ÉCOLE DES MINES DE PARIS
Collège doctoral

N° attribué par la bibliothèque
/__/__/__/__/__/__/__/__/__/__/__/__/

Thèse

pour obtenir le grade de

Docteur de l'École des Mines de Paris
Spécialité : Informatique Temps Réel, Robotique et Automatique

présentée et soutenue publiquement par

Thi Viet Nga NGUYEN

le 19 novembre 2002

VÉRIFICATIONS EFFICACES DES APPLICATIONS SCIENTIFIQUES PAR ANALYSE STATIQUE ET INSTRUMENTATION DE CODE

EFFICIENT AND EFFECTIVE SOFTWARE VERIFICATIONS FOR SCIENTIFIC APPLICATIONS USING STATIC ANALYSIS AND CODE INSTRUMENTATION

Directeur de thèse : François IRIGOIN

Jury

M.	Nicolas HALBWACHS	Président, Rapporteur
M.	Paul FEAUTRIER	Rapporteur
Mme.	Corinne ANCOURT	Examinateur
M.	Guillaume GACON	Examinateur
M.	François IRIGOIN	Examinateur
M.	Jacques RAGUIDEAU	Examinateur

Résumé

La vérification des logiciels est essentielle pour les systèmes critiques. C'est une procédure lourde pouvant représenter jusqu'à 80% du coût de développement. Il existe plusieurs techniques de vérification qui sont actuellement utilisées, comme le test, la preuve de théorèmes, la vérification de modèles. Nous montrons dans cette thèse comment les analyses statiques de programmes peuvent être utilisées automatiquement pour assister la vérification par rapport à la norme d'un langage.

Quatre types de vérification sont étudiés. Le *redimensionnement de tableaux* résout le problème lié aux déclarations incomplètes de variables dans certains langages de programmation. Les *tests de débordement de tableaux*, intra- et inter-procéduraux, sont souhaitables pour n'importe quel programme, indépendamment du langage de programmation employé. L'optimisation de ces tests est nécessaire pour réduire le surcoût de temps d'exécution. L'*analyse des alias* détecte les violations des restrictions spécifiques dans la norme FORTRAN, qui est critique pour appliquer correctement la plupart des optimisations. L'utilisation des *variables non initialisées*, une des raisons connues des accidents de logiciels, est aussi détectée par notre analyse.

Grâce aux propriétés de programme fournies par l'analyse statique, la quantité de code nécessaire pour instrumenter l'application est réduite. Les tests dynamiques sont générés seulement quand nous ne savons pas comment prouver qu'ils sont inutiles. Les erreurs causant potentiellement l'échec du système sont détectées. L'information donnée par nos analyses est utile pour la mise au point, la maintenance et la réutilisation de programmes de grande taille. Comme la sémantique d'un programme bogué n'est pas définie, les interactions entre les différentes analyses et leur ordonnancement sont aussi traités.

Toutes ces vérifications sont implantées dans PIPS, notre compilateur de recherche dédié aux codes scientifiques. Des expériences qualitatives et quantitatives en ont montré la robustesse, ainsi que suggéré des adaptations possibles pour des applications industrielles contenant des centaines de milliers de lignes de code.

Mots-clés : vérification des logiciels, analyse statique de programmes, analyse interprocédurale, instrumentation du code, redimensionnement de tableaux, test de débordement de tableaux, analyse des alias, variable non initialisée.

Abstract

Software verification is essential for almost critical systems whose failure is unacceptable. It is a very expensive process, typically costing 80% of the total software development costs. Many verification techniques exist and are being actively used such as testing, theorem proving, model checking, etc. We show in this thesis how program analyses can be used automatically to assist software verification with respect to a language standard.

Four types of program verifications are studied. *Array resizing* solves the problem of incomplete variable declarations that prevent program analyses in some programming languages. *Array bound checking*, intra- or inter-procedurally, is desirable for any program, regardless of the programming language used, since bound violations are among the most common programming errors. *Alias analysis* deals with the violation on the FORTRAN alias restriction, which is critical for performing most optimizations correctly and therefore improving software performance. *Uninitialized variables* are known reasons of software system crashes, the thesis deals with this issue too.

With useful knowledge of the program properties provided by static analyses, the amount of additional code necessary to instrument an application is reduced to a minimum. Run-time checks are generated only when we do not know how to prove that they are useless. Dynamic errors that may cause potentially failure are caught. Information provided by our analyses is helpful for the debugging, maintenance and reuse of large scale programs. Since the semantics of a bugged program is undefined, the interactions among different analyses and a safe order of verifications are also studied.

All these verifications are implemented in PIPS, our research compiler dedicated to scientific code. Qualitative and quantitative experiments have shown its robustness, as well as possible adaptations for industrial applications up to hundreds of thousand lines of code.

Keywords: software verification, program analysis, static analysis, interprocedural analysis, code instrumentation, array resizing, array bound checking, alias analysis, uninitialized variable.

Remerciements

Je voudrais d'abord remercier François Irigoin, mon directeur de thèse, sans qui cette thèse n'aurait jamais existé. Je lui dois une profonde reconnaissance pour m'avoir initiée à la recherche dans les domaines de la compilation et de la vérification. Ses suggestions, ses critiques ainsi que les discussions constructives que nous avons eues ont contribué à l'aboutissement de ce travail.

Je remercie également le Professeur Robert Mahl, directeur du Centre de Recherche en Informatique de l'École Nationale Supérieure des Mines de Paris, pour m'avoir chaleureusement accueillie dans son laboratoire.

Je désire remercier plus particulièrement Corinne Ancourt, ma responsable de thèse, pour son aide inestimable et sa gentillesse. Je suis aussi très reconnaissante à Fabien Coelho qui m'a fourni des supports stratégiques pour résoudre certains problèmes. J'ai aussi eu de nombreuses discussions avec Pierre Jouvelot, qui m'a fait des remarques et posé des questions intéressantes sur mes travaux de recherche. Georges-André Silber a consacré du temps à la lecture de ma thèse avec des commentaires constructifs.

Je tiens à remercier tous les membres de mon jury de thèse, tout particulièrement le Professeur Paul Feautrier et le Professeur Nicolas Halbwachs qui m'ont fait le plaisir et l'honneur d'accepter de rapporter ma thèse. Le travail considérable de lecture qu'ils ont effectué, leurs questions et leurs commentaires pertinents m'ont aidée à améliorer ma thèse. Je voudrais aussi remercier Monsieur Guillaume Gacon et Monsieur Jacques Raguideau pour avoir accepté de faire partie de mon jury. Je leur adresse toute ma gratitude.

Mes remerciements vont également à l'ensemble des membres du Centre de Recherche en Informatique où cette thèse a été préparée dans une ambiance très agréable et dans les meilleures conditions.

Enfin, je voudrais tout particulièrement remercier mes parents, mon frère et mon mari à qui je dois un grand amour et un soutien gigantesque.

Acknowledgments

First and foremost, I am deeply indebted to François Irigoin, my supervisor, without whom this work would not have been possible. He has introduced me to the compilation and verification research domains. His suggestions, his rigorous remarks and the serious discussions that I have had with him have contributed to the achievements of this thesis. I am fortunate to have had the opportunity to work closely with him.

I would like to thank Professor Robert Mahl, director of the Centre de Recherche en Informatique de l'École Nationale Supérieure des Mines de Paris, for welcoming me in his department.

I am also especially grateful to Corinne Ancourt, my dissertation advisor, for her inestimable support and her kindness. I gratefully acknowledge Fabien Coelho who has given me insightful advice to solve some technical problems. I also had many fruitful discussions with Pierre Jouvelot whose remarks and questions helped me to have a more general view in my research domain. Georges-André Silber spent time to read a draft version of the thesis and gave me useful comments.

I would sincerely like to thank all the members of my thesis committee, particularly Professor Paul Feautrier and Professor Nicolas Halbwachs who have accepted to report my dissertation. Their challenging questions and comments have been very helpful for the improvements of the thesis. My sincerest thanks also to Guillaume Gacon and Jacques Raguideau who have accepted to be in my thesis committee.

Special thanks are due to all the members of the Centre de Recherche en Informatique with whom I shared three years of my life in a very warm atmosphere.

Last but not least, I would like to thank my parents, my brother and my husband for their unwavering support, encouragement, patience and love.

Contents

Chapter 1

INTRODUCTION

The English speaking reader will find the translation of this chapter in Section 1.2.

1.1 Français

Cette thèse est l'aboutissement d'une recherche sur la vérification des logiciels en utilisant l'analyse de programmes. La *vérification des logiciels* détermine si un système logiciel est conforme à sa spécification. Cette vérification est critique pour garantir la fiabilité, la robustesse et la qualité du système. Elle essaye de prouver la correction, souvent partielle, du programme, et réduire ainsi les risques qui peuvent causer l'échec du système. Un exemple bien connu est l'explosion d'Ariane 5, un projet de dix années coûtant sept milliards de dollars, qui a été causée par une erreur de logiciel dans l'ordinateur responsable de calculer le mouvement de la fusée. Beaucoup de méthodes de vérification sont actuellement utilisées, y compris l'analyse de programmes.

L'*analyse de programmes* peut être statique ou dynamique. L'*analyse statique* vise à découvrir statiquement des propriétés du comportement d'un programme au moment de l'exécution, sans l'exécuter. L'information obtenue lors de l'analyse statique peut être utilisée pour la compréhension, la mise au point, l'optimisation, la documentation et la vérification de programmes. Cette dernière est essentielle pour les systèmes critiques, car les catastrophes seront évitées si les bogues du logiciel ont été trouvés et corrigés avant l'exécution du programme. L'*analyse dynamique* collecte l'information pendant l'exécution du programme et l'utilise pour conclure sur les propriétés de celui-ci. Les différentes méthodes incluent la surveillance, l'analyse prédictive, l'instrumentation et la correction en ligne du comportement des programmes, etc.

La vérification des logiciels est une procédure coûteuse qui peut atteindre jusqu'à 50% de la dépense totale du développement, éventuellement 80% dans les systèmes informatiques critiques comme les réacteurs nucléaires, le contrôle de la navigation aérienne, les systèmes embarqués, la télécommunication. Il existe plusieurs techniques de vérification dont la définition, les fondations et les domaines d'application sont différents. *Le test* *de logiciel* détecte des erreurs d'un programme en l'exécutant sur des entrées choisies et en observant le résultat. La *preuve de théorèmes* peut être employée pour s'assurer de la correction du programme, mais cette méthode exige de l'expertise dans les disciplines mathématiques. La *vérification de modèles* est utilisée pour vérifier des propriétés comportementales d'un système parallèle à états finis. Chaque méthode est rendue aussi automatique que possible pour faire face aux systèmes réels de taille importante.

Plus récemment, il a été proposé de combiner ces différentes techniques, approche prometteuse dans le domaine de vérification. L'efficacité de la vérification de modèles et la puissance de la preuve de théorèmes peuvent être combinées pour une vérification complète d'un système complexe. L'analyse statique peut être associée au test, à la preuve de théorèmes et à la vérification de modèles pour simplifier le processus de vérification. Bien que les analyses statique et dynamique soient d'habitude appliquées séparément, il peut être avantageux de combiner ces deux approches. Puisqu'elles se complétent, une technique hybride automatisée devrait permettre de réduire les dépenses et augmenter la qualité des logiciels de grande taille.

Le premier but de cette thèse est de développer une technique de vérification hybride qui combine les analyses statiques et l'instrumentation du code. Diverses analyses interprocédurales sont développées. Ces analyses permettent de détecter des erreurs dans les procédures, ainsi que celles causées par des interactions entre des procédures différentes. Elles aboutissent à trois résultats possibles :

- Un programme est sans erreurs et la correction du programme est établie;
- Une erreur est détectée statiquement au moment de la compilation et l'information est fournie aux programmeurs pour tracer l'erreur;
- Parfois, l'information suffisante n'est pas disponible pour conclure sur la correction du programme. L'analyse dynamique, ou plus exactement l'instrumentation de code, est nécessaire pour détecter des erreurs arrivant au moment de l'exécution.

