Utilisation de l’analyse statique comme outil d’aide au développement

par

Yves Gauthier

essai présenté au Département d'informatique
en vue de l'obtention du grade de maître en technologies de l’information
(maîtrise en génie logiciel incluant un cheminement de type cours en technologies de l’information)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Longueuil, Québec, Canada, décembre 2008

Sommaire

Historiquement, les programmes sont vérifiés à l'aide d'outils permettant de suivre à la trace l'exécution d'un programme et d'observer le comportement de l'exécution du code. L'utilisation de tels outils est fastidieuse, car elle nécessite la construction de données de test pour chacun des cas possibles. Chacun de ces cas d'essai doit être exécuté manuellement pour vérifier le comportement du programme sur ces données.

Assurer que les programmes ont le bon comportement sous les différentes conditions d'utilisation a toujours été un objectif important des concepteurs de logiciels et des gestionnaires de projets. Les langages de programmation avec des types de données bien définis et des compilateurs de plus en plus sophistiqués ont beaucoup aidé à améliorer la qualité des programmes en détectant plusieurs erreurs sémantiques.

Les différentes techniques d'analyse statique représentent l'étape suivante en ce qui concerne les outils d'aide au développement de programmes. Ces outils permettent de détecter automatiquement plusieurs classes d'erreurs de programmation sans avoir à exécuter les programmes.

Le présent document présente une méthode d'analyse statique qui peut être utilisée en complément des essais pour permettre la détection automatique de certains problèmes de programmation.

L'essai présente l'analyse statique, décrit sommairement l'étendue des applications où ces méthodes peuvent être utilisées et guide le lecteur dans les détails d'implémentation d'une méthode d'analyse statique.

Une entreprise désirant utiliser l'analyse statique comme outil d'aide au développement a plusieurs décisions à prendre.

La première décision consiste à définir clairement le ou les problèmes à détecter. Le ou les problèmes sélectionnés vont aider à choisir une représentation des données. Le choix d'une représentation adéquate permet de simplifier la détection des anomalies ciblées.

La seconde décision revient à faire un choix sur la méthode de transformation du programme source. Les techniques de compilation sont très utiles à ce stade, car elles fournissent plusieurs algorithmes de transformation de code qui sont utilisables dans le cadre de l'analyse statique.

La dernière décision à prendre est de déterminer le format de présentation du diagnostic produit par l'outil d'analyse. Lorsque le programme a été transformé, l'algorithme d'analyse statique est exécuté sur la représentation interne et des erreurs sont détectées. La détection d'une erreur possible est le but des outils d'analyse. Par contre, pour être vraiment utile, il faut que les outils d'analyse produisent des diagnostics qui aident à identifier la source de ces anomalies et guident l’utilisateur vers leur solution.

Remerciements

J’aimerais, premièrement et tout particulièrement, remercier tous les membres de ma famille qui m’ont encouragé et qui ont dû subir des absences prolongées pendant ces longues années.

J’aimerais également remercier tous les professeurs du programme de maîtrise qui ont su capter mon attention et attiser mon imagination, tout en me fournissant une bonne dose d’encouragements.

Je remercie également mon employeur qui m’a encouragé à m’inscrire au programme et m’a donné l’opportunité de le poursuivre jusqu’à la fin.

Finalement, j’aimerais remercier tout le personnel de l’Université de Sherbrooke qui m’a aidé de près ou de loin, plus particulièrement mes directeurs d’essai qui ont accepté de m’accompagner et me guider lors de cette dernière étape malgré leur horaire très chargé.

Table des matières

Liste des tableaux

Liste des figures

Introduction

Le type d'entreprise qui sert de modèle et qui amène à rédiger cet essai est aux prises avec plusieurs problèmes qui nuisent à la qualité du code produit. Le système informatique principal a été écrit au début des années 1990 par une équipe d'une vingtaine de personnes. Ce système, qui à l'origine devait gérer une ligne de produits, a été modifié au cours du temps pour permettre la mise en marché de trois nouvelles lignes de produits. La majorité des personnes qui ont contribué à la mise en place du système original ne travaillent plus pour l'entreprise aujourd'hui. Cette situation se retrouve dans un grand nombre d'entreprises qui exploitent des systèmes propriétaires. Typiquement, ces systèmes sont assez volumineux, car ils comprennent plusieurs millions de lignes de code.

Les multiples changements apportés à ces systèmes, combinés à la documentation souvent inadéquate et un taux de roulement de personnel, obligent à valider plus en profondeur le code pour s'assurer de son bon fonctionnement. Par contre, les programmeurs sont souvent amenés à sacrifier sur la durée et la qualité des essais afin de respecter les échéanciers. Pour cette raison, plusieurs défauts sont détectés par les utilisateurs du système en production. Par exemple, il y a environ cinq ans, le nombre de demandes de corrections actives enregistrées par les utilisateurs du système était d'environ cinquante. Actuellement, il y a plus de 350 demandes de corrections actives enregistrées par les utilisateurs du système et environ cinquante nouvelles demandes sont reçues chaque jour. De plus, plusieurs transactions se terminent anormalement chaque jour en production et le traitement différé qui est effectué de nuit rencontre également des événements qui nécessitent une intervention à distance quotidiennement. Une analyse des incidents rencontrés révèle plusieurs lacunes qui reviennent fréquemment : l'utilisation de variables qui ne sont pas toujours initialisées et le débordement fréquent de tables en mémoire.

Pour faire face à un problème de qualité de plus en plus pressant, il est nécessaire de faciliter le travail des programmeurs en permettant de diminuer le temps de vérification et faciliter la détection des cas d'erreurs les plus fréquemment rencontrées. L'objectif principal de l'essai est de démontrer comment l'analyse statique du code peut être utilisée pour adresser cette problématique.

L'objectif de l'analyse de programmes est de déterminer automatiquement les propriétés d'un programme. Les outils de développement, tels que les compilateurs, les vérificateurs de programmes et les systèmes d'aide à la compréhension de programmes dépendent largement de l'analyse statique. L'essai va démontrer le fonctionnement de l'analyse statique de façon suffisamment détaillée pour permettre l'implémentation des algorithmes présentés. L'essai guide vers les sources pertinentes pour un apport théorique complet, mais n'a pas la prétention de démontrer clairement tous les points de la théorie utilisée.

Les références disponibles sur l'analyse statique sont très détaillées au point de vue de la théorie, mais touchent très peu l'aspect de l'implémentation. L'essai vise à combler ce manque en se concentrant sur trois facettes qui ne sont pas ciblées par la littérature : comment utiliser l'analyse statique pour détecter plus d'un type de problème à la fois, comment construire un outil d'analyse statique pour détecter ces incidents et la production du diagnostic des occurrences détectées.

L'essai utilise la littérature spécialisée pour couvrir l'aspect théorique et choisir un algorithme d'analyse. Cet algorithme est ensuite expliqué en détail. Dans ce sens, les références utilisées sont bien connues dans le domaine de l'analyse statique. La portion théorique est principalement tirée des travaux de Patrick Cousot et Radhia Cousot; ces derniers oeuvrant dans le domaine de l'analyse statique depuis 1975. L'information sur la théorie des compilateurs ainsi que l'algorithme provient principalement du livre d'Aho, Lam, Sethi et Ullman [1] qui est une référence classique dans le domaine des compilateurs.

L'essai commence par démontrer l'importance des outils d'analyse statique dans un environnement où les coûts de développement et de correction d'erreurs sont très élevés.

Les premières étapes de l'analyse statique ont pour but de convertir le code source du programme en une représentation intermédiaire utilisable par l'analyseur. Cette représentation intermédiaire est composée de trois portions : le code, le graphe de flot de contrôle et les variables du programme. Cette représentation intermédiaire est décrite en détail.

L'algorithme choisi pour l'essai est également décrit en détail. Le choix de cet algorithme a été effectué, car il facilite le traitement de programmes découpés en fonctions en effectuant l'analyse sur les fonctions. Le résultat de l'analyse des fonctions devient une abstraction pour chacune de ces fonctions. Cette abstraction peut ensuite être utilisée pour analyser les programmes qui utilisent ces fonctions.

L'algorithme est composé de trois étapes. La première étape consiste à regrouper les instructions en blocs de base. La seconde étape a pour but de construire une hiérarchie de régions à partir de ces blocs de base. Finalement, l'analyse est effectuée en progressant des régions de base jusqu'à ce que le programme ait été analysé.

Finalement, l'essai indique comment l'analyse peut produire un diagnostic lorsqu'une erreur est détectée. La production du diagnostic nécessite d’inclure le plus d’information possible dans le but d’aider à localiser et corriger les occurrences détectées.

Chapitre 1

Présentation de l'analyse statique

Les humains font continuellement des erreurs : un point-virgule oublié, une parenthèse de trop, un nom de variable mal écrit. La plupart du temps, ces erreurs sont sans conséquence : le compilateur les détecte, le programmeur corrige le code et le cycle de développement se poursuit. Cette rétroaction rapide avec le compilateur est en contraste marquant avec les erreurs qui ne sont pas détectées par le compilateur. Certaines de ces erreurs peuvent demeurer dans le code (possiblement pendant plusieurs années) avant de se manifester. Plus il s'écoule de temps avant de détecter une erreur, plus celle-ci est coûteuse à corriger.

En 1999, une étude effectuée pour le National Institute of Standards and Technology avait comme objectif d'estimer l'impact économique d'une infrastructure inadéquate pour tester la qualité des logiciels produits aux États-Unis.

> « […] the national annual cost estimates of an inadequate infrastructure for software testing are estimated to be $59.5 billion. The potential cost reduction from feasible infrastructure improvements is $22.2 billion. »[1] ([14], p. 11)

Ces coûts représentent les coûts absolus encourus par les entreprises américaines. Ils représentent environ 0,6 et 0,2 pour cent du produit national brut des États-Unis qui est de 10 trillions de dollars. Plus de la moitié de ces coûts sont assumés par les utilisateurs de ces logiciels sous la forme d'activités de mitigation comme conséquence des erreurs rencontrées. Le reste des coûts est imputé aux développeurs des logiciels et représente les

[1] Les coûts dûs à un environnement de test inadéquat sont estimés à 59,5 milliards de dollars par année. Une amélioration à ces environnements pourrait amener une réduction de coûts estimés à environ 22,2 milliards.

ressources additionnelles requises pour vérifier les logiciels dues à une absence d'outils et à des méthodes inadéquates.

1.1 Description de l'analyse statique

L'analyse statique consiste à analyser le texte d'un programme pour en extraire de l'information. Cette analyse est effectuée sans exécuter le programme. L'analyse statique se distingue donc ainsi de l'analyse dynamique qui consiste à exécuter le programme avec des données afin d'en vérifier le comportement. Le type d'information extraite ainsi que les techniques choisies dépendent de l'utilisation à laquelle l'analyse est destinée. L'analyse statique est utilisée pour repérer des erreurs de programmation ou de conception, mais aussi pour déterminer la facilité ou la difficulté à maintenir le code.

Il n'est pas toujours possible de détecter automatiquement tous les défauts de programmation. Les travaux de recherche d'Alan Turing ont démonté, dans la première moitié du XXe siècle, que pour tout problème sur le résultat d'un calcul mécanique, il n'existe de méthode automatisée pour résoudre ce problème que s'il est trivial. Alan Turing [16] a décrit la notion d'algorithme en définissant la machine de Turing. Il a par la suite utilisé cette définition pour démontrer qu'il y a des cas indécidables, c'est-à-dire des problèmes qui n'ont pas d'algorithme. Ce point peut être illustré à l'aide de l'exemple présenté à la figure 1.1.

```
PROBLEME (ch) {
 Si TERMINE (ch) alors faire {…} tant que (vrai);
}
```

Figure 1.1 Programme PROBLEME ().

Traduction libre
Inspiré de : Papadimitriou, C. H. (1995), p. 60

Dans cet exemple, il faut poser comme hypothèse qu'une fonction `TERMINE ()` existe. Cette fonction accepte en entrée une chaîne de caractère représentant un programme et elle

retourne comme résultat un booléen qui a la valeur « vrai » si le programme reçu en entrée se termine et la valeur « faux » s'il ne se termine pas.

Une contradiction apparaît lorsque le programme passé en paramètre au programme `PROBLEME` est le programme `PROBLEME` lui-même. En effet, si la fonction `TERMINE` retourne le booléen « vrai », cela signifie que la fonction a conclu que le programme se termine; le programme `PROBLEME` entrera alors dans une boucle infinie ce qui contredit le résultat de la fonction `TERMINE`. Par contre, si la fonction `TERMINE` retourne le booléen « faux », cela signifie que le programme ne se termine pas, mais dans ce cas nous n'entrons pas dans la boucle infinie et le programme se termine, ce qui contredit également le résultat de la fonction `TERMINE`. Il est donc impossible d'avoir une fonction `TERMINE` qui peut déterminer pour tous les programmes et de façon infaillible si ce programme peut se terminer ou non.

L'analyse de programmes, incluant la recherche d'erreurs possibles à l'exécution, est donc indécidable : il n'existe aucune méthode qui peut toujours répondre sans se tromper qu'un programme puisse ou non produire des erreurs à l'exécution.

Bien qu'il soit impossible de détecter toutes les erreurs présentes dans tous les programmes; ce n'est pas une raison pour abandonner complètement les méthodes d'analyse statique. Il est possible dans la majorité des cas de détecter une grande quantité des problèmes présents. Il faut utiliser des méthodes qui marchent raisonnablement bien sur la majorité des programmes réels. Il s'agit donc de méthodes approximatives. Ces méthodes approximatives ne sont pas un manque de rigueur. Les méthodes utilisées doivent être fiables sans nécessairement être optimales.

Puisqu'il est impossible de détecter toutes les erreurs présentes dans tous les programmes, les outils d'analyse statique vont donc rencontrer des situations où ils ne peuvent pas déterminer précisément si une erreur est présente ou non. Tous les outils risquent donc de

signaler des incidents où il n'y en a pas ou, au contraire, de ne pas détecter des erreurs qui sont présentes dans le code.

Une des plaintes les plus fréquemment entendues au sujet des outils d'analyse statique, selon Brian Chess [3], est que ces outils produisent trop de faux positifs, également appelés fausses alarmes. Dans ce contexte, un faux positif est un incident détecté alors qu'aucun problème n'existe en réalité. Un grand nombre de faux positifs peut être un obstacle à l'utilisation d'outils d'analyse statique. Parcourir une très longue liste de faux positifs peut être très frustrant pour un programmeur, allant même à décourager l'utilisation de ces outils. De plus, les programmeurs peuvent manquer les problèmes réels cachés dans cette liste de faux positifs. Un faux négatif est une faute existante que l'outil d'analyse ne détecte pas.

Tous les outils d'analyse produisent quelques faux positifs ou quelques faux négatifs. Certains produisent les deux. Les outils d'analyse visent un certain équilibre entre les deux selon le but de l'outil. Les coûts engendrés par une erreur de programmation non détectée sont relativement faibles lorsqu'on les compare aux coûts pouvant être subis lors d'une intrusion. Pour cette raison, un outil visant à détecter les erreurs de programmation tente de réduire le nombre de faux positifs et est prêt à accepter certains faux négatifs. Par contre, les outils visant à détecter les problèmes de sécurité vont plutôt accepter de produire plus de faux positifs pour réduire la possibilité d'avoir des faux négatifs.

Il y a deux grandes familles d'analyses statiques formelles de programmes : la vérification de modèle et l'interprétation abstraite.

1.1.1 Vérification de modèle

La vérification de modèle consiste à vérifier algorithmiquement si un modèle donné satisfait à une spécification. Dans ce contexte, le modèle est le système lui-même ou une abstraction du système. La spécification utilisée est souvent formulée en termes de logique temporelle.

Cette famille d'analyse consiste à abstraire un modèle qui reflète le même comportement que le système original, mais duquel certains aspects ont été omis dans le but de simplifier le modèle. Ce type d'analyse est fréquemment utilisé pour la vérification des systèmes concurrents.

Lorsque le système ne peut pas être vérifié de façon exhaustive, la méthode de vérification de modèle requiert la construction d'un modèle simplifié qui conserve les caractéristiques essentielles du système [10]. Le modèle doit être construit manuellement et cette étape est critique, car une erreur lors de la construction du modèle peut invalider les résultats de l'analyse. Une fois le modèle construit, celui-ci peut être vérifié en simulant tous les cas possibles même si cette méthode n'était pas possible à partir du programme lui-même.

1.1.2 Interprétation abstraite

Selon Cousot [4], l'interprétation abstraite peut être définie comme une exécution partielle d'un programme pour obtenir des informations sur sa sémantique, par exemple sur sa structure de contrôle ou sur son flot de données, sans avoir à en faire le traitement complet. Un programme dénote une série de traitements dans un univers donné de valeurs ou d'objets. L'interprétation abstraite consiste à utiliser cette dénotation pour décrire les traitements dans un autre univers d'objets abstraits de façon que les résultats de l'interprétation abstraite puissent fournir de l'information sur les calculs concrets du programme.

L'interprétation abstraite permet de raisonner rigoureusement sur les programmes en permettant de faire des approximations. Étant donné un langage de programmation ou de spécification, l'interprétation abstraite consiste à déterminer des sémantiques liées par des relations d'abstraction.

La sémantique la plus précise est naturellement celle qui décrit l'exécution réelle d'un programme de manière très fidèle. Cette sémantique est appelée sémantique concrète. Par exemple, la sémantique concrète d'un langage de programmation impératif peut associer à

chaque programme l'ensemble complet des traces qu'il produit. Une trace d'exécution étant une séquence des états consécutifs possibles de l'exécution d'un programme. Un état est constitué de la valeur des variables utilisées.

Des sémantiques plus abstraites peuvent être déduites de la sémantique concrète. Par exemple, on pourra considérer l'ensemble des états atteignables lors des exécutions; ce qui revient à considérer uniquement les derniers états des traces.

Pour permettre d'effectuer une analyse statique d'un programme, certaines sémantiques calculables en sont déduites. Par exemple, on peut choisir de représenter les variables d'un programme par leur état. Cet état peut être choisi parmi les états suivants : initialisé ou indéterminé. De même pour les instructions représentées par l'action portée sur les variables : définir, lire, référencer, modifier ou libérer.

Ce niveau d'abstraction ne perd aucune précision pour une analyse qui désire localiser les instructions qui utilisent des variables non initialisées. Pour d'autres opérations, l'abstraction perd de la précision. Il est impossible de déterminer si un accès à une entrée dans une table est à l'intérieur des bornes qui définissent cette table. De telles pertes de précision ne peuvent pas, en général, être évitées lorsque l'on fait des sémantiques décidables. Il y a un compromis à faire entre la précision de l'analyse et sa faisabilité en termes de calculabilité ou de complexité.

1.2 Détermination de problèmes

L'analyse statique est plus utilisée que ce que la plupart des gens croient; principalement parce qu'il y a plusieurs types d'analyse statique. L'annexe 2 décrit les principales utilisations de l'analyse statique.

Le but de cet essai est de démontrer en détail comment l'analyse statique peut être utilisée pour faciliter la détection d'erreurs lors du développement de programmes. L'essai se limite donc à la détection d'erreurs de programmation fréquentes en utilisant l'approche de

l'interprétation abstraite. Les types d'erreur de programmation choisis sont très simples et les portions de code utilisées comme exemple sont très courtes pour permettre une analyse plus détaillée pouvant guider vers une implémentation possible des méthodes présentées.

Les erreurs de programmation qui sont considérées dans l'essai sont de deux types, soit l'utilisation de variables non initialisées et la validation de l'accès aux éléments d'un tableau. Ces deux types d'erreurs sont assez fréquents dans les environnements de programmation procéduraux. Tout code, même s'il est produit par un programmeur expérimenté, peut contenir des anomalies. Certains de ces problèmes sont détectés et corrigés pendant les essais unitaires. D'autres incidents sont détectés et corrigés pendant les tests formels de qualité. Le processus consistant à déterminer le niveau approprié de vérification à effectuer est subjectif. Même une quantité infinie de tests ne peut pas prouver une négative : l'absence d'erreurs dans le programme. Cependant, plus une personne vérifie un programme dans le but de trouver des problèmes, plus il y a de chances d'en trouver un. Le nombre de situations à valider pour un programme complexe peut être virtuellement infini.

La principale raison pour livrer des programmes avec des problèmes est qu'il n'est jamais économique de détecter et corriger tous les problèmes, même s'il était possible de le faire. Selon Burnstein [2], la méthode la plus répandue pour vérifier la qualité d'un système consiste à vérifier celui-ci à la fin du cycle de développement. Effectuer des essais requiert des ressources et bien que cela améliore la qualité du code, puisque le nombre de situations possibles approche l'infini, le temps et les ressources pour effectuer ces validations deviennent également infinis. Vérifier la qualité à l'aide de cas d'essai est très inefficace. Puisque la plupart du temps le nombre de cas à valider est beaucoup trop grand, les tests effectués ne couvrent qu'une fraction des situations qui peuvent être rencontrées ce qui laisse potentiellement des erreurs qui sont rencontrées après la livraison du produit.

La prévention des défauts n'est pas seulement possible, mais nécessaire dans le développement de logiciels. Toutefois, pour que la prévention des défauts soit efficace, il

faut avoir un processus formalisé pour intégrer cette stratégie dans le cycle de développement. Cette approche formalisée doit inclure l'application des meilleures pratiques de l'industrie pour éviter les problèmes communs. De plus, cette approche doit être supportée par une infrastructure adaptable qui automatise plusieurs tâches répétitives afin de propager la pratique de prévention des défauts d'un projet à l'autre. Les essais ne doivent pas être éliminés. Le processus de validation à la fin du cycle de développement n'est pas une méthode efficace pour améliorer la qualité du produit. Par contre, les tests peuvent et doivent être utilisés pour aider à mesurer et vérifier la qualité du produit.

L'utilisation d'outils d'analyse visant à détecter un maximum d'anomalies de programmation de façon automatique libère les ressources humaines qui peuvent se consacrer à détecter les autres types d'erreur et améliorer ainsi la qualité des programmes livrés. Ces outils s'insèrent donc au cœur du processus de développement et permettent de réduire le nombre de défauts plus tôt dans le processus. Ces outils permettent d'améliorer la qualité du produit à un très faible coût. L'implantation d'un outil d'analyse n'est pas très couteuse : un outil permettant de détecter les anomalies décrites dans l'essai a été écrit en 2 mois-personne. Une fois que l'utilisation d'un tel produit est intégrée au processus de développement, les coûts récurrents sont très faibles.

L'analyse statique peut s'appliquer aux différents types de langage : procéduraux, orienté objet, fonctionnel ou logique. L'essai ne couvre que les langages procéduraux en ciblant principalement le COBOL dans les exemples bien que les techniques démontrées ne se limitent pas à ce seul langage.

Certains compilateurs modernes permettent la détection de certaines erreurs de programmation que d'autres compilateurs ne détectent pas. Certains compilateurs peuvent générer du code permettant de détecter les débordements de tables lors de l'exécution, mais il s'agit là d'une option qui est souvent omise pour favoriser la rapidité de l'exécution. Sur le marché, il y a encore énormément de code développé en COBOL. Les compilateurs

COBOL, bien que robustes, ne disposent pas de toutes les facilités offertes par les compilateurs des langages plus récents.

