The Why Platform for Deductive Program Verification

Jean-Christophe Filliâtre

CNRS
Orsay, France

Porto, July 4, 2008

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

- Foundations of **ProVal**: Type Theory (the Coq project)
  - highly expressive type system, where type $\simeq$ logic specification
  - type of a program with parameters $\vec{x}$ and result y:

$$\forall \vec{x}, \mathrm{pre\text{-}condition}(\vec{x}) \rightarrow \exists y, \mathrm{post\text{-}condition}(\vec{x}, y)$$

  - program of this type $\simeq$ proof of this formula
- **functional** programs only: no side-effects
- goals of **ProVal**:
  - to deal with **imperative programs** (C, Java)
  - to apply our methods to **industrial cases**

Brief History

- 1999: a first approach for programs with side effects in Coq
- 2000-2003: EU project Verificard (verification of Java Card applets with industrial partners GemPlus, Schlumberger)
- 2001-: stand-alone WHY tool, to use both automatic and interactive provers
- 2003-: KRAKATOA tool for JAVA programs
- 2004-: CADUCEUS tool for C programs
- 2007: The WHY platform

Outline

1. overview of the Why platform
2. verification technique
3. discharging the verification conditions
4. ongoing and future work

overview of the Why platform

Overview of the Why Platform

- 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

two kinds

- **safety**, that is
  - no null pointer dereference
  - no array access out of bounds (no buffer overflow)
  - no division by zero
  - no arithmetic overflow
  - termination
- **behavioral correctness**
  - the program does what it is expected to do

Principles

- specification as **annotations** at the source code level
  - Java: an extension of JML (Java Modeling Language)
  - C: our own language (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 (Alt-Ergo, Simplify, Yices, Z3, CVC3, etc.)
  - proof assistants (Coq, PVS, Isabelle/HOL, etc.)

A Toy 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 (C 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

we want to prove:

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

Binary Search: Safety

- no division by zero
- no array access out of bounds

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

DEMO

Binary Search: Termination

we add a **variant** to prove termination

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

DEMO

Binary Search: Behavioral Specification

we add a **postcondition** for the success case

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

DEMO

Binary Search: Behavioral Specification (cont'd)

we add a postcondition for the failure case ⇒ we need a precondition which says that the array is sorted

```
/*@ 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: Behavioral Specification (cont'd)

requires a stronger invariant

```
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 ) {
    ...
  }
}
```

DEMO

Binary Search: Arithmetic Overflows

finally, let's prove that there is no arithmetic overflow... **there is one!**

in statement

```
int m = (l + u) / 2;
```

a possible overflow is signaled; a possible fix is

```
int m = l + (u - l) / 2;
```

see

- Google: "Read All About It: Nearly All Binary Searches and Mergesorts are Broken"
- "Types, Bytes, and Separation Logic" POPL'07

Binary Search: Arithmetic Overflows

finally, let's prove that there is no arithmetic overflow... **there is one!**

in statement