Des analyses statiques précises et efficaces peuvent être employées, comme un démonstrateur de la correction de programme et un détecteur d'erreurs. Grâce aux propriétés de programme fournies par ces analyses statiques, la quantité de code nécessaire pour instrumenter l'application est réduite autant que possible. Les tests dynamiques sont générés seulement quand nous ne savons pas comment prouver qu'ils sont inutiles. Les erreurs causant potentiellement l'échec du système sont détectées et leurs conséquences peuvent être réduites par le traitement des exceptions.

Le deuxième but de la thèse est l'application cette technique combinée sur les systèmes logiciels de grande taille. Il demande d'une implantation complète et ainsi que des expérimences qualitatives et quantitatives sur des applications réelles. Les analyses spécifiques étudiées dans cette thèse sont le *redimensionnement de tableaux*, le *test de débordement de tableaux*, l'*analyse des alias* et l'*analyse des variables non initialisées.* Elles sont développées par rapport à la norme du langage FORTRAN, un langage notamment utilisé pour les codes scientifiques. Le redimensionnement de tableaux est indispensable pour appliquer d'autres vérifications à tout le programme. L'analyse des alias garantit la correction d'autres analyses et transformations de programme en vérifiant si le programme respecte la spécification de la norme sur les alias. Le débordement de tableaux et l'utilisation de variables non initialisées sont des erreurs typiques de programmation pouvant produire des résultats incorrects, des violations mémoire, des comportements imprévisibles ou l'échec du programme. Leurs vérifications sont importantes car des simulations scientifiques complexes sont de plus en plus utilisées pour améliorer la performance de logiciels de contrôle critiques en temps réel.

Le problème principal rencontré par nos vérifications est que les analyses sont d'habitude utilisées pour manipuler des programmes corrects. La sémantique d'un programme incorrect n'est pas définie. Un programme bogué est correct et utile tant que les bogues ne sont pas encore révélés par des données particulières. En utilisant des analyses de programmes pour vérifier des programmes incorrects, la correction des méthodes de vérification doit être prouvée. Il est aussi évident qu'une vérification est d'habitude dépendante d'autres vérifications ou analyses de programme. Par exemple, l'analyse de variables non initialisées ne peut pas être correcte sans avoir traité les alias. Cette dernière analyse dépend de la vérification des accès aux éléments de tableaux, parce que le débordement d'un tableau peut créer d'autres alias. Pour obtenir une vérification complète d'un programme entier, nous devrons tenir compte de l'ordre d'application des différentes instrumentations et analyses de programme. Le troisième objectif de la thèse est de prouver la correction de chaque vérification de programme elle-même et d'étudier les interactions entre les différentes vérifications. À notre connaissance, ce dernier aspect n'a pas été traité par d'autres recherches.

Cette thèse est organisée de la façon suivante.

Le Chapitre 2 décrit certaines méthodes bien connues de vérification de logiciels, à savoir le test de logiciels, la preuve de théorèmes, la vérification de modèles, les analyses statique et dynamique.

Le Chapitre 3 présente l'analyse de programme et ses deux approches principales : l'analyse de flot de programme et l'interprétation abstraite. Les fondations mathématiques de ces deux approches ainsi qu'une comparaison entre elles sont présentées dans ce chapitre.

Le Chapitre 4 décrit quelques analyses préexistantes de programme dans notre compilateur de recherche PIPS, dédié à FORTRAN. Ces analyses sont utilisées pour les vérifications introduites dans les chapitres suivants. Il s'agit des analyses sémantiques fondamentales comme les transformeurs, les préconditions et les analyses avancées des accès aux éléments de tableaux, sous forme de régions convexes de tableau.

Le Chapitre 5 aborde le redimensionnement de tableaux, une transformation importante du programme qui est indispensable pour la plupart des vérifications sur tableaux comme le test de débordement, les alias entre tableaux, la détection des éléments non initialisés de tableaux, etc.

Le Chapitre 6 examine différentes méthodes de vérification des accès aux tableaux. En utilisant les analyses disponibles, deux vérifications intraprocédurales sont mises en oeuvre : l'élimination des tests redondants et l'insertion des tests inévitables. Une nouvelle vérification interprocédurale sur l'inconsistance entre la vraie taille du tableau et celle de tableau passé en paramètre est aussi étudiée et implantée.

Le Chapitre 7 illustre l'analyse des alias, qui fournit des informations sur les alias de variables dans le programme. Elle vérifie si un programme donné viole la norme FORTRAN concernant la modification d'un paramètre formel ou d'une variable commune qui est aliasé avec un autre paramètre formel. L'impact des alias sur les transformations et optimisations de programme est aussi étudié dans ce chapitre.

Le Chapitre 8 analyse le problème de l'utilisation d'une variable avant son initialisation. En utilisant des régions de tableaux, les utilisations des variables non définies qui peuvent mener à des résultats incorrects, des violations mémoire et des comportements imprévisibles de programme sont détectées avec une instrumentation minimale de code.

Le Chapitre 9 donne une synthèse des relations entre les diverses vérifications de programme et propose un ordonnancement des applications qui assure la correction d'une vérification complète.

Le Chapitre 10 résume les contributions principales de la thèse et signale les secteurs principaux où des travaux futurs sont possible pour d'autres langages de programmation.

La présentation de la littérature du domaine n'est pas inclue dans un chapitre séparé, mais est abordée dans les différents chapitres, selon le problème concerné.

1.2 English

This thesis is a detailed study of software verification by using program analysis. *Software verification* determines whether a software system meets its specification. This verification is critical to guarantee the reliability, robustness, and quality of software systems. It tries to prove the program correctness, usually partial, or to reduce risks that can cause the system to fail. A well known example of system failure is the explosion of Ariane 5, a 10 year project costing $7 billions, that was caused by a software error in the computer responsible for calculating the rocket's movement. Nowadays, many software verification methods exist and are being actively researched, including program analysis.

Program analysis consists of both static and dynamic program analyses. *Static analysis* aims at statically discovering properties of the run-time behavior of a program without actually running it. Information obtained from static analysis can be used to help program comprehension, program debugging, compiler optimization, program documentation and to assist program verification. The latter is essential for critical systems since catastrophes will not be produced if software bugs have been found and fixed before executing the program. *Dynamic analysis* gathers information during program execution and uses it to conclude properties about the program. The different dynamic methods include program monitoring, program guidance, program instrumentation, program behavior correction on-the-fly, predictive analysis, etc.

Software verification is an expensive process, typically costing upwards of 50% of the total software development costs, even 80% in some safety-critical systems such as nuclear reactors, air traffic control systems, embedded systems, telecommunication. There are many other software verification techniques which, by their definition, foundations and application domains are different. *Software testing* detects program errors by executing the program on selected inputs and observing the resulting outputs. *Theorem proving* can be used for reasoning about program correctness and the method requires expertise in mathematical disciplines. *Model checking* is used to verify behavioral properties of a finite state concurrent system. Each method is developed as automatically as possible to cope with realistic practical software systems.

More recently, we have seen the emergence of different techniques which has a very exciting future in the field of software verification. The effectiveness of model checkers and the power of theorem provers can be combined for a complete verification of a complex system. Static analysis techniques can support software testing, theorem proving and model checking to simplify the verification process. Although static and dynamic analyses have usually been applied separately, possible advantages can be achieved in combining these approaches. Since they complement each other, an automated hybrid technique has the potential to reduce costs and to increase the quality of large scale software systems.

The first goal of this thesis is to develop a hybrid verification technique combining static analyses and code instrumentation. Different powerful interprocedural static analyses are developed. These analyses allow checking for errors in a program unit and errors caused by interactions among different program units. There are three possible results for these analyses:

- A program is verified to be free of errors and the program correctness is established.
- An error is detected statically at compilation time and debugging information is provided to programmers to locate the error.
- There are some places where sufficient information is not available to conclude program correctness; dynamic analysis, or more exactly code instrumentation, is needed to emit errors occurring at execution time.

So precise and efficient static analyses can be used as both a program correctness prover and an error checker. With useful knowledge of the program properties provided by these static analyses, the amount of additional code necessary to instrument the application is reduced to a minimum. Run-time checks are generated only when we do not know how to prove that they are useless. Dynamic errors that may cause potentially failure are caught and their consequences can be reduced by exception handling.

The second goal of the thesis is the application of the combined technique on large scale systems. This requires a complete implementation as well as qualitative and quantitative experiments on real applications. Specific analyses studied in this thesis are *array resizing*, *array bound checking*, *alias analysis* and *uninitialized variables analysis*. They are developed with respect to FORTRAN standard, the language that is mainly used by scientific programmers. Array resizing is mandatory to apply other verifications on the whole program. Alias analysis guarantees the correctness of other program analyses and transformations by verifying whether the program respects the standard specification about aliasing. Exceeding the bound of an array and using uninitialized variables are typical semantic programming errors that may produce incorrect results, memory violations, unpredictable behavior and program failure. Their verifications are important because more and more complex scientific simulations are used on line to improve the performance of real-time critical software controllers.

The main problem encountered by our verifications is that program analysis is usually used to handle correct programs. The semantics of incorrect programs is undefined. A bugged program is correct and useful as long as the bugs are not activated by its inputs. When using program analysis to verify incorrect programs, the correctness of the verification methods must be proved. It is evident that one verification is usually related to other program verifications or analyses. For instance, used-before-set analysis cannot be correct without having treated aliasing. The latter depends on array bound checking because exceeding array bounds can cause other aliases. To obtain a complete verification of a whole program, we must take into account the order of applying different program instrumentation and analyses. The third objective of the thesis is to prove the correctness of each program verification itself and to study the interactions among different program verifications. To the best of our knowledge, this problem has not been treated by other research.

A brief overview of the content of the following chapters is given below.

Chapter 2 outlines some well-known software verification methods, namely software testing, theorem proving, model checking, static and dynamic analyses.

Chapter 3 introduces program analysis and its two main approaches: program flow analysis and abstract interpretation. The basic mathematical concepts of program analysis as well as a comparison between the two approaches are represented.

Chapter 4 describes some preexisting program analyses in our research compiler PIPS, which are used for the verifications in the next chapters. They are fundamental semantics analyses including transformers, preconditions and advanced array element accesses analyses called array regions.

Chapter 5 discusses array resizing, an important program transformation that is necessary for most program verifications on array variables such as array alias, array bound checking, uninitialized array element, etc.

Chapter 6 examines the different array bound checking methods. Two intraprocedural checking techniques are implemented by using available analyses: the elimination of redundant tests and the insertion of unavoidable tests. A novel interprocedural checking dealing with the actual and formal array sizes mismatch is also studied and experimented.

Chapter 7 illustrates the alias verification procedure, which provides information about the alias of variables in the program. It verifies if the program violates the FORTRAN standard on the modification of a formal parameter or a common variable that is aliased with another formal parameter. The impact of alias on program optimizing transformations is also studied in this chapter.

Chapter 8 analyses the uninitialized variable problem. By using array regions, uses of undefined variables that can lead to incorrect results, memory violations and unpredictable program behavior are detected with a minimum code instrumentation.

Chapter 9 gives a synthesis of the relationship among the above program verifications and figures out the order of applications that ensures the correctness of an overall verification.

Chapter 10 summaries the main contributions of the thesis and points out the main areas where future work is needed for more general purpose programming languages.

Because of the different verification problems studied in this thesis, the presentation of related work is not included in a separate chapter, but is spread out among different chapters, according to the problem concerned.

Chapter 2

SOFTWARE VERIFICATION METHODS

From the beginning of software engineering, many different software verification and validation methods have been developed such as code inspection, code reviewing, simulation, testing, correctness proofs, model checking, static analysis and dynamic analysis, etc. These techniques have largely been applied as isolated technologies in different domains although there are similarities among them and sometimes, they complement each other.

In this chapter, we describe briefly five main approaches for software verification and validation: software testing, theorem proving, model checking, static analysis and dynamic analysis.

2.1 Software Testing

Testing is one of the most widely used methods of validating software. As defined by Whittaker in [Whi00], software testing is the process of executing a software system to determine whether it matches its specification and executes in its intended environment. This process is normally effective in the very early stages of debugging, when the program is still infected with multiple bugs. Testers execute the program on selected inputs and check that the resulting outputs match the specified values. Additional information may also be reported to help evaluate how thoroughly the software has been tested.

