Vérification déductive de programmes avec la plateforme Why

Jean-Christophe Filliâtre

ProVal
INRIA Saclay – Île-de-France

PPS, 23 octobre 2008

Le projet ProVal — http://proval.lri.fr/

- Fondations de **ProVal** : la théorie des types (le projet Coq)
  - système de types très expressif, où type $\simeq$ spécification logique
  - type d'un programme

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

  - programme de ce type $\simeq$ preuve de cette formule
- programmes **purement applicatifs** uniquement
- objectifs de **ProVal** :
  - traiter les **programmes impératifs** (C, Java)
  - appliquer nos méthodes à des **études de cas industrielles**

Un peu d'histoire...

- 1999 : une première approche des programmes impératifs dans Coq
- 2000-2003 : projet européen Verificard (vérification d'applets Java Card avec GemPlus, Schlumberger)
- 2001- : outil WHY, pour utiliser des prouveurs autres que Coq
- 2003- : outil KRAKATOA pour les programmes Java
- 2004- : outil CADUCEUS pour les programmes C
- 2007 : La plateforme WHY

Plan

1. présentation de la plateforme Why
2. technique de vérification
3. utilisation de prouveurs existants
4. travail en cours et perspectives

présentation de la platforme Why

Présentation de la platforme Why

- **objectif** : preuve de propriétés fonctionnelles de **programmes avec pointeurs**
- programme avec pointeurs = programme manipulant des structures de données avec **champs modifiés en place**
- pour l'instant, on considère **C** et **Java**

Quelles propriétés

de deux sortes

- **sûreté**, c'est-à-dire
  - pas de déréférencement du pointeur nul
  - pas d'accès en dehors des bornes (pas de *buffer overflow*)
  - pas de division par zéro
  - pas de dépassement de capacité arithmétique
  - terminaison

- **correction fonctionnelle**
  - le programme fait ce qu'il est censé faire

Principes

- spécification = **annotations** dans le code source
  - Java : extension de JML (*Java Modeling Language*)
  - C : notre propre langage (inspiré de JML)
- génération d'**obligations de preuve** (VCs)
  - logique de Hoare / plus faibles préconditions
  - approches similaires : analyse statique (ESC/Java, SPEC#), méthode B, etc.
- approche **multi-proveurs**
  - utilisation d'autant de prouveurs existants que possible
  - prouveurs automatiques (Alt-Ergo, Simplify, Yices, Z3, CVC3, etc.)
  - assistants de preuve (Coq, PVS, Isabelle/HOL, etc.)

Un exemple historique : Binary Search

binary search : recherche d'une valeur dans un tableau trié

exemple célèbre ; voir *Programming Pearls* de J. Bentley
la plupart des programmeurs se trompent la première fois en écrivant ce programme

Binary Search (code C)

```
int binary_search(int* t, int n, int v) {
  int l = 0, u = n-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
      return m;
  }
  return -1;
}
```

Binary Search

on veut prouver

1. l'absence d'erreur à l'exécution
2. la terminaison
3. la correction fonctionnelle

Binary Search : sûreté

- pas de division par zéro
- pas d'accès en dehors des bornes du tableau

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

DÉMO

Binary Search : terminaison

on ajoute un **variant** pour établir la terminaison

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

DÉMO

Binary Search : correction fonctionnelle

on ajoute une **postcondition** pour le cas d'une recherche positive

```
/*@ 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 ;
  /*@ invariant 0 <= l && u <= n-1
    @ variant u - l
    @*/
  while (l <= u ) {
    ...
}
```

DÉMO

Binary Search : correction fonctionnelle (suite)

