Preuves de programmes et méthodes formelles

Jean-Jacques Lévy
INRIA

Microsoft TechDays - 9 février 2010

CENTRE DE RECHERCHE COMMUN

INRIA MICROSOFT RESEARCH

Plan

- Logique de Hoare
  - fibonacci
  - terminaison
  - drapeau hollandais
  - formules et règles d'inférence
  - fonctions récursives
- Logiques de programmes
- Mathématiques formelles
- Conclusion

Logique de Hoare

CENTRE DE RECHERCHE
COMMUN

INRIA
MICROSOFT RESEARCH

La suite de Fibonacci (1/6)

- Récurrence linéaire d'ordre 2

  $u_0 = 0$

  $u_1 = 1$

  $u_n = u_{n-1} + u_{n-2}$

  $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, \ldots$

- ```
  static int fib (int n) {
    if (n < 2) return n;
    else return fib (n-1) + fib (n-2);
  }
  ```

- ou en faisant un peu de programmation dynamique

  ```
  static int fib1 (int n) {
    int[ ] res = new int[n+1];
    res[0] = 0; res[1] = 1;
    for (int i=2; i <= n; ++i)
      res[i] = res[i-1]+res[i-2];
    return res[n];
  }
  ```

  ```

# La suite de Fibonacci (2/6)

- ou en ne gardant que les deux dernières valeurs $x$ et $y$

```
static int fibonacci (int n) {
  int x = 0;
  if (n != 0) {
    x = 1; int y = 0;
    for (int k=1; k != n; ++k) {
      int t = y;
      y = x;
      x = x + t;
    }
  }
  return x;
}
```

- On veut montrer que $n \geq 0 \Longrightarrow x = \mathsf{fib}(n)$ à la fin.

La suite de Fibonacci (3/6)

- ou en décomposant l'instruction for.

```
static int fibonacci (int n) {
  int x = 0;
  if (n != 0) {
    x = 1; int y = 0;
    int k = 1;
    while (k != n) {
      int t = y;
      y = x;
      x = x + t;
      ++k;
    }
  }
  return x;
}
```

La suite de Fibonacci (4/6)

- ```
  static int fibonacci (int n) {
    //P ={n ≥ 0}
    int x = 0;
    if (n != 0) {
      x = 1; int y = 0;
      int k = 1;

      while (k != n) {
        int t = y;
        y = x;
        x = x + t;
        ++k;
      }
    }
    //R ={x = fib(n)}
    return x;
  }
  ```
- $\mathcal{P}$ assertion d'entrée

  $\mathcal{R}$ assertion de fin
- Montrer que de $\mathcal{P}$, on peut dériver $\mathcal{R}$.
  ```