```
int m = (l + u) / 2;
```

a possible overflow is signaled; a possible fix is

```
int m = l + (u - l) / 2;
```

see

- Google: "Read All About It: Nearly All Binary Searches and Mergesorts are Broken"
- "Types, Bytes, and Separation Logic" POPL'07

Cases Studies so far

academic case studies

- Schorr-Waite algorithm [SEFM'05]
- selection sort, insertion sort, heapsort, quicksort
- Dijkstra's shortest path
- Bresenham's line drawing
- Knuth-Morris-Pratt string searching
- n-queens (backtracking counting of solutions)
- several MIX programs from *The Art of Computer Programming*

Cases Studies so far

industrial case studies

- Java applets
  - Java Card transactions at Gemalto [N. Rousset, SEFM'06]
    - Industrial banking applet Payflex (Banking) - 4600 loc
    - SIMSave: SIM/Server synchro - 3800 loc
    - IAS: government security platform - 20 000 loc
  - Demoney applet provided by Trusted Logic
  - PSE applet provided by Gemalto [AMAST'04]
- avionics software from Dassault Aviation [T. Hubert, HAV'07]
  - embedded C code checked for safety - 70 000 loc
- undergoing collaboration with CEA, Airbus, France Télécom, Continental SA, Dassault Aviation

verification technique

Problems to Solve

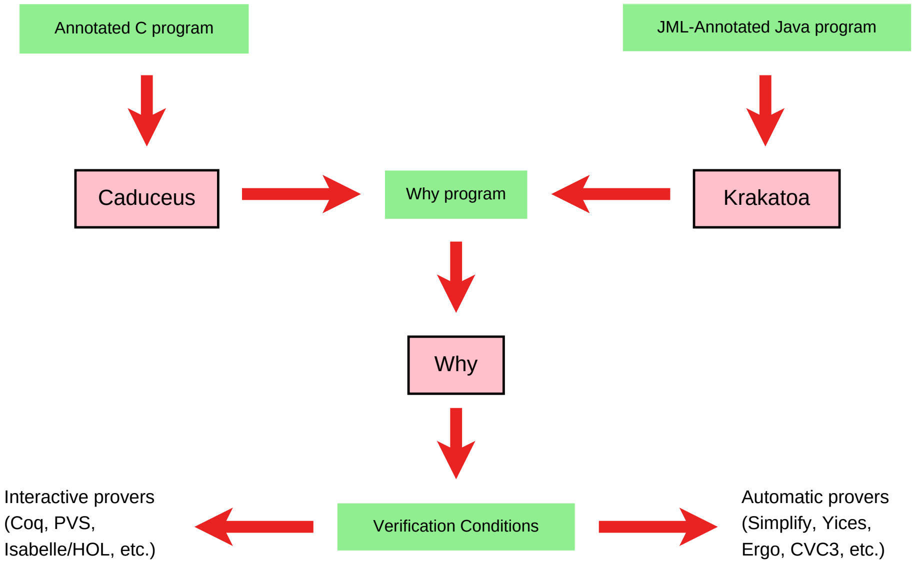
we need

1. to get VCs from annotated programs
   - how to model the memory
   - what can be shared between C and Java
2. to discharge the VCs
   - how to use both automatic and interactive theorem provers

our solution: the use of an intermediate language, **Why**, which is

- a VC generator