> « There are 180-200 billion lines of COBOL code in use worldwide. Most transactions in this world are Cobol-based. Billions of lines of business-critical Cobol code are written every year. »[2] ([12], p. 99)

Il y a donc avantage à ne pas négliger les techniques d'analyse statique de base qui peuvent par la suite être étendues à la détection d'autres erreurs de programmation. Le choix du langage COBOL est basé sur l'expérience de l'auteur avec ce langage, sur le volume de programmes COBOL en existence ainsi que le fait que le langage COBOL est conçu spécifiquement pour les applications financières et qu'il semble qu'aucun autre langage ne soit aussi bien adapté aux besoins de ces applications.

Robert L. Glass [9] mentionne quatre caractéristiques jugées essentielles aux applications financières : gérer des structures de données hétérogènes, effectuer de l'arithmétique décimale précise, générer facilement des rapports ainsi que traiter facilement une très grande quantité de données. La majorité de ces caractéristiques ne sont pas présentes dans les langages de programmation moderne, à l'exception du langage COBOL qui est adéquat dans les quatre caractéristiques.

[2] De 180 à 200 milliards de lignes de code écrit en COBOL sont actuellement utilisées dans le mode. La majorité des transactions sont en partie basées sur du code COBOL. Chaque année, plusieurs milliards de lignes de code critique aux entreprises sont écrites en COBOL.

Chapitre 2

Interprétation abstraite

L’interprétation abstraite est basée sur la définition d'une abstraction de la sémantique d'un programme écrit dans un langage donné. Cette abstraction du programme utilise une vue moins complexe du code que le programme source original, ce qui en simplifie l'analyse. L’utilisation proposée d’une abstraction sacrifie de l'information lors du processus d'abstraction. Ainsi, une solution à un problème basé sur l'interprétation abstraite peut ne pas être une solution au problème original. L'art de l'analyse symbolique des programmes consiste donc à développer une abstraction convenable pour un programme donné qui permet de résoudre ce type d’anomalie.

L'objectif de l'analyse statique est de déterminer automatiquement certaines propriétés d'intérêt d'un système. Le code source est une description de la spécification d’une situation que l'application est destinée à résoudre. Les instructions du programme forment un modèle concret représentant cette spécification. En analysant le code source tel quel, il est possible d'analyser toutes les propriétés observables sur celui-ci. Ces propriétés peuvent être aussi diverses que de déterminer le temps passé à traiter les itérations, s'assurer que les accès à une variable ne se font que lorsque cette variable est initialisée, valider le respect des bornes d'un tableau lors des accès à un élément de celui-ci.

Il existe de nombreuses propriétés observables selon les besoins de l'analyse. Habituellement, les propriétés d'intérêt pour un programme donné consistent en un sous-ensemble de toutes les propriétés observables. Il est donc utile, dans ce cas, de simplifier la représentation du comportement du programme pour faciliter le traitement de l'analyse. Par exemple, si le but de l'analyse est d'observer le comportement d'un programme sur les variables numériques pour s'assurer qu'aucune division par zéro ne peut se produire, il est

inutile de considérer les instructions qui n'ont pas d'effet sur les variables numériques. Il est donc possible d'ignorer les instructions effectuées sur les chaînes de caractères dans le cadre de cette analyse.

En fonction des propriétés à observer, la première étape de l'interprétation abstraite consiste à choisir une abstraction qui servira à simplifier le programme à analyser. En entrée, le programme possède des instructions qui ont pour but d'effectuer le traitement pour lequel il a été conçu. Ce programme est alors transformé puis l'analyse est effectuée sur une version simplifiée du programme. Cette version simplifiée représente une abstraction du programme original. Le choix de l'abstraction doit être tel que les propriétés observées sur cette version simplifiée sont identiques à celles qui auraient été obtenues si l'analyse détaillée avait été effectuée sur le programme original.

Dans ce chapitre, l'interprétation abstraite est d'abord introduite informellement à l'aide d'un exemple. Par la suite, une description plus rigoureuse de l'interprétation abstraite est formulée. Finalement, un modèle d'abstraction est choisi pour permettre de résoudre les deux types d'incidents visés par l'essai : s'assurer qu'une variable est initialisée avant qu'elle ne soit référencée et valider les bornes des tableaux lors d'un accès à un des éléments du tableau.

2.1 Définition informelle

L'interprétation abstraite est introduite à l'aide d'un exemple très simple et intuitif. Cet exemple est issu d'un article de Cousot [6], lui-même basé sur un article de Michel Sintzoff [15]. Dans l'article de Sintzoff, l'auteur utilise l'arithmétique des signes comme modèle d'abstraction. Dans l'exemple présenté ici, la propriété à observer est le signe du résultat.

Le texte « `-1515 * 17` » peut dénoter les calculs dans un univers abstrait `{(+), (-), (±)}` où la sémantique des opérateurs arithmétiques est définie par la règle des signes.

L'exécution abstraite de `-1515 * 17` ⇒ `-(+) * (+)` ⇒ `(-) * (+)` ⇒ `(-)` prouve que « `-1515 * 17` » est un nombre négatif. L'interprétation abstraite s'intéresse à une propriété spécifique de l'univers des calculs (le signe du résultat dans l'exemple présenté). L'interprétation abstraite donne un sommaire d'une facette de l'exécution d'un programme. En général, ce sommaire est facile à obtenir, mais imprécis. Ainsi, `-1515 + 17` ⇒ `-(+) + (+)` ⇒ `(-) + (+)` ⇒ `(±)`, ce qui signifie qu'il est impossible de déterminer le signe du résultat uniquement à partir du signe des opérandes.

L'exemple précédent a démontré que l'application d'une abstraction permet de simplifier l'analyse d'un programme. La section suivante présente une définition plus formelle de l'interprétation abstraite des programmes.

2.2 Définition formelle

L'analyse statique des programmes consiste en une évaluation abstraite de ces programmes. Lors de l'évaluation abstraite d'un programme, des valeurs abstraites sont associées aux variables à la place des valeurs concrètes qui seraient obtenues lors de l'exécution du programme. Les opérations de base du langage sont interprétées en fonction de l'abstraction choisie. L'interprétation abstraite consiste alors à observer les opérations du programme sur ces valeurs abstraites. Il est possible d'utiliser n'importe quel ensemble de valeurs abstraites, pourvu que ces valeurs respectent la propriété suivante : l'ensemble de valeurs abstraites choisies doit permettre à l'analyse d'atteindre un état stable après un nombre fini d'itérations.

L'interprétation des instructions de base du langage ainsi que le choix de l'abstraction des valeurs dépendent des propriétés dynamiques à extraire du programme. Dans la mesure où le choix de l'abstraction respecte certaines propriétés, la méthode d'analyse présentée au chapitre 4 permet de garantir que l'analyse se terminera en un nombre fini d'itérations et que le résultat de l'analyse est correct, bien qu'il puisse contenir un certain degré d'imprécision.

L'ensemble des valeurs abstraites doit former un ensemble ordonné de valeurs appelé un treillis. L'annexe 3 présente certaines propriétés de la théorie des ensembles utiles à la discussion sur l'abstraction.

L'abstraction d'un programme est présentée dans le cadre de la validation des accès aux éléments d'un tableau. Une variable servant à indexer un tableau peut prendre une ou plusieurs valeurs entières. L'ensemble des valeurs que peut prendre cette variable sera abstrait en un intervalle de valeurs. La limite inférieure de l'intervalle représente la plus petite valeur possible alors que la borne supérieure de l'intervalle représente la plus grande valeur.

La théorie utilisée pour l'interprétation abstraite est basée en grande partie sur les travaux de Patrick Cousot [5].

2.3 Valider les accès à un tableau

Dans cette section, l'utilisation d'un intervalle de valeurs pour valider les accès aux éléments d'un tableau est tirée de l'article de Patrick Cousot [5] qui est présenté à l'annexe 4.

Le choix d'une abstraction appropriée est effectué en tenant compte du but de l'analyse. Afin de détecter les accès à un tableau en dehors des bornes de celui-ci, il suffit de déterminer l'intervalle des valeurs possibles des variables servant à indexer les entrées de ce tableau. Pour ce type de situations, il est possible d'utiliser des intervalles de valeurs comme domaine de valeurs abstraites pour les variables numériques.

L'ensemble V_C des valeurs concrètes est donc l'ensemble des entiers $\mathbb{Z}$ (entre les limites $-\infty$ et $+\infty$) et V_A des valeurs abstraites est l'ensemble des intervalles de nombres entiers dénotés [a, b] où $a \in \mathbb{Z}$, $b \in \mathbb{Z}$ et $a \leq b$.

La fonction d'abstraction α est définie comme suit :

$$\alpha: \mathcal{P}(V_C) \longrightarrow V_A, \ \alpha(e) = [\min(e), \max(e)]$$

Lorsque la valeur concrète consiste en une seule valeur, par exemple 3, alors l'intervalle obtenu est [3, 3]. Pour un ensemble de plusieurs valeurs, par exemple {5, -6, 7}, l'intervalle est [-6, 7]. Dans ce dernier cas, l'abstraction amène une perte de précision, car ce qui était un ensemble de 3 valeurs devient un intervalle de 14 valeurs. Cette perte de précision ne nuit pas à l'analyse, car les valeurs limites de l'intervalle sont toujours des valeurs provenant de l'ensemble des valeurs concrètes.

Pour la situation consistant à valider l'accès aux éléments d'un tableau, si les accès sont valides pour les valeurs limites de l'intervalle, les accès sont également valides pour tous les autres éléments de l'intervalle. Par exemple soit l'ensemble de valeurs concrètes {3, 7, 18, 34}, si ces valeurs sont utilisées pour accéder à un tableau de 50 éléments, alors tous les accès sont valides. L'intervalle correspondant à ces valeurs est [3, 34]. Si les accès sont valides pour la valeur 3 et pour la valeur 34, alors les accès sont valides pour tous les éléments de l'intervalle. Puisque l'intervalle contient par définition tous les éléments de l'ensemble des valeurs concrètes, alors si les accès sont valides pour l'intervalle de valeurs abstraites ils sont valides également pour les valeurs concrètes.

La fonction de concrétisation γ est définie comme suit :

$$\gamma : V_A \longrightarrow V_C, \gamma([a, b]) = \{x \mid (x \in \mathbb{Z}) \wedge (a \leq x \leq b)\}$$

La fonction α permet d'obtenir un intervalle de valeurs à partir d'un ensemble d'entiers. La fonction γ retourne l'ensemble des entiers qui font partie de l'intervalle. Avec la valeur abstraite [3, 34], la fonction γ retourne l'ensemble des éléments qui font partie de l'intervalle, soit l'ensemble {3, 4, 5, …, 33, 34}.

Il apparaît que l'ensemble des valeurs concrètes contient plus de valeurs que l'ensemble original. Une imprécision apparaît à ce point puisque l'ensemble des valeurs concrètes qui

proviennent de $\gamma(\alpha(e))$ est différent de e. Il est important de s'assurer que cette imprécision n'a pas d'effet sur la validité de l'analyse. Les valeurs limites de l'intervalle proviennent toujours de l'ensemble des valeurs concrètes par la définition de α ci-dessus. De plus, les seules valeurs qu'il est important de conserver pour pouvoir valider les accès aux éléments d'un tableau sont les limites de l'intervalle. L'analyse est donc valide malgré l'imprécision introduite par la fonction γ.

La figure 2.1 illustre une partie d'un treillis. Ce treillis représente l'abstraction basée sur les intervalles de valeurs. Ce treillis contient les intervalles en ordre croissant du bas de la figure vers le haut. Plus l'intervalle est haut sur la figure, plus il est élevé selon l'ordre $\subseteq$. L'opération $\cup$ est illustrée en trouvant l'intersection des lignes provenant des deux intervalles, par exemple [2,3] $\cup$ [4,6] est l'intervalle [2,6] qui se retrouve sur l'intersection de la ligne liant l'intervalle [2,3] et l'intervalle [5,6]. Il s'agit du premier intervalle à inclure les deux intervalles.

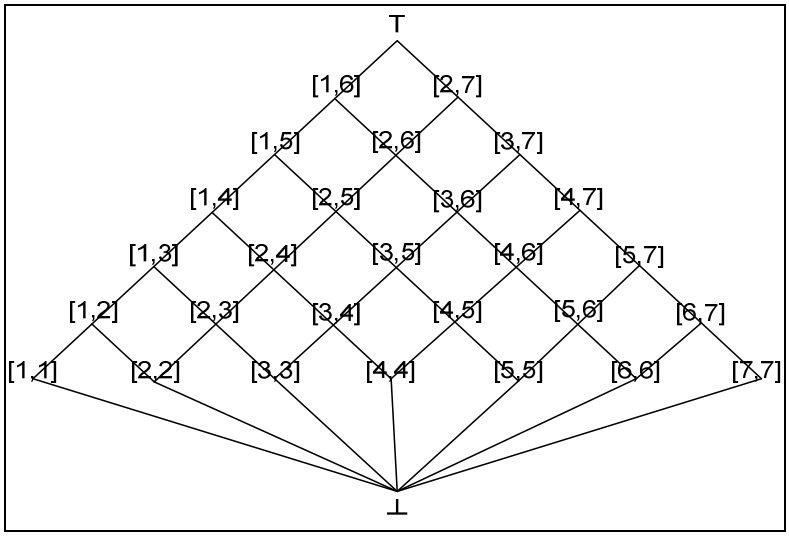


Figure 2.1 Treillis partiel pour des intervalles de valeurs entières

La ligne du bas de la figure 2.1 contient l'élément « $\perp$ » indiquant une variable sans valeur. La seconde ligne représente les valeurs concrètes que sont les nombres entiers. Ces valeurs sont représentées sous forme d'intervalle ne contenant qu'une seule valeur. Chaque ligne

représente des intervalles de plus en plus grands jusqu'à la ligne du haut qui représente l'intervalle infini [-∞, +∞].

2.4 Accès à une variable

Dans la section précédente, une abstraction a été choisie pour représenter les variables numériques servant à accéder aux éléments d'un tableau. L'essai a également pour but de détecter les occurrences d'utilisation d'une variable avant que celle-ci ne soit initialisée.

Pour ce second problème, il faut déterminer si une variable a une valeur ou non. Pour les variables numériques, nous avons déjà la valeur « ⊥ » qui indique que la variable n'a pas de valeur et un intervalle de valeur pour représenter les autres situations. Pour les autres variables du programme, le domaine de valeurs abstraites peut consister en deux valeurs : « ⊥ » indiquant qu'une variable n'a pas de valeur et « d » qui indique que la variable est définie.

En considérant que le langage COBOL est principalement utilisé pour le traitement d'information provenant de fichiers, il est possible d'ajouter des valeurs au domaine de valeurs abstraites {⊥, d} pour pouvoir procéder à une analyse plus complète. Certaines valeurs proposées par Kao et Chen [11] peuvent être utilisées à ces fins. Le domaine utilisé pour les variables peut donc devenir {⊥, d, f, o, c, ⊤}. La valeur « d » indique qu'une variable est définie, « f » indique qu'une valeur a été obtenue d'un fichier, « o » indique que la valeur de la variable provient d'un fichier et que ce fichier est ouvert, mais aucun enregistrement n'a été lu donc la variable n'est pas encore définie et finalement « c » indique que le fichier est fermé et la variable correspondante n'est plus définie. Le domaine peut être étendu pour refléter toutes les situations à couvrir selon les propriétés à observer.

Dans l'essai, il est suffisant de déterminer si une variable est initialisée au moment de l'utiliser. Le domaine d'abstraction {⊥, d, ⊤} est utilisé pour valider qu'une variable soit initialisée avant son utilisation. La valeur « ⊤ » indique qu'en un point de contrôle donné, il est impossible de savoir si la variable est initialisée ou non.

Chapitre 3

Représentation interne du programme

Les outils d'analyse travaillent habituellement à partir du code source des programmes. Ces programmes sont transformés pour en extraire les structures syntaxiques et en faciliter le traitement. Ce chapitre décrit la représentation utilisée dans cet essai sans toutefois spécifier les détails d'implémentation.

Un programme est composé d'instructions et des données sur lesquelles agissent ces instructions. Les instructions sont de deux types : les instructions de transformation des données et les instructions qui gouvernent la séquence d'exécution du programme. La première partie du chapitre traite des instructions de transformations des données. Les instructions servant à déterminer la séquence d'exécution du traitement sont présentées par la suite. La représentation des données est adressée à la fin du chapitre.

3.1 Transformation du programme

L'analyse statique est effectuée sur un programme spécifique fourni selon un format spécifique. La majorité des outils d'analyse de code traitent les programmes comme le font les compilateurs, c'est-à-dire à partir du code source. La figure 3.1 illustre les phases d'un compilateur alors que la figure 3.2 illustre les phases d'un outil d'analyse statique. Ces deux figures démontrent qu'un outil d'analyse statique effectue en grande partie le même travail de traitement du code source qu'un compilateur.

Les quatre premières phases du traitement ont pour but de transformer le programme du code source vers une représentation interne. La représentation du programme en entrée est sous un format qui est facile à manipuler par un être humain. La transformation du

programme source a pour but de produire une représentation qui peut être plus facilement manipulée par une application.

La transformation du code source en code intermédiaire par un outil d'analyse statique est identique à ce que fait un compilateur. Lorsque le code intermédiaire est obtenu, le traitement de l'outil d'analyse n'a plus rien en commun avec un compilateur. Le compilateur doit produire des instructions en un langage machine déterminé. Ces instructions doivent donner une représentation fidèle de la sémantique concrète du programme telle qu'exprimée par le code source.

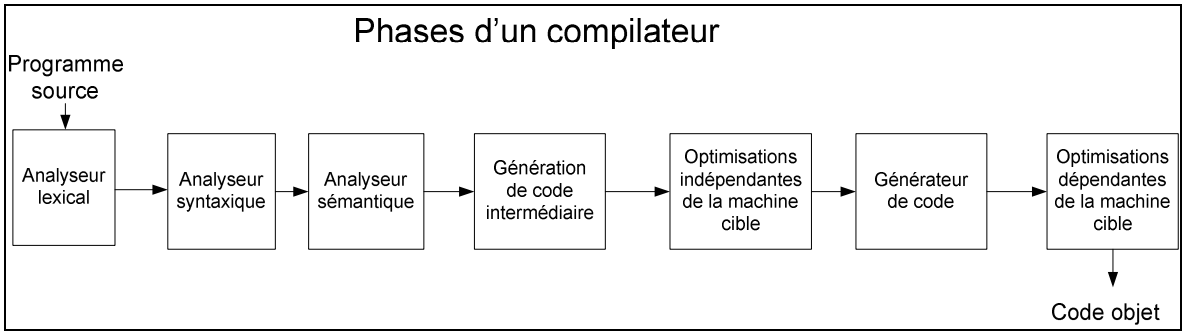


Figure 3.1 Phases de la compilation.

Traduction libre

Inspiré de : Aho, A.V., Lam, M.S., Sethi, R. et Ullman, J.D. (2007), p.5

L'outil d'analyse statique ne préserve pas la sémantique complète du programme; il conserve assez d'information afin de produire le diagnostic des incidents potentiels détectés. La représentation intermédiaire doit être en mesure de manipuler des valeurs abstraites. Le compilateur produit du code qui manipule des valeurs concrètes.

Dans l'essai, les quatre premières phases des figures 3.1 et 3.2 ne sont pas adressées. Ces phases sont très bien couvertes par la littérature sur les compilateurs, dont le livre d'Aho, Lam, Sethi et Ullman [1]. La quatrième phase, bien qu'elle ait le même nom pour un compilateur que pour un outil d'analyse statique, peut produire un résultat différent selon

l'outil. Le résultat est la représentation interne du programme et cette représentation fait l'objet de ce chapitre.

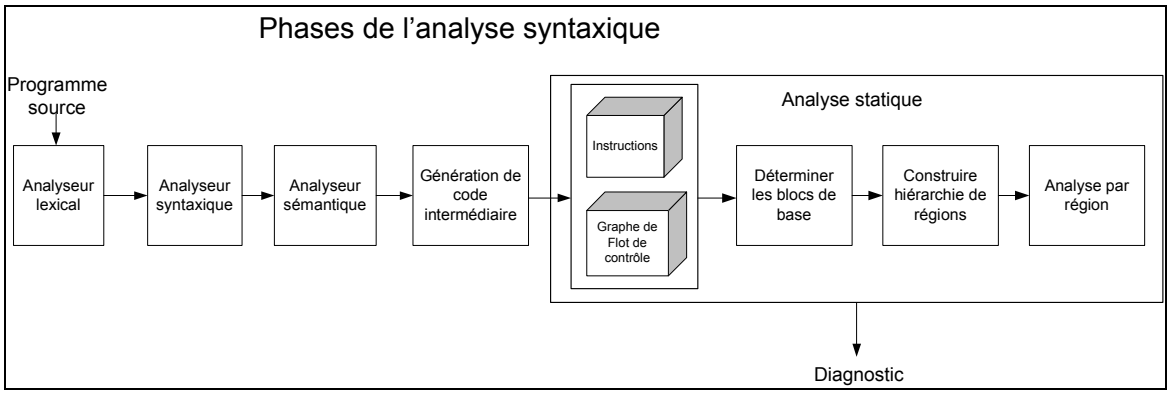


Figure 3.2 Phases de l'analyse statique

L'étape de génération de code intermédiaire produit une représentation du programme. Cette représentation est plus facile à utiliser lors de l'analyse statique que ne le serait le code objet. Si la représentation intermédiaire du programme est indépendante du langage source original et qu'elle est suffisamment générique alors, cette représentation intermédiaire permet la réutilisation du module d'analyse statique pour d'autres langages. Puisque les langages procéduraux et les langages orientés objet, une fois compilés, peuvent être exécutés sur le même processeur, il est possible d'obtenir une représentation intermédiaire qui tient compte de cette particularité. Le processeur ne connaît pas la notion d'objets ou de structure de données. Le processeur a des instructions de manipulation de données, des instructions pour gérer le flot d'exécution du programme et un certain nombre de données élémentaires qu'il connaît. Pour y parvenir, il faut toutefois tenir compte des différences de représentation des objets et d'appel des méthodes, par exemple les méthodes virtuelles qui rendent difficile la détermination des méthodes réellement utilisées lors de l'appel.

Puisque les compilateurs et les outils d'analyse statique ont beaucoup de traitements en commun; il est naturel de se baser sur la théorie des compilateurs pour décider d'une représentation interne des programmes facile à manipuler. Le langage intermédiaire

proposé dans cet essai reprend plusieurs idées tirées du livre d'Aho, Lam, Sethi et Ullman [1], en particulier l'utilisation d'instructions internes à longueur fixe, le regroupement des instructions en blocs ainsi que la représentation du flot de contrôle à l'aide d'un graphe.

3.2 Instructions de manipulation des données

Un programme qui s'exécute accède à des données; il conserve ces données sous forme de valeurs assignées à des variables; il transforme le contenu de ces variables et, finalement, il produit le résultat de ces transformations. Le but de l'analyse statique est d'observer certaines propriétés lors de cette manipulation des données.

Le programme original à analyser est écrit dans le but de manipuler des valeurs concrètes. L'abstraction choisie pour l'analyse a pour but de faciliter l'analyse des propriétés recherchées. Les instructions originales doivent donc être remplacées par des instructions pouvant agir sur l'abstraction choisie plutôt que sur les valeurs concrètes du programme original.

Bien que les instructions soient modifiées lors de la transformation du code source, les instructions résultantes continuent à utiliser des valeurs pour produire de nouvelles valeurs. Par contre, ces valeurs sont des valeurs abstraites plutôt que des valeurs concrètes.