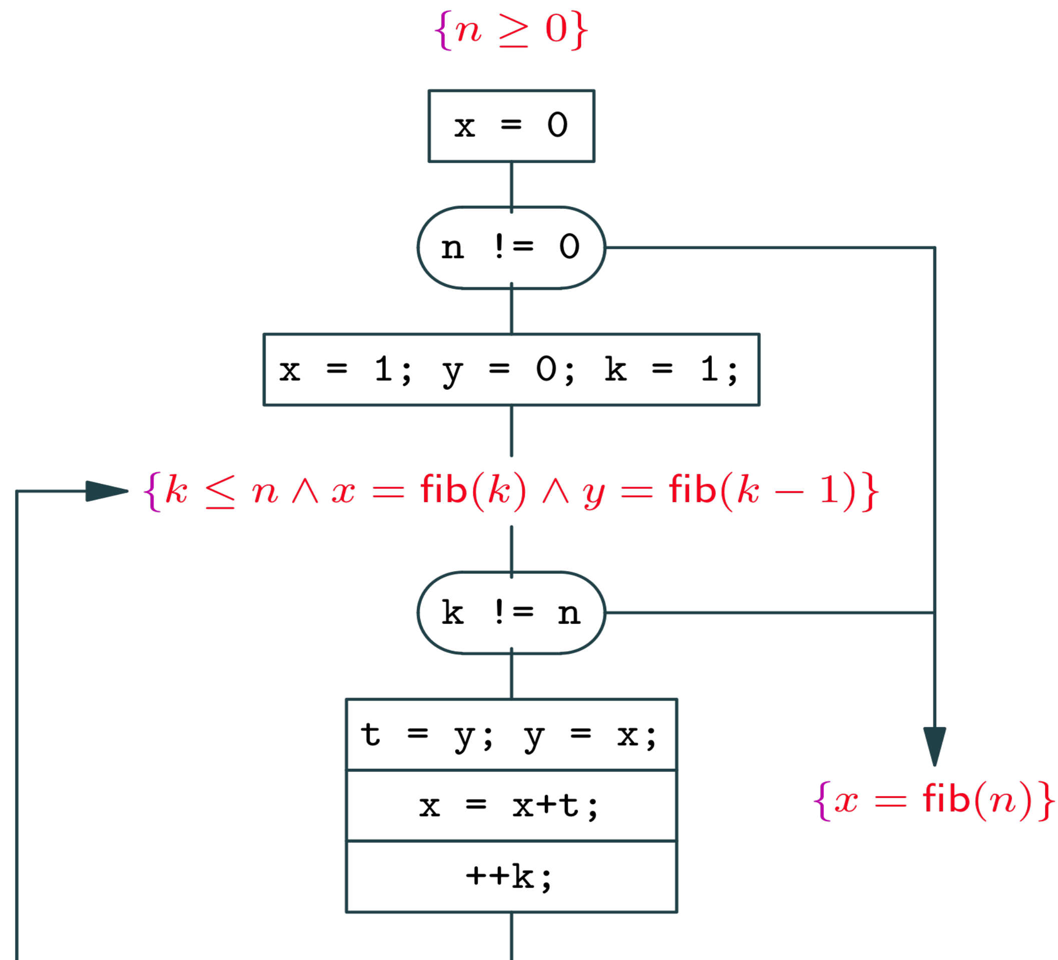
La suite de Fibonacci (4/6)

- ```
  static int fibonacci (int n) {
    //P ={n ≥ 0}
    int x = 0;
    if (n != 0) {
      x = 1; int y = 0;
      int k = 1;
      //Q ={k ≤ n ∧ x = fib(k) ∧ y = fib(k − 1)}
      while (k != n) {
        int t = y;
        y = x;
        x = x + t;
        ++k;
      }
    }
    //R ={x = fib(n)}
    return x;
  }
  ```

Trouver l'invariant !!!

- $\mathcal{P}$ assertion d'entrée
  $\mathcal{Q}$ assertion invariant de boucle
  $\mathcal{R}$ assertion de fin
- Montrer que de $\mathcal{P}$, on peut dériver $\mathcal{R}$.
  ```

La suite de Fibonacci (5/6)



La suite de Fibonacci (6/6)

- ```
  static int fibonacci (int n) {
    {n ≥ 0}
    int x = 0;

    if (n != 0) {
      x = 1; int y = 0;

      int k = 1;
      {k ≤ n ∧ x = fib(k) ∧ y = fib(k − 1)}
      while (k != n) {

        int t = y;

        y = x;

        x = x + t;

        ++k;
      }

    }
    {x = fib(n)}
    return x;
  }
  ```
  ```

La suite de Fibonacci (6/6)

- ```
  static int fibonacci (int n) {
    {n ≥ 0}
    int x = 0;
    {n ≥ 0 ∧ x = 0}
    if (n != 0) {
      x = 1; int y = 0;

      int k = 1;
      {k ≤ n ∧ x = fib(k) ∧ y = fib(k − 1)}
      while (k != n) {

        int t = y;

        y = x;

        x = x + t;

        ++k;
      }

    }
    {x = fib(n)}
    return x;
  }
  ```

CENTRE DE RECHERCHE COMMUN

INRIA MICROSOFT RESEARCH
  ```

La suite de Fibonacci (6/6)

