type bag

//@ logic bag empty_bag()

//@ logic bag singleton_bag(int x)

//@ logic bag union_bag(bag b1, bag b2)

/*@ logic bag add_bag(int x, bag b)
  @   { union_bag(b, singleton_bag(x)) } */

//@ logic int occ_bag(int x, bag b)

/*@ predicate is_max_bag(bag b, int m) {
  @   occ_bag(m, b) >= 1 &&
  @   \forall int x; occ_bag(x,b) >= 1 => x <= m
  @ } */
```

Bags

```
//@ type bag

//@ logic bag empty_bag()

//@ logic bag singleton_bag(int x)

//@ logic bag union_bag(bag b1, bag b2)

/*@ logic bag add_bag(int x, bag b)
  @   { union_bag(b, singleton_bag(x)) } */

//@ logic int occ_bag(int x, bag b)

/*@ predicate is_max_bag(bag b, int m) {
  @   occ_bag(m, b) >= 1 &&
  @   \forall int x; occ_bag(x,b) >= 1 => x <= m
  @ } */
```

Bags

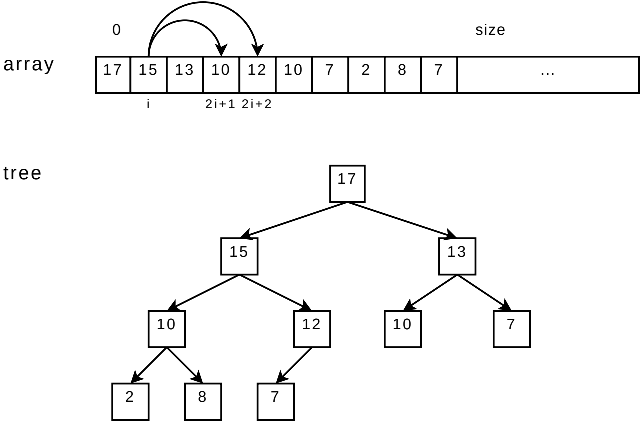
```
//@ type bag

//@ logic bag empty_bag()

//@ logic bag singleton_bag(int x)

//@ logic bag union_bag(bag b1, bag b2)

/*@ logic bag add_bag(int x, bag b)
  @   { union_bag(b, singleton_bag(x)) } */

//@ logic int occ_bag(int x, bag b)

/*@ predicate is_max_bag(bag b, int m) {
  @   occ_bag(m, b) >= 1 &&
  @   \forall int x; occ_bag(x,b) >= 1 => x <= m
  @ } */
```

Priority Queues (spec)

```
//@ logic bag model()

//@ ensures model() == empty_bag()
void clear();

//@ ensures model() == add_bag(x, \old(model()))
void push(int x);

//@ ensures is_max_bag(model(), \result)
int max();

/*@ ensures  is_max_bag(\old(model()), \result) &&
  @          \old(model()) == add_bag(\result, model())  */
int pop();
```

Implementing Priority Queues

implementation: heap encoded in an array



bag { 2, 7, 7, 8, 10, 10, 12, 13, 15, 17 }

Trees

```
//@ type tree

//@ logic tree Empty()

//@ logic tree Node(tree l, int x, tree r)
```

Heaps

```
//@ predicate is_heap(tree t)

//@ axiom is_heap_def_1: is_heap(Empty())

...
```

Trees and Bags

```
//@ logic bag bag_of_tree(tree t)

/*@ axiom bag_of_tree_def_1:
  @   bag_of_tree(Empty()) == empty_bag()
  @*/

...
```

Trees and Arrays

```
//@ logic tree tree_of_array(int *t, int root, int bound)

/*@ axiom tree_of_array_def_2:
  @   \forall int *t; \forall int root; \forall int bound;
  @      0 <= root < bound =>
  @      tree_of_array(t, root, bound) ==
  @      Node(tree_of_array(t, 2*root+1, bound),
  @           t[root],
  @           tree_of_array(t, 2*root+2, bound))
  @*/
```

...

Priority Queues (spec)

```
#define MAXSIZE 100

int heap[MAXSIZE];

int size = 0;

//@ invariant size_inv : 0 <= size < MAXSIZE

//@ invariant is_heap: is_heap(tree_of_array(heap, 0, size))