3.2.1 Variables

Un programme reconnaît un certain nombre de variables et ces variables ont un type qui dépend du langage ou du processeur. Une liste de ces variables incluant leur type et les attributs de ces variables est produite lors de l'analyse du programme et utilisée lors de l'analyse.

3.2.2 États d'un programme

Un état de la mémoire, également appelé un état, est la relation entre des variables et des valeurs. Par exemple, un état donné peut associer la valeur 1 à la variable « `I` » après une instruction d'initialisation. Les états sont modifiés par les instructions du programme. Pour l'analyse statique, un état doit représenter l'ensemble des variables d'un programme avec leur valeur abstraite. Par exemple, un programme avec deux variables « `x` » et « `y` » peut avoir un état `{(x, [1, 2]), (y, [-10, 20])}`. Cette notation dénote un ensemble constitué de deux couples. Chacun des couples `(x, [1, 2])` et `(y, [-10, 20])` représente la relation entre la variable et sa valeur abstraite. Pour un langage procédural, le type de la variable n'est pas requis dans l'état, car ce type ne change pas lors de l'exécution du programme et il serait alors redondant de répéter ce type dans tous les états où la variable se retrouve. Le type d'une variable est disponible à partir de la liste des variables produites lors de la transformation du programme source. Pour un langage orienté objet, le type de l'objet doit faire partie de l'état car le type de l'objet peut changer au cours de l'exécution.

Une expression provenant d'une instruction est essentiellement une fonction permettant d'obtenir une valeur à partir des valeurs de variables d'un état donné. Par exemple, nous pouvons résoudre l'expression « `x + y` » en utilisant la valeur des variables « `x` » et « `y` » de l'état précédant l'instruction qui contient cette expression. Le résultat de l'expression est une nouvelle valeur. Dans le but de simplifier l'essai, il est présumé que l'évaluation d'une expression n'a pas d'effet secondaire sur les variables qui composent cette expression. C'est le cas du langage COBOL, mais pas du langage C où une expression comme « `x = i++;` » modifie non seulement la variable « `x` », mais également la variable « `i` ».

3.2.3 Point de contrôle

Un point de contrôle désigne un emplacement dans le code du programme. Les points de contrôle sont situés entre chaque instruction, ainsi qu'au début et à la fin du programme.

Une instruction peut être vue comme une fonction modifiant un état. L'état modifié est l'état au point de contrôle précédant l'instruction. L'instruction crée un nouvel état au point de contrôle suivant cette instruction. Un point de contrôle est donc un endroit stable où les propriétés des variables peuvent être observées. Une instruction peut être vue comme une fonction qui reçoit un état en entrée pour produire un nouvel état.

3.2.4 Contexte

Dans un système de gestion de base de données, un uplet occupe une ligne qui croise des colonnes représentant chacune un attribut. Dans cet essai, le terme uplet est utilisé pour représenter un ensemble d'attributs décrivant un objet. Un contexte est un uplet composé d'un point de contrôle et d'un état. Un contexte ne révèle rien sur les calculs passés ni sur la façon dont le flot de contrôle est arrivé au point de contrôle. Par contre, le contexte détermine exactement le traitement futur, c'est-à-dire la suite de l'exécution. Si l'exécution du programme débute à partir de ce point de contrôle et avec un état de la mémoire, ce qui se passera dans le programme à partir de ce point de contrôle est complètement déterminé.

Le but de l'analyse est d'obtenir tous les contextes du programme, c'est-à-dire de déterminer pour chaque point de contrôle l'ensemble des valeurs que peuvent prendre les variables du programme.

3.2.5 Manipulation des valeurs abstraites

Lors de l'analyse statique, il n'est pas toujours possible de déterminer précisément la valeur d'une variable et il faut alors utiliser une approximation de cette valeur. Par exemple, dans le programme de la figure 3.3, il n'est pas possible de déterminer la valeur exacte de la variable « `I` » après l'exécution de l'instruction « `IF` », car il est impossible de connaître la valeur de la variable « `B` » obtenue lors de la lecture du fichier.

```
READ FILE-A INTO B.
IF B > 10
    MOVE 1 TO I
ELSE
    MOVE 3 TO I
END-IF.
ADD +1 TO A(I).
```

Figure 3.3 Exemple de valeur impossible à déterminer

Bien qu'il soit impossible de déterminer avec précision la valeur de la variable « `I` » après l'instruction `IF`, il est possible d'établir que cette valeur est 1 ou 3. Il est également possible de considérer que la valeur se situe entre 1 et 3 inclusivement. La valeur de la variable « `I` » peut donc être représentée par un intervalle de valeur. Après l'exécution de l'instruction `IF`, cet intervalle est [1, 3]. Cet intervalle introduit une valeur 2 qui ne fait pas partie des états possibles de la variable « `I` » à ce point de contrôle. Cette imprécision n'a cependant pas d'importance pour permettre d'évaluer l'accès aux éléments d'un tableau. En effet, si l'accès à l'élément « `I` » du tableau « `A` » est valide pour les valeurs limites de l'intervalle, soient les valeurs 1 et 3; alors, l'accès est valide pour la valeur 2. La valeur 2 peut donc être ignorée.

En tenant compte des abstractions mentionnées, les instructions du langage original peuvent être transformées en action sur l'état de l'abstraction choisie. Le but de la transformation du programme est donc de modifier les instructions du langage original en instructions sur les valeurs abstraites. Par exemple, l'instruction COBOL « `MOVE 1 TO I` » qui a pour effet d'initialiser la variable « `I` » à la valeur 1; peut être traduite par l'initialisation de l'intervalle représentant la variable « `I` » à la valeur [1, 1]. De même, une expression arithmétique qui incrémente une variable de 1 aura comme effet de modifier l'intervalle des valeurs de la variable correspondante.

Le choix effectué pour représenter les variables numériques consiste en un intervalle de valeurs. La valeur concrète de la variable peut être modifiée par les instructions du programme. L'intervalle de valeur qui est utilisé pour représenter l'état de la variable doit

alors être modifié pour refléter l'expression évaluée. Par exemple, à partir de l'état (`I`, [2, 7]), l'instruction « `COMPUTE I = I * 2 + 1` » doit produire l'état (`I`, [5, 15]) dans le contexte du point de contrôle suivant l'instruction.

Une variable numérique non initialisée est représentée par la valeur « ⊥ ». Une variable dont la valeur est obtenue d'une source extérieure au programme est initialisée selon l'intervalle maximum que peut prendre cette variable. Par exemple, si une variable « `I` » est définie « `PICTURE S99` », alors l'intervalle le plus grand que peut prendre cette variable est [-99, 99]. Si la variable est définie non signée, par exemple « `PICTURE 99` », alors l'intervalle maximum est [0, 99].

Les instructions intermédiaires utilisées dans la représentation interne du programme sont conçues de manière à refléter l'effet des instructions originales du programme sur les valeurs abstraites des variables. Le langage intermédiaire a donc des instructions permettant de modifier des intervalles de valeur. Des instructions existent également pour valider qu'à un point donné du programme une variable soit initialisée et que sa valeur fasse partie d'un intervalle donné. Pour en faciliter le traitement, toutes les instructions internes ont la même taille. Ces instructions sont composées de trois champs : le code d'opération qui définit l'opération à effectuer, un premier opérande qui indique la variable qui fait l'objet de l'opération et un second opérande dont la signification dépend de l'opération. Les instructions du langage intermédiaire sont énumérées dans le tableau 3.1.

L'instruction `INTERV` sert à détecter les situations visées par l'analyse statique. Au moment où cette instruction est rencontrée, si le contexte contient un intervalle de valeurs qui ne sont pas incluses dans les bornes du tableau spécifié par l'instruction, un diagnostic d'incident est alors produit.

L'instruction `ISDEF` permet de déterminer, au moment où cette instruction est rencontrée, si la variable correspondante est initialisée ou non.

Les instructions de TEST sont le résultat de la transformation des tests du programme original. Ces instructions permettent d'orienter le flot de contrôle selon la valeur d'une variable. Les autres instructions ont pour but de modifier les valeurs abstraites correspondant aux variables en fonction de la sémantique des instructions originales du programme. Il faut noter que les instructions modifiant le flot de contrôle, les instructions de test et les instructions de contrôle pour les boucles ne sont pas représentées par des instructions intermédiaires. Ces instructions dirigent le flot de contrôle lors de l'exécution et font l'objet de la section suivante.

Tableau 3.1 Liste des instructions intermédiaires

Opération	**1er opérande**	**2e opérande**	**Description**
INTERV	tableau	variable	Valide que l'intervalle de valeurs de la variable soit contenu au complet dans les bornes du tableau.
INIT	Variable numérique	Valeur ou variable	Remplace l'intervalle par un nouvel intervalle construit à partir de la valeur ou l'intervalle courant de la variable spécifiée.
*	Variable numérique	Valeur ou variable	Modifie l'intervalle courant de la variable.
+	Variable numérique	Valeur ou variable	Modifie l'intervalle courant de la variable.
-	Variable numérique	Valeur ou variable	Modifie l'intervalle courant de la variable.
/	Variable numérique	Valeur ou variable	Modifie l'intervalle courant de la variable.
ISDEF	Variable	Rien.	Valide que la variable possède une valeur à ce point.
TEST= TEST> TEST<	Variable	Valeur ou variable	Permets d'effectuer un test élémentaire.

3.3 Flot de contrôle

Les instructions de manipulation des données sont essentielles dans l'analyse des programmes. L'ordre dans lequel ces instructions sont exécutées est également crucial pour pouvoir observer le comportement du programme. La transformation du programme source doit donc extraire et conserver l'ordre d'exécution des instructions du programme original.

Une des principales difficultés de l'analyse statique réside dans l'analyse des instructions répétitives. La modélisation des instructions d'itération implique l'analyse des variables et zones mémoires qui sont modifiées à l'intérieur de ces boucles. Ces variables sont appelées variables de récurrences. Par exemple, dans une itération simple de la forme « `PERFORM VARYING I FROM 1 BY 1 UNTIL I > N` », la difficulté provient de la variable de récurrence « `I` » dont la valeur change pour prendre successivement les valeurs de l'intervalle [1, `N`].

Ce changement peut être représenté sous forme de récurrences. Dans le cas de l'exemple précédent, la relation de récurrence est donnée par $I_{k+1} = I_k + 1$ où I_{k+1}, $k \geq 0$ est la valeur de la variable « `I` » à la fin de l'itération k+1. Avant l'entrée dans la première itération, la variable « `I` » est initialisée à la valeur 1 ce qui permet d'initialiser la récurrence par $I_0 = 1$. À partir de la relation de récurrence, il est possible de déterminer la valeur de la variable « `I` » à n'importe quelle itération de la boucle.

Il existe plusieurs méthodes pour déterminer les relations de récurrences. Une des approches est basée sur les fonctions génératrices. Les fonctions génératrices sont couvertes en détail dans le livre de Wilf [17].

Pour les besoins de l'essai, il n'est pas nécessaire d'obtenir la relation de récurrence des boucles. Il est suffisant d'identifier la plus petite et la plus grande valeur que peuvent prendre les variables de récurrence. La figure 3.4 illustre la forme typique d'une instruction d'itération dans un programme COBOL.

```
PERFORM VARYING I
        FROM    1 BY 1
        UNTIL   I > MAX
        OR      . . .
. . .
END-PERFORM.
```

Figure 3.4 Format d'une boucle en COBOL

Lorsque ce format de l'instruction PERFORM est rencontré, il est possible de déterminer l'intervalle de valeurs que peut prendre la variable « I » dans cette boucle. Cet intervalle est [1, MAX]. Si la portion de l'instruction « VARYING … FROM … BY … » est omise, il est impossible de déterminer la valeur de l'intervalle avant d'entreprendre l'analyse statique. De même, si la portion de test de l'instruction PERFORM ne contient pas de prédicats sur la variable « I », il est impossible de déterminer l'intervalle complet de valeurs que peut prendre la variable « I » avant d'effectuer l'analyse statique. Dans ce cas, l'analyse statique doit présumer que la variable « I » n'a pas de limites supérieures sauf celles imposées par la représentation interne de la variable.

Lors de la transformation du programme source, il faut donc extraire de l'information sur les instructions itératives rencontrées. L'information requise est de deux types : les variables modifiées lors des différentes itérations et la valeur maximum que peuvent prendre ces variables.

Dans la figure 3.4, la variable « I » est une variable modifiée lors des itérations, il est possible qu'il y ait d'autres variables dans le corps de la boucle. La valeur maximum que peut prendre la variable « I » est « MAX », car lorsque la variable a une valeur supérieure à « MAX », la boucle se termine. S'il y a d'autres conditions dans le prédicat de l'instruction itérative, il est possible qu'il y ait d'autres valeurs maximums à extraire de la boucle.

Si les propositions qui forment le prédicat de la boucle ne sont pas liées par des opérateurs logiques « ou », il est difficile de déterminer des valeurs maximums pour les variables. Dans ce cas, il est plus facile d'ignorer ces prédicats et de présumer que les variables n'ont

pas de valeur maximum. Cette situation peut amener l'analyse statique à détecter une anomalie à ce point et produire un faux positif. Il est désirable, pour limiter le nombre de faux positifs, de donner une valeur particulière aux variables dans cette situation. Cette valeur doit être différente des valeurs que peut prendre la variable. Cela permet de différencier cette valeur d'une valeur obtenue lors du traitement. La production de diagnostic peut alors ignorer ces problèmes lorsqu'une variable avec cette valeur particulière est rencontrée. La majorité des outils d'analyse permettent aux utilisateurs de faire un choix leur permettant de limiter le nombre de faux positifs ou de faux négatifs. Thomas Fahringer et Bernhard Scholz [8] proposent une méthode d'analyse symbolique permettant de limiter le nombre de faux positifs dans de telles situations. Cette méthode est cependant plus complexe que celle présentée dans cet essai.

3.3.1 Bloc de base

Lorsque chaque instruction du programme source a été traduite en instructions intermédiaires, les instructions intermédiaires sont partitionnées en blocs élémentaires. Un nouveau bloc est créé avec la première instruction du programme et les instructions suivantes y sont ajoutées jusqu'à ce qu'une instruction de branchement ou une étiquette soit rencontrée. En l'absence de saut et d'étiquette, le contrôle procède séquentiellement d'une instruction à l'instruction suivante et cette instruction fait alors partie du bloc de base. Un bloc de base est donc un bloc constitué d'instructions séquentielles où le contrôle ne peut arriver que par l'instruction de tête qui est la première instruction du bloc.

Aho, Lam, Sethi et Ullman [1] présentent un algorithme qui consiste à déterminer les instructions de tête puis à construire les blocs élémentaires. Les règles servant à déterminer les instructions de tête sont les suivantes : la première instruction du programme est une instruction de tête, une instruction qui est la cible d'un branchement est une instruction de tête et finalement toute instruction qui suit immédiatement un branchement est une instruction de tête. Les instructions de tête sont les instructions susceptibles de recevoir le contrôle d'un endroit autre que l'instruction précédente.

Une fois les instructions de tête déterminées, les blocs élémentaires sont constitués d'une instruction de tête et de toutes les instructions qui la suivent jusqu'à l'instruction de tête suivante ou la fin du programme.

3.3.2 Graphe de flot de contrôle

Les instructions intermédiaires présentées dans le tableau 3.1 ne contiennent aucune instruction servant à diriger le flot de contrôle. Le flot de contrôle est plutôt représenté à l'aide d'un graphe. Dans un graphe de flot de contrôle, les sommets sont les instructions intermédiaires du programme, sous forme de blocs de base. Il y a un arc entre deux blocs, s'il est possible que le second bloc suive directement le premier dans une exécution possible du programme.

Le graphe utilisé pour représenter le flot de contrôle est décrit sous forme d'organigramme et il est adapté de l'article de Cousot [5]. L'avantage d'une telle représentation est de limiter le nombre d'éléments différents ce qui a comme effet de simplifier l'analyse statique telle que présentée au chapitre 4. La représentation choisie utilise les symboles présentés à la figure 3.5. Cette représentation graphique est utilisée pour des fins d'illustration dans cet essai. Dans la représentation interne utilisée par l'outil d'analyse, une structure de données telle qu'illustrée à la figure 4.4 est utilisée. Toutefois, il y a une correspondance un à un entre la représentation graphique et la représentation interne.

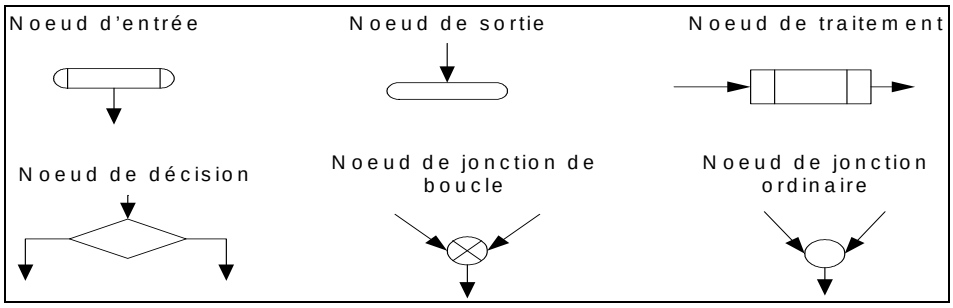
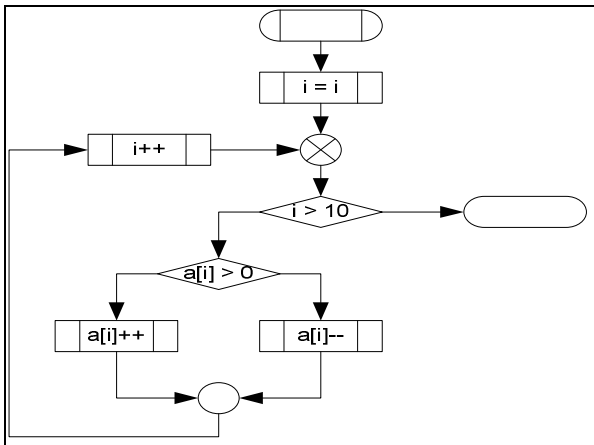


Figure 3.5 Éléments de base du flot de contrôle

Le graphe de flot de contrôle est constitué des éléments suivants :

- un nœud d'entrée est utilisé pour refléter le point d'entrée unique du programme, ce nœud n'ayant aucun arc en entrée reflète le fait qu'il s'agit du début du programme;
- un nœud de traitement ou de décision est utilisé pour chaque bloc de base, ces derniers sont constitués d'instructions générées lors de la traduction du programme;
- lorsque plusieurs arcs se rejoignent à une instruction donnée, un nœud de jonction est inséré avant l'instruction pour y recevoir ces arcs, un nœud de jonction ne reçoit que deux arcs en entrée;
- un nœud de sortie est utilisé à la fin du programme, ce nœud est le seul nœud qui n'a aucun arc de sortie.

Il doit exister au moins un chemin qui part du nœud d'entrée unique pour se rendre à chacun des autres nœuds du graphe. Ceci implique que tout cycle dans le graphe inclut au moins un nœud de jonction.




```
PERFORM VARYING I
        FROM    1 BY 1
        UNTIL   I > 10
    IF A(I) > 0
        SUBTRACT 1 FROM A(I)
    ELSE
        ADD 1 TO A(I)
    END-IF
END-PERFORM.
```

```
For (i = 1; i <= 10; i++)
{
    if (a[i] > 0)
        a(i)--;
    else
        a(i)++;
}
```

Figure 3.6 Exemple d'un diagramme de flot de contrôle

La figure 3.6 présente l'exemple d'un flot de contrôle pour une portion de programme. Le programme correspondant en COBOL et en C est à la droite de la figure.

Pour toute application particulière de l'algorithme d'évaluation abstraite, il faut définir l'évaluation des blocs de base : le nœud de traitement, le nœud de décision ainsi que les nœuds de jonction. La théorie est présentée dans l'annexe 5.

3.4 Variables

Puisque la méthode d'analyse abstraite utilise une abstraction des données du programme, il va de soi que la représentation interne des variables dépend de l'abstraction choisie.

Les variables entières sont représentées sous forme d'intervalles de valeur. Pour ces variables, il suffit de savoir si la variable est indéfinie, ou de connaître l'intervalle des valeurs que cette variable peut représenter dans un contexte donné. Pour les tableaux, il faut connaître le nombre d'entrées provenant de la définition du tableau. Pour les autres variables, il suffit de savoir si la variable est indéfinie (⊥), initialisée (d) ou s'il est impossible de déterminer si la variable est initialisée ou non (⊤). Cette dernière situation est possible dans une situation telle qu'illustrée à la figure 3.7.

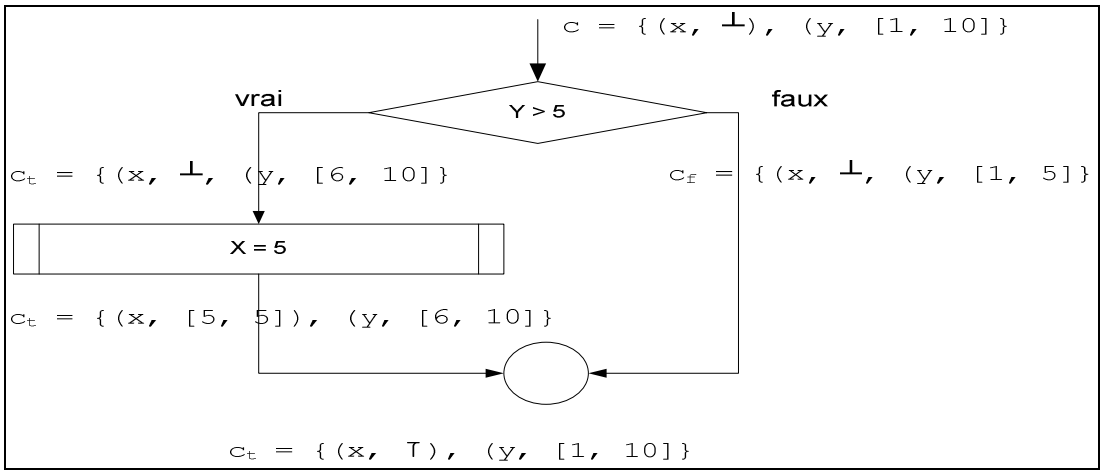


Figure 3.7 Décision entraînant une valeur indéfinie

Chapitre 4

Analyse du programme

Ce chapitre présente l'algorithme d'analyse choisi pour résoudre les problèmes ciblés par l'essai et de démontrer l'utilisation de l'algorithme à l'aide d'un exemple.

L'algorithme présenté dans ce chapitre provient d'Aho, Lam, Sethi et Ullman [1]. Cet algorithme est appelé « analyse par région ».

Dans l'analyse par région, un programme est vu comme une hiérarchie de régions. Une région est une portion du graphe de flot de données qui n'a qu'un seul point d'entrée. Ce concept, qui permet de voir le code comme une hiérarchie, devrait être intuitif puisqu'une procédure structurée en blocs est naturellement organisée en hiérarchie de régions.

Chaque instruction d'un programme structuré est une région, puisque le flot de contrôle débute au début de l'instruction. Chaque niveau d'imbrication des instructions correspond à un niveau dans la hiérarchie des régions. Plus formellement, une région d'un graphe de flot de contrôle est une collection de nœuds N et d'arcs A tels que :

- Il y a une tête t dans N qui domine tous les nœuds de N.
- Si un nœud m peut atteindre un nœud n dans N sans passer par t, alors m est également dans N.
- A constitue l'ensemble de tous les arcs dans le flot de contrôle entre les nœuds n_1 et n_2 dans N sauf, possiblement, pour certains qui entrent dans t.