```
• static int fibonacci (int n) {
    {n ≥ 0}
    int x = 0;
    {n ≥ 0 ∧ x = 0}
    if (n != 0) {
      x = 1; int y = 0;
      {n > 0 ∧ x = fib(1) ∧ y = fib(0)}
      int k = 1;
      {k ≤ n ∧ x = fib(k) ∧ y = fib(k − 1)}
      while (k != n) {

        int t = y;

        y = x;

        x = x + t;

        ++k;
      }

    }
    {x = fib(n)}
    return x;
  }
```

CENTRE DE RECHERCHE COMMUN

INRIA
MICROSOFT RESEARCH

La suite de Fibonacci (6/6)

- static int fibonacci (int n) {
  $\{n \geq 0\}$
  int x = 0;
  $\{n \geq 0 \wedge x = 0\}$
  if (n != 0) {
    x = 1; int y = 0;
    $\{n > 0 \wedge x = \mathsf{fib}(1) \wedge y = \mathsf{fib}(0)\}$
    int k = 1;
    $\{k \leq n \wedge x = \mathsf{fib}(k) \wedge y = \mathsf{fib}(k-1)\}$
    while (k != n) {

      int t = y;

      y = x;

      x = x + t;

      ++k;
    }
    $\{k = n \wedge x = \mathsf{fib}(k) \wedge y = \mathsf{fib}(k-1)\}$
  }
  $\{x = \mathsf{fib}(n)\}$
  return x;
}

CENTRE DE RECHERCHE COMMUN

INRIA
MICROSOFT RESEARCH

La suite de Fibonacci (6/6)

- ```
  static int fibonacci (int n) {
    {n ≥ 0}
    int x = 0;
    {n ≥ 0 ∧ x = 0}
    if (n != 0) {
      x = 1; int y = 0;
      {n > 0 ∧ x = fib(1) ∧ y = fib(0)}
      int k = 1;
      {k ≤ n ∧ x = fib(k) ∧ y = fib(k − 1)}
      while (k != n) {
        {k < n ∧ x = fib(k) ∧ y = fib(k − 1)}
        int t = y;

        y = x;

        x = x + t;

        ++k;
      }
      {k = n ∧ x = fib(k) ∧ y = fib(k − 1)}
    }
    {x = fib(n)}
    return x;
  }
  ```
  ```

La suite de Fibonacci (6/6)

- `static int fibonacci (int n) {`

  $\{n \geq 0\}$
  ```
  int x = 0;
  ```
  $\{n \geq 0 \wedge x = 0\}$
  ```
  if (n != 0) {
    x = 1; int y = 0;
  ```
  $\{n > 0 \wedge x = \mathsf{fib}(1) \wedge y = \mathsf{fib}(0)\}$
  ```
    int k = 1;
  ```
  $\{k \leq n \wedge x = \mathsf{fib}(k) \wedge y = \mathsf{fib}(k-1)\}$
  ```
    while (k != n) {
  ```
  $\{k < n \wedge x = \mathsf{fib}(k) \wedge y = \mathsf{fib}(k-1)\}$
  ```
      int t = y;
  ```
  $\{k < n \wedge x = \mathsf{fib}(k) \wedge y = \mathsf{fib}(k-1) \wedge t = \mathsf{fib}(k-1)\}$
  ```
      y = x;

      x = x + t;

      ++k;
    }
  ```
  $\{k = n \wedge x = \mathsf{fib}(k) \wedge y = \mathsf{fib}(k-1)\}$
  ```
  }
  ```
  $\{x = \mathsf{fib}(n)\}$
  ```
  return x;
  }
  ```

La suite de Fibonacci (6/6)