There are a lot of existing testing methods that can be classified by different criteria, Beizer [Bei90]. Generating test cases can be based on program control flow (*path testing*), transaction flows (*transaction flow testing*), data flow anomalies (*data flow testing*), specification or implementation information on input domain (*domain testing*), syntax (*syntax testing*), decision tables (*logic based testing*), state graphs (*transition testing*), etc. The most common classification is the *structural* (or *white-box*) techniques versus the *functional* (or *black-box*) techniques. Structural testing requires that inputs be based solely on the structure of the source code or its data structures. Functional testing requires the selection of test scenarios based on attributes of the specification or operational environment, without regard to source code structure.

Since testing is an effectively practical method of discovering software bugs, its primary goal is the development of systematic means of selecting test sets that are small enough to be feasible in practice, yet which cover a wide variety of situations the software could potentially face. However, the objective of software testing is not to prove the correctness of programs. An exhaustive testing is neither theoretically nor practically possible. For structural testing, one should design enough tests to ensure that every path through the routine is exercised at least once, but the number of possible execution paths may be infinite or too large to make it practical. Furthermore, all possible input combinations for functional testing may go beyond the memory limitation or processing capacity. So fatal bugs can be hidden in unexploited paths or go undetected by untested input. Another limit of testing is its non-reproducible results. Finally, we cannot be certain that the verification system itself is correct, a never-ending sequences of correctness proof.

2.2 Theorem Proving

Theorem proving, or so called *deductive verification*, uses mathematical foundations to prove that programs meet their specifications. For example, as explained in [Gor88], Floyd-Hoare logic can be used to establish the correctness of programs written in conventional imperative programming languages like FORTRAN, ALGO, PASCAL, ADA, etc. The λ-calculus and the theory of combinators can be used for formal correctness analysis of compilers for functional programming languages such as LISP, ML, MIRANDA, etc. The importance of theorem proving is widely recognized by computer scientists. The main focus in its earlier research is on guaranteeing the correctness of critical systems. This is achieved by using the expertise of a mathematician or a logician. Initially, such proofs were established entirely by hand. This process can be partially automated by using a software tool called *theorem prover*. We present briefly a representative deductive verification process to prove the correctness of programs by using Floyd-Hoare logic, studied by Gordon [Gor88].

2.2.1 Partial and Total Correctness Specifications

Let C be a program from the considered programming language, P and Q conditions on program variables.

Definition 2.2.1 The expression $\{P\}C\{Q\}$ is called a *partial correctness specification* and is true if whenever C is executed in a state satisfying P and if the execution of C terminates, then the state in which C's execution terminates satisfies Q; P is called *precondition* and Q *postcondition*.

Definition 2.2.2 $\{P\}C\{Q\}$ is a *total correctness specification* and is true if and only if the two following conditions are satisfied:

- Whenever C is executed in a state satisfying P, then the execution of C terminates.
- After termination Q holds.

The relationship between partial and total correctness can be informally expressed by the equation:

$$\text{Total correctness} = \text{Termination} + \text{Partial correctness}$$

It is well known that decision problems connected to programs such as termination are algorithmically unsolvable. So theorem proving, usually heuristic, often limits itself to partial correctness proofs.

2.2.2 Program Verification Process

A proof in Floyd-Hoare logic is a sequence of lines, each of which is either an *axiom* of the logic or follows from earlier lines by a *rule of inference* of the logic. An imperative programming language can be defined semantically by the axioms and rules of the logic such as assignment axiom, precondition strengthening, postcondition weakening, sequencing rule, conditional rule, `WHILE` rule, `FOR` rule, etc. Then formal proofs of partial correctness can be established by using these axioms and rules.

The verification process takes as input a partial correctness specification annotated with mathematical assertions describing relationships among variables. These assertions are passed to a *theorem prover* program which attempts to prove them automatically. When the theorem prover finishes its task, the final result is either completed, which means that the specification has been proved, or a list of simplified assertions that the prover could not prove. An expert advice will then be needed.

The aim of theorem proving research is to build systems that reduce the role of the time-consuming and expensive human expert to a minimum, by increasing the power of the theorem prover. However, not all mathematical tasks are solved by an algorithm. In general, correct termination of programs cannot be verified automatically, so most proof systems cannot be completely automated and expert intervention is required.

2.3 Model Checking

Model checking is a method of verifying a desired behavioral property of a reactive system through exhaustive enumeration of all the states reachable by the system and the behavior that goes through them. This automated process receives a model of the system and a temporal logic formula, and determines whether the formula holds in the model. If the formula does not hold, a counter-example, or a sequence of steps leading to the problem, is returned. Because of automation and counter-example generation facility, model checkers have become popular debugging tools and have been used in reactive systems such as hardware and circuit designs, communication protocols, telecommunication switches and even user interfaces. The application domain of model checking is usually finite-state concurrent system. As described by Clarke et al. in [CGP99], the process of model checking consists of the following tasks:

2.3.1 Modeling

This phase converts a system into a formal model accepted by a model checking tool. Abstraction may be used to eliminate irrelevant or unimportant details with respect to the specified properties that we check for the correctness.

2.3.2 Specification

The properties that the system must satisfy are stated by this process. The specification often uses a *temporal logic* formalism, Manna and Pnueli [MP89], which can assert how the behavior of the system evolves over time. This formalism is used for describing sequences of transitions among states in a reactive system. A temporal logic formula might specify that *eventually* some selected state is reached, or that an error state is *never* entered, by using special *temporal operators*. However, a problem in property specification is the *completeness*. It is impossible to determine whether a given specification covers all the properties that the system should satisfy.

2.3.3 Verification

Given a model with a set of states that represents a finite-state concurrent system and a temporal formula describing the specification, the model checking problem is to find the set of all states that satisfy this formula. The verification is done automatically by a model checker but the analysis of the verification results often involves human assistance. In case of negative results, a counter example of the checked property is used to help the designer to find out where the error occurred. However, an error trace can also result from incorrect modeling of the system or from an incorrect specification. Another possibility is that the verification fails to terminate normally, due to the size of the model, which is too large with respect to the memory capacity. In either case, the modeling or the specification or the verification phase must be adjusted and then the process be repeated.

The major problem with model checking is the state explosion problem. The size of the systems under analysis grows exponentially with the number of variables in the system. This makes model checking impractical for all but small systems. Most software systems are so large to analyze directly that the application of model checking to software has many challenges. Some existing advanced techniques can deal with the state explosion problem using symbolic representations based on ordered binary decision diagrams, partial order reduction, modular structure with the assume-guarantee reasoning strategy, abstraction, symmetry, induction.

Another disadvantage of model checking is the restriction to finite state systems. One possible solution is the integration with the theorem proving method, which has so far been the only viable alternative.

2.4 Dynamic Analysis

Software verification using dynamic analysis consists of methods applied during the execution of programs. They are lightweight methods in comparison to methods proving program correctness before their execution, such as model checking and theorem proving. Dynamic program analysis can be a predictive analysis that forecasts, from one execution trace, possible errors in other traces. It also consists of algorithms for detecting multi-thread errors in concurrent and distributed systems, such as deadlocks and data races.

Program instrumentation, at the source or object code level, aims at emitting events during execution to an observer. Tests are generated to discover program errors at run-time. Many compilers offer users run-time checking options to detect errors such as out-of-bound array accesses, undefined variable references, stack overflows, memory leaks, etc. These errors may cause potentially failure and they can be very difficult and time-consuming to locate. The earlier they are caught in the software development cycle, the more efficient and effective the cost to correct them. There are different approaches to instrumenting the code: before, during and after compilation [Sha00]. Instrumenting the source code directly prior to compilation can be effective in identifying errors, but requires the additional step of instrumenting the code and may not work with a particular compiler. Post-compilation instrumentation, object or binary code insertion, is independent of the compiler. However, in this approach, every operation must be instrumented, resulting in excessive code instrumentation that may cause the program to become too large to fit in a memory constrained environment. Furthermore, this approach does not offer a high degree of control over which modules should be checked or which types of errors should be identified, which would also help the developer to reduce the overhead.

In general, code instrumentation is costly. As summarized in [Ste92], adding run-time checking to C can take 40% longer time to compile, 10 times slower to run and the code size is 3 times larger. Some specified techniques such as instrumentation sampling [AR01b] can reduce the overhead of code instrumentation. In addition, instrumenting the code at compilation time can offer many advantages in comparison to other approaches. The increased code size and run-time overhead in the form of code instrumentation can be reduced by using available information provided by static analysis. In fact, this method is a combination of static analysis and code instrumentation that is studied in the thesis. It gives the developer more control over which and how errors are tracked and reported.

2.5 Static Analysis

Contrary to dynamic analysis, static program analysis attempts to discover properties of a program without running it. It can be used to detect from simple errors such as syntax errors, type mismatches, undeclared identifiers to complicated errors like exceeding the bound of an array, failure to initialize variables, overflow or underflow, attempted division by zero, illegal mixing of data types and incorrect transfers of control, etc. There are a range of static analysis techniques with different tradeoffs between the required effort and analysis complexity. Standard compilers offer analyses for simple program checking at the low effort. Lightweight static analysis is used to detect vulnerabilities in [Wag00, EL02]. The more efficient and precise the static analysis is, the less difficult the verification task is, without sacrificing soundness and completeness.

Since static analysis is included in the main studies of this thesis, we shall leave its description to the following chapters. And it is also noted that, traditionally, the terminology of program analysis often refers to static analysis. In the continuation of the thesis, we use program analysis for static analysis, dynamic analysis is specified each time it is referenced to.

2.6 Conclusion

All the above methods are very useful for software verification, in order to reduce risks existing in computers and related systems and to gain the confidence of users (Neumann [Neu01, Neu02], Frankl and Weyuker [FW00]). As mentioned, the combination of verification techniques is currently studied in many research. For example, Kerboeuf et al. [KNT00] combine SIGNAL, a model checking tool, with COQ, a theorem prover, to verify a steam-boiler system. Static analysis is used by Brew and Johnson in [BJ01], Souter et al. in [SWSP01] to assist program testing. The complementary of the formal verification methods and automatic testing is used to validate synchronous reactive systems by Halbwachs and Raymond in [HR99]. The emergence of model checking and testing can also be found in [Hol00, SP01, LMW01], etc.

However, the application domain of each method is different. One disadvantage of software testing and formal methods (theorem proving and model checking) is large scale applications. The efficiency of software testing decreases dramatically and the cost greatly increases with the size of programs. Formal methods seem promising, but they are not applicable to industrial scale applications. But this is the advantage of static analysis since it is completely automatic; no assertion or user interaction is needed. In addition, static analysis can be integrated in compilers to detect program bugs at the very early stage of development. In this thesis, the power of static analysis is used to assist dynamic checking. We develop a program analysis based approach that is proved to be an adequate answer to software verification.

Chapter 3

PROGRAM ANALYSIS

Program analysis offers static techniques for predicting safe and computable approximations of the set of behavioral properties arising dynamically at run-time when executing a program. Information obtained from program analysis can be used to support compiler optimization, program comprehension, debugging, verification and documentation. Several approaches to program analysis exist such as program flow analysis, denotation- or logical formulations-based approaches, abstract interpretation, but important similarities and connections among them have been found. In this thesis, we are interested in program flow analysis, the most classical and widely used program analysis approach, and in abstract interpretation, a methodological approach for program analysis correctness.

This chapter summarizes well-known concepts in program analysis that are used in the remaining of the thesis. It begins with some basic mathematical concepts, namely, lattices, functions and fixed points (Section 3.1). The program control and data flow analyses as well as interprocedural analysis are discussed in the second section (Section 3.2). The semantic foundations of programs with the abstract interpretation methodology are described in the third section (Section 3.3). This approach can be integrated with program flow analysis since there are profound similarities between them. A comparison between the two approaches is given in the last section (Section 3.4).

3.1 Basic Concepts

This section recalls some definitions and theorems on lattice theory, based on Birkhoff [Bir67] and the approximation of fixed points, based on Nielson et al. [NNH99].

