Vérification déductive de programmes avec la plateforme Why

Jean-Christophe Filliâtre

ProVal
INRIA Saclay – Île-de-France

LABRI, 18 septembre 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. 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

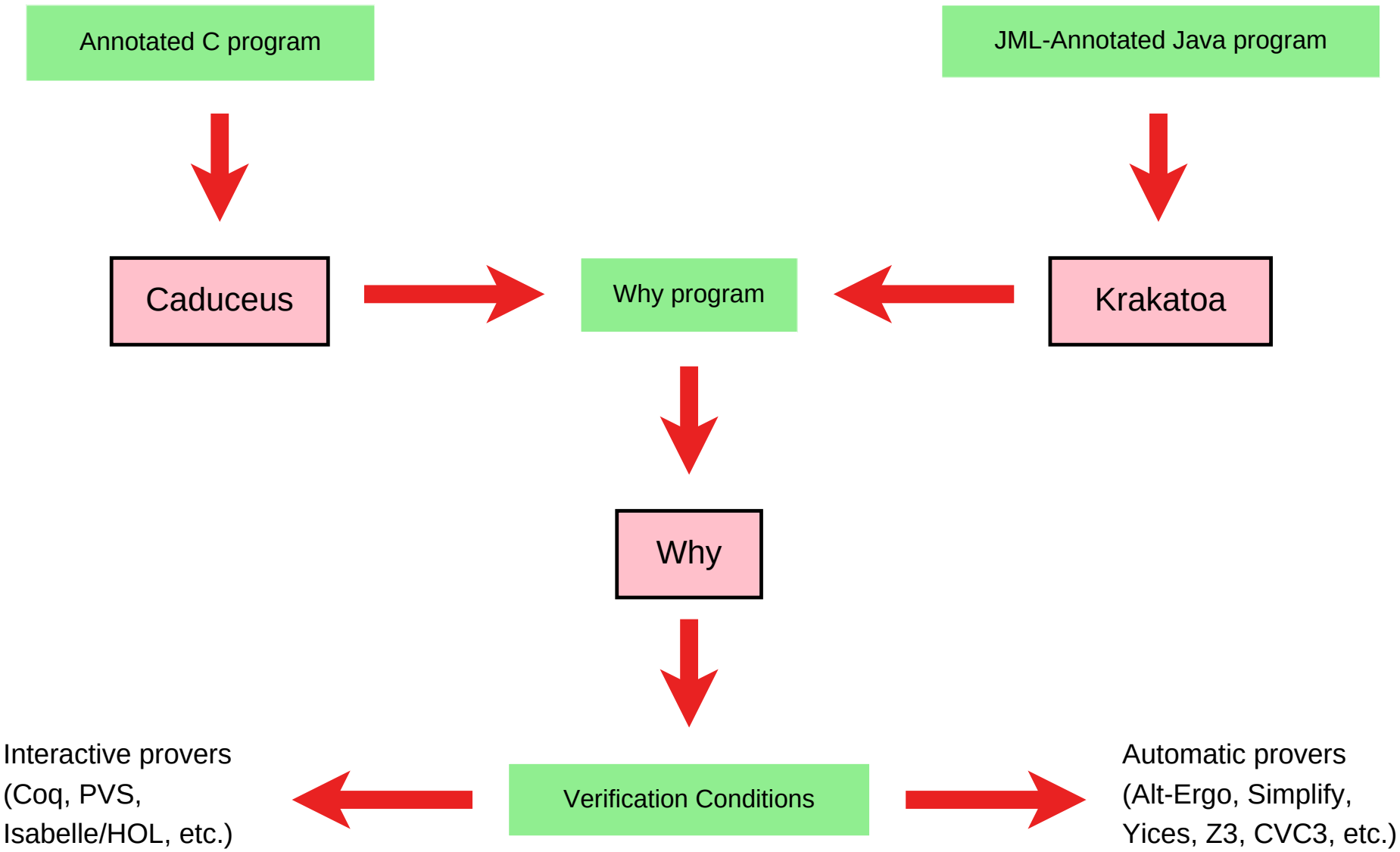
on a besoin

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

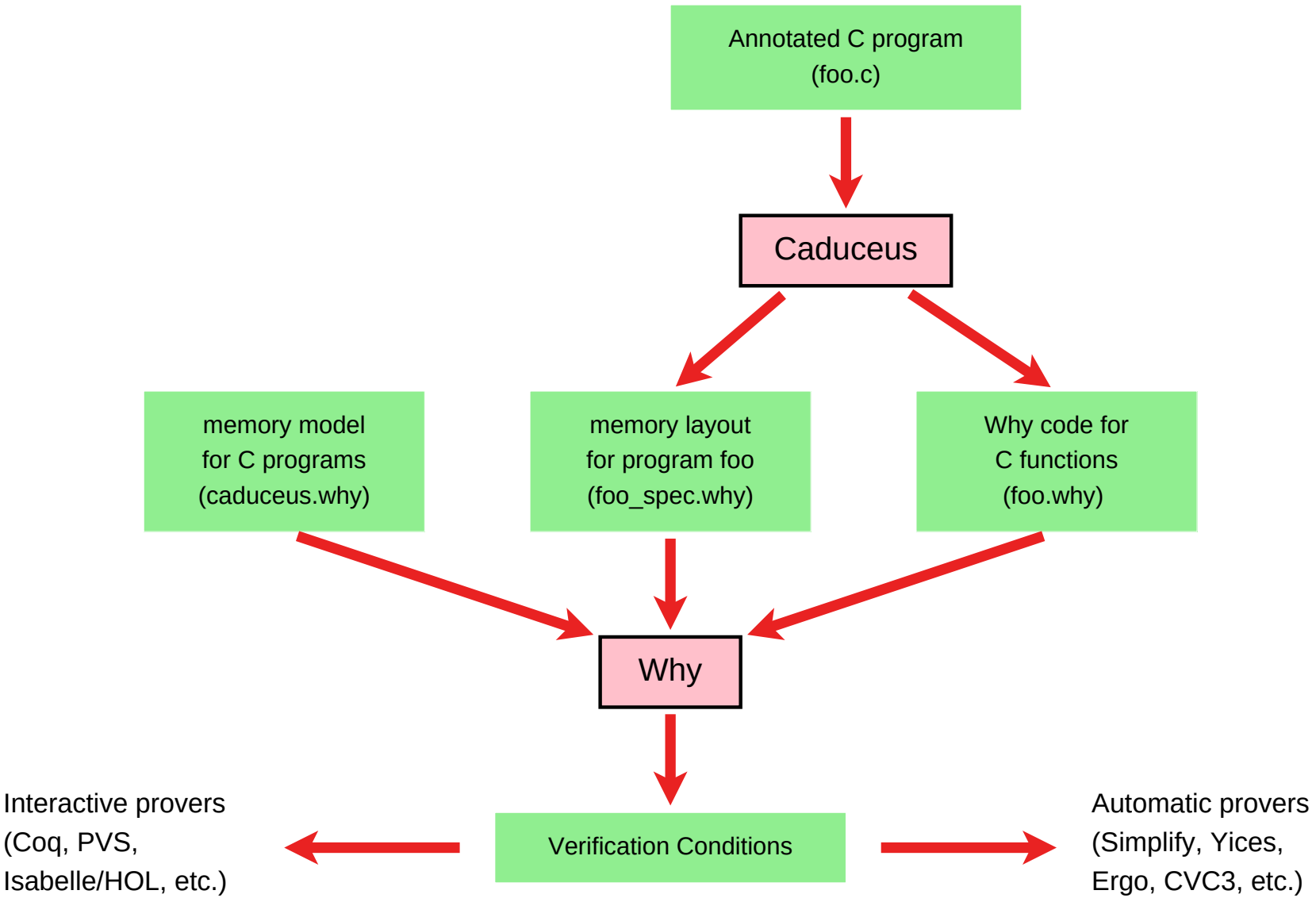
Why est un **générateur d'obligations de preuve** pour un langage avec