- static int fibonacci (int n) {

```
  {n ≥ 0}
  int x = 0;
  {n ≥ 0 ∧ x = 0}
  if (n != 0) {
    x = 1; int y = 0;
    {n > 0 ∧ x = fib(1) ∧ y = fib(0)}
    int k = 1;
    {k ≤ n ∧ x = fib(k) ∧ y = fib(k − 1)}
    while (k != n) {
      {k < n ∧ x = fib(k) ∧ y = fib(k − 1)}
      int t = y;
      {k < n ∧ x = fib(k) ∧ y = fib(k − 1) ∧ t = fib(k − 1)}
      y = x;
      {k < n ∧ x = fib(k) ∧ y = fib(k) ∧ t = fib(k − 1)}
      x = x + t;

      ++k;
    }
    {k = n ∧ x = fib(k) ∧ y = fib(k − 1)}
  }
  {x = fib(n)}
  return x;
}
```

CENTRE DE RECHERCHE COMMUN INRIA MICROSOFT RESEARCH

La suite de Fibonacci (6/6)

- static int fibonacci (int n) {

```
  {n ≥ 0}
  int x = 0;
  {n ≥ 0 ∧ x = 0}
  if (n != 0) {
    x = 1; int y = 0;
    {n > 0 ∧ x = fib(1) ∧ y = fib(0)}
    int k = 1;
    {k ≤ n ∧ x = fib(k) ∧ y = fib(k − 1)}
    while (k != n) {
      {k < n ∧ x = fib(k) ∧ y = fib(k − 1)}
      int t = y;
      {k < n ∧ x = fib(k) ∧ y = fib(k − 1) ∧ t = fib(k − 1)}
      y = x;
      {k < n ∧ x = fib(k) ∧ y = fib(k) ∧ t = fib(k − 1)}
      x = x + t;
      {k < n ∧ x = fib(k + 1) ∧ y = fib(k) ∧ t = fib(k − 1)}
      ++k;
    }
    {k = n ∧ x = fib(k) ∧ y = fib(k − 1)}
  }
  {x = fib(n)}
  return x;
}
```

Terminaison

```
static int fibonacci (int n) {
  {n ≥ 0}
  int x = 0;
  if (n != 0) {
    x = 1; int y = 0;
    int k = 1;
    {Ω(n, k) = n − k}
    while (k != n) {
      int t = y;
      y = x;
      x = x + t;
      ++k;
    }
  }
  {x = fib(n)}
  return x;
}
```

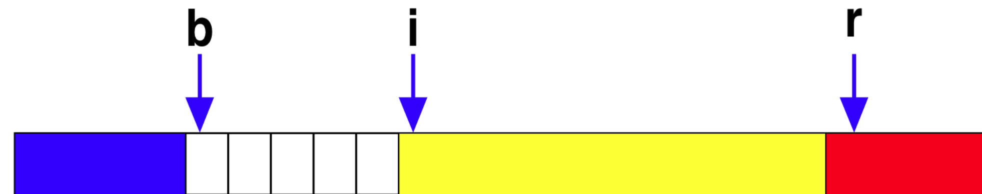
En un tour de boucle comme (n, k) devient $(n, k+1)$, on a $\Omega(n, k) = n - k > n - k - 1 = \Omega(n, k+1)$.

L'instruction while s'arrête donc.

Assertions

- les variables ont des valeurs modifiables
- une assertion est une propriété logique sur les valeurs de ces variables
- une assertion est attachée à un point du programme
- on montre l'assertion de fin à partir de l'assertion d'entrée par implications successives grâce à des assertions intermédiaires

Assertions et tableaux




```
static void drapeauHollandais (int[ ] a) { [Dijkstra]
  int n = a.length; int b = 0, i = 0, r = n;
  while (i < r) {
    switch (a[i]) {
    case BLEU:
      int t = a[b]; a[b] = a[i]; a[i] = t;
      ++b; ++i;
      break;
    case BLANC:
      ++i;
      break;
    case ROUGE:
      --r;
      int u = a[r]; a[r] = a[i]; a[i] = u;
      break;
    }
  }
}
```