on ajoute une positive pour le cas négatif $\Rightarrow$ exige une précondition indiquant que le tableau est trié

```
/*@ 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 : correction fonctionnelle (suite)

exige un invariant plus fort

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

DÉMO

Binary Search : dépassement de capacité arithmétique

pour terminer, prouvons qu'il n'y a pas de dépassement de capacité arithmétique ... **il y en a un !**

dans l'instruction

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

un dépassement possible est signalé ; se corrige en

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

voir

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

Binary Search : dépassement de capacité arithmétique

pour terminer, prouvons qu'il n'y a pas de dépassement de capacité arithmétique ... **il y en a un !**

dans l'instruction

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

un dépassement possible est signalé ; se corrige en

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

voir

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

Études de cas à ce jour

études de cas académiques

- algorithme de Schorr-Waite [SEFM'05]
- tri sélection, insertion, par tas, rapide
- algorithme de Dijkstra de plus court chemin
- algorithme de Bresenham de dessin de ligne
- recherche de sous-chaîne de Knuth-Morris-Pratt
- *n*-reines (nombre de solution par *backtracking*)
- plusieurs programmes MIX de *The Art of Computer Programming*

Études de cas à ce jour

études de cas industrielles

- applets Java
  - transactions Java Card chez Gemalto [N. Rousset, SEFM'06]
    - applet bancaire Payflex (Banking) - 4600 loc
    - SIMSave : SIM/Server synchro - 3800 loc
    - IAS : plateforme sécuritaire pour l'administration - 20 000 loc
  - applet Demoney fournie par Trusted Logic
  - applet PSE fournie par Gemalto [AMAST'04]
- code avionique chez Dassault Aviation [T. Hubert, HAV'07]
  - sûreté d'un code C embarqué - 70 000 loc
- collaboration en cours avec le CEA, Airbus, France Télécom, Continental SA, Dassault Aviation

technique de vérification

Problèmes à résoudre

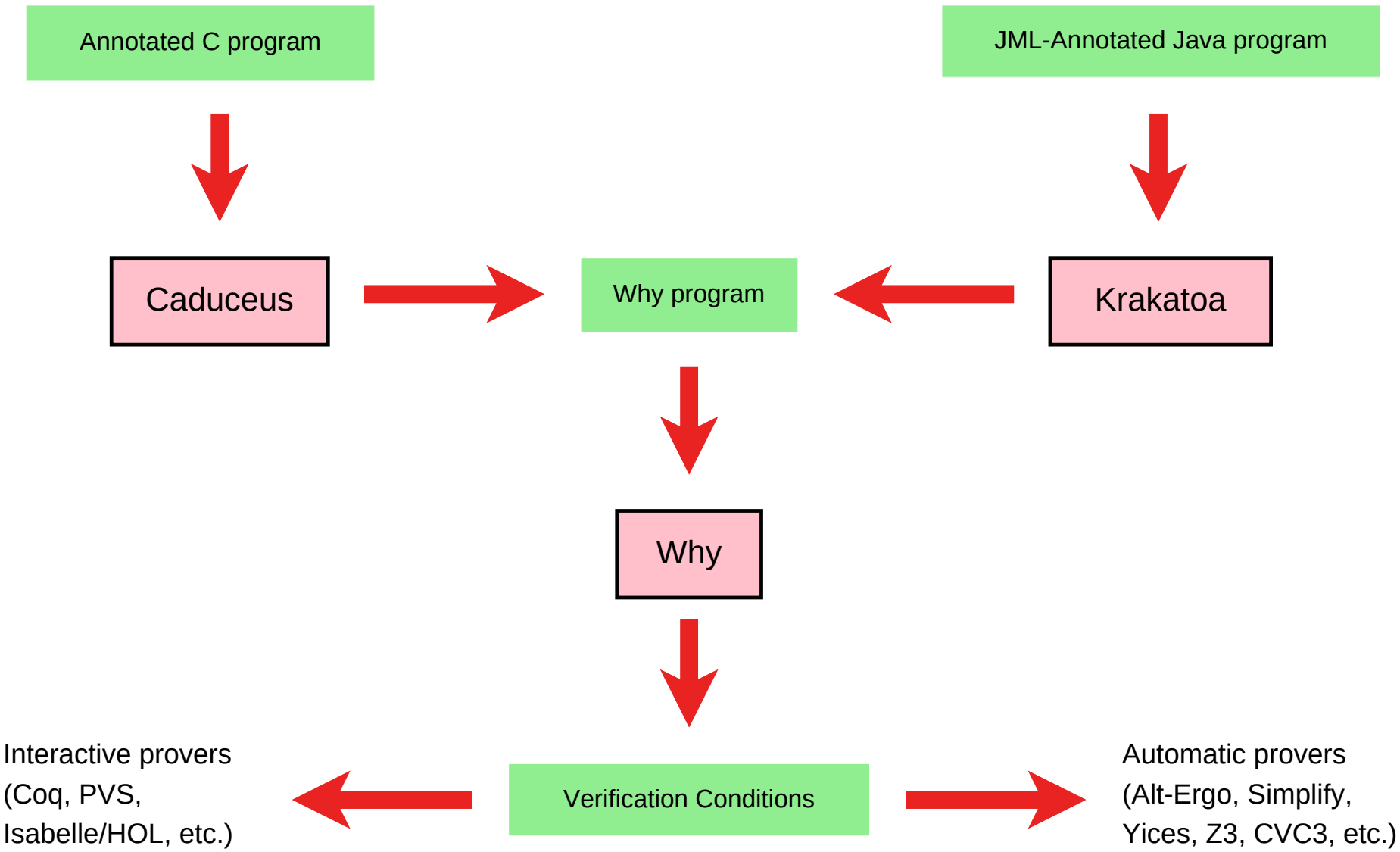
on a besoin

1. de produire les obligations de preuve à partir de programmes annotés
   - comment modéliser la mémoire
   - que peut-on partager entre C et Java
2. de prouver ensuite les obligations de preuve
   - comment utiliser autant les prouveurs automatiques qu'interactifs

notre solution : l'utilisation d'un langage intermédiaire, **Why**, qui est

- un générateur d'obligations de preuve
- un *front-end* commun à tous les prouveurs

Aperçu de la plateforme



Why : un générateur d'obligations de preuve

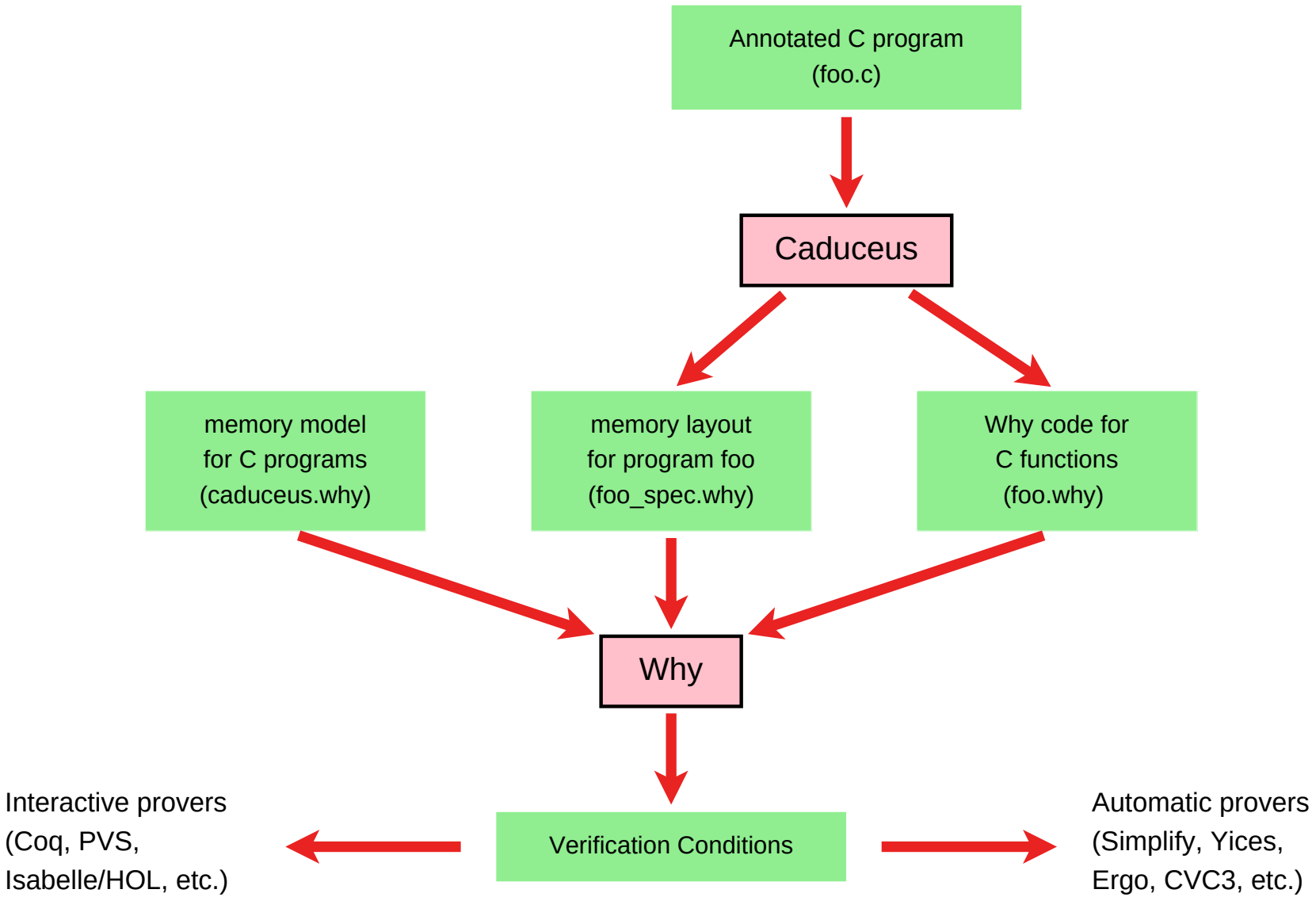
Why est un **générateur d'obligations de preuve** pour un langage avec

- des variables contenant des valeurs pures, sans alias
  ($\sim$ logique de Hoare)