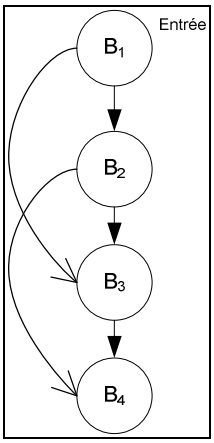


Figure 4.1 Exemple de régions

Traduction libre

Source : Aho, A., Lam, M. Sethi, R. et Ullman, J. D. (2007), p. 673

Une boucle naturelle est une région, mais une région n'a pas nécessairement un arc de retour et peut ne pas avoir de cycles. Dans la figure 4.1, les nœuds B_1 et B_2, incluant l'arc $B_1 \rightarrow B_2$, forment une région; de même que les nœuds B_1, B_2 et B_3 avec les arcs $B_1 \rightarrow B_2$, $B_2 \rightarrow B_3$ et $B_1 \rightarrow B_3$. Toutefois, le graphe formé des nœuds B_2 et B_3 avec l'arc $B_2 \rightarrow B_3$ ne forme pas une région parce que le contrôle peut entrer dans ce graphe par les nœuds B_2 et B_3. En d'autres termes, aucun des nœuds B_2 ou B_3 ne domine l'autre, donc la condition (1) des règles des régions n'est pas respectée. Même si on choisissait le nœud B_2, par exemple, comme tête, la condition (2) n'est pas respectée puisque le nœud B_3 peut être atteint de B_1 sans passer par B_2 et B_1 n'est pas dans la « région ».

Pour les besoins de l'algorithme d'analyse par région, trois types de régions sont considérées : les régions élémentaires, les régions corps et les régions boucles.

4.1 Hiérarchies de régions

Pour construire une hiérarchie de régions, il faut trouver les itérations naturelles. Dans les langages qui n'utilisent pas d'instruction de branchement direct comme les « `go to` », deux boucles dans un programme sont soit disjointes ou incluses l'une dans l'autre. Le processus de décomposition d'un graphe de flot de contrôle en sa hiérarchie de zones

itératives commence en considérant chaque bloc de base comme une région en soi; il s'agit des régions élémentaires. Par la suite, il faut ordonner les boucles naturelles de l'intérieur vers l'extérieur, c'est-à-dire en partant des itérations les plus imbriquées. Le traitement d'une boucle consiste à la remplacer par un nœud en deux étapes :

- Premièrement, le corps de la boucle B (qui inclus tous les nœuds et les arcs à l'exception de l'arc de retour vers la tête) est remplacé par un nœud représentant une région R. Les arcs vers la tête de la boucle B entrent maintenant dans le nœud de la région R. Un arc de n'importe quelle sortie de la boucle B est remplacé par un arc qui va de la région R vers la même destination. Toutefois, si l'arc en question est l'arc de retour vers la tête de la boucle, il devient alors une boucle sur R. La région R est alors appelée une région corps.
- Finalement, il faut construire une région R' qui représente la boucle naturelle B au complet. R' est appelé une région boucle. La seule différence entre la région R et la région R' est que cette dernière inclut l'arc de retour vers la tête de la boucle B.

Ces deux étapes sont répétées en réduisant des boucles de plus en plus larges en nœuds simples, premièrement avec un arc de retour puis sans cet arc. À la longue, toutes les boucles naturelles sont réduites en un nœud simple ou il peut y avoir plusieurs nœuds sans boucles, formant un graphe acyclique de plus d'un nœud. Dans le premier cas, le processus est terminé, mais dans le second cas, il faut construire un autre nœud région pour le graphe de flot complet. L'algorithme de construction des régions est détaillé dans l'annexe 6.

Un exemple de l'application de l'algorithme est présenté à l’annexe 8.

4.2 Survol de l'analyse par région

Pour chaque région R et chaque sous-région R' de R, il faut créer une fonction de transfert $f_{R,ENTRÉE[R']}$ qui résume les effets de l'exécution de tous les chemins possibles qui vont de l'entrée de R à l'entrée de R', tout en restant dans R.

Un bloc B dans R est un bloc de sortie de la région R s'il a un arc sortant du bloc B vers un bloc à l'extérieur de R. Il faut également créer une fonction de transfert pour chaque bloc de sortie B de R, appelé $f_{R,SORTIE[B]}$ qui résume les effets de l'exécution de tous les chemins possibles dans R, menant de l'entrée de la région R vers la sortie du bloc B.

L'algorithme procède ainsi, en remontant la hiérarchie des régions et en créant des fonctions de transfert pour des régions de plus en plus grandes. Il faut débuter avec les blocs de base B, où $f_{B,ENTRÉE[B]}$ est la fonction identité et $f_{B,SORTIE[B]}$ est la fonction de transfert pour le bloc B lui-même. Au fur et à mesure que le traitement remonte dans la hiérarchie,

- Si R est une région corps, les arcs appartenant à R forment un graphe acyclique des régions qui sont incluses dans R. Il faut procéder à la création des fonctions de transfert en ordre topologique des régions incluses.
- Si R est une région boucle, il faut simplement considérer l'effet de l'arc de retour vers la tête de R.

Finalement, le sommet de la hiérarchie est atteint et la fonction de transfert pour la région R_n est créée, c'est-à-dire la fonction pour le graphe au complet.

La prochaine étape consiste à calculer les contextes à l'entrée et à la sortie de chaque bloc. Les régions sont traitées en ordre inverse, en commençant par la région R_n et en descendant dans la hiérarchie. Pour chaque région, il faut calculer les contextes à l'entrée. Pour la région R_n, il faut appliquer $f_{Rn,ENTRÉE[R]}$(ENTRÉE[CONTEXTE]) pour obtenir les valeurs de flot de données à l'entrée des régions incluses R dans R_n. Il faut répéter ce processus jusqu'à ce que les blocs de base de la hiérarchie de régions soient traités.

L'algorithme d'analyse par région est présenté en détail à l'annexe 7. Un exemple d’application de l’algorithme d’analyse par région est présenté à l’annexe 9.

Chapitre 5
Diagnostic d'erreur

Le chapitre précédent a démontré, puis illustré à l'aide d'un exemple, l'utilisation de l'algorithme d'analyse par région pour déceler deux erreurs simples de programmation : l'utilisation d'une variable non initialisée et l'accès à un tableau en dehors des bornes de celui-ci.

Lorsqu'une erreur potentielle est détectée, il faut produire un diagnostic pour permettre au programmeur de déterminer les situations où ces erreurs peuvent se produire et corriger ces erreurs.

L'information requise pour produire le diagnostic des erreurs a été omise dans le traitement de l'exemple 4.1 dans le but de ne pas le rendre plus complexe. Ce chapitre décrit l'information qui doit être extraite des différentes étapes de l'analyse du programme pour être en mesure de produire un diagnostic des erreurs détectées.

La première section du chapitre indique l'information qui devrait se retrouver dans un diagnostic pour que celui-ci puisse être utile. La section suivante démontre comment les conditions rencontrées lors de la séquence d'exécution peuvent aider à produire un diagnostic plus précis des erreurs possibles. Par la suite, le reste du chapitre est consacré à indiquer comment l'étape d'analyse du programme permet l'extraction de l'information requise à la production du diagnostic.

5.1 Diagnostic

Pour qu'un diagnostic puisse être utile, il n'est pas suffisant de déterminer l'instruction où l'erreur peut se produire. Il est possible que seulement certaines séquences d'exécution rencontrent une erreur à cette instruction. Cette situation est illustrée à l'aide d'un exemple.

Dans la majorité des exemples de ce chapitre, le code est présenté en COBOL plutôt qu'en langage intermédiaire pour en faciliter la compréhension. À la droite de chaque instruction, le contexte résultant de l'instruction est présenté pour les variables d'intérêt.

Dans la figure 5.1, la variable « I » n'est initialisée lors de l'exécution de l'instruction 5 que lorsque l'instruction 3 a été exécutée. Si la variable « X » a la valeur zéro, alors l'instruction 3 n'est pas exécutée et la variable « I » n'a pas de valeur lors de l'exécution de l'instruction 5. Lorsqu'une instruction conditionnelle est rencontrée, le contexte en entrée de l'instruction est transformé en deux contextes, l'un pour la branche « vrai » de la condition et l'autre pour la branche « faux ». Les contextes de fin de la branche « vrai » et de la branche « faux » sont ensuite fusionnés après l'instruction de test. Lors de cette fusion, il est possible, comme l'illustre la figure 5.1, qu'une variable soit définie dans un des deux contextes, mais pas dans l'autre. Dans ce cas, la valeur « ⊤ » est utilisée pour indiquer que la variable peut ou non être définie à ce point.

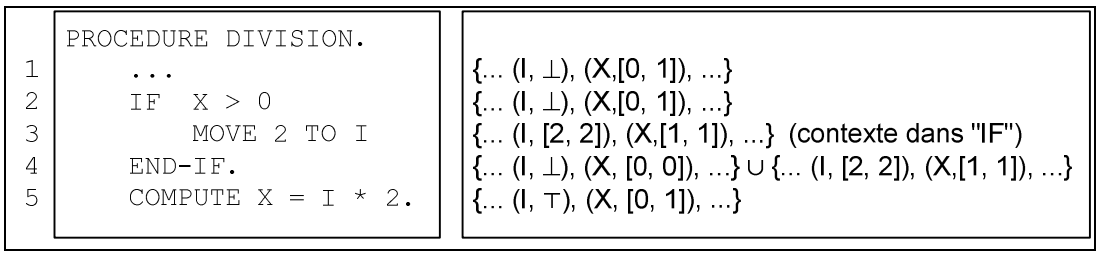


Figure 5.1 Programme où une erreur dépend du chemin d'exécution.

Dans la figure 5.1, une erreur est détectée à l'instruction 5, car le contexte à l'entrée de cette instruction a la valeur « ⊤ » pour la variable « I » qui est utilisée par cette instruction. À ce point, il est possible de produire le diagnostic suivant : « la variable I peut être indéfinie à l'instruction 5 ». Avec un tel diagnostic, le programmeur n'a pas d'indices pour l'aider dans la détermination des circonstances où la variable peut être indéfinie.

Il serait plus utile d'avoir la trace d'exécution qui a mené à l'erreur : « la variable `I` est indéfinie à l'instruction 5. Trace : {1, 2, 5} ». Dans ce diagnostic, la trace d'exécution est la séquence des instructions exécutées jusqu'à la détection de l'erreur. En suivant cette séquence d'instructions, il est plus facile de déterminer que le programme n'exécute pas l'instruction 3 ce qui indique que l'instruction conditionnelle 2 donne la condition où la variable « `I` » est définie, et par conséquent l'instruction indique également les conditions sous lesquelles la variable n'est pas définie.

Cette information peut être ajoutée au diagnostic : « la variable `I` est indéfinie à l'instruction 5. Trace : {1, 2 (`X` = 0), 5} ». Dans ce diagnostic, la trace d'exécution est jumelée avec l'information du contexte pour les variables utilisées dans l'instruction conditionnelle de manière à produire un diagnostic plus précis. Le contexte avant l'instruction indique que les valeurs possibles pour la variable « `X` » sont dans l'intervalle [0, 1] et la condition est vraie pour les valeurs supérieures à zéro. Par conséquent, la seule valeur pour laquelle l'instruction qui initialise la variable « `I` » n'est pas exécutée est la valeur zéro.

5.2 Information du contexte dans le diagnostic

L'exemple de la section précédente a démontré que l'information du contexte peut être utilisée pour aider à produire un diagnostic qui aide à déterminer les circonstances où une erreur se produit. Cette section illustre l'application de cette technique lorsque l'instruction conditionnelle est constituée de prédicats liés ensemble avec les opérateurs logiques « et » et « ou » pour former une condition plus complexe.

La figure 5.2 illustre comment une condition complexe est décomposée en une série d'opérations intermédiaires `TEST`. Chacune de ces instructions effectue un test sur une seule variable. Le graphe de flot de contrôle relie ces opérations pour conserver la sémantique de la condition complexe originale.

```
   PROCEDURE DIVISION.
1       ...
2       IF  X > 0
        AND (Y = 18
          OR Y = 25)
3           MOVE 2 TO I
4       END-IF.
5       COMPUTE X = I * 2.
```

```
2.1   TEST>   X    0   2.2   5
2.2   TEST=   Y   18   3     2.3
2.3   TEST=   Y   25   3     5
3     INIT    I    2
5     ...
```

Figure 5.2 Exemple de test complexe.

Le graphe de flot de contrôle est illustré à la figure 5.3.

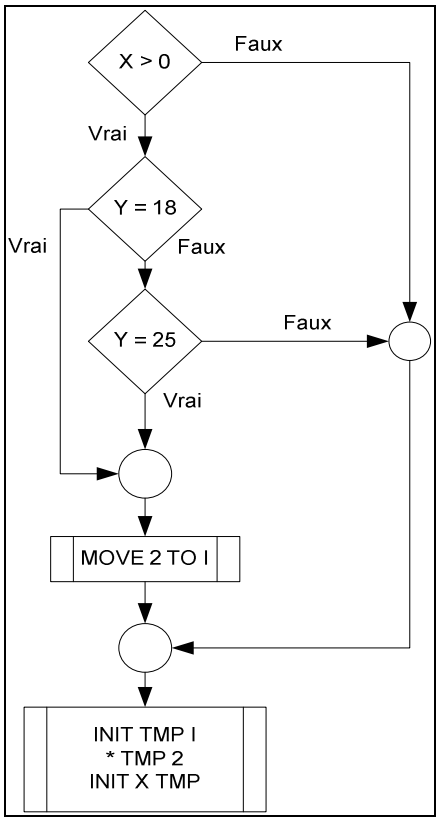


Figure 5.3 Flot de contrôle pour un test complexe.

La figure 5.4 présente les contextes associés au code intermédiaire de l'instruction conditionnelle. Pour les instructions conditionnelles, deux contextes sont produits. Le

premier contexte, celui de gauche, est le contexte passé à la première instruction de la branche « vrai » de l'instruction `IF`, le second contexte est passé à la première instruction de la branche « faux » de l'instruction `IF`. Lorsqu'un nœud de jonction est rencontré, les deux contextes qui se rejoignent à ce nœud de jonction sont fusionnés pour produire un contexte qui est ensuite passé à l'instruction suivante.

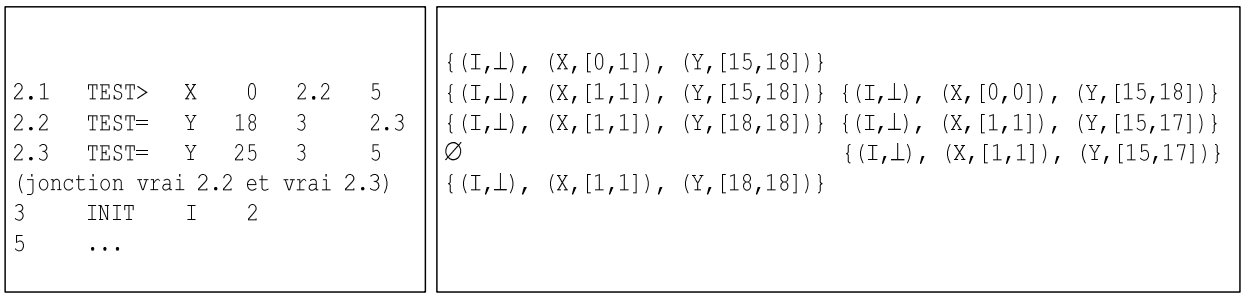


Figure 5.4. Contextes résultant d'un test complexe.

Dans la figure 5.4, il faut remarquer que l'instruction de test 2.3 ne produit pas de contexte pour la branche « vrai », car en entrée à ce test la variable « Y » ne peut pas avoir la valeur 25. Il est donc impossible que la branche « vrai » du test 2.3 soit exécutée. Lors de l'exécution de l'instruction 3, la variable « I » est indéfinie et, dans ce cas, le contenu des variables « X » et « Y » qui produit l'erreur est obtenu de façon précise.

5.3 Trace d'exécution

Cette section indique comment déterminer la trace d'exécution qui permet de reproduire cette erreur.

La méthode d'analyse par région de l'annexe 7 ne traite pas le programme instruction par instruction. La trace d'exécution doit donc être produite en partant du point où l'erreur est détectée et en reculant dans les instructions jusqu'à ce que l'instruction qui a donné une mauvaise valeur à la variable soit atteinte ou jusqu'à ce que le début du programme soit détecté.

La méthode proposée dans l'essai consiste à annoter les contextes avec de l'information supplémentaire permettant de déterminer les instructions qui ont fourni une valeur aux variables. La construction du graphe de flot de contrôle facilite ce traitement en imposant certaines règles lors de l'élaboration du graphe. La construction du graphe de flot de contrôle a été décrite à la section 3.3.2. Une règle importante est de limiter le nombre d'arcs de contrôle qui peut entrer ou sortir d'un nœud. Seulement deux arcs de contrôle peuvent entrer dans un nœud de jonction. Pour tous les autres types de nœuds, un seul arc peut y entrer. De plus, le seul nœud où plus d'un arc peut sortir est le nœud de décision.

Chaque instruction transforme son contexte en entrée pour produire un contexte en sortie. Le contexte est constitué d'un état. L'état est la liste des variables avec leur valeur à ce point de contrôle. Pour chaque variable, il faut ajouter un autre élément d'information qui est constitué d'un couple indiquant une ou deux instructions qui peuvent être à l'origine de la valeur de la variable. Ceci est illustré à l'aide du programme de la figure 5.5.

```
    PROCEDURE DIVISION.
1       READ FICH1 INTO X.
2       IF  X > 0
3           MOVE 2 TO I
        ELSE
4           MOVE 1 TO I
5           MOVE X TO J
6       END-IF.
7       COMPUTE I = I * J.
```

Figure 5.5 Exemple de contexte avec état étendu.

La première instruction initialise la variable « X » à partir d'un enregistrement lu. La variable « X » peut donc avoir n'importe quelle valeur permise par sa définition. L'état de la variable « X » est donc (X, [0,999], (1,0)). La première partie de cet uplet est le nom de la variable, la seconde partie est l'intervalle de valeurs que peut prendre cette valeur dans l'état et la troisième partie est constituée de deux numéros d'instructions qui ont pu contribuer à produire cette valeur. Après l'instruction 1, la valeur de la variable « X » provient de l'instruction 1.

Le second numéro d'instruction est toujours zéro sauf pour le contexte sortant d'un nœud de jonction. Dans ce cas, les deux numéros d'instruction indiquent la dernière instruction de la portion « vrai » et de la portion « faux » de l'instruction conditionnelle qui a modifié la variable.

L'instruction 2 ne modifie aucune variable, mais produit deux contextes en sortie pour les branches « vrai » et « faux » de l'instruction conditionnelle. Le contexte de la branche « vrai » est {(X, [1, 999], (1,0))} et le contexte pour la branche « faux » est {(X, [0, 0], (1,0))}.

L'instruction 3 donne une valeur à la variable « I » : (I, [2, 2], (3,0)). L'instruction 4 donne également une valeur à la variable « I » : (I, [1, 1], (4,0)).

L'instruction 5 initialise la variable « J » en y copiant la valeur de la variable « X ». L'état de la variable « J » devient (J, [0, 0], (5, 0)), puisque la variable « X » a la valeur zéro dans le cas où la condition de l'instruction 2 est fausse, comme indiqué par le contexte qui provient de l'instruction 2 et qui entre dans la portion « faux » du test.

L'instruction 6 représente la jonction des deux branches provenant du test. Les deux états de la variable « X » sont réunis pour donner : (X, [0, 999], (1, 0)) et les deux états de la variable « I » donnent (I, [1, 2], (3, 4)). L'état de la variable « I » indique que la valeur de la variable « I » à ce point peut être soit 1 ou 2 et que cette valeur provient de l'instruction 3 ou 4. L'état de la variable « J » indique que la variable peut être indéfinie à ce point ou qu'elle peut avoir une valeur : (J, ⊤, (6, 0)). Comme indiqué dans ce contexte, ce n'est pas l'instruction 5 qui a modifié la valeur de la variable « J », mais plutôt la fusion des deux contextes provenant du test, donc l'instruction 6.

L'instruction 7 essaie d'accéder à la variable « J » et détecte une anomalie, car la variable « J » peut être indéfinie à ce point. Pour construire la trace d'exécution, il faut utiliser les numéros d'instructions qui ont été ajoutés aux états. Le contexte en entrée de l'instruction 7

pour la variable « J » est le suivant : (J, ⊤, (6, 0)). Ce contexte nous dit que la dernière instruction à avoir modifié à variable « J » est l'instruction 6. L'instruction 6 est la jonction de deux contextes pour la valeur de la variable « J ». Pour la variable « J », un de ces contextes a une valeur qui ne reflète pas l'erreur, il s'agit de l'instruction 5 et ce n'est donc pas la source de l'erreur. L'autre contexte indique que la variable « J » est indéfinie et il s'agit du contexte à inclure dans la trace d'exécution. Il a donc été possible de remonter de l'instruction 7, à l'instruction 6 puis à l'instruction 3 pour construire une trace d'exécution partielle. Ces traces ont pu être déduites des modifications à la variable « J ».

5.4 Construction de la trace d'exécution

Pour obtenir une trace d'exécution complète, il faut retrouve toutes les instructions exécutées même si celles-ci ne modifient pas la variable. Le contexte doit être annoté des numéros de lignes d'une ou des deux instructions qui ont permis d'atteindre l'instruction.

Un contexte forme donc maintenant la liste des variables avec leur valeur, et le contexte a en plus un couple indiquant une ou deux instructions ayant donné le contrôle à l'instruction d'où provient le contexte. Comme pour le cas des variables, l'uplet ajouté au contexte aura un numéro de ligne et le second numéro de ligne est à zéro, sauf lorsque le contexte est obtenu d'un nœud de jonction.

Dans le cas de la figure 5.5, le contexte de l'instruction 1 devient : {(X, [0, 999], (1, 0))} / (1,0). La notation adoptée est d'ajouter l'uplet contenant les instructions qui conduisent au contexte en précédent cet uplet du caractère « / » pour séparer l'uplet de la liste des états des variables.

Le contexte de l'instruction 6 est : {(X, [0, 999], (1, 0)), (I, [1, 2], (3, 4)), (J, ⊤, (6, 0))} / (3, 5). Ce contexte indique que le contrôle peut provenir de l'instruction 3 ou de l'instruction 5 selon la valeur de la condition de l'instruction 2.

Le tableau 5.1 illustre les contextes de l'exemple 5.1. Ces contextes contiennent l'instruction où cette variable a possiblement obtenu sa valeur. De plus, le contexte contient le numéro d'une ou deux instructions d'où provient le contexte qui a permis de créer le contexte en sortie.