- 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

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

l'obligation de preuve est

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

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

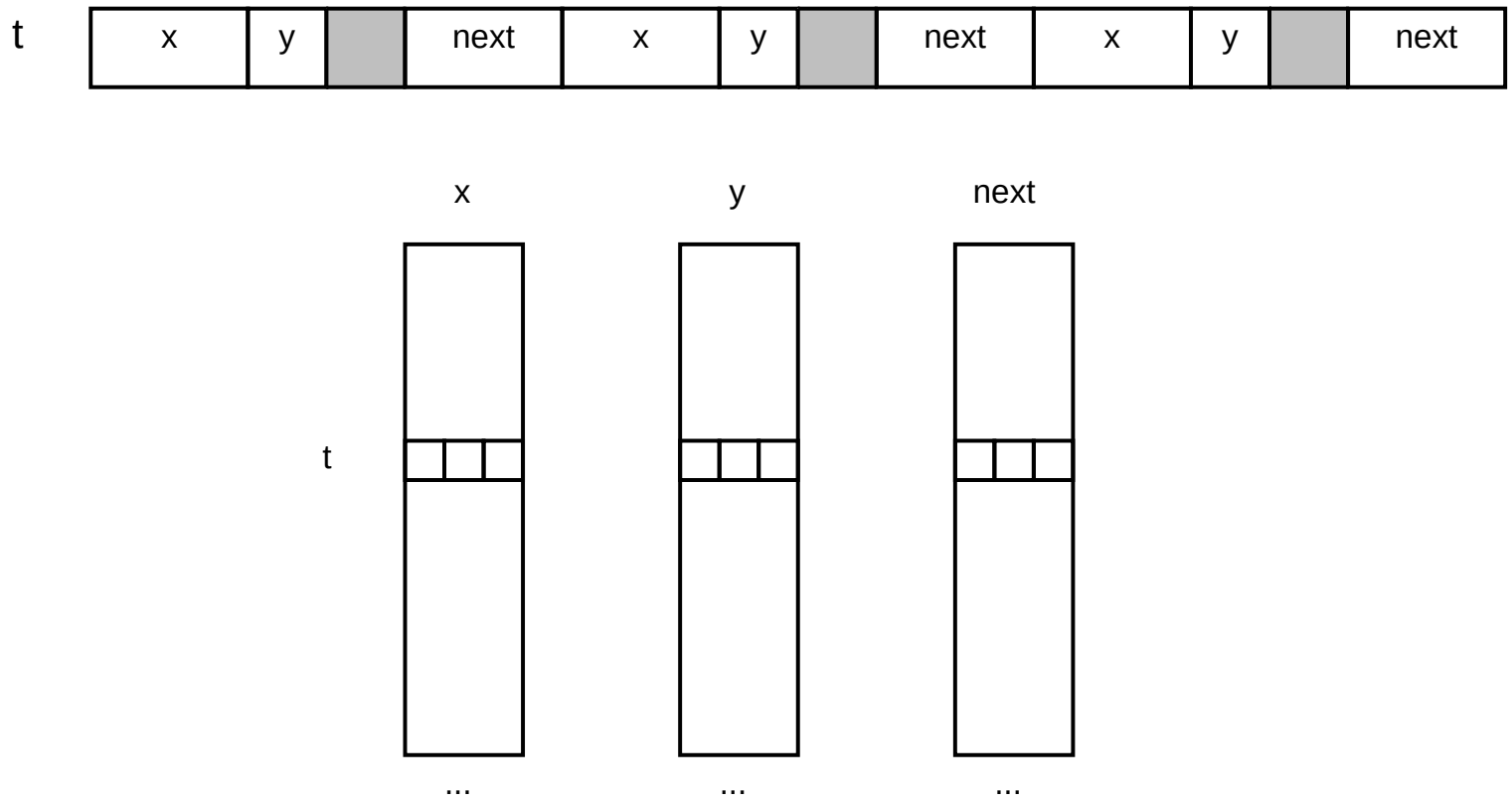
$$x := \mathtt{upd}(x, p, 0);$$
$$y := \mathtt{upd}(y, p, 1);$$
$$\mathtt{assert}\ \mathtt{acc}(x, p) = 0$$