3.1.1 Partially Ordered Set, Lattices and Chains

Partially Ordered Set

Definition 3.1.1 A *partially ordered set (poset)* is a set in which a binary relation $x \leqq y$ is defined, which satisfies for all x, y, z the following conditions:

1. *Reflexive*: for all x, $x \leqq x$
2. *Antisymmetry*: if $x \leqq y$ and $y \leqq x$, then $x = y$
3. *Transitivity*: if $x \leqq y$ and $y \leqq z$, then $x \leqq z$

Lattices

An *upper bound* of a subset X of a poset P is an element $a \in P$ such that $x \leqq a$ for all $x \in X$. a is called the *least upper bound* of X if a is an upper bound of X and if for any upper bound b of X, $a \leqq b$. The notions of *lower bounds* and the *greatest lower bound* of X are defined dually.

Definition 3.1.2 A *lattice* is a partially ordered set P any two of whose elements have a greatest lower bound denoted by $x \sqcap y$ (*meet* operator), and a least upper bound denoted by $x \sqcup y$ (*join* operator).

Definition 3.1.3 A lattice L is *complete* when each of its subsets X has a least upper bound and a greatest lower bound in L. A non-void complete lattice L has a greatest lower bound denoted by $\perp$ (*least element*), and a least upper bound denoted by $\top$ (*greatest element*).

Chains

Definition 3.1.4 A subset X of a partially ordered set $L(\leqq)$ is a *chain* if $\forall x_1, x_2 \in X : (x_1 \leqq x_2) \vee (x_2 \leqq x_1)$. A chain is a *finite chain* if it is a finite subset of L.

A sequence $x_0, x_1, \ldots, x_n, \ldots$ is an *ascending chain* if $n \leq m \Longrightarrow x_n \leqq x_m$. Similarly, a sequence $x_0, x_1, \ldots, x_n, \ldots$ is a *descending chain* if $n \leq m \Longrightarrow x_m \leqq x_n$.

We shall say that a sequence $x_0, x_1, \ldots, x_n, \ldots$ *eventually stabilizes* if and only if $\exists n_0 \in \mathbb{N} : \forall n \in \mathbb{N} : n \geq n_0 \Longrightarrow x_n = x_{n_0}$.

Definition 3.1.5 The partially ordered set L satisfies the *ascending chain condition* if and only if all ascending chains eventually stabilize. Similarly, it satisfies the *descending chain condition* if and only if all descending chains eventually stabilize.

3.1.2 Fixed Points Theorems

Definition 3.1.6 Let f be a function from a partially ordered set $L_1(\leqq_1)$ to a partially ordered set $L_2(\leqq_2)$.

- f is *surjective* if $\forall x_2 \in L_2 : \exists x_1 \in L_1 : f(x_1) = x_2$
- f is *injective* if $\forall x_1, x_1' \in L_1 : f(x_1) = f(x_1') \Longrightarrow x_1 = x_1'$
- f is *monotone* if $\forall x_1, x_1' \in L_1 : x_1 \leqq_1 x_1' \Longrightarrow f(x_1) \leqq_2 f(x_1')$

- f is an *isomorphism* if f is monotone and there exists an unique monotone function $f^{-1} : L_2 \longrightarrow L_1$ such that $f \circ f^{-1} = id_2$ and $f^{-1} \circ f = id_1$ (where id_i is the identity function over $L_i, i = 1, 2$)

Theorem 3.1.1 (Tarski) Let L be a complete lattice and f a monotone function on L into L. Then $f(a) = a$ for some $a \in L$.

An element $a \in L$ is called a *fixed point* of f if $f(a) = a$ and the set of fixed points of f is denoted by:

$$Fix(f) = \{a \in L : f(a) = a\}$$

Tarski's Fixed Point Theorem states that the set of fixed points of a monotone function f on a complete lattice $L(\leqq, \sqcup, \sqcap, \perp, \top)$ is a nonempty complete lattice with the partial ordering $\leqq$.

Definition 3.1.7 The function f is *reductive* at $x \in L$ if and only if $f(x) \leqq x$ and the set of elements upon which f is reductive is written:

$$Red(f) = \{x \in L : f(x) \leqq x\}$$

Similarly, the function f is *extensive* at $x \in L$ if and only if $x \leqq f(x)$ and the set of elements upon which f is extensive is written:

$$Ext(f) = \{x \in L : x \leqq f(x)\}$$

Tarski's Theorem also ensures that the *least fixed point* of f, symbolized as $lfp(f)$, is the greatest lower bound of the set $Red(f)$. Dually the *greatest fixed point* of f, symbolized as $gfp(f)$, is the least upper bound of the set $Ext(f)$.

$$lfp(f) = \sqcap Fix(f) = \sqcap Red(f)$$

$$gfp(f) = \sqcup Fix(f) = \sqcup Ext(f)$$

These propositions correspond to the two following principles:

- *Recursion Induction Principle*:

$$\forall x \in L, (f(x) \leqq x) \Longrightarrow (lfp(f) \leqq x))$$

- *Dual Recursion Induction Principle*:

$$\forall x \in L, (x \leqq f(x)) \Longrightarrow (x \leqq gfp(f)))$$

3.1.3 Approximation of Fixed Points

Without loss of generality, we consider the least fixed point of f since the case of the greatest fixed point is dual. To obtain the approximations of $lfp(f)$, the *widening* and *narrowing* operators are used to speed up the convergence.

Let $x_0, x_1, \ldots, x_n, \ldots$ be a sequence of elements of L and $\phi : L \times L \longrightarrow L$ a total function on L. A new sequence $x_0^\phi, x_1^\phi, \ldots, x_n^\phi, \ldots$ is defined by:

$$x_n^\phi = \begin{cases} x_n & \text{if } n = 0 \\ \phi(x_{n-1}^\phi, x_n) & \text{if } n > 0 \end{cases}$$

Widening Operator

Definition 3.1.8 An operator $\triangledown : L \times L \longrightarrow L$ is a *widening operator* if and only if :

- $(x_1 \leqq x_1 \triangledown x_2) \wedge (x_2 \leqq x_1 \triangledown x_2)$ for all $x_1, x_2 \in L$
- for any ascending chain $x_0, x_1, \ldots, x_n, \ldots$ the ascending chain $x_0^\triangledown, x_1^\triangledown, \ldots, x_n^\triangledown, \ldots$ eventually stabilizes.

So, given a monotone function $f : L \longrightarrow L$ on a complete lattice L and given a widening operator $\triangledown$ on L, we can compute the sequence $f_\triangledown^0, f_\triangledown^1, \ldots, f_\triangledown^n, \ldots$, which is an ascending chain and eventually stabilizes, defined by:

$$f_\triangledown^n = \begin{cases} \bot & \text{if } n = 0 \\ f_\triangledown^{n-1} & \text{if } n > 0 \wedge f(f_\triangledown^{n-1}) \leqq f_\triangledown^{n-1} \\ f_\triangledown^{n-1} \triangledown f(f_\triangledown^{n-1}) & \text{otherwise} \end{cases}$$

So there exists $m \in \mathbb{N}$ such that $\forall n \in \mathbb{N} : n \geq m \Longrightarrow f_\triangledown^n = f_\triangledown^m$, and we have $f_\triangledown^{m+1} = f_\triangledown^m$, or $f(f_\triangledown^m) \leqq f_\triangledown^m$. This means that f is reductive at $f_\triangledown^m$:

$$f_\triangledown^m \in Red(f) \text{ or } lfp(f) \leqq f_\triangledown^m$$

We can take $f_\triangledown^m$ as an upper approximation of $lfp(f)$.

Narrowing Operator

To improve the approximation, we next consider the iterative sequence $f_\triangledown^m, f(f_\triangledown^m), f^2(f_\triangledown^m), \ldots, f^n(f_\triangledown^m), \ldots$, a descending chain which does not necessarily eventually stabilize.

Definition 3.1.9 An operator $\triangle : L \times L \longrightarrow L$ is a *narrowing operator* if:

- $x_2 \leqq x_1 \Longrightarrow x_2 \leqq x_1 \triangle x_2 \leqq x_1$ for all $x_1, x_2 \in L$
- for any descending chain $x_0, x_1, \ldots, x_n, \ldots$ the sequence $x_0^\triangle, x_1^\triangle, \ldots, x_n^\triangle, \ldots$ eventually stabilizes.

We construct a descending chain that eventually stabilizes as follows:

$$[f]^n_\triangle \quad = \begin{cases} f^m_\triangledown & \text{if } n = 0 \\ [f]^{n-1}_\triangle \triangle f([f]^{n-1}_\triangle) & \text{if } n > 0 \end{cases}$$

So there exists m' where $[f]^{m'}_\triangle = [f]^{m'+1}_\triangle$, $[f]^{m'}_\triangle \in Red(f)$ and we can take $[f]^{m'}_\triangle$ as an upper bound of the least fixed point of f, which is more exact than $f^m_\triangledown$.

3.2 Program Flow Analysis

Various definitions of program flow analysis are given but in general, it is a method for describing what a program does to its data. For each possible control point in the program, flow analysis gives a finite description of the set of data states the program could be in when actual execution passes through that point. The resulting description may be either exact or approximate, in the sense that it does not provide all information, not in the sense that it may be incorrect. In this section, we describe the control flow and data flow analyses that apply to a procedure as well as an interprocedural analysis to deal with whole programs. These analyses are vital for doing correct optimization and verification. The material in the next subsections is based on the book *Advanced Compiler Design and Implementation* of Muchnick [Muc97].

3.2.1 Control-Flow Analysis

The purpose of control-flow analysis is to discover the hierarchical control flow within each procedure. The control structure of a subroutine in the program is represented by a rooted and directed graph $G = (N, E)$ where N is the set of nodes and $E \subseteq N \times N$ is the set of edges; G is called the *control flow graph* of the routine. Different control structures are defined as follows.

A *basic block* is a maximal sequence of instructions that can be entered only at the first instruction and exited only from the last one. Set N contains a unique *entry* node, a unique *exit* node and the other basic blocks. A *branch node* is a node that has more than one successor and a *join node* is a node that has more than one predecessor. We say that node d *dominates* node n if every possible execution path from the entry node to n includes d. Dually, node p *post-dominates* node n if every possible execution path from n to the exit node includes p. A *back edge* in a graph is an edge whose head dominates its tail.

Given a back edge (m, n), the *natural loop* of (m, n) is the subgraph consisting of the set of nodes containing n and all the nodes from which m can be reached in the flow graph without passing through n and the edge set connecting all the nodes in its node set. n is called the *loop header*. If two natural loops do not have the same header, they are either disjoint or one is nested within the other.

A *strongly connected component* (a *region*) of a directed graph is a subgraph $G_s = (N_s, E_s)$ such that every node in N_s is reachable from every other node by a path that includes only edges in E_s.

Definition 3.2.1 A control flow graph $G = (N, E)$ is *reducible* (*well-structured*) if and only if E can be partionned into disjoint sets E_F (the *forward* set) and E_B (the *backward* set) such that (N, E_F) forms a directed acyclic graph (DAG) in which each node can be reached from the entry node, and the edges in E_B are all back edges.

Figure 3.1 shows an example of a well-structured control flow graph.

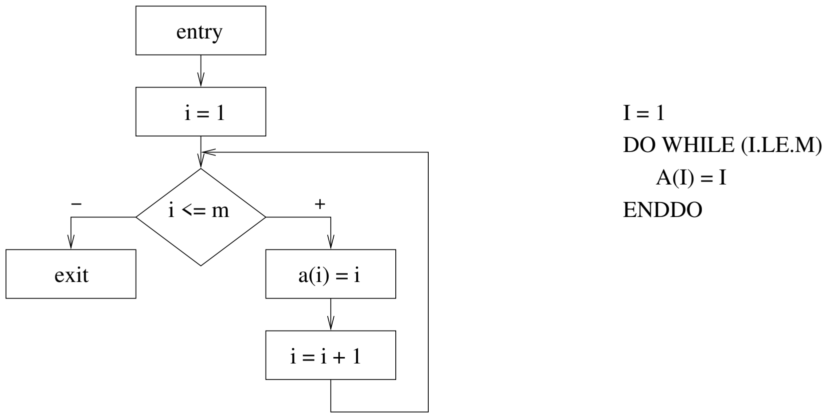

Figure 3.1: Example of a control flow graph

Certain control flow patterns such as `gotos` in loop bodies can make flow graphs *irreducible*. These irreducible flow graphs occur rarely in practice, but we must make sure that the control flow and data flow analyses can deal with them. Three practical approaches to dealing with irreducibility are: *i*) *node splitting* that transforms irreducible regions into reducible ones; *ii*) doing iterative data flow analysis on irreducible regions and plugging the results into the data flow equations for the rest of the flow graph; *iii*) performing an induced iteration on the lattice of monotone functions from the lattice to itself.