Assertions et tableaux

```
static void drapeauHollandais (int[ ] a) {
  int n = a.length; int b = 0, i = 0, r = n;
  {ϕ(0, b, BLEU) ∧ ϕ(b, i, BLANC) ∧ ϕ(r, n, ROUGE) ∧ i ≤ r}
  while (i < r) {
    switch (a[i]) {
    case BLEU:
      {ϕ(0, b, BLEU) ∧ ϕ(b, i, BLANC) ∧ ϕ(r, n, ROUGE) ∧ a[i] = BLEU ∧ i < r}
      int t = a[b]; a[b] = a[i]; a[i] = t;
      {ϕ(0, b + 1, BLEU) ∧ ϕ(b + 1, i + 1, BLANC) ∧ ϕ(r, n, ROUGE) ∧ i < r}
      ++b; ++i; break;
    case BLANC:
      {ϕ(0, b, BLEU) ∧ ϕ(b, i + 1, BLANC) ∧ ϕ(r, n, ROUGE) ∧ i < r}
      ++i; break;
    case ROUGE:
      {ϕ(0, b, BLEU) ∧ ϕ(b, i, BLANC) ∧ ϕ(r, n, ROUGE) ∧ a[i] = ROUGE ∧ i < r}
      --r;
      {ϕ(0, b, BLEU) ∧ ϕ(b, i, BLANC) ∧ ϕ(r + 1, n, ROUGE) ∧ a[i] = ROUGE ∧ i ≤ r}
      int u = a[r]; a[r] = a[i]; a[i] = u;
      break;
  } }
  {ϕ(0, b, BLEU) ∧ ϕ(b, r, BLANC) ∧ ϕ(r, n, ROUGE)}
}
```

où $\phi(i, j, c) = \forall k.\ i \le k < j \Rightarrow a[k] = c$

Quelques principes logiques

- On a toujours
  $\{P(E)\}$ `x = ` E`;` $\{P(x)\}$

- Pour montrer
  $\{P\}$ if (E) S else S' $\{Q\}$
  il faut montrer
  $\{P \wedge E\}$ S $\{Q\}$
  et
  $\{P \wedge \neg E\}$ S' $\{Q\}$

- Pour montrer
  $\{P\}$ while (E) S $\{Q\}$
  il faut deviner l'invariant I, et montrer $P \Rightarrow I$ et $I \wedge \neg E \Rightarrow Q$ et
  $\{I \wedge E\}$ S $\{I\}$

cf. logique de `Floyd-Hoare`.

Dans le triplet $\{P\}$ S $\{Q\}$, on appelle P et Q des pré-condition et post-condition de S.

Logique de Hoare

$$\frac{}{\{P(E)\}\ \texttt{x = E;}\ \{P(x)\}} \qquad \frac{\{P \wedge E\}\ \texttt{S}\ \{Q\} \qquad \{P \wedge \neg E\}\ \texttt{S'}\{Q\}}{\{P\}\ \textsf{if}\ \texttt{(E)}\ \texttt{S}\ \textsf{else}\ \texttt{S'}\ \{Q\}}$$

$$\frac{\{P\}\ \texttt{S}\ \{Q\} \qquad \{Q\}\ \texttt{S'}\{R\}}{\{P\}\ \texttt{S}\ \texttt{S'}\{R\}} \qquad \frac{\{P\}\ \texttt{S}\ \{Q\}}{\{P\}\ \texttt{\{S\}}\ \{Q\}}$$

$$\frac{\{P \wedge E\}\ \texttt{S}\ \{P\}}{\{P\}\ \textsf{while}\ \texttt{(E)}\ \texttt{S}\ \{P \wedge \neg E\}} \qquad \frac{P \Rightarrow P' \qquad \{P'\}\ \texttt{S}\ \{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\ \texttt{S}\ \{Q\}}$$

Les formules sont des triplets $\{P\}S\{Q\}$.

Les prémisses d'une règle d'inférence sont au dessus de la barre.

La conclusion d'une règle d'inférence est en dessous.

Un axiome est une règle sans prémisse.

