Ariane 502: analyse des variables partagées

(une application de la théorie de la concurrence?)

jeanjacqueslevy.net

INRIA Rocquencourt

École polytechnique

5 Octobre 2005

avec les participations de
Alain Deutsch, Damien Doligez, Robert Ehrlich,
Georges Gonthier, François Rouaix, et Marcin Skubiszewski

Plan

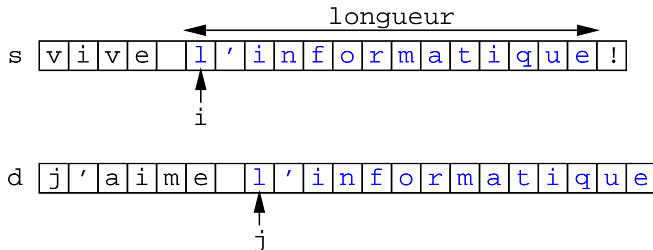
Alias

Ariane 501

Conclusion

Débarras

Un programme tout simple (1/5)

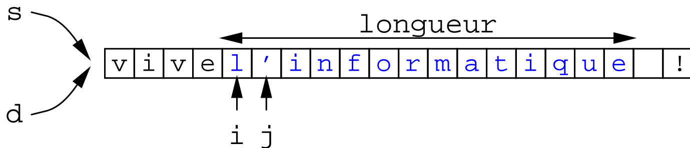



```
static void copie(char[ ] s, int i, char[ ] d, int j,
                  int longueur) {
  for (int k = 0; k < longueur; ++k)
    d[j + k] = s[i + k];
}
```

Complexité $O($`longueur`$)$. Programme correct ?

Un programme tout simple (2/5)

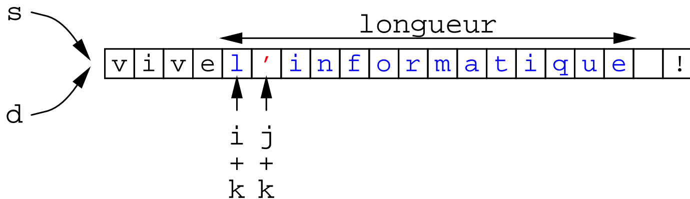
Non, car bogue si s et d sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

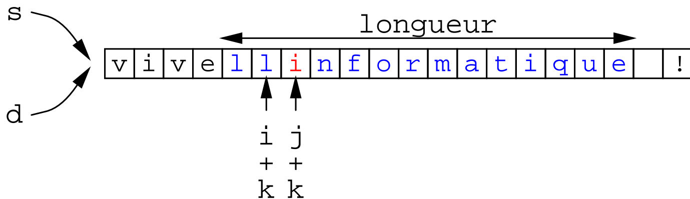
Non, car bogue si s et d sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

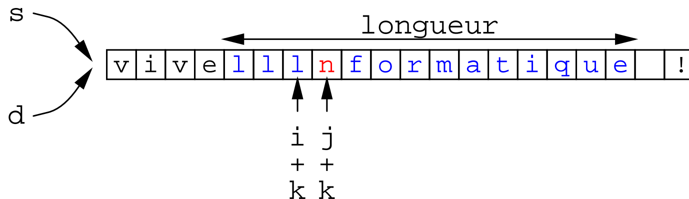
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

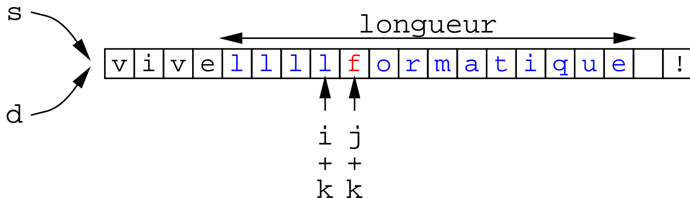
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

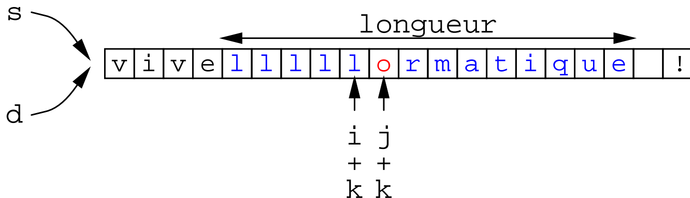
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