Two main approaches have been defined to perform control flow analysis. Both start by determining the basic blocks that make up the routine and by constructing its flow graph. The first approach uses dominators to discover loops, the only control structure to be concentrated on. This approach is sufficient for iterative data flow analyzers and is the least time-consuming to implement.

The second approach, interval analysis, includes a series of methods that analyze the overall structure of the routine and that decompose it into nested regions called intervals. They have recently been employed by optimizing compilers because they are faster and make it easier to update data flow information which has already been computed.

3.2.2 Data-Flow Analysis

Data flow analysis aims at providing global information about data manipulation within a procedure. In general, each data-flow analysis problem is attached to a complete lattice which is called *property space* and to a *monotone function space* defined on this lattice. We can cite here some important data-flow analyses such as *reaching definition, available expression, constant propagation, live variable* or *partial redundancy elimination*, etc. A data-flow analysis can be a *forward* problem if the direction of information flow is in the direction of program execution, a *backward* problem if the direction is opposite to the direction of execution and a *bidirectional* problem if it is in both directions. A *flow-insensitive* problem does not depend on the type of control flow encountered. However, control-flow analysis is needed to solve the *flow-sensitive* problems. The *may* and *must* data-flow information distinguish what may occur on some path through a flow graph and what must occur on all paths through it. In summary, data-flow analysis with its data-flow equations must provide a correct and conservative information that tells us if a transformation or an optimization can be applied or not.

Many approaches are used to solve data-flow problems such as strongly connected region method, iterative algorithm, interval analysis, structural analysis, slot-wise analysis, etc. As for control-flow analysis, there are two corresponding main approaches: *i*) iterative data-flow analysis that associates with loops and, *ii*) interval analysis. The interval analysis is significantly harder to implement than the iterative one, requiring node splitting, iteration, or solving of a data-flow problem on the lattice of monotone functions to handle irreducible regions. Both methods have their advantages and disadvantages. To give a representative description of data-flow analysis, we present in the next part the iterative data-flow approach.

Iterative Data-flow Analysis

This is the most frequently used method because the forward and backward analyses as well as bidirectional analyses are not so difficult to implement. Let us consider the forward and backward analyses with a given flow graph $G = (N, E)$. N contains an entry block and an exit block. For each node $B \in N$, we desire to compute $in(B), out(B) \in L$ where L is the underlying lattice; $in(B)$ represents the data-flow information on entry to B and $out(B)$ represents the data-flow information on exit from B. The data-flow equation for the forward problem is formulated as follows:

$$in(B) = \begin{cases} Init & \text{for } B = entry \\ \displaystyle\bigsqcup_{P \in Pred(B)} F_P(in(P)) & \text{otherwise} \end{cases}$$

where $Init$ specifies the initial value for the data-flow information on entry to the procedure, F_P is the flow function of the data-flow information corresponding to executing block P and $\bigsqcup$ models the effect of combining the data-flow information on the edges reaching a block.

Dually, we can formulate the data-flow equation for a backward problem as follows :

$$out(B) = \begin{cases} Init & \text{for } B = exit \\ \bigsqcup\limits_{P \in Succ(B)} F_P(out(P)) & \text{otherwise} \end{cases}$$

where $Init$ specifies the initial value for the data-flow information on exit from the procedure, F_P is the flow function of the data-flow information corresponding to executing block P in reverse and $\bigsqcup$ models the effect of combining the data-flow information on the edges leaving a block.

$\bigsqcup$ can be the join operator $\sqcup$ or the meet operator $\sqcap$ of the lattice. If we take the meet operator, we are able to detect properties satisfied by all paths of execution reaching the block (for forward analysis) or leaving the block (for backward analysis) and this is a *must* analysis. In case of the join operator, we can find properties satisfied by at least one execution path and we have a *may* analysis.

We now want to solve the above data-flow equations by beginning with the initial information and applying the flow functions along all possible paths in the flow graph and forming the result for each node. This desired solution is called *Meet Over all Paths* (MOP). Unfortunately, the MOP solution is sometimes uncomputable so we have to be satisfied in computing the *Maximal Fixed Point* solution (MFP). This is an iterative algorithm that computes the least solution to the data flow equations. The MFP solution safely approximates the MOP solution and, in case all flow functions are distributive, the two solutions are equal.

To get a maximum benefit from analyses, we seek to pose data-flow problems as aggressively as possible. Data-flow analysis on scalar variables is easier than analysis on variables such as arrays, pointers and records because treating these variables as scalar variables does not give much information. Advanced array and pointer analyses can provide more precise information, but at the expense of considerable additional computation. However, these analyses are important for aggressive optimization as well as efficient verifications. In the following chapters, we describe our advanced data-flow analyses on array variables. It is used for program verifications in Chapters 5, 7, 6 and 8.

3.2.3 Interprocedural Analysis

The program flow analysis techniques that have been considered in the previous sections are called *intraprocedural analyses* because they are applied within a single procedure, without regard to the calling context in which that procedure is used or the procedures it calls. *Interprocedural analysis* refers to gathering information about the entire program instead of a single procedure. It studies the interactions among procedures in the whole program, including control-flow and data-flow.

Interprocedural Control-flow Analysis

Interprocedural control-flow of a program can be represented by a static *call graph* which is a multi-graph with multi-directed edges from one node to another, defined as follows.

Given a program P with procedures $p_1, p_2, \dots, p_n$, the call graph of P is the graph $G = (N, S, E, p_1)$ with the node set $N = \{p_1, p_2, \dots, p_n\}$, the set S of call site labels for different calls, the set $E \subseteq N \times S \times N$ of labeled edges, and the distinguished entry node for the main program $p_1 \in N$. For each e in E with $e = (p_i, s_k, p_j)$, s_k denotes a call site in p_i to p_j. Figure 3.2 contains a call graph with $N = \{A, B, C, D\}$, $S = \{2, 3, 4, 7, 8, 11\}$, $p_1 = A$ and $E = \{(A, 2, B), (A, 3, C), (A, 4, C), (B, 7, D), (B, 8, C), (C, 11, B)\}$.

```
1  PROGRAM A          6  SUBROUTINE B( )
2  CALL B( )          7  CALL D( )
3  CALL C( )          8  CALL C( )
4  CALL C( )          9  END
5  END

10 SUBROUTINE C( )    13 SUBROUTINE D( )
11 CALL B( )          14 END
12 END
```

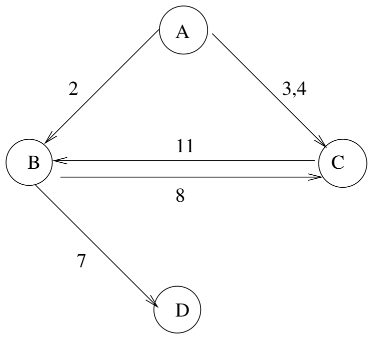

Figure 3.2: Example of a call graph

To traverse the call graph of non-recursive programs, the *invocation order* processes a procedure before all its callees and the *reverse invocation order* processes a procedure after its callees.

Except separate compilation and programs with procedure-valued variables or recursion, call graphs can be constructed efficiently. But they do not allow flow-sensitive interprocedural analyses because intraprocedural control flow of individual procedures is not taken into account. We can have more precise approaches that perform flow-sensitive analyses and reduce at the same time the burden of representing the internal control-flow of procedures such as the *program summary graph* [CK88], the *sparse data flow evaluation graph* [CCF91]. We can also improve the accuracy of analyses by detecting the *unrealizable paths* [RHS95] or taking into account path specific information [LR92].

Interprocedural Data-flow Analysis

Interprocedural data-flow analysis determines a conservative approximation of how a program manipulates data at the level of its call graph. Many useful interprocedural analyses exist, including alias analysis, side effect analysis and interprocedural constant propagation.

Interprocedural aliases are generally created by parameter passing and by accesses to global variables. Aliasing information is necessary to apply any transformation or optimization to the called procedure. Side effect analysis gives information about what variables may be used or modified when the procedure is invoked. Furthermore, knowing that some parameters have some constant values in a given call allows us to improve the intraprocedural data-flow information in the procedure, or to apply for example, procedure cloning.

May versus must information, flow-sensitive versus flow-insensitive information are also distinguished in interprocedural analysis. In addition, interprocedural analysis can be classified into *context-insensitive* or *context-sensitive* analysis. The context-insensitive approach cannot distinguish the different call sites of a procedure. Information about calling states is combined for all call sites and the resulting information is returned at all return points. In contrast, the context-sensitive approach considers interprocedurally *realizable paths* (Reps et al. [RHS95], Bodik et al. [BGS97], Ball and Rajamani [BR01]) by maintaining the relationship between procedure calls and procedure returns. Normally, flow-sensitive, context-sensitive and must information are expensive to compute, being generally NP-complete problems. In some cases we have to turn to flow-insensitive, context-insensitive or may information, in order to reduce the analysis cost. Interprocedural analyses improve the quality of intraprocedural ones by initializing data-flow information for procedure entries and exits, and by narrowing down the scope of possible effects of the called procedures.

3.3 Abstract Interpretation

The *semantics* of a program is a computation model describing all possible behaviors of a computer system executing this program. The *concrete semantics* of a syntactically correct program is a standard semantics that specifies exactly the effect of the program and the values it manipulates. In general, questions related to the concrete semantics of a non trivial program are undecidable, in the sense that no computer can always answer them in finite time. An *abstract semantics* is a non-standard semantics that gives a conservative approximation of the concrete semantics. The abstraction does not allow to answer all questions about the program semantics but all answers given by the abstract semantics are always correct with respect to the concrete semantics. *Abstract interpretation*, pioneered by the Cousots [CC77, Cou97, CC00], is a method of formalizing the approximation relation between the concrete semantics and the abstract semantics.

Each semantics is based on a semantic domain which is a partially ordered set, or a stronger set, a complete lattice. To represent the relation between the concrete and abstract semantic domains, we need the concept of Galois connection that is defined below.

3.3.1 Galois Connection

Let $L_1(\leqq_1, \sqcup_1, \sqcap_1, \perp_1, \top_1)$ and $L_2(\leqq_2, \sqcup_2, \sqcap_2, \perp_2, \top_2)$ be two complete lattices corresponding to the concrete and abstract semantic domains. To express the relationship between these two domains, we have an *abstraction function*

$$\alpha : L_1 \longrightarrow L_2$$

giving an abstract representation of the elements belonging to the concrete domain in the abstract domain, and a *concretization function*

$$\gamma : L_2 \longrightarrow L_1$$

that expresses the meaning of some abstract information in concrete terms.

Definition 3.3.1 $(L_1, \alpha, \gamma, L_2)$ is a *Galois connection* between the complete lattices L_1 and L_2 if and only if $\alpha : L_1 \longrightarrow L_2$ and $\gamma : L_2 \longrightarrow L_1$ are monotone functions that satisfy:

$$\lambda l_1.l_1 \leqq_1 \gamma \circ \alpha \text{ and } \alpha \circ \gamma \leqq_2 \lambda l_2.l_2$$

The partial orderings $\leqq_1$ and $\leqq_2$ formalize the loss of information. $l_1 \leqq_1 \gamma(\alpha(l_1))$ means that the concretization of the abstract representation of l_1 is a safe approximation of l_1. The above conditions express that we do not lose safety by going back and forth between the two domains although we may lose precision.

Lemma 3.3.1 If $(L_1, \alpha, \gamma, L_2)$ is a Galois connection then:

$$\forall l_1 \in L_1, \forall l_2 \in L_2 : \alpha(l_1) \leqq_2 l_2 \Longleftrightarrow l_1 \leqq_1 \gamma(l_2)$$

Definition 3.3.2 Let f be a function from a complete lattice L_1 to a complete lattice L_2.

