Pour aller plus loin : autres points, bibliographie

L'interprétation abstraite et l'analyse statique forment un domaine très riche.

Une bonne introduction générale à l'interprétation abstraite :

Patrick Cousot, Radhia Cousot, Abstract interpretation and application to logic programs, *Journal of Logic Programming*, 1992. `http://www.di.ens.fr/~cousot/COUSOTpapers.shtml`

Reinhard Wilhelm, Dieter Maurer, *Les compilateurs — théorie, construction, génération*, Masson, 1994.

Analyse dataflow (cas particuliers) : n'importe quel bon livre de compilation :

Alfred V. Aho, Ravi Sethi, Jon D. Ullman, *Compilateurs : principes, techniques et outils*, InterEditions, 1989.

Richard J. LeBlanc, Charles N. Fischer, *Crafting a compiler*, Benjamin-Cummings, 1988.

Quelques points subtils

• les élargissements perdent parfois énormément d'information; on peut alors raffiner le résultat d'un élargissement par l'utilisation d'un rétrécissement Δ permettant de regagner un peu de l'information perdue : cf. Cousot&Cousot, JLP 1992 (op.cit.)

• l'analyse statique consiste à résoudre des équations sémantiques abstraites; par ex., si $1 \stackrel{\texttt{x:=x+y}}{\longrightarrow} 2$, l'équation correspondante pour la sémantique avant est $\rho_2^\sharp = \rho_1^\sharp[x \leftarrow \mathcal{E}[\![x + y]\!]^\sharp \rho_1^\sharp]$. Il est parfois intéressant (efficacité) de ne pas chercher à les résoudre directement, mais à les produire par interprétation abstraite, puis à les résoudre dans un ordre indépendant de l'ordre de parcours du graphe de contrôle : ce sont les itérations chaotiques. (Cas typique : équations sur les polyèdres convexes de Cousot&Halbwachs 1978, cf. transparent suivant.)

Analyses relationnelles

Permettent d'exprimer des relations entre variables, par exemple $x = y + 3$.

– Équations affines (efficace, précis, simple) : Michael Karr, Affine relationships among variables of a program, *Acta Informatica* 6, 133–151, 1978

– Équations et inégalités affines (précis, complexe, fondamental en parallélisation automatique de code Fortran) : Patrick Cousot, Nicolas Halbwachs, Automatic discovery of linear restraints among variables of a program, *5th ACM Symposium on Principles of Programming Languages (POPL'78)*, 84–97, 1978.

+ toutes publications par Paul Feautrier, François Irigoin, et d'autres.

– Évaluation symbolique : J. H. Reif, H. R. Lewis, Symbolic evaluation and the global value graph, *4th ACM Symposium on Principles of Programming Languages (POPL'77)*, 104–118, 1977.

Pointeurs

En présence de pointeurs, les analyses sont plus difficiles :

```
x->a = 3;
y->a = 5;
z = x->a; /* peut-on remplacer ceci par z = 3 ? */
```

... uniquement si on est sûr que `&x->a`≠`&y->a`, autrement dit que `x->a` et `y->a` ne peuvent pas être des alias.

Il est en général nécessaire de détecter les alias possibles :

• Solution simple : toutes les données dans le tas sont des alias possibles;

• Dans un langage fortement typé (pas en C), on peut raffiner : toutes les données dans le tas *de même type* sont des alias possibles; dans les langages orientés objet (Java), toutes les données *de types comparables* dans la relation de sous-classage sont des alias possibles.

Analyses d'alias précises

– Très précis, très complexe : Alain Deutsch, A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations, *4th International Conference on Computer Languages*, 2–13, 1992

Alain Deutsch, Interprocedural may-alias analysis for pointers: beyond k-limiting, *ACM Conference on Programming Language Design and Implementation (PLDI'94)*, 230–241, 1994

– Assez simple, suffisamment précis en général : Mooly Sagiv, Thomas Reps, Reinhardt Wilhelm, Solving shape-analysis problems in languages with destructive updating, *ACM Transactions on Programming Languages and Systems* 20(1), 1998, pp. 1-50. En général, voir `http://www.cs.wisc.edu/~reps/`

Analyses interprocédurales

Les langages réels ont toujours en plus des notions de procédures (= fonctions, sous-programmes, etc.). L'analyse du corps d'une procédure est dite intraprocédurale; si elle se prolonge à plusieurs procédures, elle est interprocédurale.