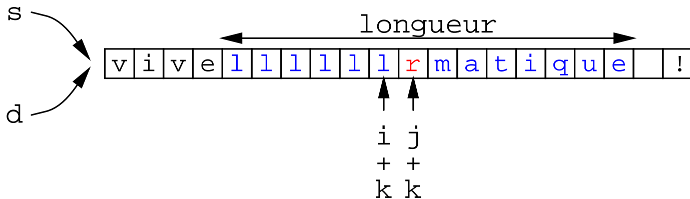
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

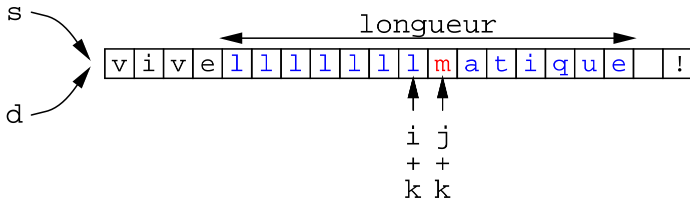
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

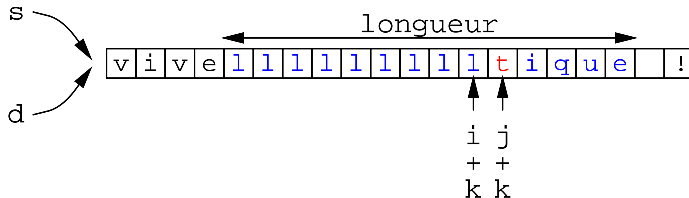
Un programme tout simple (2/5)

Non, car bogue si s et d sont deux alias ($s == d$)

Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

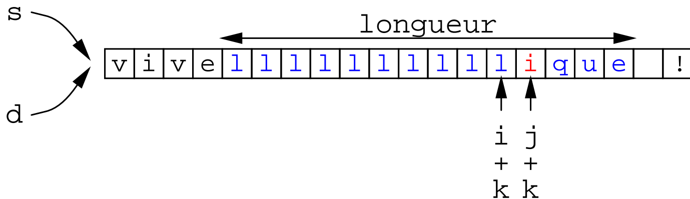
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

Non, car bogue si *s* et *d* sont deux alias ($s == d$)

s
d
longueur

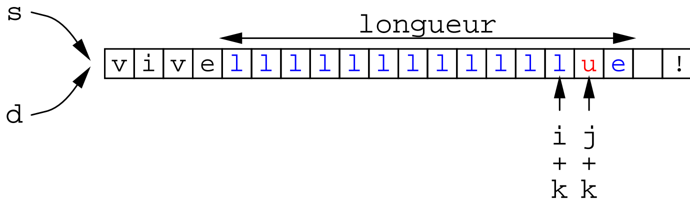
v	i	v	e	l	l	l	l	l	l	l	l	l	l	l	q	u	e		!

i + k j + k

Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

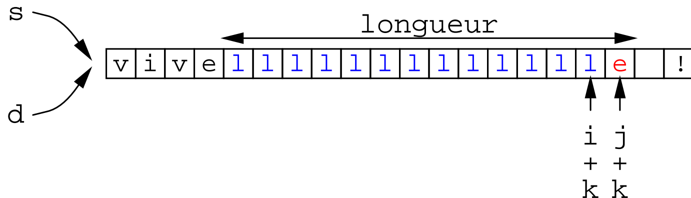
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

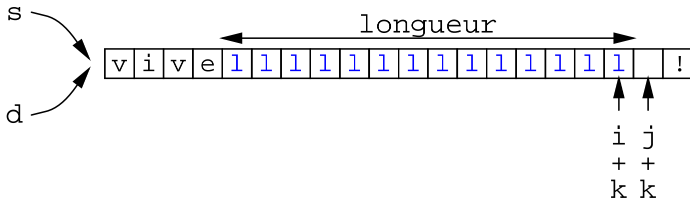
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

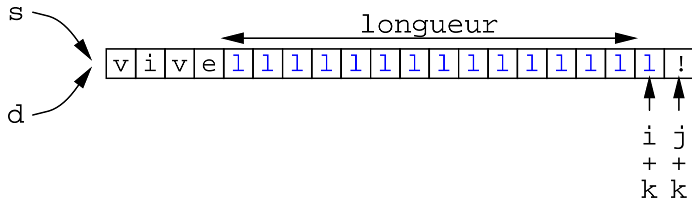
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