- f is *completely additive* (or *completely join morphism*) if

$$\forall X \subseteq L_1 \;\; f(\sqcup_1 X) = \sqcup_2\{f(x) : x \in X\}$$

- f is *completely multiplicative* (or *completely meet morphism*) if

$$\forall X \subseteq L_1 \;\; f(\sqcap_1 X) = \sqcap_2\{f(x) : x \in X\}$$

Here are some important properties of a Galois connection:

Lemma 3.3.2 If $(L_1, \alpha, \gamma, L_2)$ is a Galois connection then:

- α uniquely determines γ by $\gamma(l_2) = \sqcup_1\{l_1 : \alpha(l_1) \leqq_2 l_2\}$ and γ uniquely determines α by $\alpha(l_1) = \sqcap_2\{l_2 : l_1 \leqq_1 \gamma(l_2)\}$
- α is completely additive and γ is completely multiplicative

In particular $\alpha(\bot_1) = \bot_2$ and $\gamma(\top_2) = \top_1$

This result shows that a Galois connection is fully determined by either one of the abstraction and concretization functions. The next property shows that it suffices to specify either a completely additive abstraction function or a completely multiplicative concretization function in order to obtain a Galois connection.

Lemma 3.3.3 If $\alpha : L_1 \longrightarrow L_2$ is completely additive then there exists $\gamma : L_2 \longrightarrow L_1$ such that $(L_1, \alpha, \gamma, L_2)$ is a Galois connection. Similarly, if $\gamma : L_2 \longrightarrow L_1$ is completely multiplicative then there exists $\alpha : L_1 \longrightarrow L_2$ such that $(L_1, \alpha, \gamma, L_2)$ is a Galois connection.

Lemma 3.3.4 If $(L_1, \alpha, \gamma, L_2)$ is a Galois connection then $\alpha \circ \gamma \circ \alpha = \alpha$ and $\gamma \circ \alpha \circ \gamma = \gamma$

This last property shows that we neither lose nor gain precision by iterating abstraction and concretization.

Definition 3.3.3 $(L_1, \alpha, \gamma, L_2)$ is a *Galois insertion* between the complete lattices L_1 and L_2 if and only if $\alpha : L_1 \longrightarrow L_2$ and $\gamma : L_2 \longrightarrow L_1$ are monotone functions that satisfy:

$$\lambda l_1.l_1 \leqq_1 \gamma \circ \alpha \text{ and } \alpha \circ \gamma = \lambda l_2.l_2$$

The concept of Galois insertion is to ensure that the abstract semantic domain cannot contain elements that do not describe elements of the concrete domain. We have the following lemma:

Lemma 3.3.5 For a Galois connection $(L_1, \alpha, \gamma, L_2)$ the following claims are equivalent:

- $(L_1, \alpha, \gamma, L_2)$ is a Galois insertion
- α is surjective
- γ is injective
- γ is an order-similarity: $\forall l_2, l_2' \in L_2 : \gamma(l_2) \leqq_1 \gamma(l_2') \Longleftrightarrow l_2 \leqq_2 l_2'$

$$\begin{tikzcd}
L_1 \arrow[r, "f_1"] \arrow[d, shift right, "\alpha"'] & L_1 \arrow[d, shift right, "\alpha"'] \\
L_2 \arrow[r, "f_2"'] \arrow[u, shift right, "\gamma"'] & L_2 \arrow[u, shift right, "\gamma"']
\end{tikzcd}$$

Figure 3.3: Commuting diagram of fixed points

If the programming language constructs involve iteration and recursion, the least fixed point of the monotone semantic function f_1 of the concrete semantics is abstracted by:

$\alpha(lfp(f_1)) = lfp(f_2)$ where f_2 is defined by the commuting diagram in Figure 3.3. We have : $f_2 = \alpha \circ f_1 \circ \gamma$, $\alpha \circ f_1 = f_2 \circ \alpha$ and $f_1 \circ \gamma = \gamma \circ f_2$.

3.3.2 Program Analysis Correctness

In this subsection, we describe abstract interpretation as a methodological approach to prove the correctness of program analyses with respect to the semantics of programs. The material is based on the book *Principles of Program Analysis* of Nielson et al. [NNH99].

Each program analysis is considered as an abstract semantics whose domain is called *property space*, which is a complete lattice. If the calculation on this property space is too costly or even uncomputable, we can replace it by another space. Rather than performing the analysis in a costly lattice, the idea is to perform it in a simpler lattice. We have to ensure that all these program analyses are correct with respect to the concrete semantics of the program. The *correctness relation* approach and the Galois connection are used to verify this correctness.

The semantics of a programming language specifies how a value v_1 is transformed to another v_2 by a program p:

$$p \vdash v_1 \rightarrow v_2$$

and a program analysis specifies how a program p transforms one property l_1 to another l_2:

$$p \vdash l_1 \rightarrow l_2$$

The set of values V usually refers to the set of program states and the set of properties L depends on the specific program analysis. If the program analysis is in the class of analyses where properties directly describe sets of values, such as constant propagation analysis and shape analysis, a correctness relation $R : V \times L \longrightarrow \{true, false\}$ where vRl means that the value v is described by the property l can be established. R is preserved under computation if the following condition is verified: if the relation holds between the initial value and the initial property then it also holds between the final value and the final property.

$$(v_1 R l_1) \wedge (p \vdash v_1 \rightarrow v_2) \wedge (p \vdash l_1 \rightarrow l_2) \Longrightarrow v_2 R l_2$$

The following relationship is imposed between R and L:

$$(vRl_1) \wedge (l_1 \leqq l_2) \Longrightarrow vRl_2 \tag{3.1}$$

which means that the smaller the property is with respect to the partial ordering, the more precise it is; and

$$(\forall l \in L' \subseteq L : vRl) \Longrightarrow vR(\sqcap L') \tag{3.2}$$

which means that there is always a best property for describing a value.

If the set of properties L is replaced by a simpler lattice M and there exists a Galois connection (L, α, γ, M) between L and M, the correctness relation still holds between V and M. It is established as:

$$\begin{array}{rcl} S : V \times M & \longrightarrow & \{true, false\} \\ vSm & \Longleftrightarrow & vR(\gamma(m)) \end{array}$$

We prove S is a correctness relation fulfilling conditions corresponding to Equations 3.1 and 3.2. We have:

$$\begin{array}{rll} (vSm_1) \wedge m_1 \leqq m_2 & \Longrightarrow (vR(\gamma(m_1))) \wedge \gamma(m_1) \leqq \gamma(m_2) & (\gamma \text{ is monotone}) \\ & \Longrightarrow vR(\gamma(m_2)) & (R \text{ satisfies 3.1}) \\ & \Longrightarrow vSm_2 & \end{array}$$

and:

$$\begin{array}{rll} (\forall m \in M' : vSm) & \Longrightarrow (\forall m \in M' : vR(\gamma(m))) & \\ & \Longrightarrow vR(\sqcap\{\gamma(m) : m \in M'\}) & (R \text{ satisfies 3.2}) \\ & \Longrightarrow vR(\gamma(\sqcap M')) & (\gamma \text{ is completely multicative}) \\ & \Longrightarrow vS(\sqcap M') & \end{array}$$

Hence S defines a correctness relation between V and M.

A program analysis can be developed in several steps of refinement, by using different combinations of Galois connections such as sequential composition, parallel combinations, etc. Galois connections are indeed useful for transforming computations into more approximate computations that have better time, space or termination behavior and whose correctness properties are still preserved.

3.4 Conclusion

Program analysis aims at statically discovering the run-time behavioral properties of a program without actually running it. There are numerous approaches to program analysis. Program flow analysis considers a program as a graph whose nodes are basic blocks and whose edges describe how control might pass from one block to another. In interprocedural analysis, the node set is the set of procedures and the edge set represents procedure calls. Each program analysis is formalized by a set of data-flow equations. Abstract interpretation is a general theory for calculating and combining analyses rather than specifying them. The abstraction of a semantics provides an approximation which is less precise but computable. In general, this abstraction is the fixed point solution of an equation system that can be solved iteratively by using the widening and narrowing operators to speed up convergence.

There is a large amount of commonality among these two approaches. For instance, in order to remain computable, both program flow analysis and abstract interpretation can only give approximate answers. These answers do not provide precise information but may give useful information. In general, the program analysis produces a possibly larger set of possibilities than what will ever happen during execution of the program. In addition, all program analyses should be semantics based, which means that information obtained from the analysis can be proved to be correct with respect to the semantics of the programming language.

In fact, abstract interpretation and program flow analysis are related by the principle of lattice duality. In abstract interpretation, the greatest element is trivially safe and conveys no information whereas in program flow analysis it is the least element that has this role. Similarly, in abstract interpretation we are interested in least fixed points whereas in program flow analysis we are interested in greatest fixed points. A more detailed comparison can be found in [NNH99].

In the following chapters, our analyses are described with respect to the principles of program flow analysis and abstract interpretation.

Chapter 4

PREEXISTING ANALYSES

This chapter provides basic information about analyses developed by other researchers but used through Chapters 5, 6, 7 and 8. Summary presentations are given here to keep the dissertation self-contained. More detailed presentations are available in [TFI86, IJT91, Iri93, CI95, Cre96]. In addition, we provide the semantic meaning of procedure calls for these analyses, which is not presented in the above papers.

The chapter begins with a toy language definition (Section 4.1). It is used to formalize the analyses in the following sections. The convex polyhedron representation that is used for program analyses is described in the next section (Section 4.2). Transformers represent the effects of a statement on program variables (Section 4.3). Precondition analysis discovers the constraints holding among variables of the program (Section 4.4). Array regions collect information about the way array elements are used and defined by programs (Section 4.5). Each preexisting analysis is computed intra- and inter-procedurally, approximated by under- and over-approximations. The three analyses were first introduced by Triolet, Feautrier, Irigoin and Jouvelot, later refined by Irigoin, Coelho and Creusillet. Related work and conclusion are discussed in the last section (Section 4.6).

4.1 Programming Language

The source language used in this study is a subset of FORTRAN 77, an imperative programming language. In fact, the input of our research compiler PIPS and its analyses is almost the whole language, in order to deal with realistic programs. It also includes High Performance Fortran directives. However, for the sake of simplicity, we present here only a subset of the language.