• Solution simple : les points de programme sont affectés à chaque instruction de chaque procédure. Récursion $\sim$ boucles $\Rightarrow$ calculs de points fixes et élargissements.

En pratique, ce n'est pas assez précis; si :

```
int f (int i) {
1:  int j = i+1; return j; }
```

et on appelle `f` sur 3, 10 et -2, alors `1 :` $j : ([-2, 10])$, donc `f (3)`: $([-2, 10])$, `f (10)`: $([-2, 10])$, `f (-2)`: $([-2, 10])$.

• Solution par évaluation symbolique (Callahan, Cooper, Kennedy & Torczon 1986) : le corps de `f` est analysé par propagation de constante symbolique, et est résumé par des relations de la forme `j`-final$=$`i`-initial$+1$. Donc `f (3)`$= 3 + 1 = 4$. Problème : ne fonctionne pas en présence de récursion. Amélioration en : Mooly Sagiv, Thomas Reps, Susan Horwitz, *Precise interprocedural dataflow analysis with applications to constant propagation*, Colloquium on Formal Approaches in Software Engineering (FASE'95), Springer Verlag LNCS 915, 651–665, 1995. `http://www.cs.wisc.edu/wpis/papers/tapsoft95.ps`

• Partitionnement statique : $\equiv$ faire k copies de `f` (par ex. $k = 3$), et décider à l'avance quels sites d'appels `f (...)` appellent quelles versions de `f`

• Partitionnement dynamique : François Bourdoncle, *Abstract interpretation by dynamic partitioning*, Journal of Functional Programming, 2(4), 407-435, 1992.

Parallélisme

• Détection de problèmes dûs au parallélisme : Interblocages (deadlocks : 2 processus s'attendent mutuellement indéfiniment), indéterminisme (race conditions : 2 processus accèdent à une même ressource, le plus rapide l'acquérant), etc.

P. Lacan, J.N. Montfort, Le Vinh Quy Ribal, Alain Deutsch, Georges Gonthier, The software reliability verification process: The ARIANE 5 Example. *DAta Systems In Aerospace (DASIA'98)*, ESA Publications, SP-422, 1998

• Détection de problèmes usuels (valeurs arithmétiques, etc.) en présence de parallélisme : complexité accrue à cause de l'explosion combinatoire du nombre d'entrelacements possibles (interleavings).

Typage

… une autre façon d'exprimer des propriétés sur des programmes, parfois plus facile à comprendre, mais parfois plus difficiles à étendre.

– Découverte d'expressions parallélisables par systèmes de types et d'effets : J. M. Lucassen, Dave M. Gifford, Polymorphic effect systems, *15th ACM Symposium on Principles of Programming Languages (POPL'88)*, 1988

Jean-Pierre Talpin, Pierre Jouvelot, Polymorphic type, region and effect inference, *Journal of Functional Programming* 2(3), 1992.

– Sécurité du code mobile : François Rouaix, Xavier Leroy, Security properties of typed applets, *25th ACM Symposium on Principles of Programming Languages (POPL'98)*, 1998.

– En principe (si correcte), le typage est une interprétation abstraite :

Patrick Cousot, Types as abstract interpretations, *24th ACM Symposium on Principles of Programming Languages (POPL'97)*, 1997.

Vérification de propriétés complexes

Nous n'avons vu que des propriété de sûreté (safety) : "quoi qu'il arrive, la propriété P est vraie". En particulier, invariants : "au point p, $x \in [-2, 10]$".

On peut s'intéresser à d'autres propriétés :
- "le programme P termine toujours/parfois";
- "dans toute trace, P finit toujours par écrire sur `stdout`";
- "dans toute trace, P écrit sur `stdout` infiniment souvent"; etc.