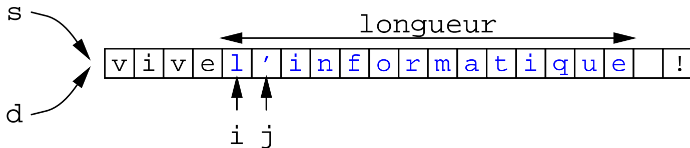
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

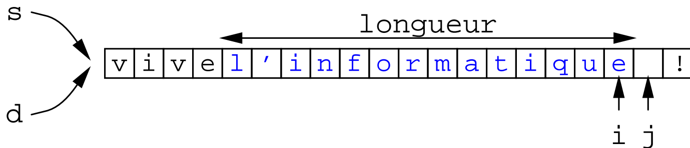
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

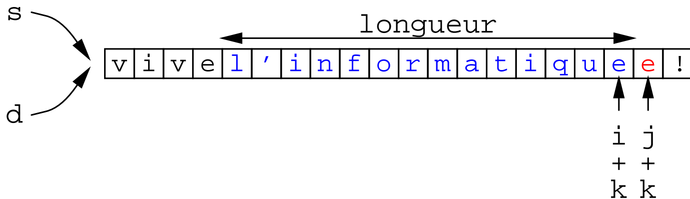
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

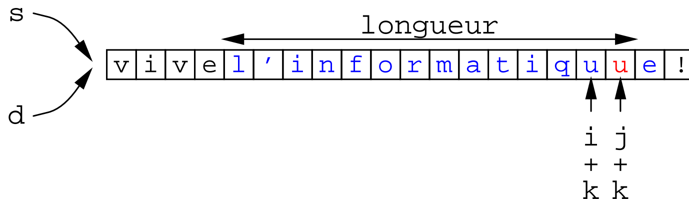
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

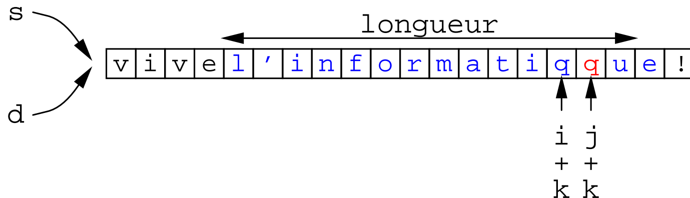
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

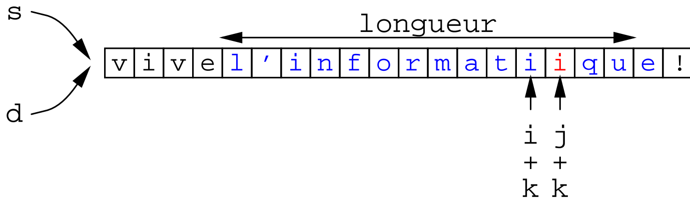
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

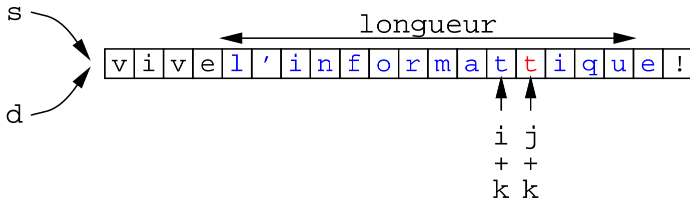
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

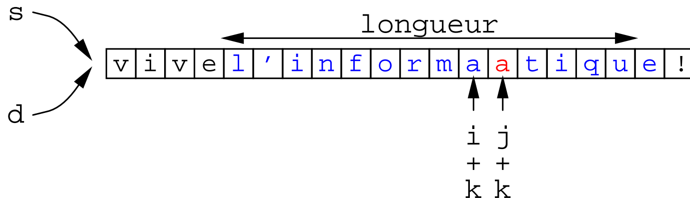
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

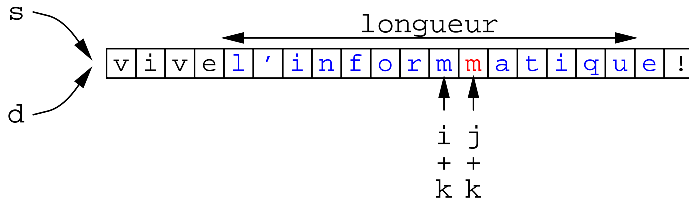
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