/*@ logic bag model()
  @   { bag_of_tree(tree_of_array(heap, 0, size)) } */
```

Another (More Challenging) Example

challenge for **the verified program of the month**:

```
t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)for(f=0;d=(e-=d)&-e;f+=t(a-d,(b+d)*2,(
c+d)/2));return f;}main(q){scanf("%d",&q);printf("%d\n",t(~(~0<<q),0,0));}
```

appears on a web page collecting C signature programs

due to Marcel van Kervinck (author of MSCP, a chess program)

Unobfuscating...

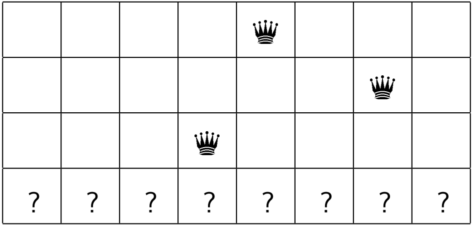
```
int t(int a, int b, int c) {
  int d, e=a&~b&~c, f=1;
  if (a)
    for (f=0; d=e&-e; e-=d)
      f += t(a-d, (b+d)*2, (c+d)/2);
  return f;
}

int main(int q) {
  scanf("%d", &q);
  printf("%d\n", t(~(~0<<q), 0, 0));
}
```

this program reads an integer n
and prints the number of solutions to the n-queens problem

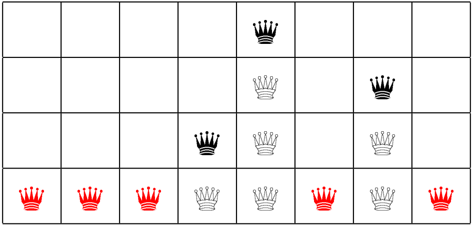
How does it work?

- backtracking algorithm (no better way to solve the n-queens)
- integers used as **sets** (bit vectors)

integers	sets
`0`	$\emptyset$
`a&b`	$a \cap b$
`a+b`	$a \cup b$, when $a \cap b = \emptyset$
`a-b`	$a \setminus b$, when $b \subseteq a$
`~a`	$\complement a$
`a&-a`	$min\_elt(a)$, when $a \neq \emptyset$
`~(~0<<n)`	$\{0, 1, \ldots, n-1\}$
`a*2`	$\{i + 1 \mid i \in a\}$, written $S(a)$
`a/2`	$\{i - 1 \mid i \in a \wedge i \neq 0\}$, written $P(a)$

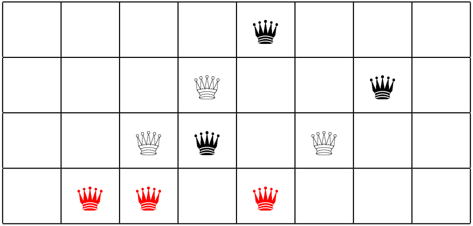


What a, b and c mean



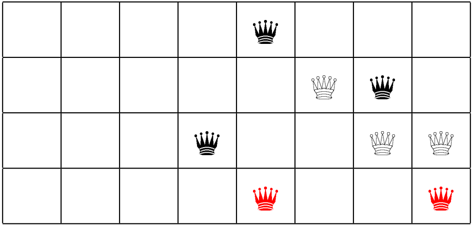
a = columns to be filled $= 11100101_2$

What a, b and c mean



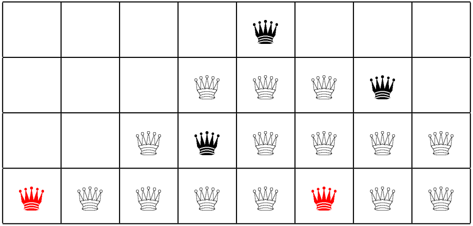
b = positions to avoid because of left diagonals = 01101000_2

What a, b and c mean



c = positions to avoid because of right diagonals = 00001001_2

What a, b and c mean



$a\&\tilde{}b\&\tilde{}c$ = positions to try = 10000100_2

Now it is clear

```
int t(int a, int b, int c) {
  int d, e=a&~b&~c, f=1;
  if (a)
    for (f=0; d=e&-e; e-=d)
      f += t(a-d, (b+d)*2, (c+d)/2);
  return f;
}

int queens(int n) {
  return t(~(~0<<n), 0, 0);
}
```

Abstract finite sets

```
//@ type iset

//@ predicate in_(int x, iset s)