- les structures de contrôle usuelles (boucles, tests, etc.)
- des exceptions
- des fonctions (éventuellement récursives)
- une logique du premier ordre polymorphe avec égalité et arithmétique

Why est semblable à Boogie (projet SPEC# chez Microsoft Research)

Why est aussi chargé de **traduire** les obligations de preuve vers les **logiques natives** de tous les prouveurs

Générer les obligations de preuve



Modéliser la mémoire

objectif : traduire du code avec pointeurs vers du code sans alias

idée naïve : modéliser la **mémoire comme un grand tableau**

en utilisant la théorie des tableaux

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

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

Modèle mémoire naïf

le programme C

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

devient

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

l'obligation de preuve est

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

Modèle mémoire pour les programmes avec pointeurs

on utilise le modèle **Burstall-Bornat** (*component-as-array*)

chaque champ de structure/objet est représenté par un tableau différent

repose sur la propriété « **deux champs différents ne peuvent être partagés (*aliasés*)** »

conséquence importante : empêche les *casts* et les unions (a priori)

Bénéfices

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

devient

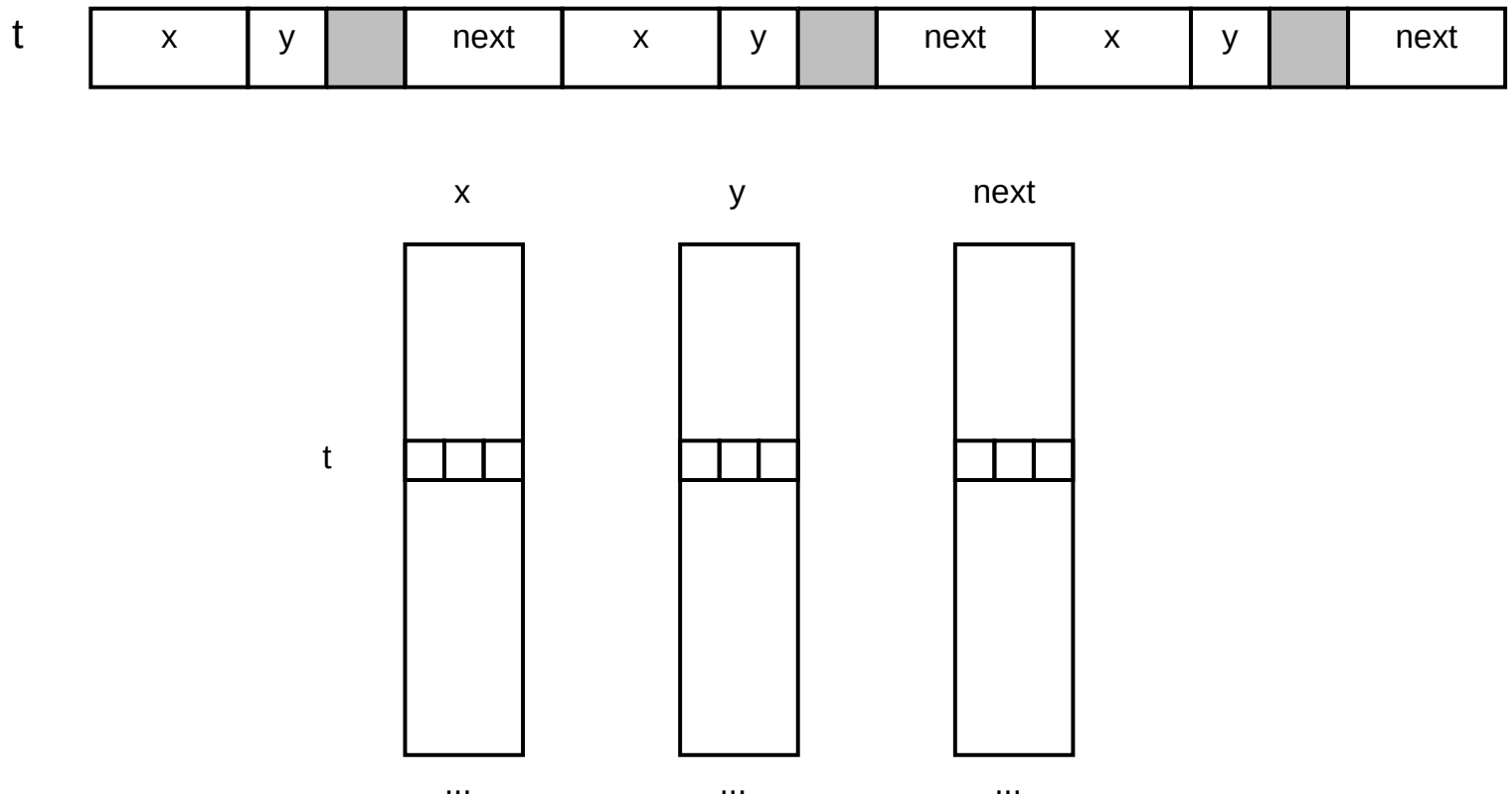
$$x := \mathtt{upd}(x, p, 0);$$