Correction partielle – Correction totale

- dans un triplet $\{P\}\ S\ \{Q\}$, rien ne garantit que S termine
- il y a correction partielle
- pour la correction totale, on utilise d'autres méthodes (ordinaux, ...) pour démontrer la terminaison
- les assertions doivent correspondre aux spécifications
- il y a toujours un fossé entre spécifications et assertions

Récursion et assertions

- On veut montrer $\{n \geq 0\}$ `r = fact(n);` $\{fact(n) = n!\}$
  Pour cela, on suppose (comme pour les boucles) que les appels (récursifs) utilisés vérifient déjà cette formule.

- ```
  static int fact (int n) {
    int r;
    {n ≥ 0 ∧ ∀n fact(n) = n! }
    if (n == 0) {
      {n = 0}
      r = 1;
      {n = 0 ∧ r = 1 = 0!}
    } else {
      {n − 1 ≥ 0 ∧ fact(n − 1) = (n − 1)!}
      r = n * fact(n-1);
      {r = n(n − 1)! = n!}
    }
    {r = n!}
    return r;
  }
  ```

- A nouveau, il s'agit de correction partielle.
  Un autre argument montre la terminaison.
  ```

Récursion et assertions

- On veut montrer $\{n \geq 0\}$ `r = f(n);` $\{f(n) = \text{if } n > 100 \text{ then } n - 10 \text{ else } 91\}$

- static int f (int n) { // ———————— fonction 91 [McCarthy]

```
  int r;
  {n ≥ 0 ∧ ∀n′ f(n′) = if n′ > 100 then n′ − 10 else 91}
  if (n > 100) {
    {n > 100}
    r = n-10;
    {n > 100 ∧ r = n − 10}
  } else {
    {n ≤ 100 ∧ ∀n′ f(n′) = if n′ > 100 then n′ − 10 else 91}
    r = f(f(n+11));
    {n ≤ 100 ∧ r = Φ(n)}     ↔     {n ≤ 100 ∧ r = 91}
  }
  {r = if n > 100 then n − 10 else 91}
  return r;
} E
```

$$
\begin{aligned}
\Phi(n) &= f(\text{if } n + 11 > 100 \text{ then } n + 1 \text{ else } 91) \\
&= \text{if } n \geq 90 \text{ then } f(n+1) \text{ else } f(91) \\
&= \text{if } n \geq 90 \text{ then if } n \geq 100 \text{ then } n - 9 \text{ else } 91 \text{ else } 91 \\
&= \text{if } n \geq 100 \text{ then } n - 9 \text{ else } 91
\end{aligned}
$$

Logiques de programme

CENTRE DE RECHERCHE
COMMUN

INRIA
MICROSOFT RESEARCH

Hoare et le reste

tests

environnements de programmation

CAO matériel

analyse statique

vérification formelle

spécification formelle

logique de Hoare

logiques temporelles

logique de la séparation

model checking

logiques d'ordre supérieure

Hoare et le reste

tests

environnements de programmation

emacs, visual studio, eclipse

CAO matériel

analyse statique

astrée, slam, fluctuat

vérification formelle

spécification formelle

logique de Hoare

krakatoa, spec#, frama-c

logiques temporelles

z, tla+

logique de la séparation

smallfoot

coq, isabelle, hol

logiques d'ordre supérieure

model checking

bdds, sat solvers, smt (z3)

pvs

Binary Search

Krakatoa Frama-C

prouvons la sûreté d'exécution