l'obligation de preuve est

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

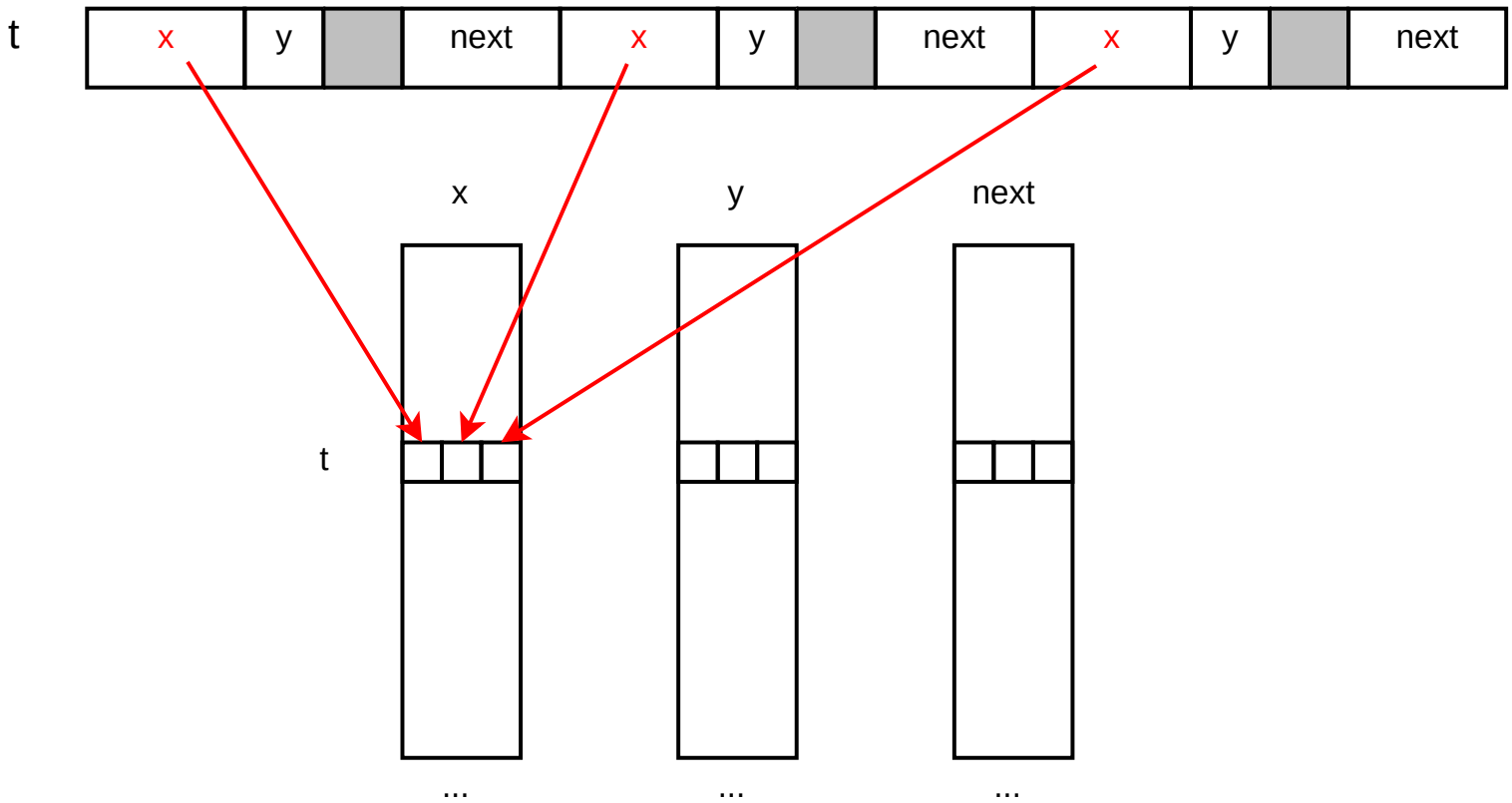
Modèle Burstall-Bornat et arithmétique de pointeurs

```
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

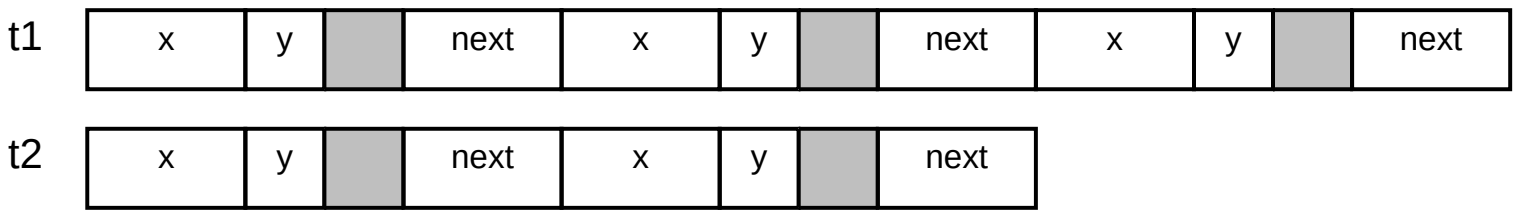