/*@ predicate included(iset a, iset b)
  @  { \forall int i; in_(i,a) => in_(i,b) } */

//@ logic iset empty()

//@ axiom empty_def : \forall int i; !in_(i,empty())
```

...

total: **66 lines** of functions, predicates and axioms

C ints as abstract sets

```
//@ logic iset iset(int x)

/*@ axiom iset_c_zero : \forall int x;
  @   iset(x)==empty() <=> x==0 */

/*@ axiom iset_c_min_elt :
  @   \forall int x; x != 0 =>
  @     iset(x&-x) == singleton(min_elt(iset(x))) */

/*@ axiom iset_c_diff : \forall int a, int b;
  @   iset(a&~b) == diff(iset(a), iset(b)) */
```

...

total: **27 lines**

Termination

```
int t(int a, int b, int c){
  int d, e=a&~b&~c, f=1;
  if (a)
    //@ variant card(iset(e-d))
    for (f=0; d=e&-e; e-=d) {
      f += t(a-d,(b+d)*2,(c+d)/2);
    }
  return f;
}
```

3 verification conditions, all proved automatically

similarly for the termination of the recursive function:
7 verification conditions, all proved automatically

Soundness

how to express that we compute the right number,
since the program is not storing anything,
not even the current solution?

answer: by introducing **ghost code** to perform the missing operations

Ghost code

ghost code can be regarded as regular code, as soon as

- ghost code does not modify program data
- program code does not access ghost data

ghost data is purely logical $\Rightarrow$ ne need to check the validity of pointers

Program instrumented with ghost code

```
//@ int** sol;
//@ int s;
//@ int* col;
//@ int k;

int t(int a, int b, int c) {
  int d, e=a&~b&~c, f=1;
  if (a)
    for (f=0; d=e&-e; e-=d) {
      //@ col[k] = min_elt(d);
      //@ k++;
      f += t3(a-d, (b+d)*2, (c+d)/2);
      //@ k--;
    }
  //@ else
  //@   store_solution();
  return f;
}
```

Program instrumented with ghost code (cont'd)

```
/*@ requires solution(col)
  @ assigns  s, sol[s][0..N()-1]
  @ ensures  s==\old(s)+1 && eq_sol(sol[\old(s)], col)
  @*/
void store_solution();