```
//@ requires t != null ;
static int binary_search(int t[], int v) {
    int l = 0, u = t.length - 1 ;
    //@ loop_invariant 0 <= l && u <= t.length-1 ;
    while (l <= u ) {
        int m = (l + u) / 2 ;
        if (t[m] < v) l = m ;
        else if (t[m] > v) u = m ;
        else return m ;
    }
    return -1 ;
}
```


CENTRE DE RECHERCHE COMMUN INRIA MICROSOFT RESEARCH

Binary Search

l'autre sens : si le résultat vaut -1 alors v n'apparaît pas dans t

⇒ il faut maintenant indiquer que le tableau est trié

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


CENTRE DE RECHERCHE COMMUN INRIA MICROSOFT RESEARCH

Binary Search

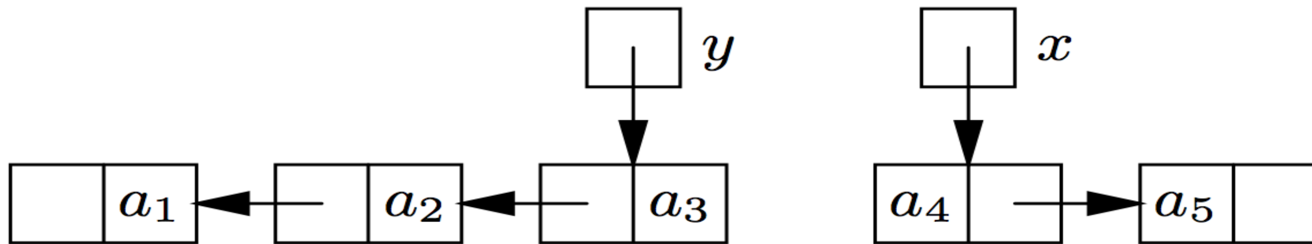
cela ne suffit pas : il faut aussi renforcer l'invariant de boucle

```
static int binary_search(int t[], int v) {
    int l = 0, u = t.length - 1;
    /*@ loop_invariant
      @   0 <= l && u <= t.length-1 &&
      @   \forall integer k;
      @       0 <= k < t.length ==> t[k] == v ==> l <= k <= u;
      @ ...
      @*/
    ...
}
```


CENTRE DE RECHERCHE COMMUN INRIA MICROSOFT RESEARCH

Avec des pointeurs

logique de la séparation




```
static Liste miroir (Liste y, Liste x) {
    while (x != null) {
        Liste z = x.suivant;
        x.suivant = y;
        y = x; x = z;
    }
    return y;
}

miroir (null, x)
```

Invariant $x'_0 = x' \cdot y$ où x_0 est la liste initiale

et x' est l'image miroir de x

Avec des pointeurs

- cet invariant n'est vrai que si les deux listes sont disjointes
- la logique de la séparation formule des propriétés par rapport à un certain tas `[o'hearn, reynolds]`
- un connecteur logique de base exprime que 2 tas sont disjoints
- propriétés modulaires sur les tas mémoire
- ...

Les diverses logiques de programme

- logique de Hoare pour programmes impératifs
- logique de la séparation pour programmes impératifs avec pointeurs
- logique temporelle pour programmes concurrents
- logique d'ordre supérieur pour programmes fonctionnels et mathématiques formelles
- démonstration automatique (model checking) pour calcul booléen et logique sur domaines finis.

Mathématiques formelles

(logique mathématique)

CENTRE DE RECHERCHE COMMUN

INRIA
MICROSOFT RESEARCH

Mathématiques formelles

- calcul booléen `[bdds,sat solvers]`
- logique de 1er ordre `[smt solvers, z3, zenon]`
- logique de 1er ordre + théorie des ensembles `[tla+]`
- logique d'ordre supérieure `[coq, ssreflect]`
- ...
- théories utiles pour les preuves de programme
- théorème des 4 couleurs `[gonthier]`, Feit-Thompson `[?]`
- ...

Conclusion

Quelques succès

- spécification système Météor (ligne 14) en Z [abrial]
- analyse statique PV Ariane 5 [deutsch et al, inria]
- analyse statique A380 [astrée, cousot et al, ens]
- compilateur C minor certifié [leroy, inria]
- slam/terminator/spec# teste les drivers de Windows [ball, cook, leino, rajamani, msr]
- ...
- protocoles de sécurité [msr, inria, ens, cachan]

Futur

- méthodes formelles moins chères (bibliothèques, modularité)
- programmes certifiés (avec certificat vérifiable)
- langages de programmation avec plus de facilités pour des spécifications logiques [F7]
- environnements de programmation avec analyses statiques
- vérification des programmes mobiles
- formation des ingénieurs
- processus biologiques vérifiés

CENTRE DE RECHERCHE COMMUN

INRIA
MICROSOFT RESEARCH