Tableau 5.1 Contexte de l'exemple 5.1

Instruction	Contexte à la fin de l'instruction
1	{(X, [0, 999], (1, 0))} / (0, 0)
2	{(X, [1, 999], (1, 0))} / (1, 0) et {(X, [0, 0], (1, 0))} / (1, 0)
3	{(X, [1, 999], (1, 0)), (I, [2, 2], (3, 0))} / (2, 0)
4	{(X, [0, 0], (1, 0)), (I, [1, 1], (4, 0))} / (2, 0)
5	{(X, [0, 0], (1, 0)), (I, [1, 1], (4, 0)), (J, [0, 0], (5, 0))} / (4, 0)
6	{(X, [0, 999], (6, 0)), (I, [1, 2], (3, 4)), (J, ⊤, (5, 0))} / (3, 5)
7	{(X, [0, 999], (6, 0)), (I, ⊤, (7, 0)), (J, ⊤, (5, 0))} / (6, 0)

Le contexte de l'instruction 6 montre que la variable « J » a déjà une valeur indiquant que cette variable peut être indéfinie. Aucune erreur n'est indiquée à ce point, car il peut être possible que cette situation soit normale : si la variable n'est plus référencée dans le reste du programme ou si la valeur de la variable est modifiée à nouveau avant que celle-ci ne soit utilisée.

L'instruction 7 a besoin du contenu de la variable « J » et à ce point la valeur « ⊤ » indique que celle-ci peut être indéfinie. Une erreur est donc détectée à ce point. La trace d'exécution doit être construite. Puisque la trace est construite à partir de l'instruction en erreur et en remontant le flot d'exécution, cette trace est produite en ordre inverse.

L'instruction 7 produit la trace partielle : {7}. L'instruction 7 indique que le contexte en entrée de l'instruction provient de l'instruction 6.

L'instruction 6 permettrait d'ajouter de l'information à la trace : {6, 7}. Par contre, puisque l'instruction 6 est un nœud de jonction, et non pas une instruction exécutable dans le programme original, la trace reste inchangée : {7}. L'instruction 6 indique que deux

contextes ont servi à produire le contexte, les instructions 3 et 5. Le contexte de l'instruction 7 indique que la valeur de l'instruction 7 peut provenir du contexte 5. Dans le contexte produit par l'instruction 5, la variable « `J` » est définie. Ce n'est donc pas le contexte qui doit être ajouté à la trace d'exécution. Le contexte de l'instruction 3 ne contient pas la variable « `J` » ce qui signifie que la variable « `J` » n'a pas de valeur à ce point. L'instruction 3 est alors ajoutée à la trace d'exécution : {3, 7}. L'instruction 3 indique que le contexte utilisé pour cette instruction provient de l'instruction 2.

L'instruction 2 est une instruction conditionnelle. La condition consiste en un test sur la valeur de la variable « `X` ». Puisque l'instruction 3 fait partie de la branche « vrai » du test, et que la condition est « `X > 0` », l'analyse détermine les valeurs de la variable « `X` » qui sont passées dans le contexte « vrai », et celles-ci sont copiées dans la trace d'exécution : {2 (`X` = [1, 999]), 3, 7}. Le contexte de l'instruction 2 provient de l'instruction 1.

L'instruction 1 est ajoutée à la trace d'exécution : {1, 2 (`X` = [1, 999]), 3, 7}. Puisque le contexte de l'instruction 1 ne donne aucun numéro d'instruction, le début du programme a été atteint. La trace d'exécution est donc {1, 2 (`X` = [1, 999]), 3, 7}.

Cette section a donné une idée de l'information requise pour pouvoir déterminer une trace d'exécution qui permet de guider lors de l'identification des anomalies présentes dans un programme. De l'information supplémentaire peut être fournie puisque le contexte de chacune des instructions est connu.

5.5 Exemple de diagnostic

La section précédente a démontré comment recueillir l'information requise à la production du diagnostic. Pour que le programmeur puisse utiliser cette information, le diagnostic doit être facile à comprendre et à interpréter. Cette section reprend la trace d'exécution produite à la section précédente et donne un exemple de diagnostic produit.

La représentation du diagnostic dépend naturellement du médium utilisé pour présenter celui-ci. Si l'outil d'analyse est intégré à un environnement de développement, il est fort possible que le diagnostic soit affiché dans une fenêtre et permette, à l'aide de liens, la navigation entre le diagnostic et le programme source original. Dans le cas d'un outil spécifique et indépendant d'un environnement de développement, le diagnostic peut prendre la forme d'un simple rapport. Dans ce dernier cas, il est important de permettre de retrouver facilement les instructions du programme original à partir du diagnostic.

Le diagnostic présenté dans cette section suppose que le diagnostic est produit indépendamment de l'environnement de développement. Il présente les instructions avec l'information de la trace d'exécution et la valeur des variables. La figure 5.6 donne un exemple d'un diagnostic pour le programme de la figure 5.5.

Dans la figure 5.6, chaque instruction de la trace d'exécution est imprimée avec son numéro de ligne. Sur les lignes précédant l'instruction, les variables utilisées par l'instruction sont affichées avec leur valeur. Sur les lignes suivant l'instruction, les variables modifiées par l'instruction sont affichées avec leur nouvelle valeur. Il faut remarquer que le diagnostic n'inclut pas toutes les variables du programme. Le programme exemple est très simple, il contient sept instructions et trois variables. Un vrai programme compte beaucoup plus d'instructions et de variables. Pour ne pas produire un rapport de diagnostic trop volumineux, il est important d'inclure de l'information pertinente qui permet au programmeur de bien analyser l'anomalie identifiée.

Une ligne de séparation est insérée entre chaque instruction pour bien délimiter les étapes de la trace d'exécution. La trace est présentée dans l'ordre d'exécution des instructions, et non pas l'ordre dans lequel la trace est construite, qui donnerait la trace à partir de l'instruction en erreur en reculant dans l'exécution du programme. Cette dernière méthode rend la consultation du diagnostic moins intuitive pour l'utilisateur.

```
                                << { }
1       READ FICH1 INTO X.
                                >> X = [0, 999]
    ------------------------------------------------------
                                << X = [0, 999]
2       IF  X > 0
                                >> (VRAI) X = [1, 999]
                                >> (FAUX) X = [0, 0]
    ------------------------------------------------------
                                << X = [1, 999]
3           MOVE 2 TO I
                                >> I = [2, 2]
    ------------------------------------------------------
                                << (VRAI) X = [1, 999]
                                << (FAUX) X = [0, 0]
6       END-IF.
                                >> X = [0, 999]
                                >> I = [1, 2]
                                >> J = ⊤
    ------------------------------------------------------
                                << I = [1, 2]
                                << J = ⊤
7       COMPUTE I = I * J.
    >>> Problème: la variable J peut être indéfinie à l'instruction 7.
```

Figure 5.6 Exemple de diagnostic avec valeur des variables.

Le diagnostic de la figure 5.6 peut être interprété comme suit. À l'instruction 7, un incident potentiel est détecté et le diagnostic d'erreur y est produit. La variable qui provoque l'anomalie fait partie du message d'erreur. Il s'agit de la variable « J ». La valeur de la variable « J » en entrée à l'instruction indique une valeur « ⊤ » qui signifie qu'à ce point, il est possible que la variable ne soit pas définie. Donc, au moins un chemin d'exécution se rend à cette instruction alors que la variable « J » n'est pas définie. L'instruction précédente de la trace d'exécution est l'instruction 6. Après l'exécution de cette instruction, la variable « J » a effectivement la valeur « ⊤ ». Cependant, la valeur de la variable « J » n'apparaît plus dans la trace d'exécution. Il n'est pas évident à ce point de savoir pourquoi l'erreur se produit. Il serait plus utile d'inclure dans le diagnostic à chaque étape de la trace d'exécution la variable en erreur même lorsque celle-ci n'est pas utilisée par une instruction. Ce diagnostic est illustré à la figure 5.7.

```
                                << J = ⊥
1       READ FICH1 INTO X.
                                >> X = [0, 999]
                                >> J = ⊥
    ---------------------------------------------------
                                << X = [0, 999]
                                << J = ⊥
2       IF  X > 0
                                >> (VRAI) X = [1, 999]
                                          J = ⊥
                                >> (FAUX) X = [0, 0]
                                          J = ⊥
    ---------------------------------------------------
                                << X = [1, 999]
                                << J = ⊥
3           MOVE 2 TO I
                                >> I = [2, 2]
                                >> J = ⊥
    ---------------------------------------------------
                                << (VRAI) X = [1, 999]
                                          J = ⊥
                                << (FAUX) X = [0, 0]
                                          J = [0, 0]
6       END-IF.
                                >> X = [0, 999]
                                >> I = [1, 2]
                                >> J = ⊤
    ---------------------------------------------------
                                << I = [1, 2]
                                << J = ⊤
7       COMPUTE I = I * J.
    >>> Problème: la variable J peut être indéfinie à l'instruction 7.
```

Figure 5.7 Exemple de diagnostic avec les valeurs de la variable en erreur

Comme l'indique la figure 5.7, la valeur de la variable « `J` » apparaît avant et après chacune des instructions de la trace d'exécution. Le diagnostic indique, à l'entrée de l'instruction 6, qu'au moment de fusionner les contextes de la portion « vrai » de l'instruction conditionnelle avec le contexte de la portion « faux » que la variable « `J` » n'est pas définie dans le contexte de la portion « vrai ». Cela indique que la variable « `J` » n'est pas initialisée lorsque la condition « `X > 0` » est vrai.

La valeur « ⊤ » signifie que deux contextes ont été fusionnés et qu'un de ces contextes contient une valeur pour la variable alors que l'autre contexte indique que la variable est indéfinie. Il est donc possible de déterminer, lors de la production du diagnostic, la condition qui amène à avoir deux contextes différents en ce point. Dans la figure 5.7 à l'instruction 6, les deux contextes sont fusionnés. Dans le contexte « faux », la variable « J » est définie et elle ne l'est pas dans l'autre contexte. La branche « vrai » de l'instruction conditionnelle est exécutée lorsque la variable « X » est supérieure à zéro. Le diagnostic peut donc indiquer que la variable « J » est indéfinie à l'instruction 7 parce que la variable « X » a une valeur supérieure à zéro à l'instruction 2. Ce genre de diagnostic permet de donner un maximum d'information à l'utilisateur pour faciliter le travail de correction des défauts.

Il a été mentionné précédemment que, pour un programme volumineux, le diagnostic peut être assez long. Une caractéristique intéressante permettant de faciliter l'utilisation du diagnostic est de donner le contrôle au programmeur sur le niveau d'information fournie dans le diagnostic. Inclure la variable en erreur avec toutes les instructions de la trace d'exécution peut aider à la résolution du problème. L'outil devrait permettre au programmeur de spécifier une liste de variables qui devraient également être incluses ou omises des traces d'exécution pour permettre d'obtenir plus ou moins d'information et faciliter la localisation des erreurs.

5.6 Faux positifs

Une des difficultés qui freine l'acceptation des outils d'analyse statique est le nombre de faux positifs qui sont détectés par ces outils. Il y a des cas où un compromis doit être fait entre la complexité d'un outil d'analyse et la précision de l'analyse. Un outil d'analyse simple peut signaler des faux positifs qu'un outil plus complexe pourrait éliminer.

Si le nombre de faux positifs est très élevé par rapport au nombre de vraies anomalies détectées, il est possible que le programmeur perde confiance en l'outil s'il doit

constamment inspecter une longue liste d'incidents potentiels qui ne sont pas de vrais problèmes. De plus, si cela prend plus de temps à passer à travers la liste d'incidents potentiels que d'effectuer les essais unitaires, le processus de développement devient plus lourd. Cela réduit un des avantages des outils d'analyse

Pour ces raisons, il y a des stratégies à adopter pour limiter le nombre de faux positifs. Une de ces stratégies consiste à permettre au programmeur de spécifier le niveau de précision à apporter lors de l'analyse. Par exemple, si le prédicat qui gère l'exécution d'une boucle est trop complexe pour permettre à l'analyse de déterminer les valeurs précises des variables de récurrences, l'outil peut signaler une anomalie sans être certain qu'un incident peut se produire. Le programmeur devrait pouvoir indiquer à l'outil d'analyse d'ignorer ce genre de problèmes ou non. Cela permet de limiter les faux positifs qui sont dus au niveau de complexité de l'analyse.

Une autre stratégie consiste à permettre au programmeur d'indiquer, pour chaque erreur détectée par l'analyse statique, s'il s'agit d'un faux positif ou non. L'outil d'analyse maintient alors une base de données par programme analysé et lorsque le programme est analysé à nouveau dans le futur, ces faux positifs ne sont plus signalés.

Conclusion

Le mandat de cet essai est de présenter l'analyse statique comme outil pouvant aider une équipe de développement à construire du code de qualité plus facilement. Plus spécifiquement, l'essai a démontré comment un tel outil pouvait détecter automatiquement des problèmes fréquemment rencontrés dans l'industrie.

Dans le but de fournir aux programmeurs un outil qui permet de livrer du code de meilleure qualité, il est essentiel de cibler les principaux problèmes de qualité à adresser. Dans cet essai, deux types d'anomalies fréquemment rencontrées en pratique ont été sélectionnés, soit l'utilisation de variables qui ne sont pas initialisées avant d'être référencées et le débordement de tables.

Lorsque le choix des situations à détecter est effectué, il faut choisir une abstraction qui permet de bien localiser les anomalies ciblées. Dans l'essai, les variables numériques qui sont susceptibles d'être utilisées pour accéder à une table sont représentées sous forme d'un intervalle de valeurs. Pour les autres variables, il suffit de conserver un état indiquant si la variable est initialisée ou non.

Les programmes à analyser doivent être transformés pour pouvoir détecter la présence des problèmes ciblés. La représentation interne des instructions est choisie en fonction de l'abstraction sélectionnée. La représentation interne du programme comprend trois parties principales : la liste des données que le programme utilise, les instructions de manipulation des données et les instructions qui gèrent la séquence d'exécution du programme. L'essai a décrit une représentation interne permettant l'analyse du programme.

Plusieurs méthodes d'analyse statique sont décrites dans la littérature. Une de ces méthodes a été choisie dans cet essai : l'analyse par région. Ce choix a été guidé par le langage considéré, le COBOL, car cette méthode se prête bien à la structure d'un

programme COBOL qui est divisé en hiérarchies de paragraphes. Cette méthode d'analyse a été présentée et un exemple a été utilisé pour en démontrer le fonctionnement.

Lorsque l'analyse statique détecte une anomalie, il est important de produire un diagnostic précis et détaillé de la situation pour permettre au programmeur de localiser l'anomalie et de déterminer les conditions sous lesquelles elle peut se produire. L'essai a montré comment une trace complète d'exécution peut être fournie. À cette trace, l'information sur le contenu des variables qui mène à l'incident permet de guider le programmeur à en identifier les causes.

Une situation fréquemment rencontrée lors de l'utilisation d'un outil d'analyse statique est l'apparition de faux positifs. Plusieurs outils courent le risque d'identifier des anomalies qui ne sont pas réellement des problèmes. Un trop grand nombre de faux positifs peut amener l'utilisateur à ne pas regarder toutes les anomalies détectées. Il est possible que le programmeur ne désire pas utiliser cet outil s'il a l'impression de passer plus de temps à valider des anomalies qui n'en sont pas qu'à régler de vrais problèmes. L'essai a mentionné deux stratégies permettant de gérer le niveau de détail du diagnostic et avoir un certain contrôle sur le nombre de faux positifs produits.

L'essai a démontré comment un outil d'analyse statique fonctionne et comment développer un tel outil en considérant les situations à détecter. Cependant, il est important de considérer comment utiliser un tel outil d'analyse dans le processus de développement d'une entreprise. Pour être efficace, le programmeur doit utiliser l'outil d'analyse comme il utilise un dévermineur. Dès que le programme est compilé sans erreurs, et avant même de commencer à le tester, le programmeur devrait utiliser l'analyse statique sur le programme pour valider que les erreurs fréquentes que l'outil est en mesure de détecter ne soient pas présentes dans le programme. Si de telles erreurs sont détectées par l'outil, le programmeur peut les corriger sans avoir eu à tester le programme. Une fois que l'outil d'analyse ne trouve plus d'erreurs, le programmeur devrait alors procéder aux essais unitaires.

Il est important de noter que l’outil d’analyse n’est pas destiné à remplacer les essais, car ces outils ne peuvent pas détecter toutes les anomalies. L’analyse statique permet la détection automatique de certaines catégories d’erreur. La détection et la correction de ces erreurs se font plus rapidement avec l’analyse statique qu’avec l’utilisation de cas d’essais.

L'essai a montré que la représentation interne des données, soit l'abstraction choisie pour les variables, est importante pour faciliter l'analyse. Ce choix de représentation doit être effectué en tenant compte des situations à détecter.

Comme il a déjà été mentionné dans l'essai, aucun outil ne peut détecter toutes les anomalies, il y a donc un lien direct entre la complexité de l'outil d'analyse et la précision du résultat à obtenir.

Une avenue de recherche intéressante est de pouvoir élaborer une structure globale d'analyse indépendante de la représentation interne et qui permettrait de réutiliser les algorithmes présentés dans l'analyse avec un minimum de modification pour adresser d’autres situations problématiques.

L'essai n'a pas abordé les techniques d'analyse entre procédures qui permettent d'analyser une procédure ou une fonction et conserver le résultat de l'analyse pour l'utiliser aux points où ces fonctions sont utilisées dans le code. Ces techniques pourraient faire l'objet d'un essai.

Liste des références

[1] Aho, A. V., Lam, M. S., Sethi, R. et Ullman, J. D., *Compilers: principles, techniques, and tools*, 2e éd., Pearson Education Inc., 2007, 1009p.

[2] Burnstein, I., *Practical software testing : A Process Oriented Approach*, Springer, 2002, 400p.

[3] Chess, B. et West, J., *Secure Programming with Static Analysis*, Pearson Education Inc., juin 2007, 587p.

[4] Cousot, P. et Cousot, R., *Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, janvier 1977, p. 238–252.

[5] Cousot, P. et Cousot, R., *Static Verification of Dynamic Type Properties of Variables*, Research Report R.R. 25, Laboratoire IMAG, University of Grenoble, novembre 1975, 18 p.

[6] Cousot, P., et Cousot, R., *Towards a Universal Model for Static Analysis of Programs*, Laboratoire IMAG, University of Grenoble, janvier 1977, 90 p.

[7] Davey, B. A. et Priestley, H. A., *Introduction to Lattices and Order*, 2e éd., Cambridge University Press, 2002, 298p.

[8] Fahringer, T. et Scholz, B., *Advanced Symbolic Analysis for Compilers*, Springer, 2003, 132 p.

[9] Glass, R. L., *Cobol – a contradiction and an enigma*, Communications of the ACM, vol. 40, n° 9, septembre 1997, p. 11-13.

[10] Holzmann, G. J., *The Spin Model Checker*, Addison Wesley, 2004, 597p.

[11] Kao, H. et Chen, T. Y., *Data Flow Analysis for COBOL*, ACM SIGPLAN Notices, vol. 19, n° 7, juillet 1984, p. 18-21.

[12] Lämmel, R. et De Schutter, K., *What does aspect-oriented programming mean to Cobol?*, Proceedings of the 4th international conference on Aspect-oriented software development, mars 2005, p. 99-110.

[13] Papadimitriou, C. H., *Computational complexity*, Addison Wesley Longman, août 1995, 523p.

[14] RTI, *The Economic Impacts of Inadequate Infrastructure for Software Testing*, mai 2002, 309p.

[15] Sintzoff, M., *Calculating properties of programs by valuations on specific models*, ACM SIGPLAN Notices, vol. 7, n° 1, janvier 1972, p. 203-207.

[16] Teuscher, C., *Alan Turing: Life and Legacy of a Great Thinker*, Springer 2004, 542p.

[17] Wilf, H. S., *Generatingfunctionology*, 3e éd., A, K. Peters, Ltd, 2006, 242 p.

Annexe 1

Bibliographie

Beng Kee Kiong, D., *Compiler Technology : Tools, Translators and Language Implementation*, Kluwer academic publishers, 1997, 210p.

Benton, P. N., *On the Relationship Between Formal Semantics and Static Analysis*, ACM Computing Surveys, vol. 28, n° 2, juin 1996, p. 321-323.

Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L. et Schnoebelen, P., *Systems and Software Verification*, Springer, 2001, 190p.

Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D. et Rival, X., *A Static Analyzer for Large Safety-Critical Software*, Proceedings of the ACM SIGPLAN 2003 conference on Programming Language design and implementation PLDI '03, vol. 38, n° 5, mai 2003, p. 196-207.

Clarke, E. M., Grumbert, O. et Peled, D.A., *Model Checking*, MIT Press, 1999, 314p.

Collard, J.-F., *Reasoning About Program Transformations : Imperative Programming and Flow of Data*, Springer, New York, 2003, 235p.

Cooper, K. D. et Torczon, L., *Engineering a Compiler*, Morgan Kaufmann publishers, 2004, 801p.

Foster, J. S., Hicks, M. W. et Pugh, W., *Improving Software Quality with Static Analysis*, Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering PASTE '07, juin 2007, p. 83-84.

Ganapathy, V., Jha, S., Chandler, D., Melski, D. et Vitek, D, *Analysis and Verification: Buffer overrun detection using linear programming and static analysis*, Proceedings of the 10th ACM conference on Computer and communications security CCS '03, 2003, p. 345-354.

Grune, D., Bal, H. E., Jacobs, C. J. H. et Langendoen, K. G., *Modern Compiler Design*, Wiley, 2000, 736p.

Harrison W., *An Extensible Static Analysis Tool for COBOL programs*, Proceedings of the 15th annual conference on Computer Science CSC '87, février 1987, p. 285-291.

Holub, A. I., *Compiler Design in C*, Prentice Hall, 1990, 924p.

Janssen, J. et Corporaal, H., *Making Graphs Reducible with Controlled Node Splitting*, ACM Transactions on Programming Languages and Systems (TOPLAS), vol, 19, n° 6, novembre 1997, p. 1031-1052.

Louden, K. C., *Compiler Construction : Principles and Practice*, PWS Publishing Company, Boston, 1997, 582p.

Muchnick, S. S., *Advanced Compiler Design Implementation*, Academic Press, 1997, 856p.

Nagel, E., Newman, J. R., Göedel, K. et Girard, J-Y., *Le théorème de Gödel*, Éditions du Seuil, septembre 1989, 184p.

Nguyen, T. V. N. et Irigoin, F. *Efficient and Effective Array Bound checking*, ACM Transactions on Programming Languages and Systems, vol. 27, n° 3, mai 2005, p. 527-570.

Reps, T., Sagiv, M. et Bauer, J., *Program Analysis and Compilation, Theory and Practice*, Springer, 2007, 360p.

Rich, C. et Waters, R. C., *The Programmer's Apprentice*, ACM Press, 1990, 238p.

Scott, M. L., *Programming Language Pragmatics*, 2e éd., Morgan Kaufmann publishers, 2006, 880p.

Sneed, H., *SOFTDOC - A system for automated software static analysis and documentation*, Proceedings of the 1981 ACM workshop/symposium on Measurement and evaluation of software quality, vol. 10, no 1, janvier 1981, p. 173-177.

Srikant, Y. N. et Shankar, P., *The Compiler Design Handbook : Optimizations and Machine Code Generation*, CRC Press, 2003, 916p.

Venet, A. et Brat, G., *Precise and Efficient Static Array Bound Checking for Large Embedded C Programs*, Proceedings of the ACM SIGPLAN 2004 conference on Programming Language design and implementation, juin 2004, p.231-242.

Wang, C., Hachtel, G. D. et Somenzi, F., *Abstraction Refinement for Large Scale Model Checking*, Springer, 2006, 179p.