/*@ requires
  @   n == N() && s == 0 && k == 0
  @ ensures
  @   \result == s &&
  @   sorted(sol, 0, s) &&
  @   \forall int* t; solution(t) <=>
  @       (\exists int i; 0<=i<\result && eq_sol(t,sol[i]))
  @*/
int queens(int n) { return t(~(~0<<n), 0, 0); }
```

Finally, we get...

256 lines of code and specification

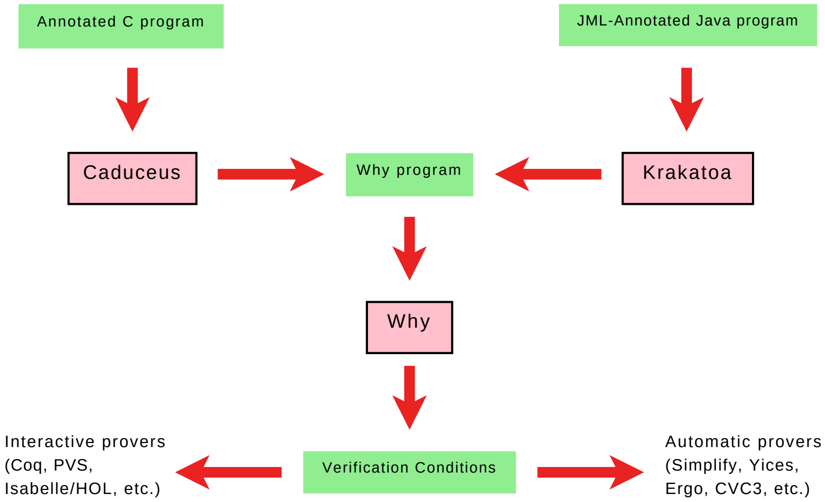
regarding VCs:

- main function `queens`: **15** verification conditions
  - **all** proved automatically (Simplify, Ergo or Yices)
- recursive function `t`: **51** verification conditions
  - **42** proved automatically: 41 by Simplify, 37 by Ergo and 35 by Yices
  - **9** proved manually using Coq (and Simplify)

part II

Generating the Verification Conditions

Generating the Verification Conditions



Why: a Verification Condition Generator

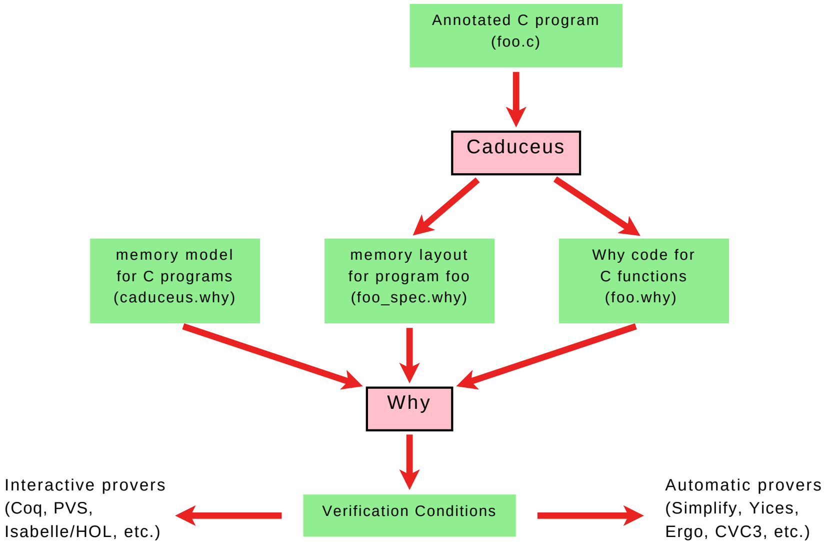
Why is a **verification condition generator** for a language with

- variables containing pure values, no alias (~ Hoare-logic language)
- usual control structures (loops, tests, etc.)
- exceptions
- (possibly recursive) functions
- polymorphic first-order logic with equality and arithmetic

Why is similar to Boogie (SPEC# project)

Why is also responsible for **translating** verification conditions to the **native logics** of all provers

Generating the Verification Conditions



To Pointer Programs to Alias-Free Programs

naive idea: model the **memory as a big array**

using the theory of arrays

$$\texttt{acc} : \texttt{mem}, \texttt{int} \rightarrow \texttt{int}$$
$$\texttt{upd} : \texttt{mem}, \texttt{int}, \texttt{int} \rightarrow \texttt{mem}$$

$$\forall m\, p\, v,\ \texttt{acc}(\texttt{upd}(m, p, v), p) = v$$
$$\forall m\, p_1\, p_2\, v,\ p_1 \neq p_2 \Rightarrow \texttt{acc}(\texttt{upd}(m, p_1, v), p_2) = \texttt{acc}(m, p_2)$$

Naive Memory Model

then the C program

```
int x;
int y;
x = 0;
y = 1;
//@ assert x == 0
```

becomes

$$m := \mathtt{upd}(m, x, 0);$$
$$m := \mathtt{upd}(m, y, 1);$$
$$\mathtt{assert}\ \mathtt{acc}(m, x) = 0$$

the verification condition is

$$\mathtt{acc}(\mathtt{upd}(\mathtt{upd}(m, x, 0), y, 0), x) = 0$$

Memory Model for Pointer Programs

we use the **component-as-array** model (Burstall-Bornat)

each structure/object field is mapped to a different array

relies on the property **"two different fields cannot be aliased"**

strong consequence: prevents pointer casts and unions (a priori)

Benefits of the Component-As-Array Model