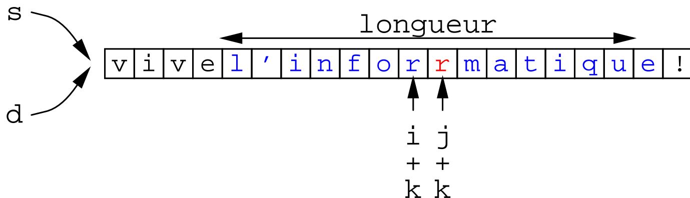
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

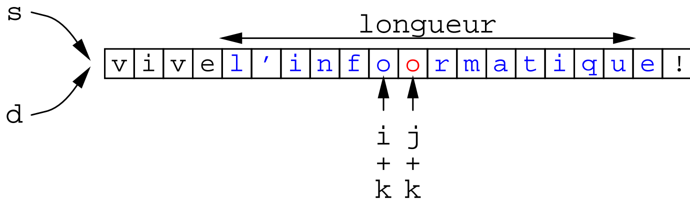
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

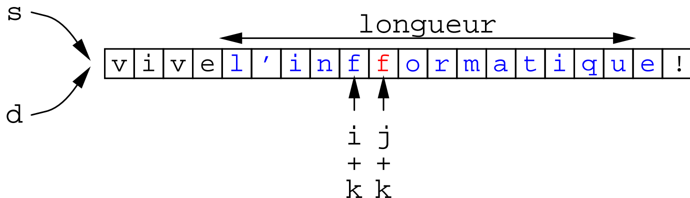
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

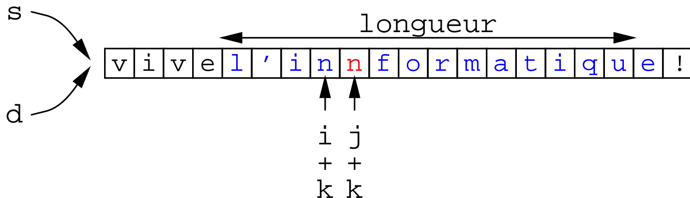
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

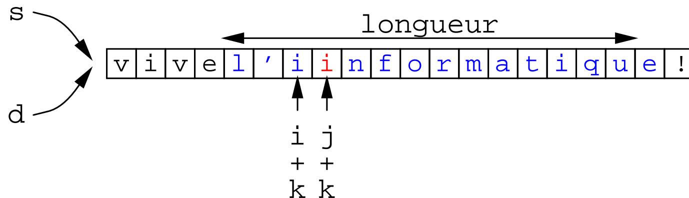
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

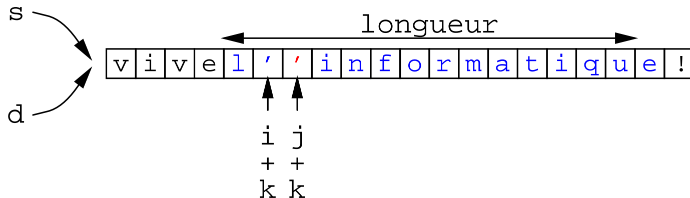
Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

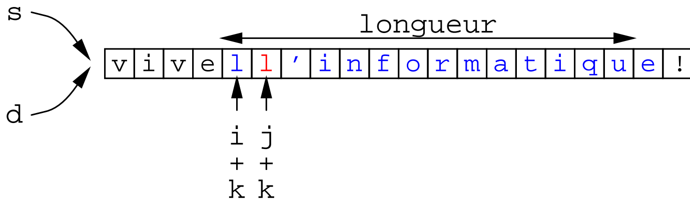
Non, car bogue si s et d sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

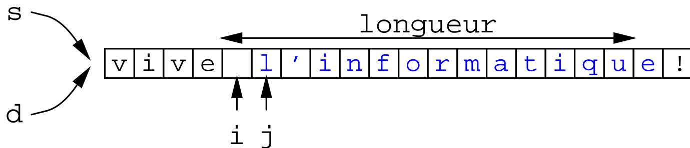
Non, car bogue si s et d sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (2/5)

Non, car bogue si *s* et *d* sont deux alias ($s == d$)



Erreur `copie` = alias mémoire

Un programme tout simple (3/5)