4.1.1 Abstract Syntax

The structure of well-formed programs in this programming language is described in Figure 4.1.

$$
\begin{array}{lll}
\mathit{Procedure} & := & \mathit{head} : \mathit{Header}\ \mathit{decls} : \mathit{Declaration}^{*}\ \mathit{body} : \mathit{Statement}\ \mathrm{END} \\
\mathit{Header} & := & \mathrm{PROGRAM}\ \mathit{name} : \mathit{Name} \\
 & | & \mathrm{SUBROUTINE}\ \mathit{name} : \mathit{Name}\ [(\mathit{formals} : \mathit{Variable}^{*})] \\
\mathit{Declaration} & := & \mathrm{PARAMETER}\ (\mathit{var} : \mathit{Variable} = \mathit{cons} : \mathit{Constant}) \\
 & | & \mathrm{DIMENSION}\ \mathit{var} : \mathit{Variable}\ (\mathit{dims} : \mathit{Dimension}^{*}) \\
 & | & \mathrm{EQUIVALENCE}\ (\mathit{var}_1 : \mathit{Variable}, \mathit{var}_2 : \mathit{Variable}) \\
 & | & \mathrm{COMMON}\ [/\mathit{com} : \mathit{Name}/]\ \mathit{vars} : \mathit{Variable}^{*} \\
 & | & \mathit{type} : \mathit{Type}\ \mathit{var} : \mathit{Variable}\ [(\mathit{dims} : \mathit{Dimension}^{*})] \\
 & | & \mathrm{DATA}\ \mathit{var} : \mathit{Variable}\ /\mathit{cons} : \mathit{Constant}/ \\
\mathit{Type} & := & \mathrm{INTEGER} \mid \mathrm{REAL} \mid \mathrm{DOUBLE\ PRECISION} \mid \\
 & & \mathrm{COMPLEX} \mid \mathrm{LOGICAL} \mid \mathrm{CHARACTER} \\
\mathit{Dimension} & := & \mathit{lower} : \mathit{Expression} : \mathit{upper} : \mathit{Expression} \\
\mathit{Statement} & := & \mathrm{CONTINUE} \\
 & | & \mathrm{STOP} \\
 & | & \mathrm{READ}\ \mathit{ref} : \mathit{Reference} \\
 & | & \mathrm{WRITE}\ \mathit{exp} : \mathit{Expression} \\
 & | & \mathit{ref} : \mathit{Reference} = \mathit{exp} : \mathit{Expression} \\
 & | & \mathrm{IF}\ (\mathit{cond} : \mathit{Expression})\ \mathrm{THEN}\ \mathit{true} : \mathit{Statement} \\
 & & \mathrm{ELSE}\ \mathit{false} : \mathit{Statement}\ \mathrm{ENDIF} \\
 & | & \mathrm{DO}\ (\mathit{cond} : \mathit{Expression})\ \mathit{body} : \mathit{Statement}\ \mathrm{ENDDO} \\
 & | & \mathrm{CALL}\ \mathit{proc} : \mathit{Name}\ [(\mathit{actuals} : \mathit{Expression}^{*})] \\
 & | & \mathit{sequence} : \mathit{Statement}^{*} \\
\mathit{Expression} & := & \mathit{cons} : \mathit{Constant} \\
 & | & \mathit{ref} : \mathit{Reference} \\
 & | & (\mathit{exp} : \mathit{Expression}) \\
 & | & \mathit{unop} : \mathit{Unary\_op}\ \mathit{exp} : \mathit{Expression} \\
 & | & \mathit{exp}_1 : \mathit{Expression}\ \mathit{binop} : \mathit{Binary\_op}\ \mathit{exp}_2 : \mathit{Expression} \\
\mathit{Unary\_op} & := & - \mid .\mathrm{NOT}. \\
\mathit{Binary\_op} & := & + \mid - \mid * \mid / \mid ** \mid .\mathrm{LT}. \mid .\mathrm{LE}. \mid .\mathrm{EQ}. \mid .\mathrm{NE}. \mid .\mathrm{GT}. \mid .\mathrm{GE}. \mid \\
 & & .\mathrm{AND}. \mid .\mathrm{OR}. \mid .\mathrm{EQV}. \mid .\mathrm{NEQV}. \\
\mathit{Reference} & := & \mathit{var} : \mathit{Variable}\ [(\mathit{exp} : \mathit{Expression}^{*})]
\end{array}
$$

Figure 4.1: FORTRAN toy language syntax

The sign := denotes a definition, the sign `[ ]` an optional item, the sign | the choice production and the sign $*$ the list production. For example, an unary operator can be a negation ($-$) or a logical negation (.NOT.); $< statement >^*$ is a sequence of zero, one or more statements.

An executable program includes the main program, function and subroutine subprograms, characterized by their header. There is no recursion in the language. Specification statements are parameter, type, common, equivalence, dimension and data declarations. Executable statements consist of continue, stop, read, write, assignment, conditional statement, loop, call to external subroutine and sequence of statements. An expression is either a constant, a reference to a variable, an unary or a binary expression. A variable reference can be a scalar variable, an array variable or an array element. The syntactic domains of this language include:

$$\begin{array}{ll} \mathit{Variable} & : \text{set of variables} \\ \mathit{Expression} & : \text{set of expressions} \\ \mathit{Statement} & : \text{set of statements} \end{array}$$

4.1.2 Semantic Domains

The semantic property of a programming language characterizes the effect of programs and the values they manipulate. To define the semantics of our language, we have the following semantic domains:

$$\begin{array}{ll} \mathbb{B} & : \text{set of boolean values} \\ \mathbb{N} & : \text{set of positive integers} \\ \mathbb{Z} & : \text{set of integers (positive, zero or negative)} \\ \mathbb{Q} & : \text{set of rational numbers} \\ \mathbb{R} & : \text{set of reals} \\ \mathit{Constant} & : \text{set of constants} \\ \mathit{Value} & : \text{set of values} \\ \mathit{State} & : \text{set of states} \end{array}$$

The set of possible states of a program is defined as the set of functions from the set of variables to the set of values. In other words, the state of a program at any instant of its execution is a function that associates each variable to its value at that instant. For the sake of simplicity, this definition does not take into account the *environment* that gives the correspondence between variables and memory locations.

$$\mathit{State} = \mathit{Variable} \longrightarrow \mathit{Value}$$

Each semantic domain is provided with a partial ordering $\leqq$, a least element $\perp$ and a greatest element $\top$ when needed. For example, with the set of states *State*, we define a partial ordering $\leqq_{\mathit{State}}$, for all $\sigma_1, \sigma_2 \in \mathit{State}$,

$$\sigma_1 \leqq_{\mathit{State}} \sigma_2 \Longleftrightarrow \mathit{dom}(\sigma_1) \subseteq \mathit{dom}(\sigma_2) \wedge \forall v \in \mathit{dom}(\sigma_1) : \sigma_1(v) \leqq_{\mathit{Value}} \sigma_2(v)$$

and the least and greatest elements:

$$\begin{array}{llrcl}
\bot_{State} \in State, & \bot_{State}: & Variable & \longrightarrow & Value \\
& & dom(\bot_{State}) & \longmapsto & \bot_{Value} \\
\top_{State} \in State, & \top_{State}: & Variable & \longrightarrow & Value \\
& & var & \longmapsto & \top_{Value} \\
& & dom(\top_{State}) & = & Variable
\end{array}$$

where $\leqq_{Value}$ is the partial ordering and $\top_{Value}$ the greatest element of the *Value* domain. From now on, to represent the partial ordering, the least and greatest elements of a semantic domain, $\leqq$, $\bot$ and $\top$ are used without the domain name whenever there is no ambiguity.

Our ultimate goal is to make sure that no error in the program is going to violate the standard and go undetected, by ensuring that every access to an array element is within bound, and that a variable is always initialized before being used, etc. The value of a variable at a program point, the modification of this value by a statement are essential information that are provided by preexisting semantic analyses. Before going into details of these analyses, we describe the approximation used for abstracting sets and relationship over the abstract sets: convex polyhedron.

4.2 Set Approximation

In PIPS, convex polyhedra [Sch86] are used to represent the data-flow information for its analyses. This representation is a trade-off which reduces time and space complexity, while still preserving accuracy. The current subsection is devoted to definitions related to convex polyhedra and basic operations on them.

4.2.1 Convex Polyhedron

Definition 4.2.1 A *convex polyhedron* over $\mathbb{Z}$ is a set of points of $\mathbb{Z}^n$ defined by:

$$P = \{x \in \mathbb{Z}^n : A.x \leq b\}$$

where A is a matrix of $\mathbb{Z}^m \times \mathbb{Z}^n$ and b a vector of $\mathbb{Z}^m$.

Definition 4.2.2 Let P be a convex polyhedron. Its *generating system* is made of three sets: a set of vertices $V = \{v_1, \ldots, v_\sigma\}$, a set of rays $R = \{r_1, \ldots, r_\rho\}$, and a set of lines $L = \{l_1, \ldots, l_\delta\}$.

- A *vertex* of a polyhedron P is a point of P which is not a linear combination of other points of P:

$$\left\{\begin{array}{l} v = \sum_{i=1}^{p} \lambda_i x_i \\ \forall i = 1, \ldots, p : x_i \in P, \lambda_i \geq 0 \\ \sum_{i=1}^{p} \lambda_i = 1 \end{array}\right. \longrightarrow \forall i = 1, \ldots, p : \lambda_i = 0 \text{ or } v = x_i$$

- A *ray* of a polyhedron P is a vector $r \in \mathbb{R}^n$ such that there exists a half-line parallel to r, and entirely included in P:

$$\forall x \in P,\ \forall \mu \in \mathbb{R}^+,\ x + \mu r \in P$$

- A *line* of a polyhedron P is a vector $l \in \mathbb{R}^n$ such that both l and $-l$ are rays of P:

$$\forall x \in P \quad \forall \nu \in \mathbb{R} \quad x + \nu l \in P$$

P is then defined by:

$$\left\{ x : \left| \begin{array}{l} \exists \lambda_1, \ldots, \lambda_\sigma \in [0,1] : \sum_{i=1}^{\sigma} \lambda_i = 1 \\ \exists \mu_1, \ldots, \mu_\rho \in \mathbb{R}^+ \\ \exists \nu_1, \ldots, \nu_\delta \in \mathbb{R} \end{array} \right. \quad x = \sum_{i=1}^{\sigma} \lambda_i v_i + \sum_{i=1}^{\rho} \mu_i r_i + \sum_{i=1}^{\delta} \nu_i l_i \right\}$$

The representations of the system of inequalities and of the generating system are equivalent [CH78]. They are complementary because some operations on convex polyhedra, such as intersection, perform easier on systems of inequalities, while others are less complicated on generating systems.

4.2.2 Convex Polyhedron Operators

This subsection describes some important operations on convex polyhedra. They are implemented in the LINEAR C^3 library, with several important algorithms in linear programming such as Simplex, Fourier-Motzkin, etc. IRISA contributed an implementation of Chernikova algorithm [Wil97] and PRISM a C implementation of PIP (Parametric Integer Programming) [Fea88].

Feasibility / Satisfiability of Constraints

A polyhedron over $\mathbb{Z}^n$ is said to be feasible when it contains at least one element in $\mathbb{Z}^n$. On the contrary, if we detect a contradiction between its constraints, it is said to be infeasible. Otherwise, when we cannot prove that the system is feasible or infeasible, we say that the system is indeterminable. There exists a number of methods to perform this feasibility test [Sch86]. The Simplex algorithm only gives solutions in $\mathbb{Q}^n$. The Fourier-Motzkin algorithm checks the existence of a solution by successively eliminating the variables of the system. All these algorithms have an exponential complexity but they perform relatively well on practical cases.

Elimination of a Variable

The Fourier-Motzkin algorithm performs the elimination of a variable by computing the convex hull of the projections of all the rational points that belong to the initial polyhedron. The result is a convex polyhedron that may contain elements that are not the projection of an integer point. PIPS uses some necessary and sufficient conditions for testing the exactness of the elimination of a variable, based on the work of Ancourt and Irigoin [AI91], Pugh [Pug92].

Intersection

The intersection of two convex polyhedra is a convex polyhedron. Indeed, a polyhedron is the intersection of half-spaces. Thus, the intersection of two polyhedra is the intersection of all their half-spaces.

Union and Convex Hull

The union of two convex polyhedra is not necessarily a convex polyhedron. The smallest convex over-approximation of the union of two polyhedra is their *integer hull*. It may contain points that do not belong to the original polyhedra. To check whether the convex hull of two polyhedra is equal to their union, we can use the feasibility test on the system built from the convex hull and the negations of the two original polyhedra.

Difference

The difference of two convex polyhedra usually is not a convex polyhedron. An over-approximation is obtained by computing the convex hull of the resulting polyhedron, whose exactness can be checked as for the union.

4.3 Transformers

```
PROGRAM MAIN
COMMON /CT/ A(52,21,60)
COMMON /CI/ K,L,M,N
DATA NC /17/
READ *,K,L,N
IF (L.GE.1.AND.N.GE.1) THEN
   NI = 4
   L = L+1
   M = 2*K+1
   K = K+1
   CALL EXTR(NI,NC)
ENDIF
END
```

```
SUBROUTINE EXTR(NI,NC)
COMMON /CT/ A(52,21,60)
COMMON /CI/ K,L,M,N
DO I = L,K
   A(I,1,NC+1) = I*(NC+1)
   J = L+M-I
   A(J,1,NC) = A(I,1,NC)
   A(J,1,NC+1) = A(I,1,NC+1)
ENDDO
END
```

Figure 4.2: Preexisting analyses example

The semantic meaning of transformers is that a statement, when executed in a certain state, yields a new state upon termination. In other words, a statement is a state-to-state transformation. We have two kinds of transfer functions that model the direction of data-flow analysis: transformer and inverse transformer. The *transformer* represents the transformation of an input state to an output state, which is the result of the execution of the concerned statement. The *inverse transformer* corresponds to the transformation from the output to the input state.