- a common front-end to various provers

Platform Overview



Why: a Verification Condition Generator

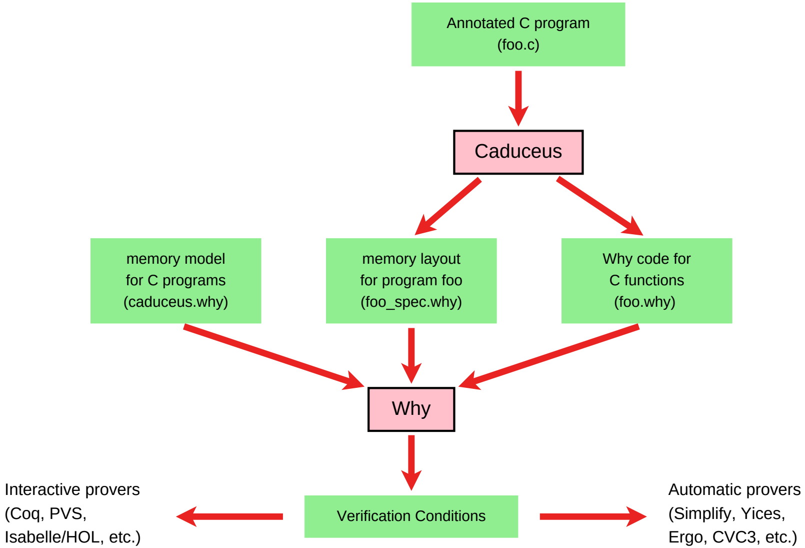
Why is a **verification condition generator** for a language with

- variables containing pure values, no alias ($\sim$ 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



Modeling the memory

we need to translate 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

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

becomes

$$
\begin{aligned}
m &:= \mathtt{upd}(m, px, 0);\\
m &:= \mathtt{upd}(m, py, 1);\\
&\mathtt{assert}\ \mathtt{acc}(m, px) = 0
\end{aligned}
$$

the verification condition is

$$\mathtt{acc}(\mathtt{upd}(\mathtt{upd}(m, px, 0), py, 1), px) = 0$$

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

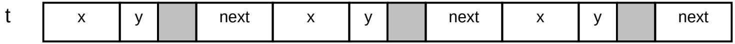
$$\begin{array}{l} x \ := \ \mathtt{upd}(x, p, 0); \\ y \ := \ \mathtt{upd}(y, p, 1); \\ \mathtt{assert}\ \mathtt{acc}(x, p) = 0 \end{array}$$

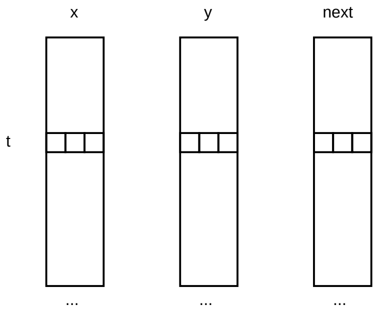
the verification condition is

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

Component-As-Array Model and Pointer Arithmetic

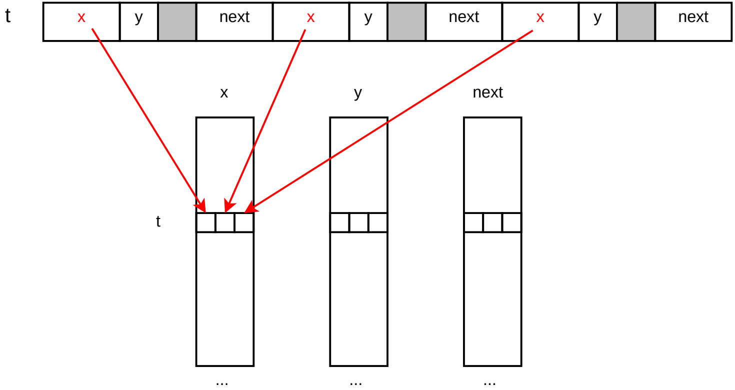
```
struct S { int x; short y; struct S *next; } t[3];
```





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

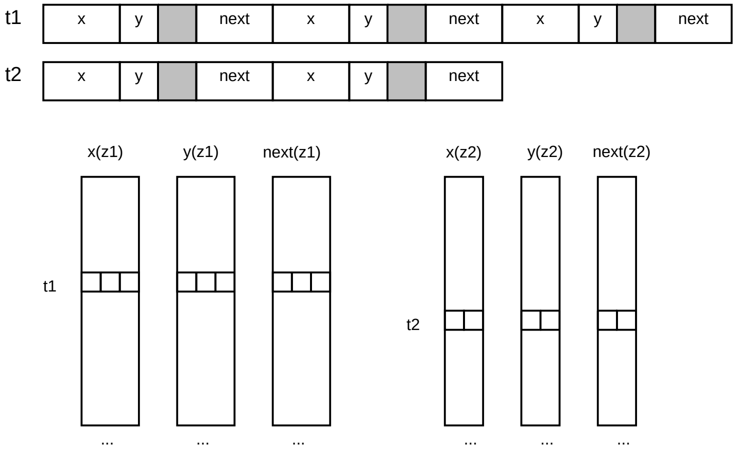
similar to ML-type inference

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

Separation Analysis for Deductive Verification [HAV'07]

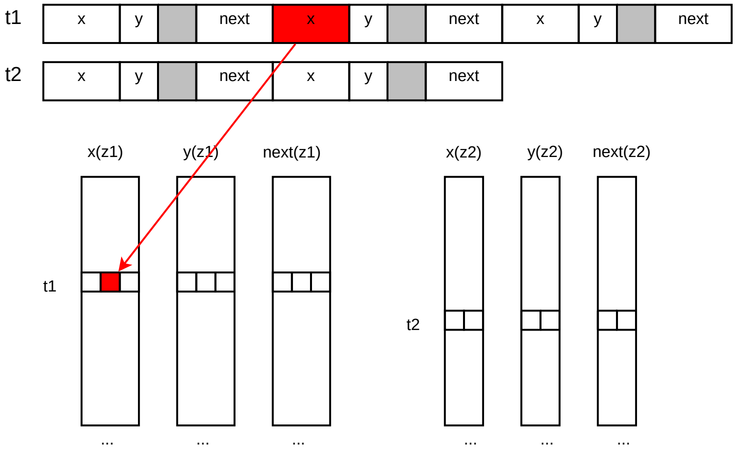
Separation Analysis