```
struct S { int x; int y; } p;
...
p.x = 0;
p.y = 1;
//@ assert p.x == 0
```

becomes

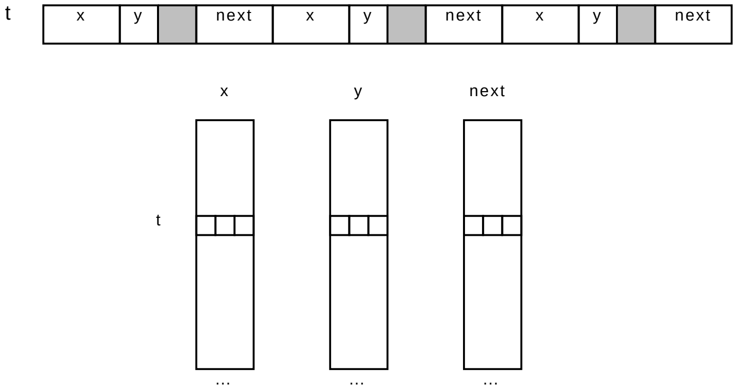
$$\begin{array}{lcl} x & := & \mathtt{upd}(x, p, 0); \\ y & := & \mathtt{upd}(y, p, 1); \\ \multicolumn{3}{l}{\mathtt{assert}\ \mathtt{acc}(x, p) = 0} \end{array}$$

the verification condition is

$$\mathtt{acc}(\mathtt{upd}(x, p, 0), p) = 0$$

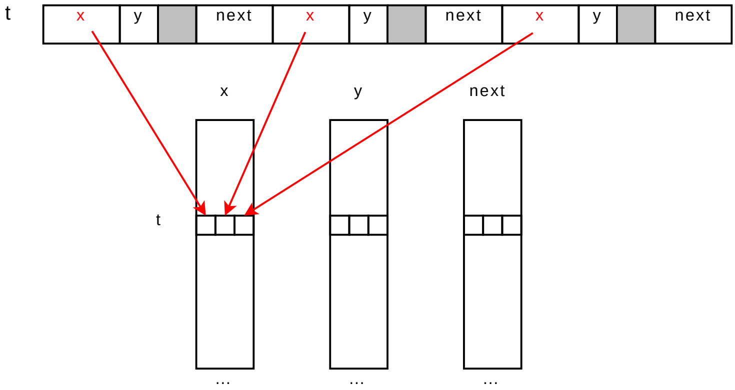
Component-As-Array Model and Pointer Arithmetic

```
struct S { int x; short y; struct S *next; } t[3];
```



Component-As-Array Model and Pointer Arithmetic

```
struct S { int x; short y; struct S *next; } t[3];
```



Separation Analysis

on top of Burstall-Bornat model, we add some **separation analysis**

- each pointer is assigned a **zone**
- zones are **unified** when pointers are assigned / compared
- functions are **polymorphic** wrt zones

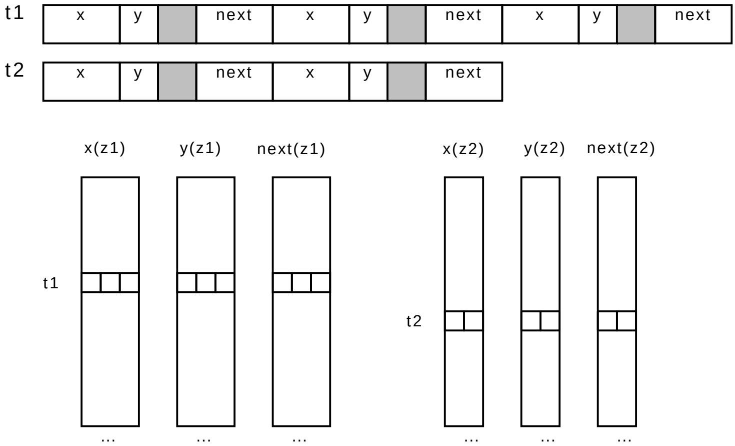
similar to ML-type inference

then the component-as-array model is refined according to zones

Separation Analysis for Deductive Verification (HAV'07)

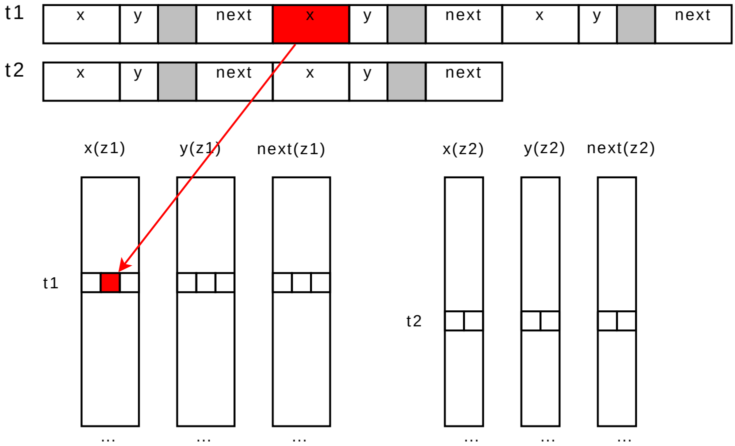
Separation Analysis