$$y := \mathtt{upd}(y, p, 1);$$
$$\mathtt{assert}\ \mathtt{acc}(x, p) = 0$$

l'obligation de preuve est

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

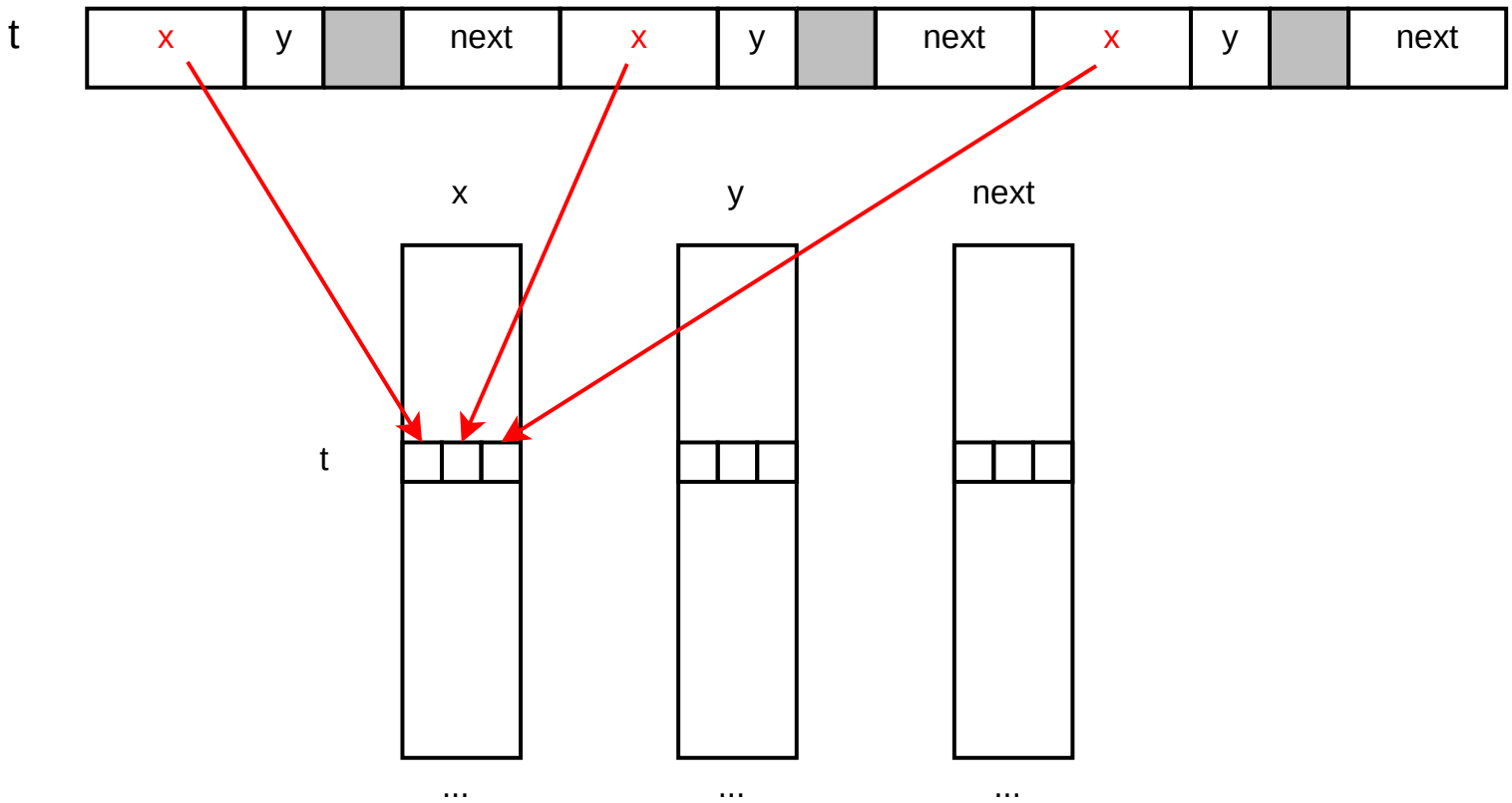
Modèle Burstall-Bornat et arithmétique de pointeurs

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



Modèle Burstall-Bornat et arithmétique de pointeurs

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



Analyse de séparation

au dessus du modèle Burstall-Bornat, on ajoute une **analyse de séparation**

- à chaque pointeur est associée une **zone**
- les zones sont **unifiés** quand les pointeurs sont affectés / comparés
- les fonctions sont **polymorphes** vis-à-vis des zones

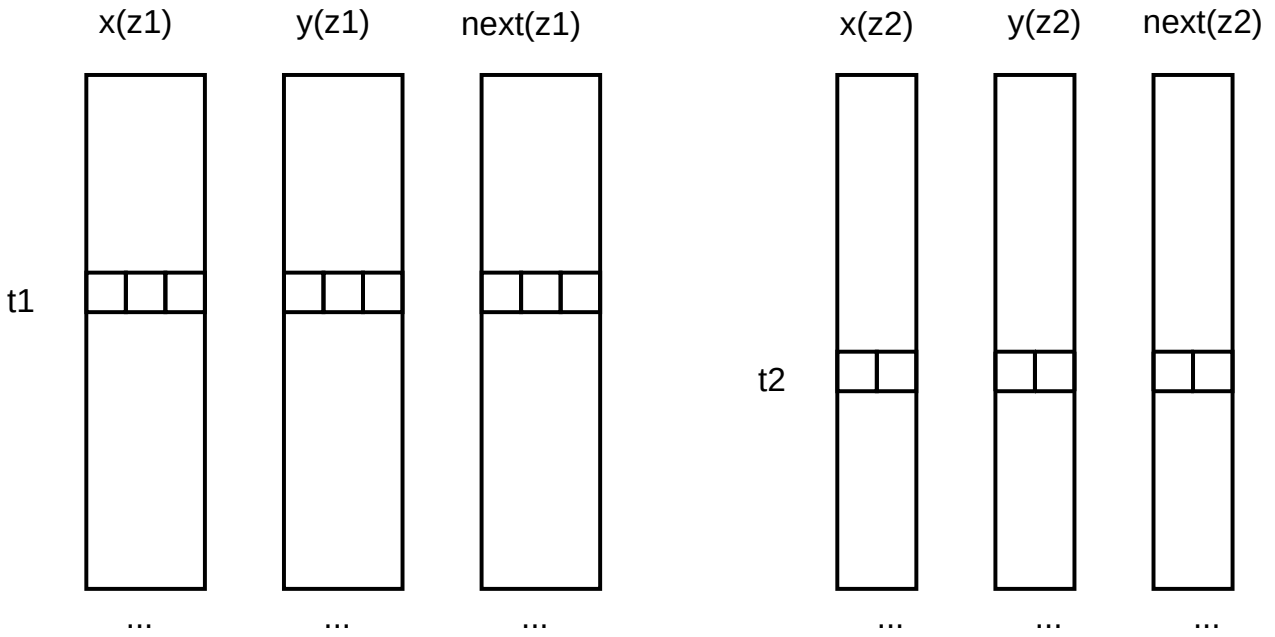
semblalbe à l'inférence de types de ML

le modèle mémoire est alors raffiné en prenant en compte les zones

Separation Analysis for Deductive Verification [HAV'07]

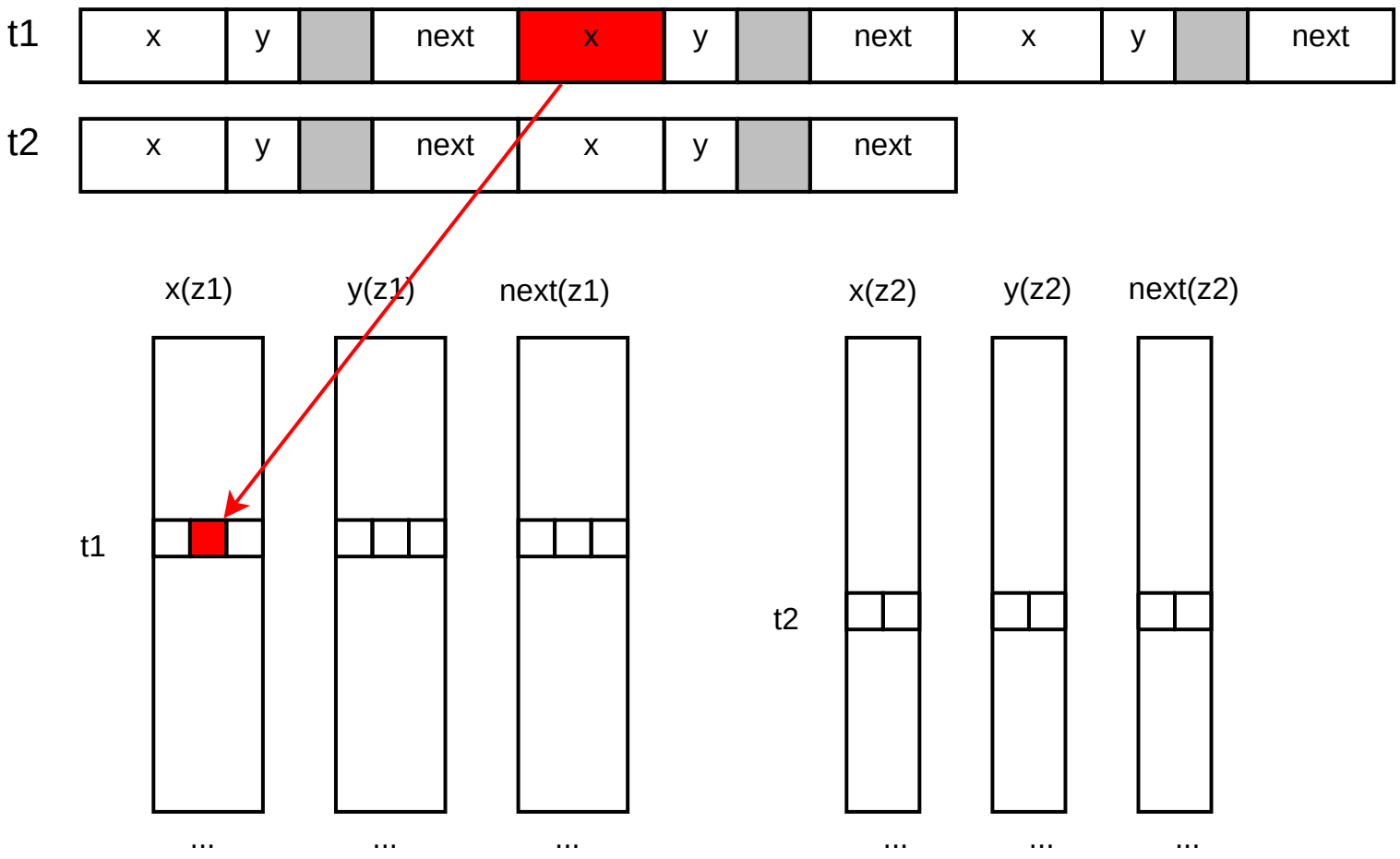
Analyse de séparation

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



Analyse de séparation

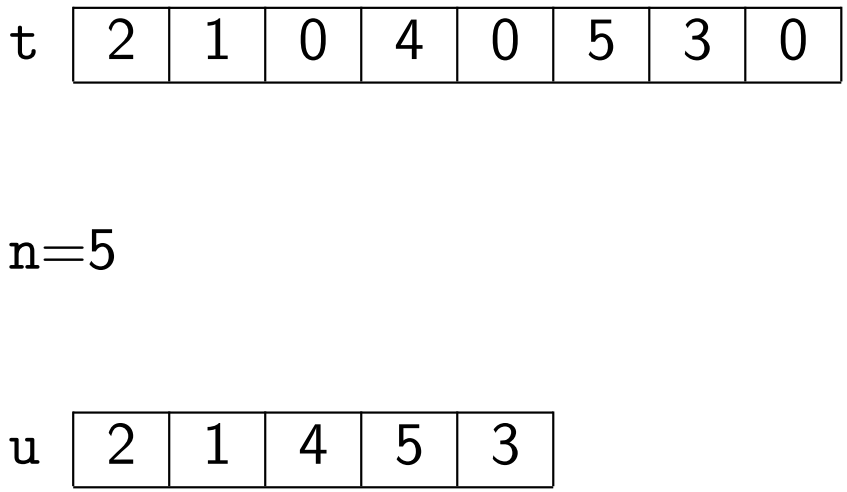
```
struct S { int x; short y; struct S *next; } t1[3], t2[2];
```



Exemple

défi pour la vérification déductive proposé par P. Müller :

> *compter le nombre n de valeurs non-nulles dans un tableau t,*
> *puis les copier dans un tableau fraîchement alloué de taille n*



L'exemple de P. Müller's (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];
}
```