Le programme correct :

```
static void copie(char[ ] s, int i, char[ ] d, int j,
                  int longueur) {
  if (s != d || j <= i)
    for (int k = 0; k < longueur; ++k)
      d[j + k] = s[i + k];
  else
    for (int k = longueur - 1; k >= 0; --k)
      d[j + k] = s[i + k];
}
```

Un programme tout simple (4/5)

- alias inoffensifs si données non modifiables.
  (au contraire, ils améliorent l'occupation mémoire) ;
- langages fonctionnels (ML, Caml, Haskell) favorisent l'utilisation de données non modifiables.
- alias dangereux si données modifiables ;
- la recherche d'alias dans un programme est un problème important (Ariane 5, Word 97, Windows XP, etc)
  $\Rightarrow$ Analyse statique de programmes ;
- absence d'alias $\Rightarrow$ déplacements de code, indépendance de modules, etc.
- alias $\Rightarrow$ détection des variables partagées entre processus concurrents

Un programme tout simple (4/5)

- alias inoffensifs si données non modifiables.
  (au contraire, ils améliorent l'occupation mémoire) ;
- langages fonctionnels (ML, Caml, Haskell) favorisent l'utilisation de données non modifiables.
- alias dangereux si données modifiables ;
- la recherche d'alias dans un programme est un problème important (Ariane 5, Word 97, Windows XP, etc)
  $\Rightarrow$ Analyse statique de programmes ;
- absence d'alias $\Rightarrow$ déplacements de code, indépendance de modules, etc.
- alias $\Rightarrow$ détection des variables partagées entre processus concurrents

Un programme tout simple (4/5)

- alias inoffensifs si données non modifiables.
  (au contraire, ils améliorent l'occupation mémoire) ;
- langages fonctionnels (ML, Caml, Haskell) favorisent l'utilisation de données non modifiables.
- alias dangereux si données modifiables ;
- la recherche d'alias dans un programme est un problème important (Ariane 5, Word 97, Windows XP, etc)
  ⇒ Analyse statique de programmes ;
- absence d'alias ⇒ déplacements de code, indépendance de modules, etc.
- alias ⇒ détection des variables partagées entre processus concurrents

Un programme tout simple (4/5)

- alias inoffensifs si données non modifiables.
  (au contraire, ils améliorent l'occupation mémoire) ;
- langages fonctionnels (ML, Caml, Haskell) favorisent l'utilisation de données non modifiables.
- alias dangereux si données modifiables ;
- la recherche d'alias dans un programme est un problème important (Ariane 5, Word 97, Windows XP, etc)
  $\Rightarrow$ Analyse statique de programmes ;
- absence d'alias $\Rightarrow$ déplacements de code, indépendance de modules, etc.
- alias $\Rightarrow$ détection des variables partagées entre processus concurrents

Un programme tout simple (4/5)

- alias inoffensifs si données non modifiables.
  (au contraire, ils améliorent l'occupation mémoire) ;
- langages fonctionnels (ML, Caml, Haskell) favorisent l'utilisation de données non modifiables.
- alias dangereux si données modifiables ;
- la recherche d'alias dans un programme est un problème important (Ariane 5, Word 97, Windows XP, etc)
  $\Rightarrow$ Analyse statique de programmes ;
- absence d'alias $\Rightarrow$ déplacements de code, indépendance de modules, etc.
- alias $\Rightarrow$ détection des variables partagées entre processus concurrents

Un programme tout simple (4/5)

- alias inoffensifs si données non modifiables.
  (au contraire, ils améliorent l'occupation mémoire) ;
- langages fonctionnels (ML, Caml, Haskell) favorisent l'utilisation de données non modifiables.
- alias dangereux si données modifiables ;
- la recherche d'alias dans un programme est un problème important (Ariane 5, Word 97, Windows XP, etc)
  $\Rightarrow$ Analyse statique de programmes ;
- absence d'alias $\Rightarrow$ déplacements de code, indépendance de modules, etc.
- alias $\Rightarrow$ détection des variables partagées entre processus concurrents

Un programme tout simple (5/5)

- méthodes flow-sensitive (CFA) sont impraticables sur gros programmes ;
- méthodes *flow-insensitive* [Andersen 94] traitent en quelques secondes plusieurs $n \times 1000$ *locs* ;
- méthodes *flow-insensitive* et *context-insensitive* sont encore plus rapides. [Steensgaard 96, Das 00, Heinze/Tardieu 02]
- autres méthodes contextuelles. [Deutsch 94]