Ce sont des propriétés temporelles. La vérification de propriétés temporelles (model-checking) sur des programmes est le debugging abstrait : François Bourdoncle, *Abstract debugging of higher-order imperative languages*, ACM SIGPLAN Notices, 28(6), 46-55, 1993.

Dave A. Schmidt, Bernhard Steffen, Data-flow analysis as model checking of abstract interpretations, *5th Static Analysis Symposium (SAS'98)*, 1998, Springer Verlag LNCS 1503.

Analyses statiques d'autres langages

Chaque langage présente des particularités intéressantes à analyser :

• Langages orientés objet (Java, C++, Smalltalk) : détermination des classes réelles à l'exécution d'un objet.

– si x est forcément de classe `C`, on peut compiler x.`method`(a) par un appel direct à la méthode `C.method`;

– si x est de classe $\mathtt{C}_1, \ldots,$ ou $\mathtt{C}_n$, compiler x.`method`(a) par un `switch`.

Dans ces deux cas, ça permet d'affiner d'autres analyses, et aide à la relecture du code (!)

Craig Chambers, Jeffrey Dean, David Grove, Whole-program optimization of object-oriented languages, Technical report, University of Washington, 1996. `http://www.cs.washington.edu/research/projects/cecil/www/www/Papers/whole-program.html`. Autres références en `http://www.cs.washington.edu/research/projects/cecil/www/www/Papers/papers.html`.

- Langages Prolog et affiliés :

– Analyses de modes : paramètre d'entrée ($-$), de sortie ($+$), inutilisé (0), d'entrée-sortie ($\pm$); permet d'éviter l'unification et de compiler la résolution de but par un appel de procédure;

– Analyses de groundness : si on sait qu'un terme passé à un prédicat est toujours *ground* (sans variable libre), on peut remplacer l'unification par le pattern-matching, moins coûteux;

– Analyses de types : permet d'optimiser les programmes; de détecter quels sous-buts peuvent être engendrés (analyse avant), quels buts peuvent réussir (analyse arrière).

cf. papiers de Manuel Hermenegildo, Saumia Daubray, Andreas Podelski, Witold Charatonik, C.S. Mellish, Alan Mycroft, etc. Consulter leurs pages Web, suivre les références à partir de Cousot&Cousot, J. Logic Programming, 1992 (op.cit.)

• Langages fonctionnels (ML, Miranda, Haskell, Lisp) :

– analyse de types : imposée par ML, Miranda, Haskell; pour optimiser le code en Common-Lisp;

– analyse de nécessité pour les langages paresseux (Miranda, Haskell) : si un argument e est de toute façon utilisé par f dans $f(e)$, on peut directement évaluer e avant de le passer à f;

Patrick Cousot, Radhia Cousot, Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), *Proceedings of the 1994 International Conference on Computer Languages (ICCL'94)*, 95–112, 1994. Suivre les références.

– analyse d'allocations (boxings) : pour éviter d'allouer des données qui ne servent qu'à lire leurs composantes;

Jean Goubault, Generalized boxings, congruences and partial inlining, *Proceedings of the 1st Static Analysis Symposium (SAS'94)*, Springer Verlag LNCS 864, 1994. Suivre les références.

– etc.

Autres applications

– Vérification d'invariants dans des bases de données (par ex., si $homme(m)$ et $fils(m) = x$ alors $pere(x) = m$); traditionnellement opérée par vérifications coûteuses à l'exécution des requêtes. Ces vérifications peuvent être éliminées par analyse statique :

Véronique Benzaken, Anne Doucet, Xavier Schaefer, Integrity constraint checking optimization based on abstract databases generation and program analysis, *Journal de l'ingénierie des systèmes d'information* 1(3), 1995.

– Vérification formelle de protocoles cryptographiques :

Jean Goubault-Larrecq, A method for automatic cryptographic protocol verification, *Workshop on formal methods in parallel programming, theory and applications (FMPPTA'2000)*, Springer Verlag, 2000; suivre les références.

Et bien d'autres domaines ! À vous d'en imaginer maintenant…