```
struct S { int x; short y; struct S *next; } t1[3], t2[2];
```



Separation Analysis

```
struct S { int x; short y; struct S *next; } t1[3], t2[2];
```



Example

little challenge for program verification proposed by P. Müller:

> *count the number n of non-zero values in an integer array t, then copy these values in a freshly allocated array of size n*

`t`	2	1	0	4	0	5	3	0

`count`=5

`u`	2	1	4	5	3

P. Müller's Example (code)

```
void m(int t[], int length) {
  int count=0, i, *u;

  for (i=0 ; i < length; i++)
    if (t[i] > 0) count++;

  u = (int *)calloc(count,sizeof(int));
  count = 0;

  for (i=0 ; i < length; i++)
    if (t[i] > 0) u[count++] = t[i];
}
```

P. Müller's Example (spec)

```
void m(int t[], int length) {
  int count=0, i, *u;
  //@ invariant count == num_of_pos(0,i-1,t) ...
  for (i=0 ; i < length; i++)
    if (t[i] > 0) count++;
  //@ assert count == num_of_pos(0,length-1,t)
  u = (int *)calloc(count,sizeof(int));
  count = 0;
  //@ invariant count == num_of_pos(0,i-1,t) ...
  for (i=0 ; i < length; i++)
    if (t[i] > 0) u[count++] = t[i];
}
```

P. Müller's Example (proof)

12 verification conditions

- without separation analysis: 10/12 automatically proved
- with separation analysis: 12/12 automatically proved

DEMO

part III

Discharging the Verification Conditions

Which Provers

we want to use **off-the-shelf provers**, as many as possible

requirements

- first-order logic
- equality and arithmetic
- quantifiers (memory model, user algebraic models)

Provers Currently Supported

automatic decision procedures

- provers *a la* Nelson-Oppen
  - Simplify, Yices, Ergo
  - CVC Lite, CVC3
- resolution based provers
  - harvey, rv-sat
  - Zenon

interactive proof assistants

- Coq, PVS, Isabelle/HOL
- HOL4, HOL Light, Mizar

Typing Issues

verification conditions are expressed in polymorphic first-order logic

need to be **translated** to logics with various type systems:

- unsorted logic (Simplify, Zenon)
- simply sorted logic (SMT provers)
- parametric polymorphism (CVC Lite, PVS)
- polymorphic logic (Ergo, Coq, Isabelle/HOL)

Typing Issues

forgetting types is unsound

```
//@ type color
//@ logic color black
//@ logic color white
//@ axiom color: \forall color c; c==white || c==black
```

$$\forall c,\ c = \texttt{white} \vee c = \texttt{black}\ \vdash\ \bot$$

Type Encoding

several type encodings are used

- monomorphization
    - may loop
- usual encoding "types-as-predicates"
    - does not combine nicely with most provers
- new encoding with **type-decorated terms**
  Handling Polymorphism in Automated Deduction (CADE 21)

Trust in Prover Results

- some provers apply the de Bruijn principle and thus are **safe**
  - Coq, HOL family
- most provers **have to be trusted**
  - Simplify, Yices
  - PVS, Mizar
- some provers output **proof traces**
  - Ergo, CVC family, Zenon

Provers Collaboration

most of the time, we run the various provers **in parallel**, expecting at least one of them to discharge the VC

if not, we turn to interactive theorem provers

- no real collaboration between automatic provers
- from Coq or Isabelle, one can call automatic theorem provers
  - proofs are checked when available
  - results are trusted otherwise

Conclusion

Summary

the Why/Krakatoa/Caduceus platform features

- behavioral specification languages for C and Java programs, at source code level
- deductive program verification using original memory models
- multi-provers backend (interactive and automatic)

successfully applied on both

- academic case studies (Schorr-Waite, N-queens, list reversal, etc.)
- industrial case studies (Gemalto, Dassault Aviation, France Telecom)

Other Features

other features not covered in this talk

- **floating point arithmetic**
  - allows to specify rounding and method errors
  - *Formal Verification of Floating-Point Programs* (ARITH 18)
- **pruning strategies** to help decision procedures on large VCs

Ongoing Work & Future Work

ongoing work

- ownership: when class/type invariants must hold?
- C unions & pointer casts

future work

- verification of ML programs