L'exemple de P. Müller's (spécification)

```
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];
}
```

L'exemple de P. Müller's (preuve)

12 obligations de preuve

- sans analyse de séparation : 10/12 prouvées automatiquement
- avec analyse de séparation : 12/12 prouvées automatiquement

DÉMO

valider les obligations de preuve

Quels prouveurs

on souhaite utiliser autant d'outils de preuve **existants** que possible

pré-requis

- logique du premier ordre
- égalité et arithmétique
- quantificateurs (modèles mémoire, modèles axiomatiques utilisateur)

Prouveurs supportés

procédures de décision

- à la Nelson-Oppen
  - Alt-Ergo [http://alt-ergo.lri.fr/, SMT'07, SMT'08]
  - Simplify, Yices, Z3, CVC3
- basés sur la résolution
  - harvey, rv-sat, Zenon

assistants de preuve interactifs

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

Utiliser plusieurs prouveurs

- théories prédéfinies / modèles axiomatiques
- les différents prouveurs n'ont pas les mêmes systèmes de type
- quelle confiance accorder aux prouveurs ?
- collaboration entre les prouveurs

Théories prédéfinies et modèles axiomatiques

les prouveurs peuvent contenir des théories prédéfinies, comme

- l'égalité (*congruence closure*)
- l'arithmétique entière, linéaire ou pas
- les tableaux applicatifs
- l'arithmétique réelle, linéaire ou pas
- les vecteurs de bits
- les n-uplets

en pratique, l'intersection se limite à l'égalité et à l'arithmétique linéaire entière

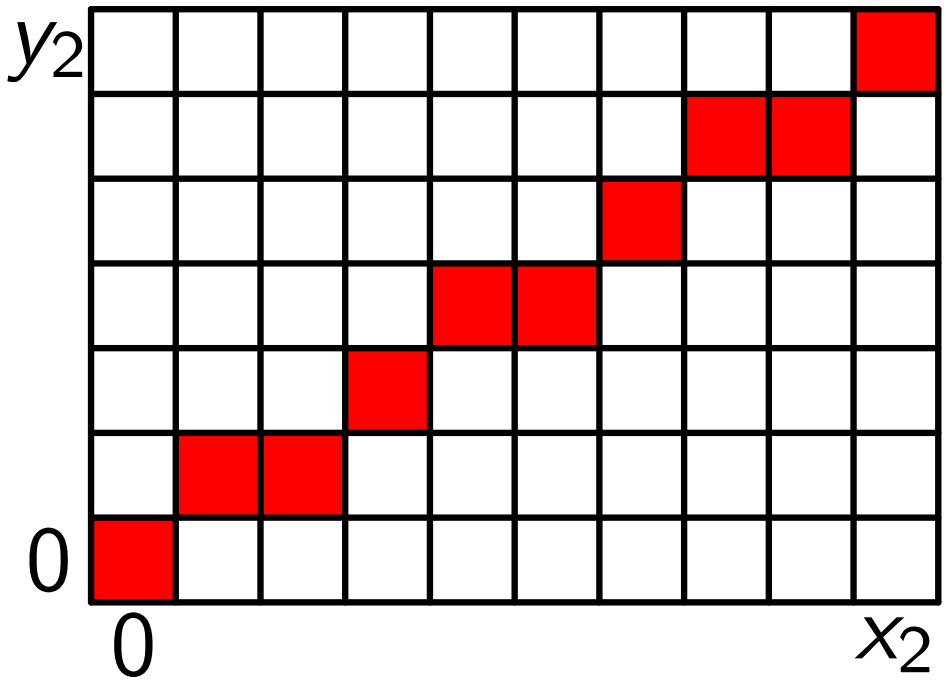
en conséquence, on **axiomatise** les théories dont on a besoin,
et on se repose sur les mécanismes d'instanciation des quantificateurs
(à la fois risqué et incomplet)

Modèes axiomatiques

illustrons les modèles axiomatiques sur plusieurs exemples

1. algorithme de Bresenham
2. compter les bits
3. files de priorité
4. programme “mystère”

Exemple 1 : Algorithme de Bresenham




```
int x2, y2;

//@ invariant first_octant : 0 <= y2 <= x2
```

Exemple 1 : Algorithme de Bresenham

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

Exemple 1 : Algorithme de Bresenham

le code utilise seulement des additions,
mais l'invariant de boucle nécessite de l'arithmétique **non linéaire**

```
/*@ 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++) {
  // ici on fait plot (x,y)
  ...
```

Exemple 1 : Algorithme de Bresenham

on peut aider les prouveurs avec les axiomes suivants

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

DÉMO

Exemple 2 : compter les bits

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

`x&-x` isole le bit à 1 le plus à droite de `x`

Exemple 2 : compter les bits

on introduit un symbole de fonction pour le nombre de 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;
}
```

Exemple 2 : compter les bits

ensuite on axiomatise `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
  @*/
```

Exemple 3 : files de priorité

structure de données globale pour une **file de priorité** contenant des entiers

```
void clear() ; // vide la file
void push(int x) ; // insère un élément
int max() ; // renvoie l'élément maximal
int pop() ; // supprime et renvoie l'élément maximal
```

Exemple 3 : files de priorité

```
//@ 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
  @ } */
```

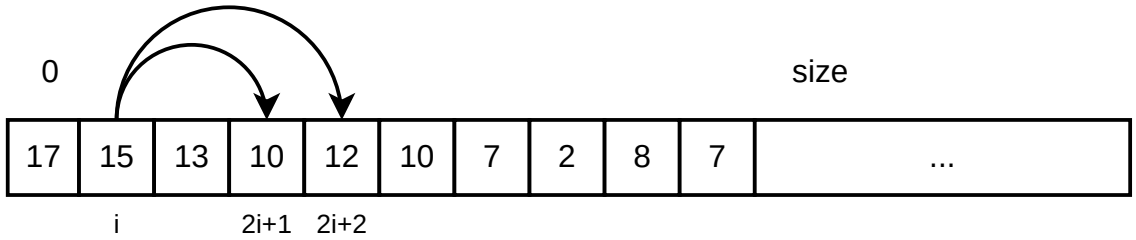
Exemple 3 : files de priorité

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

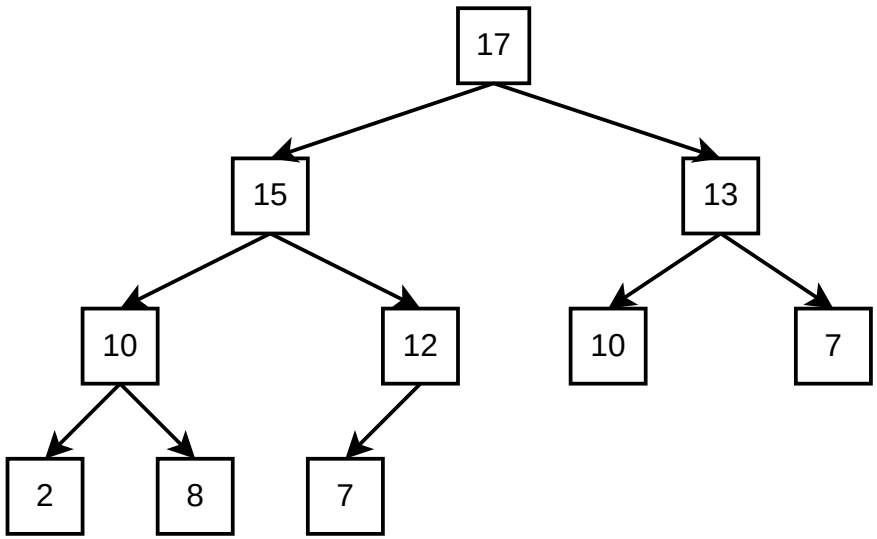
Exemple 3 : files de priorité

réalisation : tas encodé dans un tableau

array



tree



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

Exemple 3 : files de priorité

```
//@ type tree

//@ logic tree Empty()

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

Exemple 3 : files de priorité

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

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

...
```