semblalbe à l'inférence de types de ML

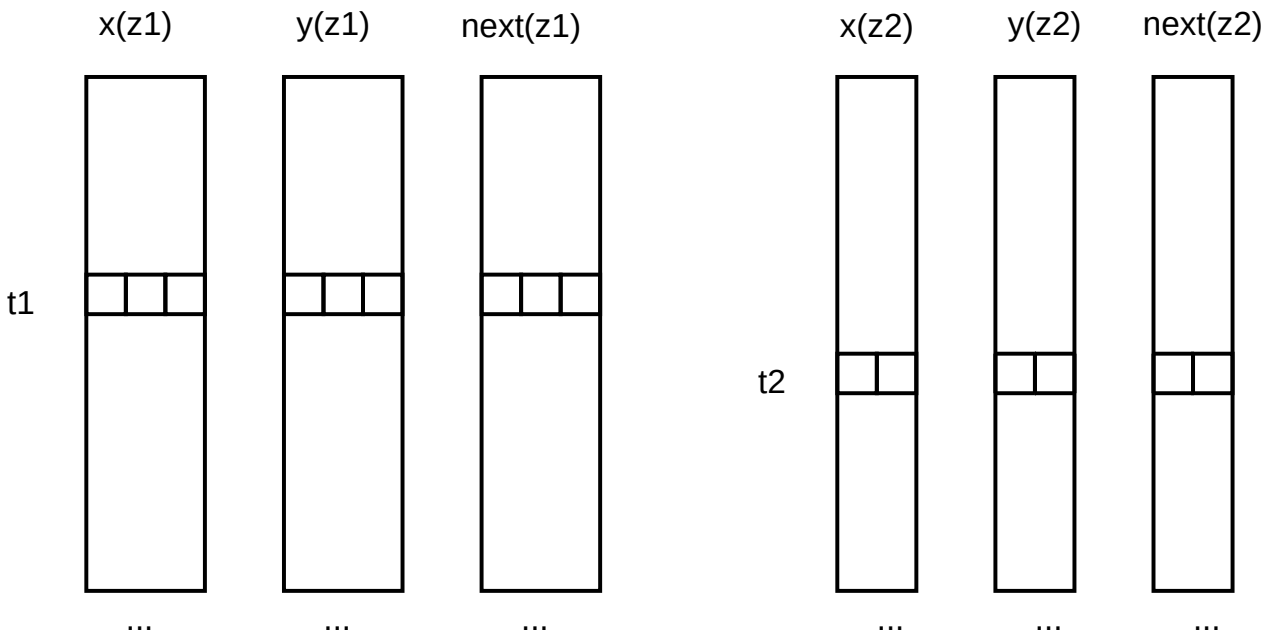
le modèle mémoire est alors raffiné en prenant en compte les zones

Separation Analysis for Deductive Verification [HAV'07]

Analyse de séparation

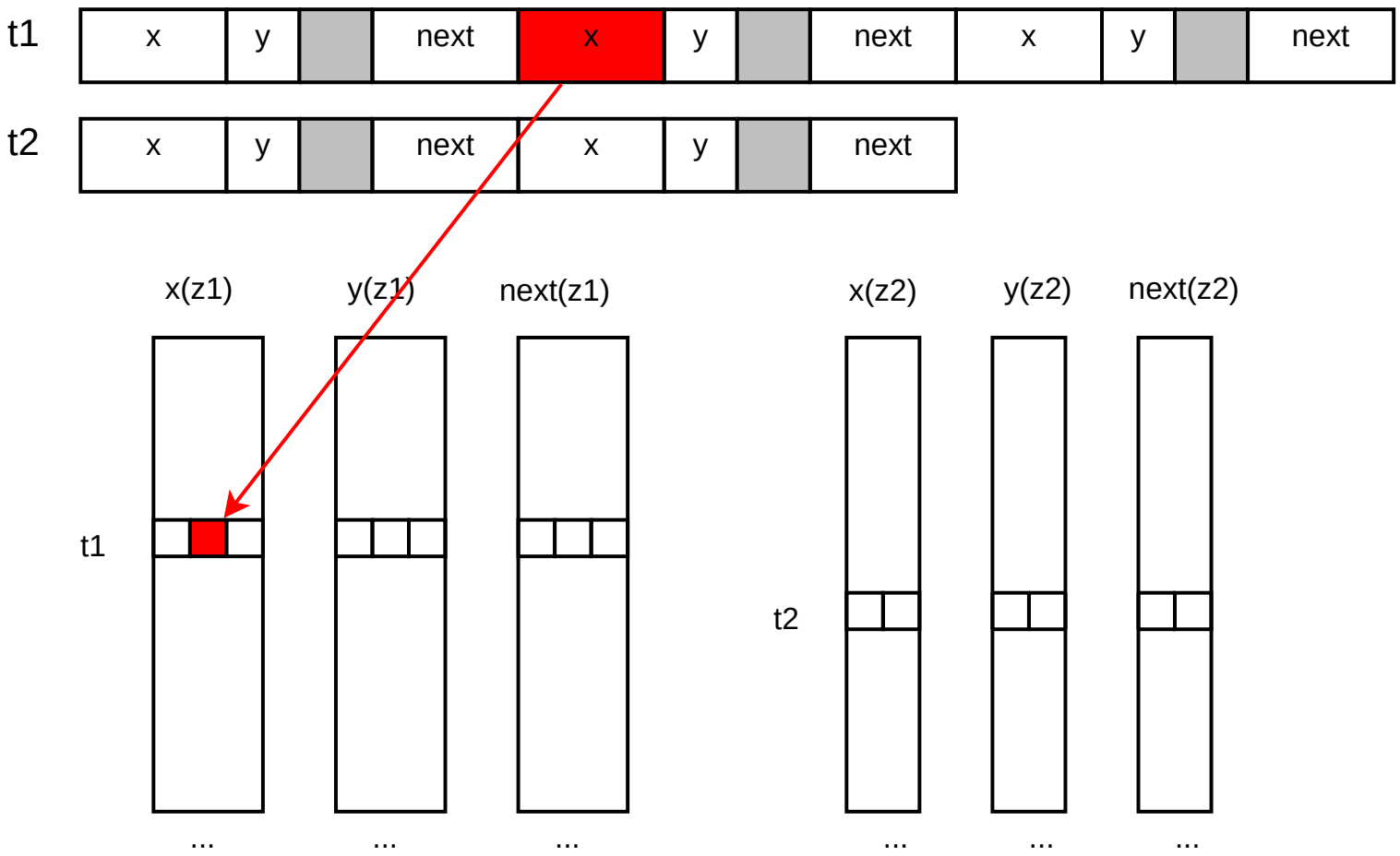
```
struct S { int x; short y; struct S *next; } t1[3], t2[2];
```





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*

t	2	1	0	4	0	5	3	0

n=5

u	2	1	4	5	3

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

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/`