Annexe 2

Utilisation de l'analyse statique

Selon Brian Chess et Jacob West [3], l'analyse statique peut être utilisée pour adresser plusieurs besoins.

La vérification de type est la forme d'analyse statique la plus utilisée et celle avec laquelle les programmeurs sont les plus habitués. La plupart des programmeurs n'y prêtent aucune attention : les règles sont fixées par la syntaxe du langage et sont validées par le compilateur. Ces règles sont donc hors du contrôle du programmeur. La vérification de type n'en est pas moins de l'analyse statique. Elle permet d'éviter plusieurs catégories d'erreurs de programmation.

Les outils de vérification de style sont également des outils d'analyse statique. Les vérificateurs de styles valident le respect de règles sur l'utilisation des espacements, de l'indentation du code, sur l'utilisation de commentaires ainsi que sur le style général du programme. Les erreurs de styles détectées par ces outils nuisent à la facilité de comprendre et maintenir le programme, mais ces erreurs de style indiquent rarement qu'une erreur se produira à l'exécution de celui-ci.

Les outils d'aide à la compréhension des programmes visent à faciliter la compréhension d'une application informatique contenant plusieurs milliers de lignes de code. Les environnements de développement intégrés ont habituellement quelques fonctionnalités d'aide à la compréhension des programmes. Par exemple, certains de ces outils permettent de trouver toutes les utilisations d'une méthode ou encore de localiser la déclaration d'une variable globale. Certains outils permettent un certain niveau de transformation de programmes; par exemple en permettant de renommer des variables ou en aidant à séparer une fonction très complexe en de multiples fonctions plus simples. Ces outils permettent également d'extraire des métriques qui permettent de mesurer la complexité des programmes.

La vérification de programmes est une autre utilisation de l'analyse statique. Habituellement, ces outils utilisent une spécification accompagnée du code qui implémente cette spécification. L'outil tente de démontrer que le code est une représentation fidèle de la

spécification. Si celle-ci contient une description complète de tout ce que le programme devrait accomplir, l'outil de vérification de programme peut procéder à une vérification d'équivalence pour s'assurer que la spécification et le code sont une représentation fidèle l'une de l'autre. Les programmeurs ont rarement à leur disposition une spécification suffisamment détaillée pour qu'elle puisse être utilisée avec un outil de vérification d'équivalence, et l'effort requis pour créer une telle spécification peut être supérieur à l'effort requis pour écrire le code. Cela explique pourquoi ce type de vérification est très peu utilisé.

La majorité des outils de vérification de programme effectuent plutôt une validation à l'aide d'une spécification partielle qui ne détaille qu'une partie du comportement du programme. Ce comportement est communément appelé la vérification de propriétés. Plusieurs de ces outils se spécialisent sur les propriétés temporelles qui détaillent une séquence temporelle d'événements qui ne devrait pas se produire. Une propriété temporelle est une séquence d'événements qui devraient se produire ou non l'un après l'autre. Par exemple, une propriété temporelle pourrait spécifier qu'une adresse en mémoire ne doit pas être accédée une fois que cette adresse a été libérée. La majorité des outils de vérification de propriétés permettent aux programmeuses et aux programmeurs de décrire leurs propres spécifications dans le but de valider certaines propriétés propres à leur programme.

La détermination d’anomalies dans un programme est une autre utilisation de l'analyse statique. Le but de ces outils est de déterminer les endroits dans un programme où ce programme ne se comportera pas comme son concepteur peut s'y attendre. La majorité des outils de cette catégorie sont faciles à utiliser, car ils contiennent déjà une bonne liste d'erreurs communes qu'ils peuvent détecter. Un outil de détection de problèmes vise généralement à minimiser le nombre de faux positifs même si cela est fait au détriment du nombre de faux négatifs. Une caractéristique essentielle des outils de détermination de problèmes est la génération d'un rapport d’anomalies accompagné d'un contre-exemple qui démontre une séquence réalisable d'événements dans le programme menant à la production de l’incident.

Les outils d'analyse statique qui se spécialisent dans la vérification de sécurité utilisent les mêmes techniques que les autres outils, mais leur but principal est l'identification de problèmes de sécurité. La première génération de tels outils se contentait de parcourir le code à la recherche de l'utilisation de fonctions telles que la fonction `strcpy()` qu'il est facile de mal utiliser, et faire une liste de ces fonctions qui devraient être inspectées dans le cadre d'une revue de code manuelle. Dans ce sens, ils ressemblent plus à des vérificateurs de styles, car les points identifiés ne sont pas nécessairement des problèmes de sécurité, mais plutôt des indications qu'une vérification plus poussée devrait être effectuée. Malheureusement, ces outils sont considérés comme ayant un haut taux de faux positifs, car les programmeurs ont tendance à utiliser leur résultat comme une liste d’anomalies plutôt que comme une aide à la revue manuelle du code.

Les outils modernes sont un hybride entre les outils de vérification de propriété et les outils de détection de problèmes. Contrairement à un outil de détection de problème, un outil de vérification de sécurité ne peut pas minimiser les faux positifs aux dépens des faux négatifs. Un outil de vérification de sécurité a tendance à pencher du côté de la prudence et à identifier des morceaux de code qui devraient être sujet à une revue de code même s'il n’est pas possible d’y démontrer la présence de vulnérabilité exploitable.

Annexe 3

Rappel sur la théorie des ensembles

Cette annexe présente quelques éléments de la théorie des ensembles provenant du livre de Davey et Priestley [7].

Un ordonnancement est établi en effectuant des comparaisons entre des objets. Si P est un ensemble, alors un ordre sur P est une relation binaire $\leq$ sur P telle que, $\forall x, y, z \in P$,

$$x \leq x,$$

$$x \leq y \wedge y \leq x \Rightarrow x = y,$$

$$x \leq y \wedge y \leq z \Rightarrow x \leq z.$$

Ces conditions sont appelées respectivement la réflexivité, l'asymétrie et la transitivité. Un ensemble P qui possède une telle relation d'ordre est appelé un ensemble ordonné.

Si l'ensemble X est n'importe quel ensemble d'éléments, alors $\mathcal{P}(X)$ est l'ensemble de tous les sous-ensembles de X. $\mathcal{P}(X)$ est appelé l'ensemble des parties. Cet ensemble des parties est ordonné par l'inclusion des ensembles : pour les ensembles $A, B \in \mathcal{P}(X)$, $A \leq B$ si et seulement si $A \subseteq B$. Par exemple, si l'ensemble X est l'ensemble des nombres entiers, {…, -1, 0, 1, 2, …}, alors $\mathcal{P}(X)$ contient l'ensemble X ainsi que tous les sous-ensembles de X : {…, -1, 0, 1, 2, …, {0, 1},…, {0, 3, 128}, …}. De même, l'ensemble $\{2, 3\} \leq \{1, 2, 3, 4\}$ car l'ensemble {2, 3} est inclus dans l'ensemble {1, 2, 3, 4}.

Si P est un ensemble ordonné, P a un élément minimum s'il y a un $\perp \in P$ avec la propriété $\perp \leq x$ pour tous les $x \in P$. P a également un élément maximum $\top \in P$ tel que $x \leq \top$ pour tous les $x \in P$. Dans le contexte d'un ensemble d'éléments d'information, $\perp$ et $\top$ peuvent avoir l'interprétation suivante : $\perp$ représente l'absence d'information alors que $\top$ correspond à un élément contradictoire ou un élément ayant plusieurs interprétations.

Pour un ensemble d'éléments, $x \vee y$ est l'équivalent à sup{x, y} qui est l'élément supérieur entre x et y. De même $x \wedge y$ est l'équivalent à inf{x, y} qui est l'élément inférieur entre x et y. Donc $x \leq y \Rightarrow (x \vee y = y)$ et $(x \wedge y = x)$. En particulier, puisque la relation $\leq$ est réflexive, $x \vee x = x$ et $x \wedge x = x$.

Soit P un ensemble ordonné non vide, alors si $x \vee y$ et $x \wedge y$ existent pour tous les $x, y \in P$, alors P est appelé un treillis. Un treillis est donc un ensemble ordonné dans lequel tout couple d'éléments a un élément inférieur et un élément supérieur.

Annexe 4

Théorie de l'interprétation abstraite

L’information de cette annexe provient de l’article de Cousot et Cousot [5].

Une valeur abstraite dénote un ensemble de valeurs concrètes définies en extension ou des propriétés d'un tel ensemble qui satisfont un certain nombre de conditions dynamiques. L'ensemble des valeurs concrètes est nommé V_C et l'ensemble des valeurs abstraites V_A.

La correspondance entre un ensemble de valeurs concrètes et une valeur abstraite est établie à l'aide d'une fonction d'abstraction α :

$$\alpha: \mathcal{P}(V_C) \longrightarrow V_A$$

Cette notation indique que la fonction d'abstraction, appliquée sur l'ensemble de tous les sous-ensembles des valeurs concrètes $\mathcal{P}(V_C)$, également appelé l'ensemble des parties, produit un élément de l'ensemble des valeurs abstraites. Donc $\alpha(S)$ doit être défini pour tout S dans $\mathcal{P}(V_C)$. Par exemple, $\alpha(\{-1, 5, 3\}) = [-1, 5]$ et $\alpha(\{3, 4, 5, \ldots\}) = [3, +\infty]$.

Une autre fonction γ permet d'obtenir une valeur concrète à partir d'une valeur abstraite. Elle est définie comme suit :

$$\gamma: V_A \longrightarrow \mathcal{P}(V_C)$$

Dans le cas où l'ensemble des valeurs abstraites est l'ensemble des intervalles des nombres entiers, la fonction de concrétisation est définie comme suit :

$$\gamma([a, b]) = \{x \mid (x \in \mathbb{Z}) \wedge (a \leq x \leq b)\}$$

Les fonctions α et γ doivent être définies de façon à respecter les propriétés suivantes :

$$(\forall e \in \mathcal{P}(V_C), e \subseteq \gamma(\alpha(e)))$$

$$(\forall v \in V_A, v = \alpha(\gamma(v)))$$

En prenant comme exemple la valeur concrète 3, $\alpha(3) = [3, 3]$ et $\gamma(\alpha(e)) = \gamma([3, 3]) = 3$. La seconde propriété peut être illustrée à l'aide de la valeur abstraite [3, 7], $\gamma([3, 7]) = \{3, 4, 5, 6, 7\}$ et $\alpha(\gamma(v)) = \alpha(\{3, 4, 5, 6, 7\}) = [3, 7]$.

De manière similaire à l'union $\cup$ de deux ensembles de valeurs concrètes, l'opération $\underline{\cup}$ des valeurs abstraites doivent également être définies pour toutes les évaluations abstraites. L'opération $\underline{\cup}$ appliquée sur deux valeurs abstraites donne une valeur abstraite :

$$\underline{\cup} : V_A \times V_A \longrightarrow V_A$$

Dans le cas où nous utilisons les intervalles de nombres entiers comme valeur abstraite, l'opérateur $\underline{\cup}$ est défini de la façon suivante :

$$[a_1, b_1] \underline{\cup} [a_2, b_2] = [\text{Min}(a_1, a_2), \text{Max}(b_1, b_2)]$$

Il est assumé que la fonction d'abstraction α transpose du treillis $(\mathcal{P}(V_C),\cup)$ dans $(V_A, \underline{\cup})$:

$$\forall (e_1, e_2) \subseteq \mathcal{P}(V_C), \alpha(e_1 \cup e_2) = \alpha(e_1) \underline{\cup} \alpha(e_2)$$

Ceci implique que l'opérateur $\underline{\cup}$ possède les propriétés d'associativité, de commutativité et d'idempotence et que l'élément zéro $\perp$ de $\underline{\cup}$ est $\alpha(\emptyset)$ où $\emptyset$ est l'ensemble vide. $\perp$ est appelé la valeur abstraite nulle.

Correspondant à l'inclusion $\subseteq$ des ensembles de valeurs concrètes, l'évaluation abstraite utilise l'inclusion $\trianglelefteq$ des valeurs abstraites qui est définie par : $\forall (v_1, v_2) \in V^2_A$, $\{v_1 \trianglelefteq v_2\} \Leftrightarrow \{ v_1 \underline{\cup} v_2 = v_2 \}$ et $\{v_1 \triangleleft v_2\} \Leftrightarrow \{ (v_1 \trianglelefteq v_2) \wedge (v_1 \blacktriangleright v_2)\}$.

Pour le domaine des intervalles des nombres entiers $[a_1, b_1] \trianglelefteq [a_2, b_2] \Leftrightarrow \{(a_2 \leq a_1) \wedge (b_2 \geq b_1)\}$. V_A est donc un semi-treillis complet avec un ordre partiel $\trianglelefteq$. Par exemple, $[2, 5] \trianglelefteq [1,18]$ car l'intervalle de valeur [2, 5] est complètement inclus dans l'intervalle [1, 18]. Par contre, aucune relation d'ordre n'est définie pour les intervalles [2,5] et [3,6], c'est pourquoi la relation est appelée un ordre partiel.

Pour l'évaluation des boucles dans les programmes, il est possible de rencontrer un problème avec l'évaluation de la limite d'une séquence qui croît de façon infinie en un nombre fini d'étapes. Pour éviter ces situations, une opération d'élargissement ∇ est définie :

$$\nabla: V_A \times V_A \longrightarrow V_A$$

Pour chaque évaluation abstraite, ∇ doit être défini de façon à respecter les deux conditions suivantes, premièrement $\forall (v_1, v_2) \in V^2_A$, $\{(v_1 \cup v_2) \trianglelefteq (v_1 \nabla v_2)\}$ et chaque séquence infinie $e_0 = \perp$, $e_1 = e_0 \nabla v_1$, …, $e_n = e_{n-1} \nabla v_n$, (où v_1, v_2, …, v_n sont des valeurs abstraites arbitraires) n'est pas strictement croissante. Dans le cas où les valeurs abstraites sont des intervalles de valeur, l'opérateur ∇ peut être défini comme suit :

$$[a_1, b_1] \nabla [a_2, b_2] = [\text{si } a_2 < a_1 \text{ alors } -\infty \text{ sinon } a_1, \text{ si } b_2 > b_1 \text{ alors } +\infty \text{ sinon } b_1]$$

Une liste ne peut donc pas croître à l'infini comme le montre la séquence d'étapes de l'exemple suivant : $\perp \nabla [1, 10] = [1, 10]$, $[1,10] \nabla [1, 11] = [1, +\infty]$, $[1, +\infty] \nabla [0, 12] = [-\infty, +\infty]$. L'intervalle $[-\infty, +\infty]$ résultant est l'élément maximum du treillis des valeurs abstraites, donc il ne change plus lors des itérations suivantes.

Annexe 5

Sémantique du langage

Cette annexe présente la sémantique utilisée dans cet essai. Cette portion théorique est tirée de l'article de Cousot et Cousot [5]. Dans la discussion qui suit, I représente l'ensemble des identifiants et $\mathfrak{C}$ l'ensemble des contextes.

La fonction $\Im$ détermine, pour tout nœud de traitement ou de décision n et contexte d'entrée c, un contexte de sortie $\Im(n, c)$, (ou deux contextes dans le cas d'un nœud de décision). La fonction $\Im$ doit être une abstraction représentant fidèlement l'exécution de l'instruction qui correspond au nœud n.

Pour un nœud de traitement, $\forall n \in N_t$ (l'ensemble des nœuds de traitement), n a la forme $v := f(v_1, \ldots, v_m)$ où $(v, v_1, \ldots, v_m) \in I^{m+1}$ et $f(v_1, \ldots, v_m)$ est une expression du langage source qui dépend des variables $v_1, \ldots, v_m$. Par exemple, l'expression « x + y * 5 » est une fonction qui peut être vue comme f (x, y). Alors

$$\{\forall c \in \mathfrak{C}, \forall i \in I, i \neq v \Rightarrow \Im(n, c)(i) = c(i)\} \text{ et}$$

$$\{\forall c \in \mathfrak{C}, \Im(n, c)(v) = \alpha[\{f(\upsilon_1, \ldots, \upsilon_m) \mid (\upsilon_1, \ldots, \upsilon_m) \in \gamma(c(v_1)) \times \ldots \times \gamma(c(v_m))\}]\}$$

La première condition indique que l'évaluation de l'expression $f(v_1, \ldots, v_m)$ n'a pas d'effet secondaire et la seconde condition que la valeur de v dans le contexte de sortie est l'abstraction de l'ensemble des valeurs de l'expression $f(v_1, \ldots, v_m)$ lorsque les valeurs $(\upsilon_1, \ldots, \upsilon_m)$ de $(v_1, \ldots, v_m)$ proviennent du contexte d'entrée c.

La figure A5.1 illustre deux nœuds de traitement, le contexte d'entrée et le contexte de sortie calculé à partir de l'instruction n et du contexte d'entrée c. La fonction $\Im$ est définie en utilisant une arithmétique des intervalles.

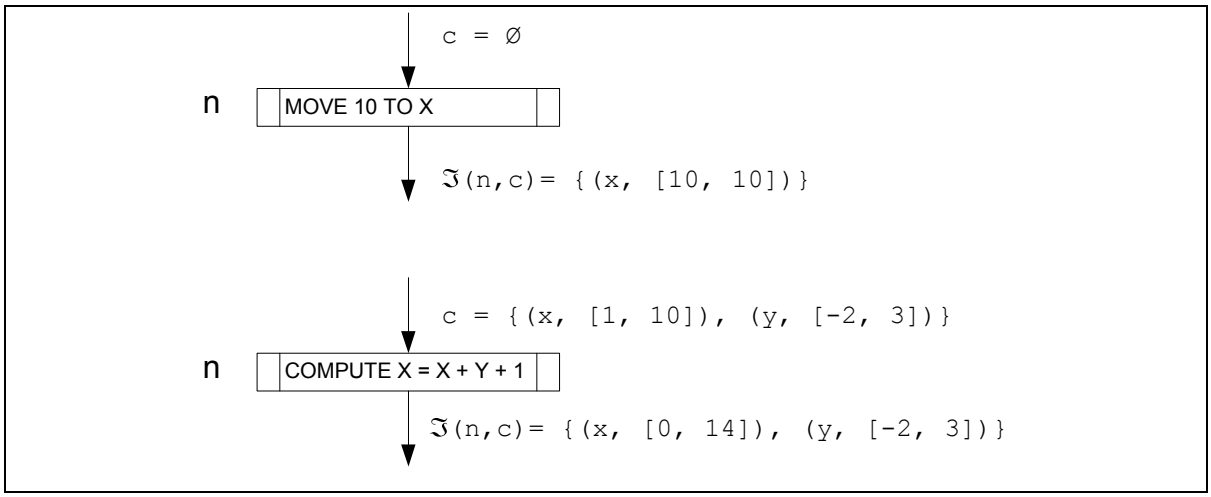


Figure A5.1 Exemple de nœud de traitement

Dans le cas d'un nœud de décision, l'utilisation de $\Im$ sur un contexte d'entrée résulte en deux contextes de sortie, c_v et c_f associés avec les arcs « vrai » et « faux » respectivement. $\forall n \in N_t$ (l'ensemble des nœuds de test), n a la forme $T(v_1, \ldots, v_m)$ où $T(v_1, \ldots, v_m)$ est une expression booléenne sans effet secondaire en fonction des variables $v_1, \ldots, v_m$. $\Im(n, c) = (c_v, c_f)$ tel que :

$$\forall i \in I$$

$$c_t(i) = \alpha(\{(\iota \mid \iota \in \gamma(c(i))) \wedge (\exists\ (v_1, \ldots, v_m) \in \gamma(c(v_1)) \times \ldots \times \gamma(c(v_m)) \mid T(v_1, \ldots, v_m))\})$$

$$c_f(i) = \alpha(\{(\iota \mid \iota \in \gamma(c(i))) \wedge (\exists\ (v_1, \ldots, v_m) \in \gamma(c(v_1)) \times \ldots \times \gamma(c(v_m)) \mid \neg T(v_1, \ldots, v_m))\})$$

Pour l'arc vrai, par exemple, la valeur abstraite d'une variable i est l'abstraction de l'ensemble des valeurs ι provenant du contexte d'entrée pour lesquelles l'évaluation du prédicat T dans le contexte c retourne la valeur « vrai ».

Dans le cas d'un nœud de jonction, deux chemins d'exécutions se rencontrent et chacun de ces chemins a son propre contexte d'entrée. Le nœud de jonction sert à synchroniser ces

chemins d'exécution et produire un contexte de sortie qui est une fonction de chacun des contextes d'entrée. Un exemple de nœud de jonction est donné à la figure A5.3.

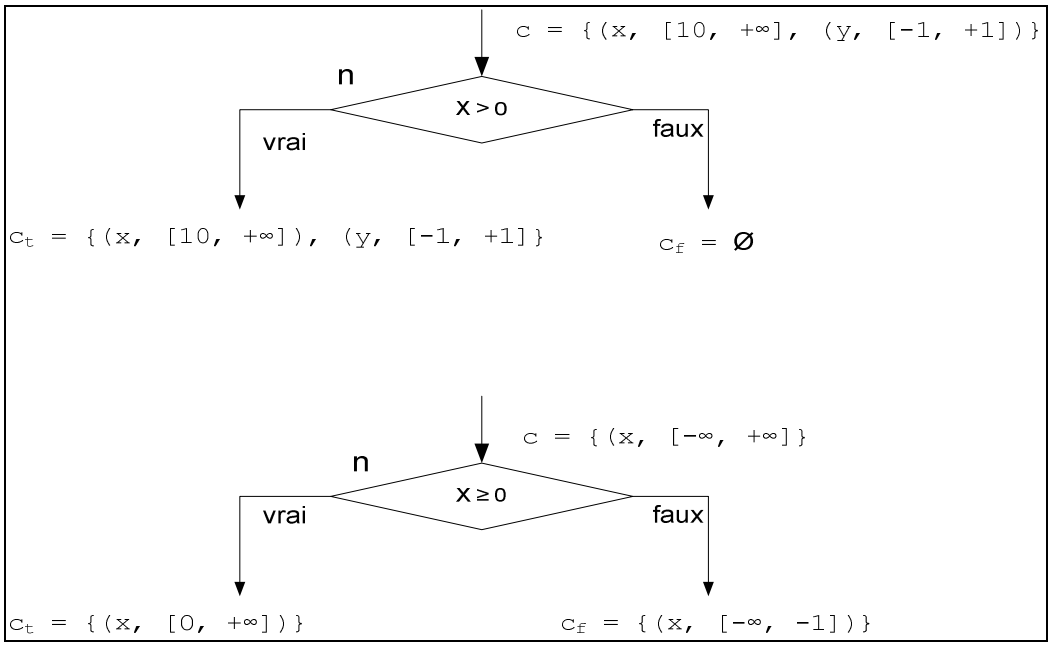


Figure A5.2 Exemple de nœud de décision

Pour pouvoir produire le contexte de sortie d'un nœud de jonction, il faut que chacun des contextes d'entrée soit déjà calculé. Pour un nœud de jonction ordinaire, le contexte de sortie $c = \underline{\cup}_{i \in [1,m]} c_i$. L'utilisation de cette notation provient de la commutativité et de l'associativité de $\underline{\cup}$.