Exemple 3 : files de priorité

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

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

...
```

Exemple 3 : files de priorité

```
//@ 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))
  @*/

...
```

Exemple 3 : files de priorité

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

Exemple 4

défi pour le **programme vérifié du mois** :

```
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));}
```

Exemple 4

```
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));
}
```

le programme lit un entier n
et affiche le nombre de solutions du problème des n **reines**

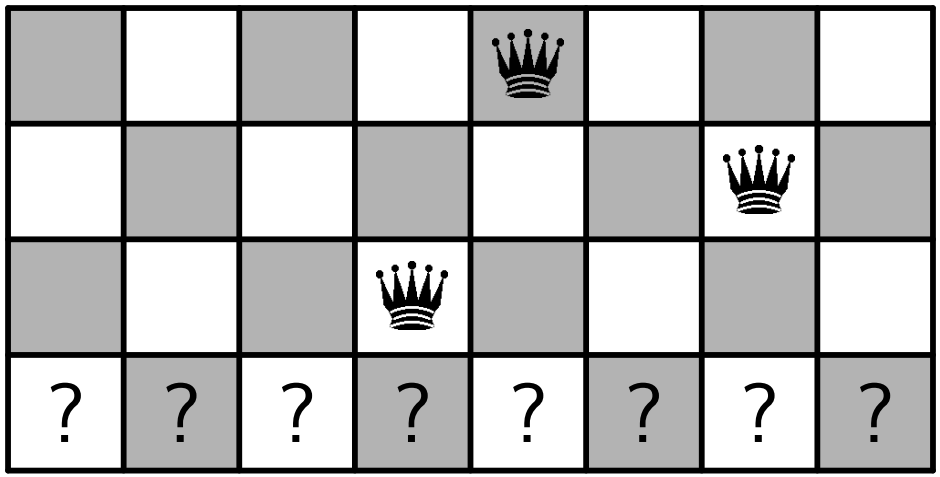
Exemple 4 : les n reines

- procède par backtracking (pas de meilleure solution connue)
- les entiers sont utilisés comme des **ensembles** (vecteurs de bits)

entiers	ensembles
`0`	$\emptyset$
`a&b`	$a \cap b$
`a+b`	$a \cup b$, quand $a \cap b = \emptyset$
`a-b`	$a \setminus b$, quand $b \subseteq a$
`~a`	$\complement a$
`a&-a`	$min_elt(a)$, quand $a \neq \emptyset$
`~(~0<<n)`	$\{0, 1, \ldots, n-1\}$
`a*2`	$\{i+1 \mid i \in a\}$, noté $S(a)$
`a/2`	$\{i-1 \mid i \in a \wedge i \neq 0\}$, noté $P(a)$

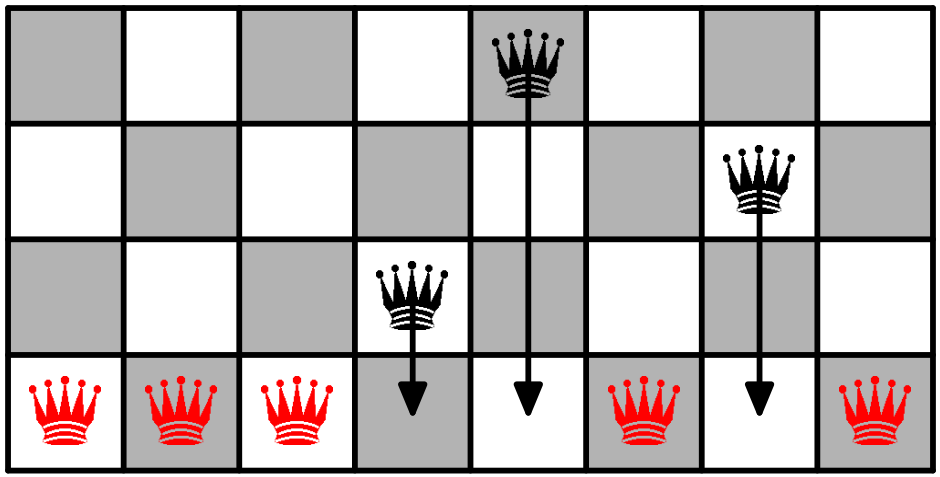
Exemple 4 : les n reines

Que signifient a, b et c ?



Exemple 4 : les n reines

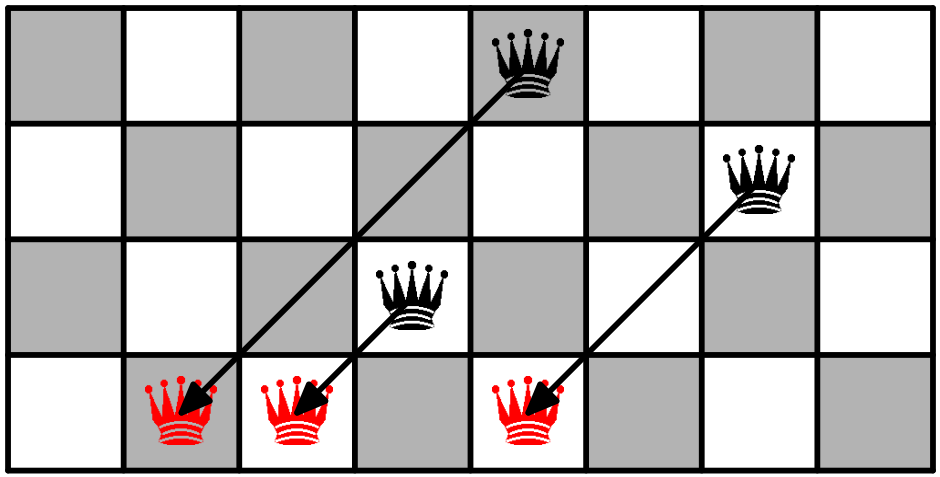
Que signifient a, b et c ?



a = colonnes à pourvoir = 11100101_2

Exemple 4 : les n reines

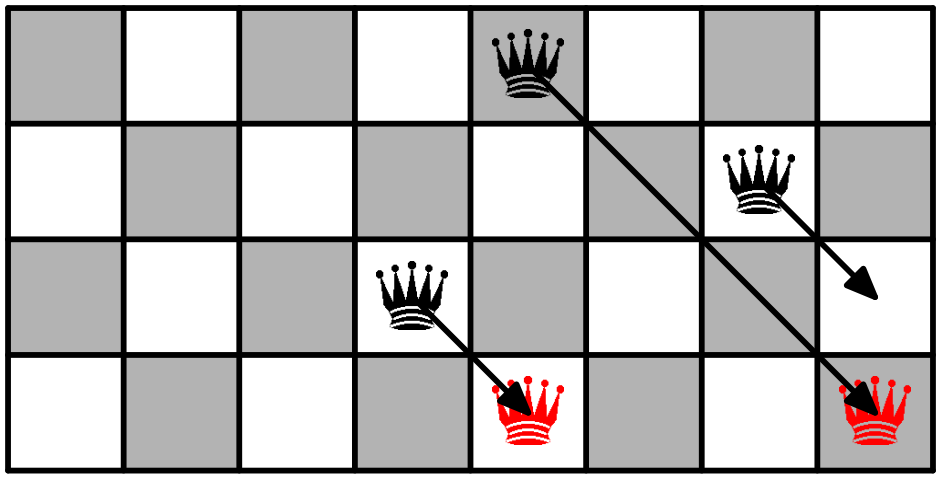
Que signifient a, b et c ?



b = positions illégales = 01101000_2

Exemple 4 : les n reines

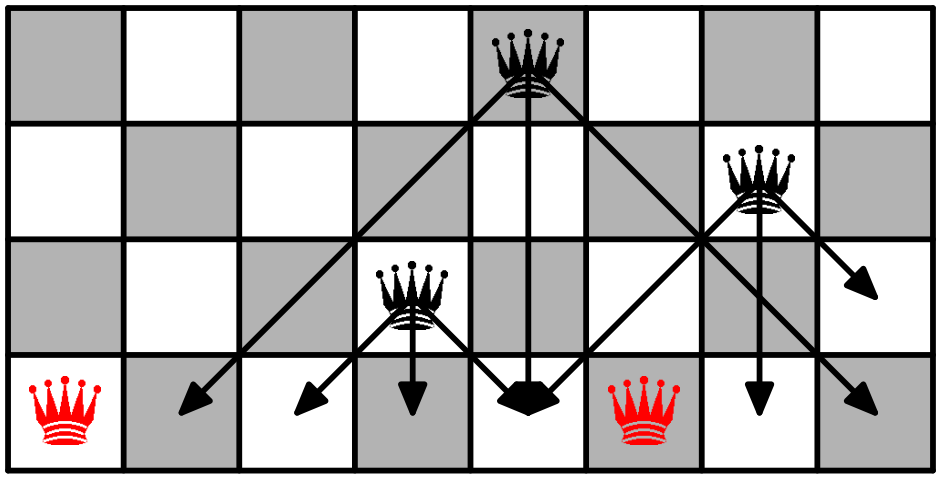
Que signifient a, b et c ?



c = autres positions illégales = 00001001_2

Exemple 4 : les n reines

Que signifient a, b et c ?



$a\&\tilde{}b\&\tilde{}c$ = positions à essayer = 10000100_2

Exemple 4 : les n reines

maintenant c'est clair