Ariane 501 (1/4)

Juin 1996

- Le lanceur a commencé à se désintégrer à environ H0 + 39 secondes sous l'effet de charges aérodynamiques dues à un angle d'attaque de plus de 20 degrés.
- cet angle d'attaque avait pour origine le braquage en butée des tuyères des moteurs à propergols solides et du moteur principal Vulcain ;
- le braquage des tuyères a été commandé par le logiciel du calculateur de bord (OBC) agissant sur la base des données transmises par le système de référence inertielle actif (SRI2). A cet instant, une partie de ces données ne contenait pas des données de vol proprement dites mais affichait un profil de bit spécifique de la panne du calculateur du SRI 2 qui a été interprété comme étant des données de vol ;
- la raison pour laquelle le SRI 2 actif n'a pas transmis des données d'attitude correctes tient au fait que l'unité avait déclaré une panne due à une exception logiciel ;
- l'OBC n'a pas pu basculer sur le SRI 1 de secours car cette unité avait déjà cessé de fonctionner durant le précédent cycle de données (période de 72 millisecondes) pour la même raison que le SRI 2 ;

Ariane 501 (2/4)

Juin 1996

- l'exception logiciel interne du SRI s'est produite pendant une conversion de données de représentation flottante à 64 bits en valeurs entières à 16 bits. Le nombre en représentation flottante qui a été converti avait une valeur qui était supérieure à ce que pouvait exprimer un nombre entier à 16 bits. Il en est résulté une erreur d'opérande. Les instructions de conversion de données (en code Ada) n'étaient pas protégées contre le déclenchement d'une erreur d'opérande bien que d'autres conversions de variables comparables présentes à la même place dans le code aient été protégées ;
- l'erreur s'est produite dans une partie du logiciel qui n'assure que l'alignement de la plate-forme inertielle à composants liés. Ce module de logiciel calcule des résultats significatifs avant le décollage seulement. Dès que le lanceur décolle, cette fonction n'est plus d'aucune utilité ;
- La fonction d'alignement est active pendant 50 secondes après le démarrage du mode vol des SRI qui se produit à H0 - 3 secondes pour Ariane 5. En conséquence, lorsque le décollage a eu lieu, cette fonction se poursuit pendant environ 40 secondes de vol. Cette séquence est une exigence Ariane 4 mais n'est pas demandée sur Ariane 5 ;

Ariane 501 (3/4)

Juin 1996

- l'erreur d'opérande s'est produite sous l'effet d'une valeur élevée non prévue d'un résultat de la fonction d'alignement interne appelé BH (Biais Horizontal) et lié à la vitesse horizontale détectée par la plate-forme. Le calcul de cette valeur sert d'indicateur pour la précision de l'alignement en fonction du temps ;
- la valeur BH était nettement plus élevée que la valeur escomptée car la première partie de la trajectoire d'Ariane 5 diffère de celle d'Ariane 4, ce qui se traduit par des valeurs de vitesse horizontale considérablement supérieures.

Erreur vol 501 = dépassement de capacité vers le haut

- Patch 502 : protection de la variable défaillante.

Ariane 501 (4/5)

Novembre 1996 – Février 1997

- code embarqué $=$ 140000 *locs* dans un sous-ensemble de Ada $+$ assembleur 68000 :
  - PV (programme de vol) [1, 2, 3]
  - LSSI (gestion de messages asynchrones avec périphériques)
  - SRI (centrale à inertie)
- analyseur d'alias (IABC) [Deutsch] pour sous-ensemble de C ($\simeq$ celui d'Ariane)
- compilation du code avec `gnat`
- code après la 1ère passe $\Rightarrow$ code C, entrée de IABC.

Ariane 501 (5/5)

Novembre 1996 – Février 1997

- index croisé des variables partagées entre processus concurrents grâce à IABC + outils publics (`graphviz`, `gnat`, `emacs`) ;
  [1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 , 16 , 17]
- « améliorations du code » [Georges Gonthier] [maintenant à Microsoft Research ☹] ;
- participations aux commissions de qualif. de 502 ;
- analyseur d'alias [Alain Deutsch] $\Rightarrow$ Polyspace tech. ;
- beaucoup de logiciels embarqués dans les satellites tournent l'analyseur de Deutsch ou d'autres (Astrée).