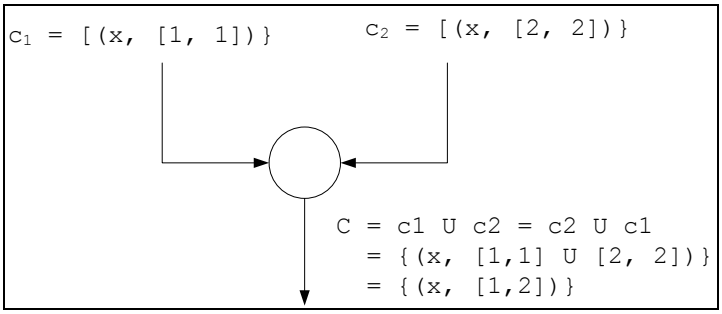


Figure A5.3 Exemple de nœud de jonction ordinaire

Dans le cas d'un nœud de jonction de boucle, puisque le contexte de sortie de l'itération est un des contextes d'entrée du nœud de jonction, il est possible qu'à chaque itération le contexte de sortie de l'itération soit modifié ce qui provoque une réévaluation du contexte de sortie du nœud de jonction de la boucle. Pour assurer que le processus d'analyse se termine, un opérateur d'élargissement est utilisé dans le calcul du contexte de sortie. Cet opérateur est défini de façon à garantir l'atteinte d'un contexte stable à l'intérieur d'un nombre fini d'itérations. Il est important de noter que cet opérateur d'élargissement introduit une perte de précision au niveau de l'abstraction des variables de récurrences. La figure A5.4 donne un exemple d'un nœud de jonction de boucle.

Si l'itération j du traitement d'un nœud de jonction de boucle a produit le contexte Sj pour l'arc de sortie du nœud, alors le contexte associé à cet arc lors de l'itération j+1 sera : $S_{j+1} = Sj \nabla (\underline{\cup}_{i \in [1,m]} c_{i,j+1})$.

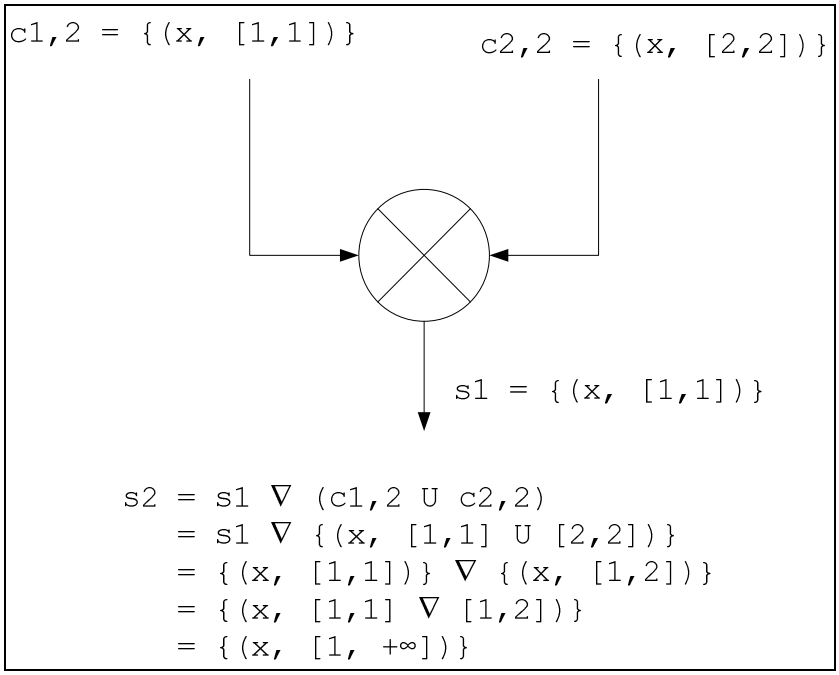


Figure A5.4 Exemple de nœud de jonction de boucle

Dans le cadre de l'essai, l'opérateur d'élargissement consiste à remplacer la valeur maximum de l'intervalle par la valeur maximum des variables de récurrences selon la méthode décrite à la section 3.3.

Annexe 6

Algorithme de construction des régions

L'algorithme de construction des régions présenté dans cette annexe provient du livre d'Aho, Lam, Sethi et Ullman [1].

ENTRÉE : un graphe de flot G.

SORTIE : une liste de régions de G qui peuvent être utilisées dans un algorithme d'analyse par région.

MÉTHODE :

1- La liste débute avec les blocs de base de G, dans n'importe quel ordre. Ces blocs de base constituent les régions élémentaires.

2- Itérativement, choisir une boucle naturelle B de façon que s'il y a d'autres boucles naturelles à l'intérieur de B, ces boucles ont déjà eu leur région corps et leur région boucle ajoutée à la liste auparavant. Premièrement, ajouter la région formant le corps de B, puis la région boucle de B.

3- Si le flot complet n'est pas en lui-même une boucle naturelle, il faut ajouter à la fin de la liste une région constituée du flot de contrôle complet.

Annexe 7

Algorithme de l'analyse par région

L'algorithme de l'analyse par région présenté dans cette annexe provient du livre d'Aho, Lam, Sethi et Ullman [1].

ENTRÉE : un graphe de flot de données réductible G.

SORTIE : les valeurs de flot de données ENTRÉE[B] pour chaque bloc B de G.

MÉTHODE :

1- Utiliser l'algorithme de l'annexe 6 pour construire la séquence, allant du bas vers le haut, des régions de G, soit R_1, R_2, …, R_n où R_n est la région la plus élevée de la hiérarchie. La région R_n correspond au programme ou à la fonction analysée.

2- Effectuer l'analyse du bas vers le haut pour construire les fonctions de transfert. Les fonctions de transfert résument l'effet d'exécuter une région. Pour chaque région R_1, R_2, …, R_n, dans l'ordre de bas en haut, effectuer :

   A. si R est une région de base, correspondant au bloc B, alors faire de $f_{R,ENTRÉE[B]} = \mathcal{I}$, et $f_{R,SORTIE[B]} = f_B$, les fonctions de transfert associées au bloc B;

   B. si R est une région corps;
   pour (chaque région S immédiatement contenue dans R, en ordre topologique)
   {
   - $f_{R,ENTRÉE[S]} = \wedge_{\text{(prédécesseurs B dans R vers la tête de S)}} f_{R,SORTIE[B]}$; /* si S est la tête de la région R, alors $f_{R,ENTRÉE[S]}$ est la jointure sur rien, ce qui revient à la fonction identité */
   - pour (chaque bloc de sortie B dans S) $f_{R,SORTIE[B]} = f_{S,SORTIE[B]} \circ f_{R,ENTRÉE[S]}$

   };

C. si R est une région boucle, alors S est la région corps immédiatement imbriqué dans R; c'est-à-dire S est R sans arc de retour de R vers la tête de R;

- $f_{R,ENTRÉE[S]} = (\wedge_{\text{prédécesseurs B dans R vers la tête de S}} f_{S,SORTIE[B]})^*$;
- pour (chaque bloc de sortie B de R) $f_{R,SORTIE[B]} = f_{S,SORTIE[B]} \circ f_{R,ENTRÉE[S]}$;

3- Effectuer une passe du haut en bas pour trouver les contextes au début de chaque région

A. ENTRÉE[R_n] = ENTRÉE[CONTEXTE];

B. pour chaque région r dans {R_1, …, R_{n-1}}, dans l'ordre de haut en bas, déterminer ENTRÉE[R] = $f_{R',ENTRÉE[R]}$(ENTRÉE[R']), où R' est la région incluant R.

Ce qui suit est une explication détaillée du fonctionnement de l'algorithme.

Dans la première ligne de 2b, chaque région incluse dans une région corps est visitée dans un ordre topologique quelconque. La seconde ligne de 2b calcule la fonction de transfert qui représente tous les chemins possibles de la tête de R vers la tête de S; puis la dernière ligne de 2b calcule les fonctions de transfert qui représentent tous les chemins possibles de la tête de R vers les sorties de R, soit vers les sorties de tous les blocs qui ont des successeurs à l'extérieur de S. Il faut noter que tous les prédécesseurs B' dans R doivent être dans des régions qui précèdent S dans l'ordre topologique construit à la première ligne de 2b. Donc $f_{R,SORTIE[B']}$ aura déjà été calculé à la dernière ligne de 2b lors d'une itération précédente de la boucle externe.

Pour les régions boucles, ce sont les étapes de la section 2c qui sont exécutées. La première ligne de 2c calcule les effets de l'exécution de la région corps de la boucle, zéro fois ou plus. La seconde ligne de 2c calcule le résultat aux sorties de la boucle après une ou plusieurs itérations.

Dans l’exécution de haut vers le bas de l'algorithme, l'étape 3a commence par assigner la condition frontière à l'entrée de la région la plus externe. Dans le cas d'un programme, il s'agit habituellement d'un contexte vide. Puis, si R est immédiatement contenu dans R' il suffit d'appliquer la fonction de transfert $f_{R',ENTRÉE[R]}$ à la valeur de flot de données de ENTRÉE[R] pour obtenir ENTRÉE[R'].

Annexe 8

Exemple de construction des régions

Cette annexe est basée sur un exemple provenant du livre d'Aho, Lam, Sethi et Ullman [1] et démontre comment l'algorithme présenté à l'annexe 6 est utilisé pour déterminer les régions d'un programme. Le programme utilisé dans cet exemple est très simple, il est présenté à la figure A8.1. La portion gauche de la figure A8.1 représente une portion des déclarations du programme alors que la portion droite donne les instructions du programme. Les instructions sont numérotées pour permettre de les référencer plus facilement dans le reste du texte.

```
IDENTIFICATION DIVISION.
    PROGRAM-ID. EXEMPLE.
ENVIRONMENT DIVISION.
    SELECT FICH1 ASSIGN TO FICH1
       FILE STATUS IS FICH1-CD-STAT.
DATA DIVISION.
FILE SECTION.
FD  FICH1.
01  ENRG.
    05  ENRG-CD-ORIG    PIC 99.
    05  ENRG-RESTE      PIC X(78).
WORKING-STORAGE SECTION.
01  I                   PIC 999.
01  FICH1-CD-STAT       PIC 99.
01  TB        OCCURS 50 TIMES.
    05  TB-ENRG         PIC X(80).
```

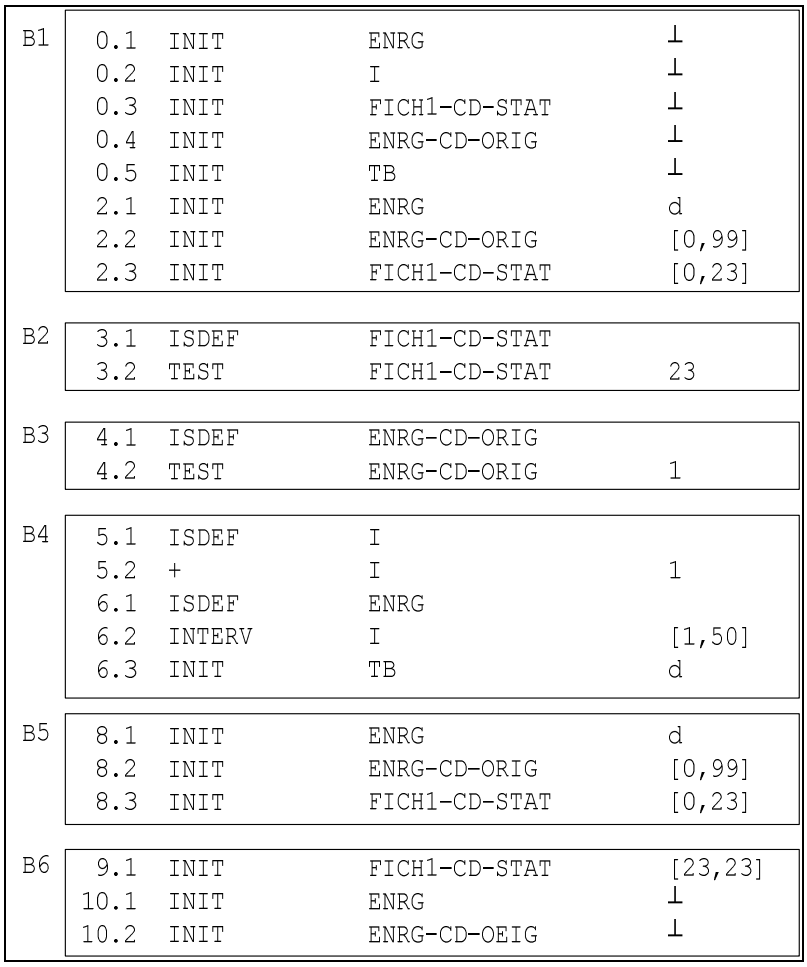
```
    PROCEDURE DIVISION.
 1  OPEN INPUT FICH1.
 2  READ FICH1.
 3  PERFORM UNTIL FICH1-CD-STAT = 23
 4      IF  ENRG-CD-ORIG = 1
 5          ADD +1        TO I
 6          MOVE ENRG     TO TB (I)
 7      END-IF
 8      READ FICH1
 9  END-PERFORM.
10  CLOSE FICH1.
```

Figure A8.1 Exemple de programme COBOL.

Avant de procéder à l'analyse, le programme est transformé pour obtenir une représentation interne. Le format de cette représentation interne a été présenté dans le chapitre 3. Le résultat de la traduction du programme exemple est présenté à la figure A8.2.

Dans la figure A8.2, les opérations intermédiaires sont numérotées en utilisant le numéro de l'instruction originale suivi d'un point décimal et d'une séquence. Ainsi, l'opération 2.2 est la seconde opération provenant de l'instruction « `READ FICH1` » du programme. Les

opérations sont également regroupées en six blocs de base numérotés B1 à B6. Le graphe de flot de contrôle provenant de la traduction du programme est présenté à la figure A8.3.



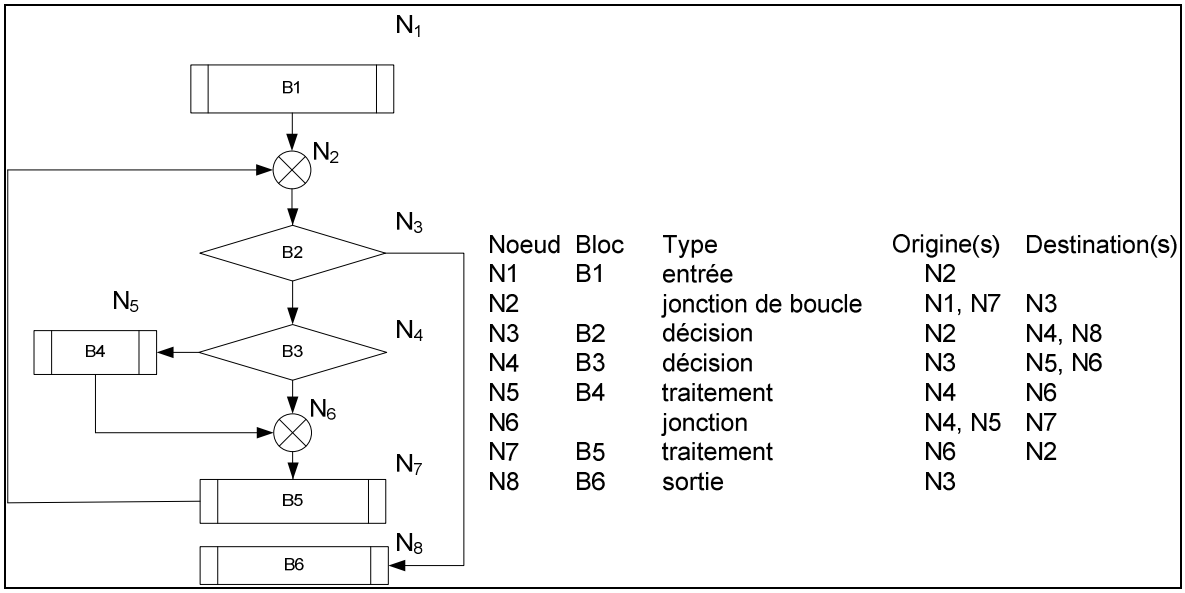
B1	0.1	INIT	ENRG	⊥
	0.2	INIT	I	⊥
	0.3	INIT	FICH1-CD-STAT	⊥
	0.4	INIT	ENRG-CD-ORIG	⊥
	0.5	INIT	TB	⊥
	2.1	INIT	ENRG	d
	2.2	INIT	ENRG-CD-ORIG	[0,99]
	2.3	INIT	FICH1-CD-STAT	[0,23]
B2	3.1	ISDEF	FICH1-CD-STAT	
	3.2	TEST	FICH1-CD-STAT	23
B3	4.1	ISDEF	ENRG-CD-ORIG	
	4.2	TEST	ENRG-CD-ORIG	1
B4	5.1	ISDEF	I	
	5.2	+	I	1
	6.1	ISDEF	ENRG	
	6.2	INTERV	I	[1,50]
	6.3	INIT	TB	d
B5	8.1	INIT	ENRG	d
	8.2	INIT	ENRG-CD-ORIG	[0,99]
	8.3	INIT	FICH1-CD-STAT	[0,23]
B6	9.1	INIT	FICH1-CD-STAT	[23,23]
	10.1	INIT	ENRG	⊥
	10.2	INIT	ENRG-CD-OEIG	⊥

Figure A8.2 Programme exemple traduit en code intermédiaire.

Les blocs de base ainsi que le graphe de contrôle constituent la représentation interne du programme. À ces deux éléments, il faut ajouter les variables utilisées par le programme.

L'exemple de cette section utilise un fichier nommé `FICH1` qui a une variable de contrôle associée : `FICH1-CD-STAT`. La valeur de cette variable est modifiée par les instructions exécutées sur le fichier. Ainsi, une opération de lecture donnera la valeur zéro à cette

variable si un enregistrement a été lu, sinon la valeur 23 signifie que la fin du fichier a été rencontrée.



Noeud	Bloc	Type	Origine(s)	Destination(s)
N1	B1	entrée	N2	
N2		jonction de boucle	N1, N7	N3
N3	B2	décision	N2	N4, N8
N4	B3	décision	N3	N5, N6
N5	B4	traitement	N4	N6
N6		jonction	N4, N5	N7
N7	B5	traitement	N6	N2
N8	B6	sortie	N3	

Figure A8.3 Graphe de contrôle.

La variable « TB » représente un tableau de 50 occurrences. Ces occurrences peuvent être accédées à l'aide d'un index dont la valeur peut varier de 1 à 50. La variable « I » dénote un nombre non signé avec trois chiffres de précision. La valeur de la variable « I » peut donc prendre des valeurs allant de zéro à 999 inclusivement. La variable « ENRG » correspond à un enregistrement lu du fichier « FICH1 ». Finalement, la variable « FICH1-CD-ORIG » est une variable qui fait partie de l'enregistrement « ENRG » et qui est donc initialisée lors de la lecture d'un enregistrement et qui devient indéfinie lorsque la fin du fichier est atteinte.

La transformation du programme source en instruction intermédiaire détecte une boucle dans le programme. Cette boucle est constituée des instructions 3 à 9 inclusivement. Cette boucle ne contient qu'une seule variable récurrente, il s'agit de la variable « I » qui est

incrémentée à l'instruction 5. Puisque le prédicat de la boucle (« `FICH1-CD-STAT = 23` ») ne contient pas de référence à la variable « `I` », il faut conclure que la boucle n'impose pas de limites à la valeur que cette variable peut prendre. La valeur de la variable « `I` » à la sortie de la boucle sera donc « ⊤ ». Ce symbole indique que la variable peut avoir la valeur maximum 999 ou même déborder et recommencer à zéro; ce symbole indique donc que la variable peut avoir une valeur quelconque entre 0 et 999.

La figure A8.4 illustre la hiérarchie de régions obtenue en utilisant l'algorithme présenté à l'annexe 6.

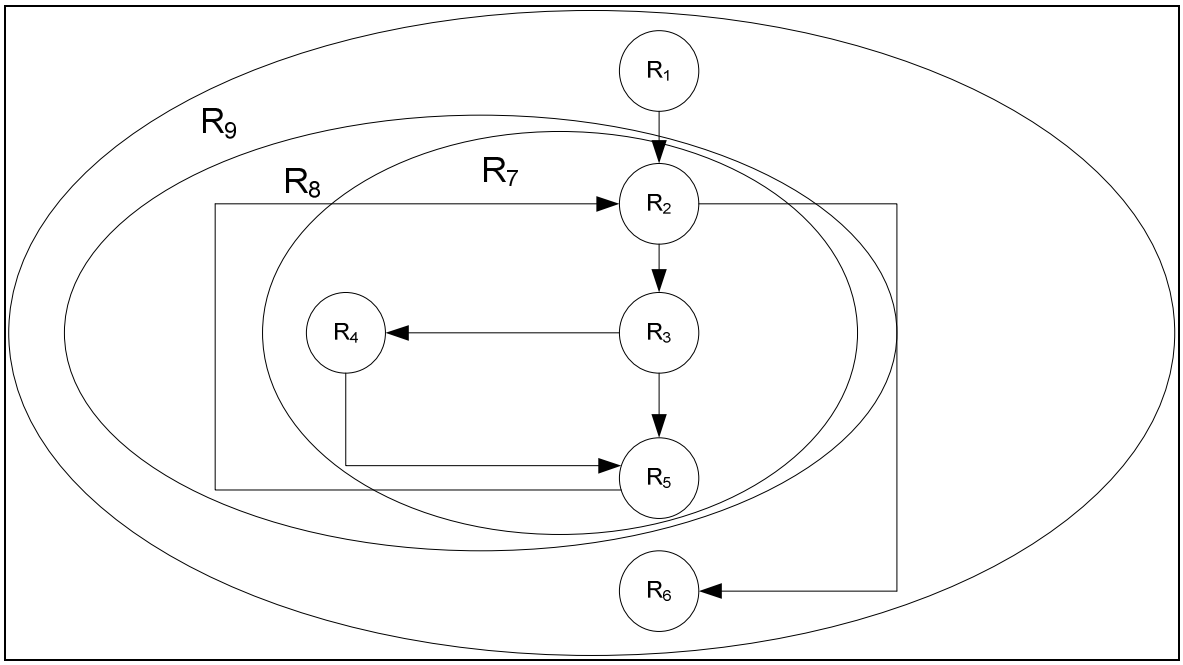


Figure A8.4 Hiérarchie de régions

Modification de : Aho, A.V., Lam, M.S., Sethi, R. et Ullman, J.D. (2007), p.675

Les régions R_1 à R_6 sont les régions élémentaires représentant les blocs B_1 à B_6 respectivement. Chaque bloc est également un bloc de sortie de sa propre région, car l'information qui sort du bloc B_1 est l'information qui sort de la région R_1.

La région corps R_7 représente le corps de la seule boucle du graphe de flot de contrôle. Elle est composée des régions R_2, R_3, R_4 et R_5 et de quatre arcs entre régions : $B_2 \rightarrow B_3$, $B_3 \rightarrow B_4$, $B_3 \rightarrow B_5$ et $B_4 \rightarrow B_5$. Elle possède un seul bloc de sortie, B_2, puisque ce bloc a un arc vers un sommet qui n'est pas inclus dans la région. La figure A8.5 (a) montre le graphe de flot où la région R_7 a été réduite à un simple nœud.

La région boucle R_8 représente la boucle au complet. Elle inclut la sous-région R_7 et un arc de retour $B_5 \rightarrow B_2$. Elle a un bloc de sortie, B_2. La figure A8.5 (b) montre le graphe de flot de contrôle lorsque la boucle complète est réduite à R_8.