```
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);
}
```

Exemple 4 : les n reines

on suppose axiomatisés des ensembles finis d'entiers (type `iset`)
on interprète alors les entiers C comme des ensembles

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

Exemple 4 : les n reines

prouvons la terminaison

```
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 obligations de preuve, toutes prouvées automatiquement

de même pour la terminaison de la fonction récursive :
7 obligations de preuve, toutes prouvées automatiquement

Exemple 4 : les n reines

sûreté, terminaison et correction nécessitent **256** lignes de code et de spécification

obligations de preuve :

- fonction `queens` : **15** VCs
  - **toutes** prouvées automatiquement (Simplify, Alt-Ergo ou Yices)
- fonction récursive `t` : **51** VCs
  - **42** prouvées automatiquement : 41 par Simplify, 37 par Alt-Ergo et 35 par Yices
  - **9** prouvées manuellement avec Coq (et Simplify)

Différents systèmes de types

les obligations de preuve sont exprimées dans la logique du premier ordre polymorphe

elles doivent être traduites vers les différents logiques des différents prouveurs :

- logique non typée (Simplify, Zenon)
- logique simplement typée (SMT provers)
- polymorphisme ad-hoc (CVC Lite, PVS)
- logique polymorphe (Alt-Ergo, Coq, Isabelle/HOL)

Différents systèmes de types

oublier les types serait incorrect

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

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

Encodage des types

plusieurs encodages des types sont envisageables

- monomorphisation
  - peut boucler
- encodage usuel "types-comme-prédicats"
  - inadapté à la plupart des prouveurs
- nouvel encodage proposé : **termes décorés par des types**
  Handling Polymorphism in Automated Deduction [CADE'07]

Confiance dans le résultat des prouveurs

- certains prouveurs appliquent le critère de de Bruijn et sont **sûrs**
  - Coq, famille HOL
- la plupart des prouveurs **ne sont pas sûrs**
  - Simplify, Yices
  - PVS, Mizar
- quelques prouveurs produisent des **traces de preuve**
  - Alt-Ergo, famille CVC, Zenon

Collaboration entre prouveurs

la plupart du temps, on utilise les différents prouveurs **en parallèle**, en escomptant que l'un au moins valide l'obligation de preuve

dans le cas contraire, on se tourne vers les prouveurs interactifs

- pas de réelle collaboration entre prouveurs automatiques
- depuis Coq ou Isabelle, on peut appeler des prouveurs automatiques
  - les preuves sont vérifiées quand elles sont disponibles
  - sinon, on fait confiance à la réponse

conclusion, travail en cours et perspectives

Conclusion

la plateforme Why

- des langages de spécification fonctionnelle pour C et Java, au niveau source
- une vérification déductive utilisant des modèles mémoires originaux
- une approche multi-prouveurs (automatiques et interactifs)

appliquée avec succès à

- des études de cas académiques
- des études de cas industrielles

logiciel libre, à `http://why.lri.fr/`

Travail en cours

- **arithmétique flottante**
  - permet de spécifier erreurs de calcul et de méthode
  - *Formal Verification of Floating-Point Programs* [ARITH'07]
  - preuve interactive pour l'essentiel (Coq, bientôt PVS)
- **ownership**
  - quand est-ce qu'un invariant de classe est vérifié ?
- **génération automatique d'invariants de boucle et de préconditions**
  - techniques d'interprétation abstraite [HAV'07]
- **plugin Eclipse (C and Java)**
- **sélection d'hypothèses** [FTP'07]
  - dans Why, dans Alt-Ergo

Le futur : Frama-C

- fragment de C plus réaliste (unions & casts de pointeurs, `goto`s, etc.)
- langage de spécification plus ambitieux

  ACSL : ANSI/ISO C Specification Language
- combinaison de vérification déductive et d'interprétation abstraite

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