```
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

`n`=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

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
  - Alt-Ergo [http://alt-ergo.lri.fr/, SMT'07, SMT'08]
  - Simplify, Yices, Z3, CVC3
- resolution based provers
  - harvey, rv-sat, Zenon

interactive proof assistants

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

Using Several Provers

- built-in theories vs algebraic models
- typing issues: provers do not implement the same logics
- trust in prover results
- provers collaboration

Built-in Theories and Algebraic Models

some provers implement built-in theories, such as

- purely applicative arrays
- real arithmetic
- bit vectors
- tuples

in practice, the intersection is limited to linear arithmetic

so we **axiomatize** the theory we need and rely on the quantifier instantiation capabilities (both risky and incomplete)

Example 1: Bresenham Line Drawing Algorithm

```
// draw a line from (0,0) to (x2,y2) assuming 0 <= y2 <= x2

void bresenham() {
  int x = 0;
  int y = 0;
  int e = 2 * y2 - x2;
  for (x = 0; x <= x2; x++) {
    // plot (x,y) at this point
    if (e < 0)
      e += 2 * y2;
    else {
      y++;
      e += 2 * (y2 - x2);
    }
  }
}
```

Example 1: Bresenham Line Drawing Algorithm

the code only uses additions,
but the loop invariant requires **non-linear** arithmetic

```
/*@ invariant
  @   0 <= x <= x2 + 1 &&
  @   e == 2 * (x + 1) * y2 - (2 * y + 1) * x2 &&
  @   2 * (y2 - x2) <= e <= 2 * y2
  @*/
for (x = 0; x <= x2; x++) {
  // plot (x,y) at this point
  ...
```

Example 1: Bresenham Line Drawing Algorithm

we can help the provers with the following axioms

```
/*@ axiom distr_left :
  @   \forall int a, int b, int c; a * (b+c) == a*b + a*c
  @*/

/*@ axiom distr_right :
  @   \forall int a, int b, int c; (b+c) * a == b*a + c*a
  @*/
```

DEMO

Example 2: Counting Bits

```
int count_bits(int x) {
  int d, c;
  for (c = 0; d = x&-x; x -= d) c++;
  return c;
}
```

`x&-x` extracts the least significant bit of `x`

Example 2: Counting Bits

we introduce a function symbol for the number of bits

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

/*@ ensures \result == nbits(x) */
int count_bits(int x) {
  int d, c;
  /*@ invariant c + nbits(x) == nbits(\at(x,init))
    @ variant   nbits(x)
    @*/
  for (c = 0; d = x&-x; x -= d) c++;
  return c;
}
```

Example 2: Counting Bits

then we axiomatize nbits:

```
//@ axiom nbits_nonneg : \forall int x; nbits(x) >= 0

//@ axiom nbits_zero : nbits(0) == 0

/*@ axiom lowest_bit_zero :
  @   \forall int x; (x&-x) == 0 <=> x == 0
  @*/

/*@ axiom remove_one_bit :
  @    \forall int x;
  @       x != 0 => nbits(x - (x&-x)) == nbits(x) - 1
  @*/
```

Example 3: 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
```

Example 3: Priority Queues

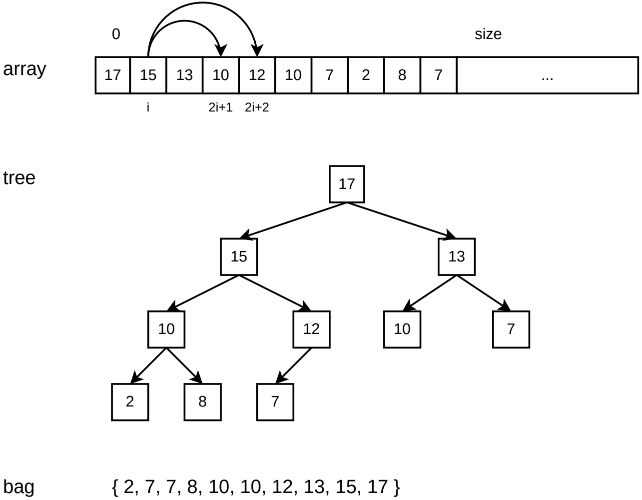
```
//@ type bag

//@ 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
  @ } */
```

Example 3: Priority Queues

```
//@ 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();
```

Example 3: Priority Queues

implementation: heap encoded in an array



Example 3: Priority Queues

```
//@ type tree

//@ logic tree Empty()

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

Example 3: Priority Queues

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

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

...
```

Example 3: Priority Queues

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

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

...
```

Example 3: Priority Queues

```
//@ 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))
  @*/
```

...

Example 3: Priority Queues

```
#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)) } */
```

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 (Alt-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'07]

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**
  - Alt-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 VCs

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, ongoing and future work

Summary

the Why 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)

Ongoing Work

- **floating point arithmetic**
  - allows to specify rounding and method errors
  - *Formal Verification of Floating-Point Programs* [ARITH'07]
  - mostly interactive proof (currently Coq, eventually PVS)
- **ownership**
  - when class/type invariants must hold?
- **automatic generation of loop invariants and preconditions**
  - using abstract interpretation techniques [HAV'07]
- **Eclipse plugin (C and Java)**
- **selection of relevant hypotheses** [FTP'07]
  - in Why, in Alt-Ergo

The Future: Frama-C

- more realistic C fragment (unions & pointer casts, `goto`'s, etc.)
- more ambitious specification language

  ACSL: ANSI/ISO C Specification Language
- combination of deductive verification and abstract interpretation

see `http://frama-c.cea.fr/`