Finalement, la région corps R_9 est la région supérieure. Elle inclut les trois régions R_1, R_8 et R_6 et deux arcs entre régions : $B_1 \rightarrow B_2$ et $B_2 \rightarrow B_6$. Lorsque le graphe de flot de contrôle est réduit à R_9, il est constitué d'un simple nœud. Puisqu'il n'y a pas d'arc de retour arrière vers le bloc de tête B_1, il n'y a pas lieu d'effectuer la dernière étape consistant à réduire la région corps en région boucle.

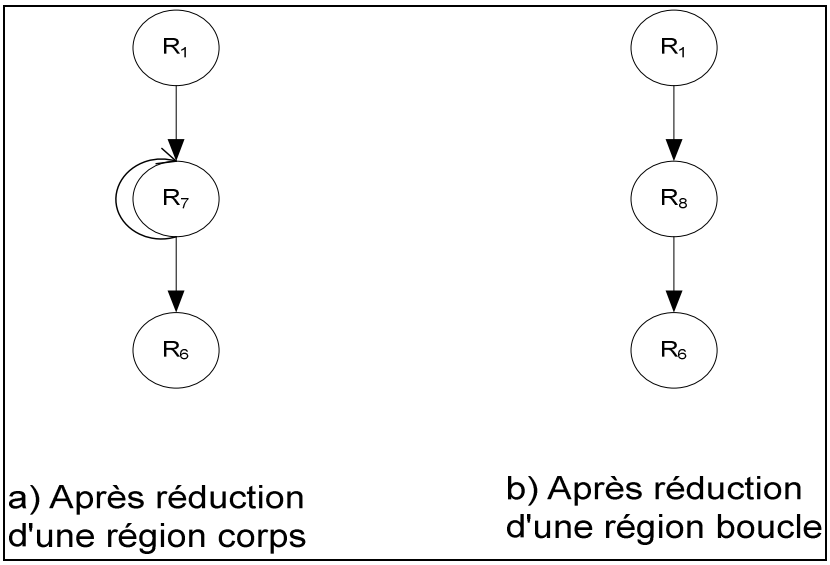


Figure A8.5 Étapes intermédiaires de la réduction du graphe de flot de contrôle

Traduction libre
Modification de : Aho, A.V., Lam, M.S., Sethi, R. et Ullman, J.D. (2007) p. 676

Annexe 9

Exemple d’analyse par région

Cette annexe est basée sur un exemple provenant du livre d'Aho, Lam, Sethi et Ullman [1] et démontre comment l'algorithme présenté à l'annexe 7 est utilisé pour effectuer l'analyse d'un programme par région. Cette annexe utilise le même programme exemple que l'annexe 8. Les régions obtenues par l'annexe 8 sont utilisées dans cet exemple.

Une fois les régions déterminées, l'algorithme de l'annexe 7 est utilisé pour déterminer les valeurs qui atteignent les différentes régions. Il faut commencer par déterminer l'ordre dans lequel les régions seront visitées. Cet ordre est du bas de la hiérarchie vers la région la plus élevée. Dans l'exemple, cet ordre est l'ordre numérique de leur indice, R_1, R_2, …., R_n.

Le but de l'analyse statique du programme exemple est de détecter les utilisations d'une variable non initialisée ou les accès aux éléments d'un tableau hors des bornes de ce dernier. Pour atteindre ces deux buts, les opérations `ISDEF` et `INTERV` ont été insérées dans la représentation interne des instructions. Lorsque les instructions sont regroupées en blocs de base, il est possible de retrouver plus d'une instruction du programme original dans un de ces blocs de base. Une des tâches de l'analyse par région est de recueillir les informations `ISDEF` et `INTERV` ainsi que les modifications apportées aux variables au niveau du bloc et de ramener ces informations au niveau des différentes régions.

À l'intérieur d'un bloc, il faut déterminer la première opération effectuée sur chacune des variables. Si cette opération est `ISDEF` ou `INTERV`, il faudra alors que cette opération soit effectuée à l'entrée du bloc de base. Si l'opération est une instruction de modification de variable, alors on peut ignorer les autres opérations `ISDEF` et `INTERV` sur cette variable dans le bloc. De plus, il faut cumuler les effets des modifications effectuées aux variables jusqu'à la fin du bloc de base.

La figure A9.1 présente un exemple d'un bloc de base construit à partir de quatre instructions COBOL. Les opérations internes sont illustrées dans la partie droite de la figure.

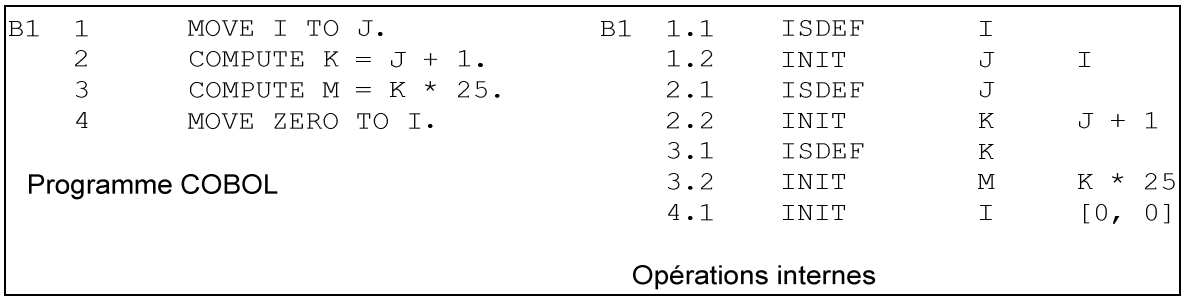


Figure A9.1 Bloc de base contenant quatre instructions

La figure A9.2 représente le bloc de base résultant lorsque l'instruction `ISDEF` 2.1 est retirée. Du point de vue du bloc de base, cette instruction est inutile, car la variable « `J` » est définie à l'instruction 1.2 ce qui permet d'éliminer l'opération 2.1. L'opération 3.1 peut également être éliminée, car la variable « `K` » est définie par l'opération précédente.

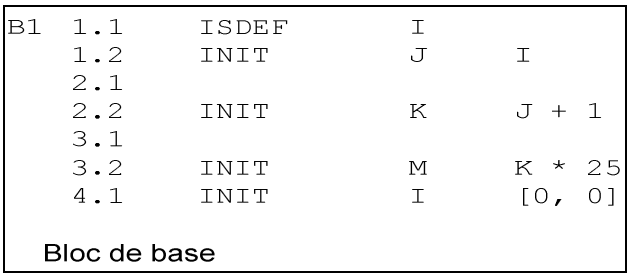


Figure A9.2 Bloc de base épuré.

Dans le contexte résultant d'un bloc de base, chaque variable a une valeur qui provient du bloc si ce dernier modifie la variable ou qui provient du contexte en entrée dans le cas contraire. Dans l'exemple précédent, si la variable « `I` » a la valeur [1, 5] avant la première instruction du bloc, alors le contexte en sortie est {(`I`, [0, 0]), (`J`, [1, 5]), (`K`, [2, 6]), (`M`, [50, 150])}.

Dans le cas du programme exemple de l'annexe 8, les valeurs des blocs de base épurés sont illustrées à la figure A9.3.

Les six premières régions R_1, …, R_6 sont les blocs B_1, …, B_6 respectivement. Ces blocs constituent les fonctions de transfert des régions R_1 à R_6.

Le reste des fonctions de transfert construites dans l'étape 2 de l'algorithme de l'annexe 7 sont résumées dans le tableau A9.1. Dans ce tableau, l'opérateur de composition « ° » est utilisé pour indiquer l'application d'une fonction sur le résultat de la fonction qui la suit. Dans le cas de blocs de base, B2 ° B1 signifie que les instructions du bloc B2 sont appliquées sur le contexte à la sortie du bloc B1.

B1	0.5	INIT	TB	⊥
	2.1	INIT	ENRG	d
	2.2	INIT	ENRG-CD-ORIG	d
	2.3	INIT	FICH1-CD-STAT	[0,23]
B2	3.1	ISDEF	FICH1-CD-STAT	
	3.2	TEST	FICH1-CD-STAT	23
B3	4.1	ISDEF	ENRG-CD-ORIG	
	4.2	TEST	ENRG-CD-ORIG	1
B4	5.1	ISDEF	I	
	5.2	+	I	1
	6.1	ISDEF	ENRG	
	6.2	INTERV	I	[1,50]
	6.3	INIT	TB	d
B5	8.1	INIT	ENRG	d
	8.2	INIT	ENRG-CD-ORIG	d
	8.3	INIT	FICH1-CD-STAT	[0,23]
B6	9.1	INIT	FICH1-CD-STAT	[23,23]
	10.1	INIT	ENRG	⊥
	10.2	INIT	ENRG-CD-OEIG	⊥

Figure A9.3 Blocs de base résultants du traitement des blocs élémentaires.

Tableau A9.1 Sommaire des fonctions de transfert.

Région	Fonction de transfert
R1	$f_{R1,ENTRÉE[B1]} = \mathcal{I}$ $f_{R1,SORTIE[B1]} = B1$
R2	$f_{R2,ENTRÉE[B2]} = \mathcal{I}$ $f_{R2,SORTIE[B2]} = B2$
R3	$f_{R3,ENTRÉE[B3]} = \mathcal{I}$ $f_{R3,SORTIE[B3]} = B3$
R4	$f_{R4,ENTRÉE[B4]} = \mathcal{I}$ $f_{R4,SORTIE[B4]} = B4$
R5	$f_{R5,ENTRÉE[B5]} = \mathcal{I}$ $f_{R5,SORTIE[B5]} = B5$
R6	$f_{R6,ENTRÉE[B6]} = \mathcal{I}$ $f_{R6,SORTIE[B6]} = B6$
R7	$f_{R7,ENTRÉE[R2]} = \mathcal{I}$ $f_{R7,SORTIE[B2]} = f_{R2,SORTIE[B2]} \circ f_{R7,ENTRÉE[R2]} = B2 \circ \mathcal{I} = B2_{vrai} + B2_{faux}$ $f_{R7,ENTRÉE[R3]} = f_{R7,SORTIE[B2]} = B2_{faux}$ $f_{R7,SORTIE[B3]} = f_{R3,SORTIE[B3]} \circ f_{R7,ENTRÉE[R3]} = (B3_{vrai} \circ B2_{faux}) + (B3_{faux} \circ B2_{faux})$ $f_{R7,ENTRÉE[R4]} = f_{R7,SORTIE[B3]} = B3_{vrai} \circ B2_{faux}$ $f_{R7,SORTIE[B4]} = f_{R4,SORTIE[B4]} \circ f_{R7,ENTRÉE[R4]} = B4 \circ B3_{vrai} \circ B2_{faux}$ $f_{R7,ENTRÉE[R5]} = f_{R7,SORTIE[B3]} \wedge f_{R7,SORTIE[B4]} = (B3_{faux} \circ B2_{faux}) \wedge (B4 \circ B3_{vrai} \circ B2_{faux})$ $f_{R7,SORTIE[B5]} = f_{R5,SORTIE[B5]} \circ f_{R7,ENTRÉE[R5]} = B5 \circ ((B3_{faux} \circ B2_{faux}) \wedge (B4 \circ B3_{vrai} \circ B2_{faux})) = (B5 \circ B3_{faux} \circ B2_{faux}) \wedge (B5 \circ B4 \circ B3_{vrai} \circ B2_{faux})$
R8	$f_{R8,ENTRÉE[B7]} = f^{*}_{R7,SORTIE[B5]} = ((B5 \circ B3_{faux} \circ B2_{faux}) \wedge (B5 \circ B4 \circ B3_{vrai} \circ B2_{faux}))^*$ $f_{R8,SORTIE[B2]} = f_{R7,SORTIE[B2]} \circ f_{R8,ENTRÉE[R7]} = B2_{vrai} \circ (B5 \circ ((B3_{faux} \circ B2_{faux}) \wedge (B4 \circ B3_{vrai} \circ B2_{faux})))^*$
R9	$f_{R9,ENTRÉE[R1]} = \mathcal{I}$ $f_{R9,SORTIE[B1]} = f_{R1,SORTIE[B1]} = B1$ $f_{R9,ENTRÉE[R8]} = f_{R9,SORTIE[B1]} = B1$ $f_{R9,SORTIE[B2]} = f_{R8,SORTIE[B2]} \circ f_{R9,ENTRÉE[R8]} = (B2_{vrai} \circ (B5 \circ ((B3_{faux} \circ B2_{faux}) \wedge (B4 \circ B3_{vrai} \circ B2_{faux})))^*) \circ B1$ $f_{R9,ENTRÉE[R6]} = f_{R9,SORTIE[B2]} = (B2_{vrai} \circ ((B5 \circ B3_{faux} \circ B2_{faux}) \wedge (B5 \circ B4 \circ B3_{vrai} \circ B2_{faux}))^*) \circ B1$ $f_{R9,SORTIE[B6]} = f_{R6,SORTIE[B6]} \circ f_{R9,ENTRÉE[R6]} = B6 \circ (B2_{vrai} \circ (B5 \circ ((B3_{faux} \circ B2_{faux}) \wedge (B4 \circ B3_{vrai} \circ B2_{faux})))^*) \circ B1$

La région R_7, qui est constituée des régions R_2, R_3, R_4 et R_5, représente le corps de la boucle et n'inclut donc pas l'arc de retour $B_5 \rightarrow B_2$. L'ordre de traitement de ces régions sera le seul ordre topologique possible : R_2, R_3, R_4, R_5. Premièrement, R_2 n'a pas de prédécesseurs dans R_7 puisque l'arc $B_5 \rightarrow B_2$ est à l'extérieur de R_7. Donc, les valeurs à l'entrée de R_7 sont les valeurs à l'entrée de B_2. Plus formellement $f_{R7,ENTRÉE[B2]}$ est la fonction identité, et $f_{R7,SORTIE[B2]}$ est la fonction de transfert du bloc B_2 lui-même. Il faut noter que l'instruction du bloc B_2 est une instruction de test qui a pour effet de produire deux contextes différents. Si le contexte en entrée contient {…, (`FICH1-CD-STAT`, [0, 23]), …}, le contexte B_2vrai contiendra la valeur {…, (`FICH1-CD-STAT`, [23, 23]), …} et le contexte B_2faux contiendra la valeur {…, (`FICH1-CD-STAT`, [0, 22]), …}.

La tête de la région B_3 possède un prédécesseur dans R_7, soit R_2. La fonction de transfert vers l'entrée de la région est simplement la fonction de transfert à la sortie de B_2, soit $f_{R7,SORTIE[B2]}$, qui a déjà été déterminée. Puisque l'arc $B_2 \rightarrow B_3$ correspond à la condition « faux » du test, la fonction de transfert ne prend que la portion du contexte qui répond à cette condition. Cette fonction est composée avec la fonction de transfert de B_3 dans sa propre région pour obtenir la fonction de transfert à la sortie de B_3.

De même, la tête de la région B_4 possède un prédécesseur dans R_7, soit R_3. La fonction de transfert vers son entrée est simplement la fonction de transfert vers la sortie de B_3, soit $f_{R7,SORTIE[B3]}$, qui a déjà été déterminée. Cette fonction de transfert est modifiée pour considérer que l'arc $B_3 \rightarrow B_4$ correspond à la condition « vrai » du test de B_3. Cette fonction est composée avec la fonction de transfert de B_4 dans sa propre région pour obtenir la fonction de transfert vers la sortie de B_4.

Finalement, pour la fonction de transfert à l'entrée de R_5, il faut déterminer $f_{R7,SORTIE[B3]} \wedge f_{R7,SORTIE[B4]}$ parce que B_3 et B_4 sont des prédécesseurs de R_5. Cette fonction de transfert est composée avec la fonction $f_{R5,SORTIE[B5]}$ pour obtenir la fonction désirée $f_{R7,SORTIE[B5]}$.

La région boucle R_8 doit maintenant être considérée. Elle contient seulement une région incluse R_7 qui représente sa région corps. Puisqu'il n'y a qu'un seul arc de retour, l'arc $B_5 \rightarrow B_2$, vers la tête de R_8, la fonction de transfert représentant l'exécution du corps de la boucle zéro ou plusieurs fois est simplement $f^*_{R7,SORTIE[B5]}$. Il y a une sortie de R_8, le bloc B_2. Par conséquent, la fonction de transfert est simplement la fonction de transfert de R_7.

Finalement, il faut considérer la région R_9, le graphe de flot de contrôle au complet. Ses sous-régions sont R_1, R_8 et R_6 et ces sous-régions sont considérées en ordre topologique. Comme avant, la fonction de transfert $f_{R9,ENTRÉE[B1]}$ est simplement la fonction identité, et la fonction de transfert $F_{R9,SORTIE[B1]}$ est simplement $F_{R1,SORTIE[B1]}$, qui est à son tour simplement F_{B1}.

La tête de R_8, qui est B_2, n'a qu'un seul prédécesseur, B_1, donc la fonction de transfert vers son entrée est simplement la fonction de transfert à la sortie de B_1 dans la région R_9. Il faut composer $f_{R9,SORTIE[B1]}$ avec la fonction de transfert à la sortie de B_2 à l'intérieur de R_8 pour obtenir la fonction de transfert correspondante dans R_9. Finalement, il faut considérer R_6. Sa tête, B_6, a un prédécesseur dans R_9, soit B_2. Donc $f_{R9,ENTRÉE[B6]}$ est simplement $f_{R9,SORTIE[B2]}$.

La dernière étape de l'algorithme détecte l'existence de problèmes en calculant les contextes à partir des fonctions de transfert. De l'étape 3 (a) de l'algorithme de l'annexe 7, ENTRÉE[R_9] = $\emptyset$ puisqu'il n'y a pas de variables initialisées au début du programme.

L'analyse procède dans l'ordre illustré dans la table A9.2. L'ordre des régions traitées est de l'extérieur vers l'intérieur, ce qui signifie que les régions sont traitées dans l'ordre R_9, suivi des régions incluses dans R_9, soit R_1, R_8 et R_6. Par la suite, la région incluse dans R8, soit R7. Puis les régions qui font partie de R7 : R2, R3, R4 et R5. La région R8 est une région boucle. Il faut traiter la région corps R7 avant de pouvoir appliquer l'effet sur les variables de récurrences.

Tableau A9.2 Traitement des régions.

	ENTRÉE[Rn]	Valeur	Contexte
1	R9		ϕ
2	R1 = $f_{R9,ENTRÉE[R1]}$	$\mathcal{I}$	ϕ
3	R8 = $f_{R9,ENTRÉE[R8]}$	B1	{(ENRG, d), (I, ⊥), (FICH1-CD-STAT, [0, 23]), (ENRG-CD-ORIG, [0, 99]), (TB, ⊥)}
4	R7 = $f_{R7,ENTRÉE[R2]}$	B1 (pour la première itération)	{(ENRG, d), (I, ⊥), (FICH1-CD-STAT, [0, 23]), (ENRG-CD-ORIG, [0, 99]), (TB, ⊥)}
5	$f_{R7,SORTIE[B2]}$	B2 ° B1	{(ENRG, d), (I, ⊥), (FICH1-CD-STAT, [0, 23]), (ENRG-CD-ORIG, [0, 99]), (TB, ⊥)}
6	$f_{R7,ENTRÉE[R3]}$	$B2_{faux}$ ° B1	{(ENRG, d), (I, ⊥), (FICH1-CD-STAT, [0, 22]), (ENRG-CD-ORIG, [0, 99]), (TB, ⊥)}
7	$f_{R7,SORTIE[B3]}$	(B3 ° $B2_{faux}$° B1)	{(ENRG, d), (I, ⊥), (FICH1-CD-STAT, [0, 22]), (ENRG-CD-ORIG, [0, 99]), (TB, ⊥)}
8	$f_{R7,ENTRÉE[R4]}$	$B3_{vrai}$ ° $B2_{faux}$ ° B1	{(ENRG, d), (I, ⊥), (FICH1-CD-STAT, [0, 22]), (ENRG-CD-ORIG, [1, 1]), (TB, ⊥)}
9	$f_{R7,SORTIE[B4]}$	B4 ° $B3_{vrai}$ ° $B2_{faux}$ ° B1 L'application de B4 amène à détecter un problème puisque l'instruction « ISDEF I » est appliquée sur la valeur (I, ⊥) du contexte.	{(ENRG, d), (I, ⊥), (FICH1-CD-STAT, [0, 22]), (ENRG-CD-ORIG, [1, 1]), (TB, d)}
10	$f_{R7,ENTRÉE[R5]}$	($B3_{faux}$ ° $B2_{faux}$° B1) ∧ (B4 ° $B3_{vrai}$ ° $B2_{faux}$° B1)	{(ENRG, d), (I, ⊥), (FICH1-CD-STAT, [0, 22]), (ENRG-CD-ORIG, [1, 1]), (TB, ⊤)} On retrouve (TB, ⊤) car dans un cas la table est initialisée, mais pas dans l'autre.
11	$f_{R7,SORTIE[B5]}$	B5 ° (($B3_{faux}$ ° $B2_{faux}$° B1) ∧ (B4 ° $B3_{vrai}$ ° $B2_{faux}$° B1))	{(ENRG, d), (I, ⊥), (FICH1-CD-STAT, [0, 23]), (ENRG-CD-ORIG, [0, 99]), (TB, ⊤)}
12	R8 = $f^*_{R7,SORTIE[B5]}$	(B5 ° (($B3_{faux}$ ° $B2_{faux}$° B1) ∧ (B4 ° $B3_{vrai}$ ° $B2_{faux}$° B1)))*	{(ENRG, d), (I, ∞), (FICH1-CD-STAT, [0, 23]), (ENRG-CD-ORIG, [0, 99]), (TB, ⊤)}

	ENTRÉE[Rn]	**Valeur**	**Contexte**
13	$f_{R8,SORTIE[B2]}$	$B2_{vrai} \circ (B5 \circ ((B3_{faux} \circ B2_{faux} \circ B1) \wedge (B4 \circ B3_{vrai} \circ B2_{faux} \circ B1)))^*$	{(ENRG, d), (I, ∞), (FICH1-CD-STAT, [23, 23]), (ENRG-CD-ORIG, [0, 99]), (TB, ⊤)}
14	$f_{R9,SORTIE[B2]}$	$(B2_{vrai} \circ (B5 \circ ((B3_{faux} \circ B2_{faux} \circ B1) \wedge (B4 \circ B3_{vrai} \circ B2_{faux} \circ B1)))^*)$	{(ENRG, d), (I, ∞), (FICH1-CD-STAT, [23, 23]), (ENRG-CD-ORIG, [0, 99]), (TB, ⊤)}
15	$R6 = f_{R9,ENTRÉE[R6]}$	$(B2_{vrai} \circ (B5 \circ ((B3_{faux} \circ B2_{faux} \circ B1) \wedge (B4 \circ B3_{vrai} \circ B2_{faux} \circ B1)))^*)$	{(ENRG, d), (I, ∞), (FICH1-CD-STAT, [23, 23]), (ENRG-CD-ORIG, [0, 99]), (TB, ⊤)}
16	$f_{R9,SORTIE[B6]}$	$B6 \circ (B2_{vrai} \circ (B5 \circ ((B3_{faux} \circ B2_{faux} \circ B1) \wedge (B4 \circ B3_{vrai} \circ B2_{faux} \circ B1)))^*)$	{(ENRG, ⊥), (I, ∞), (FICH1-CD-STAT, [23, 23]), (ENRG-CD-ORIG, ⊥), (TB, ⊤)}