Conclusion

- depuis les débuts de l'informatique, il y a des bogues ;
- bonne connaissance de la programmation
  ⇒ moins de bogues ;
- l'analyse statique de code aide à les chasser ;
- outils automatiques nécessaires dans les cas critiques ;
- besoin d'outils logiques pour raisonner sur les programmes ;
- beaucoup de ces outils servent pour la sécurité ;
- théories des analyseurs statiques sont très riches : typage, contraintes, sémantique dénotationnelle, interp. abstraite, logique mathématique, concurrence, nouveaux lang. de programmation, etc.

Conclusion

- depuis les débuts de l'informatique, il y a des bogues ;
- bonne connaissance de la programmation
  $\Rightarrow$ moins de bogues ;
- l'analyse statique de code aide à les chasser ;
- outils automatiques nécessaires dans les cas critiques ;
- besoin d'outils logiques pour raisonner sur les programmes ;
- beaucoup de ces outils servent pour la sécurité ;
- théories des analyseurs statiques sont très riches : typage, contraintes, sémantique dénotationnelle, interp. abstraite, logique mathématique, concurrence, nouveaux lang. de programmation, etc.

Conclusion

- depuis les débuts de l'informatique, il y a des bogues ;
- bonne connaissance de la programmation
  $\Rightarrow$ moins de bogues ;
- l'analyse statique de code aide à les chasser ;
- outils automatiques nécessaires dans les cas critiques ;
- besoin d'outils logiques pour raisonner sur les programmes ;
- beaucoup de ces outils servent pour la sécurité ;
- théories des analyseurs statiques sont très riches : typage, contraintes, sémantique dénotationnelle, interp. abstraite, logique mathématique, concurrence, nouveaux lang. de programmation, etc.

Conclusion

- depuis les débuts de l'informatique, il y a des bogues ;
- bonne connaissance de la programmation
  $\Rightarrow$ moins de bogues ;
- l'analyse statique de code aide à les chasser ;
- outils automatiques nécessaires dans les cas critiques ;
- besoin d'outils logiques pour raisonner sur les programmes ;
- beaucoup de ces outils servent pour la sécurité ;
- théories des analyseurs statiques sont très riches : typage, contraintes, sémantique dénotationnelle, interp. abstraite, logique mathématique, concurrence, nouveaux lang. de programmation, etc.

Conclusion

- depuis les débuts de l'informatique, il y a des bogues ;
- bonne connaissance de la programmation
  $\Rightarrow$ moins de bogues ;
- l'analyse statique de code aide à les chasser ;
- outils automatiques nécessaires dans les cas critiques ;
- besoin d'outils logiques pour raisonner sur les programmes ;
- beaucoup de ces outils servent pour la sécurité ;
- théories des analyseurs statiques sont très riches : typage, contraintes, sémantique dénotationnelle, interp. abstraite, logique mathématique, concurrence, nouveaux lang. de programmation, etc.

Conclusion

- depuis les débuts de l'informatique, il y a des bogues ;
- bonne connaissance de la programmation
  $\Rightarrow$ moins de bogues ;
- l'analyse statique de code aide à les chasser ;
- outils automatiques nécessaires dans les cas critiques ;
- besoin d'outils logiques pour raisonner sur les programmes ;
- beaucoup de ces outils servent pour la sécurité ;
- théories des analyseurs statiques sont très riches : typage, contraintes, sémantique dénotationnelle, interp. abstraite, logique mathématique, concurrence, nouveaux lang. de programmation, etc.

Conclusion

- depuis les débuts de l'informatique, il y a des bogues ;
- bonne connaissance de la programmation
  $\Rightarrow$ moins de bogues ;
- l'analyse statique de code aide à les chasser ;
- outils automatiques nécessaires dans les cas critiques ;
- besoin d'outils logiques pour raisonner sur les programmes ;
- beaucoup de ces outils servent pour la sécurité ;
- théories des analyseurs statiques sont très riches : typage, contraintes, sémantique dénotationnelle, interp. abstraite, logique mathématique, concurrence, nouveaux lang. de programmation, etc.

Débarras

- Doc Sextant
- Doc Sextant 2
- PV non scalar
- PV scalar
- PV listing