To illustrate the transformer and the other preexisting analyses in PIPS, we use the running example in Figure 4.2, extracted from an industrial application.

4.3.1 Transformers and Inverse Transformers

Transformers as well as other analyses are based on the semantic function $\mathcal{E}$ that represents the evaluation of expressions. This evaluation yields a value that depends on the current state. Semantically, the denotation of an expression is a function from the set of states to the set of values, defined as follows:

$$\mathcal{E} : \mathit{Expression} \longrightarrow \mathit{State} \longrightarrow \mathit{Value}$$

Let $e \in \mathit{Expression}, \sigma \in \mathit{State}$, we have:

$$\begin{array}{ll}
\mathcal{E}(e) := \lambda\sigma.(\text{case } e \text{ of} & \\
Constant & \longrightarrow e \\
Reference & \longrightarrow \sigma(e) \\
(exp) & \longrightarrow \mathcal{E}(exp)(\sigma) \\
unary_op\ exp & \longrightarrow unary_op(\mathcal{E}(exp)(\sigma)) \\
exp_1\ binary_op\ exp_2 & \longrightarrow binary_op(\mathcal{E}(exp_1)(\sigma), \mathcal{E}(exp_2)(\sigma)))
\end{array}$$

The transformer, denoted by $\mathcal{T}$, is a function from the set of states to itself:

$$\mathcal{T} : \mathit{Statement} \longrightarrow \mathit{State} \longrightarrow \mathit{State}$$

Let $s \in \mathit{Statement}$ and $\sigma, \top \in \mathit{State}$. $\uplus$ is the *overriding union* operator [Mey90] where $\sigma \uplus \{(var, val)\}$ returns a state identical to σ, except that the value of var becomes val. The transformer is defined below.

$$\begin{array}{ll}
\mathcal{T}(s) := \lambda\sigma.(\text{if } \sigma = \top \text{ then } \top & \text{else case } s \text{ of} \\
\text{CONTINUE} & \longrightarrow \sigma \\
\text{STOP} & \longrightarrow \top \\
\text{READ } var & \longrightarrow \sigma \uplus \{(var, stdin)\} \\
\text{WRITE } exp & \longrightarrow \sigma \\
var = exp & \longrightarrow \sigma \uplus \{(var, \mathcal{E}(exp)(\sigma))\} \\
\text{IF } e \text{ THEN } s_1 \text{ ELSE } s_2 \text{ ENDIF} & \longrightarrow \text{if } \mathcal{E}(e)(\sigma) \text{ then } \mathcal{T}(s_1)(\sigma) \text{ else } \mathcal{T}(s_2)(\sigma) \\
\text{DO } e\ s_1 \text{ ENDDO} & \longrightarrow \text{if } \mathcal{E}(e)(\sigma) \text{ then } \mathcal{T}(s)(\mathcal{T}(s_1)(\sigma)) \text{ else } \sigma \\
\text{CALL } p(e_1, \ldots, e_n) & \longrightarrow \mathcal{T}(p.body)(\sigma \uplus \{(p.f_1, \mathcal{E}(e_1)(\sigma)), \ldots, (p.f_n, \mathcal{E}(e_n)(\sigma))\}) \\
s_1; s_2; \ldots; s_n; & \longrightarrow \mathcal{T}(s_2; \ldots; s_n;)(\mathcal{T}(s_1)(\sigma)))
\end{array}$$

Transformers are computed from elementary statements such as continue, stop, read, write and assignments to compound statements such as conditional statements, loops and sequences of statements. They are also analyzed interprocedurally through procedure calls with the no side effects assumption.

The `CONTINUE` and `WRITE` statements do not change the program state. The `STOP` statement causes the termination of the whole program, and results in the $\top$ state. The `READ` *var* statement gives *var* the value from the standard input. The `CALL` statement yields a program state which is the output state of the transformer of the called procedure body. The input state of this transformer is the initial state σ, except that the value of each formal parameter is associated to the value of the corresponding actual parameter.

Given a program state reached after the execution of a particular statement, the inverse transformer characterizes the corresponding input state. The solution would be to define the inverse function of $\mathcal{T}$, that is:

$$\begin{array}{rrl} \mathcal{T}^{-1}: & \mathit{Statement} & \longrightarrow \mathit{State} \longrightarrow \mathit{State} \\ & s & \longmapsto \mathcal{T}^{-1}(s) : \mathcal{T}(s) \circ \mathcal{T}^{-1}(s) = \lambda\sigma.\sigma \end{array}$$

However, $\mathcal{T}$ is not invertible in general. Moreover, the above semantic functions for transformers are usually not computable, except for very simple programs. The recursive definition of the loop statement shows one difficulty of program analysis, the undecidable termination problem. Furthermore, the value of variables is not always known at compile-time. In addition, describing the exact behavior of any program written in any language may be impossible. To analyze complicated programs, approximations on transformers are needed.

4.3.2 Approximations of Transformers

Since the exact transformer contains only one program state, its under-approximation gives either the same result as the exact one or the least element $\perp$ of *State*, which is always safe but not interesting.

$$\begin{array}{rrl} \underline{\mathcal{T}} : \mathit{Statement} \longrightarrow & \mathit{State} \longrightarrow & \mathit{State} \\ & \underline{\mathcal{T}}(s)(\sigma) \longmapsto & \mathcal{T}(s)(\sigma) \text{ or } \perp \end{array}$$

So we are only interested in the over-approximation of transformers that computes a set of states containing the exact state:

$$\begin{array}{rrl} \overline{\mathcal{T}} : \mathit{Statement} \longrightarrow & \mathit{State} \longrightarrow & \wp(\mathit{State}) \\ & \mathcal{T}(s)(\sigma) \in & \overline{\mathcal{T}}(s)(\sigma) \end{array}$$

Theoretically, this over-approximation can be built from the definition of the exact transformer by using approximate operators. Then an over-approximation of the inverse transformer can be defined as:

$$\begin{array}{rrl} \overline{\mathcal{T}}^{-1}: & \mathit{Statement} & \longrightarrow \mathit{State} \longrightarrow \wp(\mathit{State}) \\ & \overline{\mathcal{T}}^{-1}(s)(\sigma) & \longmapsto \{\sigma_1 : \sigma \in \overline{\mathcal{T}}(s)(\sigma_1)\} \end{array}$$

In fact, a constructive definition of the over-approximated transformers depends on the representation choice. It is often impossible to represent every set of states $S \in \wp(\mathit{State})$, but only set sharing common properties. As explained, this set is represented by a convex polyhedron in PIPS and its computation is described as follows. Each transformer is associated to a list of variables whose values are modified by the statement, and a predicate containing affine equalities and inequalities. For efficiency, it is better to keep track of the few modified variables rather than to add one equation for each of the many unmodified variables. From a mathematic point of view, all implicit equations must be taken into account.

Elementary Statement

Assignments of scalar variables with affine right-hand side expressions are handled in PIPS. If the same variable appears on both sides, the initial input value and the final output value are kept in the system. For instance, `T(I) {I#new==I#init+1}` models the effect of the assignment `I=I+1`. The transformer for a `CONTINUE` statement is the identity function `T(){}`, for a `STOP` statement is `T(){0==-1}`, because the set of states reached after a stop is empty, and for an input statement such as `READ *,N` is `T(N){}`, which means that only the value of variable `N` is modified and that no relationship between its old and its new values is known.

Conditional Statement

When the test condition is a convex polyhedron, it is combined with the transformer of the true branch. When the negation of the test condition is polyhedric, it is added to the transformer of the false branch. The transformer of the test statement is the convex hull of the two above predicates, including the implicit equations for unmodified variables.

Loop Statement

To handle loops, different kinds of fixed points are provided to have a flexible choice between efficiency and precision. Fast computation uses a primitive, non-iterative fixed point algorithm based on the transition function associated to the loop body and which only handles equations. Full computation uses a *derivative* fixed point operator based on finite differences. The loop body transformer is used to find constraints on the derivatives. Invariants, both equations and inequalities, are directly deduced from these constraints using a discrete integration.

Sequence of Statements

We assume that each statement in the sequence has been associated to its transformer. To combine transformers, relevant input and output variables are renamed as intermediate values, constraints are merged and intermediate values are eliminated by projection. Transformers are propagated from bottom to top in the abstract syntax tree of the module.

Control Flow Graph

Control flow graphs (*aka* unstructured statements in [IJT91]) are also handled in PIPS and can be accurately analyzed with the non-iterative algorithm for fixed points as for loops.

4.3.3 Interprocedural Transformers

Interprocedural propagation of transformers is performed by traversing the call graph in the reverse invocation order, which processes a procedure after its callees (see Chapter 3, page 23).

Summary Transformer

The summary transformer of a procedure is the transformer computed for the procedure body after projecting its local dynamic variables. Information about formal parameters, global variables and static variables is preserved. This summary transformer is stored in the PIPS database and used each time the procedure is called.

Transformer Translation

When a call to a procedure is encountered, the transformer of the call statement is computed by translating the summary transformer of the called procedure to the frame of the calling procedure. The translation into the calling frame uses global variables information and the bindings between actual and formal parameters. Equations are built to link formal and actual arguments when they are affine. These equations are added to the summary transformer and formal parameters are eliminated. Global variables are renamed according to the caller's scope.

Transformer Example

Transformers of the running example, analyzed interprocedurally, are illustrated in Figure 4.3. Each statement is preceded by its transformer. We have no transformer information on the statements assigning the elements of array `A` in subroutine `EXTR`, because only scalar variables are handled. With `J=L+M-I`, we have `T(J){I+J==L+M}` which is also the transformer of the sequence of five statements. The transformer of the `DO` loop, `T(I,J){K+1<=I, L<=I}` is always correct, whether the loop is executed or not. Since `K`, `I` and `L` are local variables of `EXTR`, they are projected from the global transformer of the module and an identity summary transformer is obtained: no integer scalar variable in the caller's scope is modified.

`T(K,L,M,NI){K==K#init+1,2K==M+1,L==L#init+1,NI==4}` is the transformer for the sequence of five statements in the `IF` true branch. The values of variables are not changed by the false branch. The transformer of the `IF` statement is the convex hull of the two

```
      PROGRAM MAIN
C     T(K,L,N) {}
      READ *,K,L,N
C     T(K,L,M,NI) {L+K#init==L#init+K, L#init<=L, L<=L#init+1}
      IF (L.GE.1.AND.N.GE.1) THEN
C     T(K,L,M,NI) {K==K#init+1, 2K==M+1, L==L#init+1, NI==4}
C        BEGIN BLOCK
C     T(NI) {NI==4}
         NI = 4
C     T(L) {L==L#init+1}
         L = L+1
C     T(M) {M==2K+1}
         M = 2*K+1
C     T(K) {K==K#init+1}
         K = K+1
C     T() {}
         CALL EXTR(NI,NC)
C        END BLOCK
      ENDIF
      END

C     T() {}
      SUBROUTINE EXTR(NI,NC)
C     T(I,J) {K+1<=I, L<=I}
      DO I = L,K
C     T(J) {I+J==L+M}
C        BEGIN BLOCK
C     T() {}
         A(I,1,NC) = I*NC
C     T() {}
         A(I,1,NC+1) = I*(NC+1)
C     T(J) {I+J==L+M}
         J = L+M-I
C     T() {}
         A(J,1,NC) = A(I,1,NC)
C     T() {}
         A(J,1,NC+1) = A(I,1,NC+1)
C        END BLOCK
      ENDDO
      END
```

Figure 4.3: Transformer example (declarations omitted)

branch predicates. Since the value of `K` and `L` can be increased by one or not, we have the final transformer of the test statement: `T(K,L,M,NI){L+K#init==L#init+K,L#init<=L, L<=L#init+1}`. Additional information is retained because of interactions between the true and false transformers and the test condition.

4.4 Preconditions

Precondition analyses try to discover the constraints holding among variables of a program at a control point and at entry point. For any statement, the precondition and the postcondition provide a condition that holds just before the statement execution for the former and just after for the latter.

4.4.1 Preconditions and Postconditions

Preconditions and postconditions describe the set of program states reached just before or after the execution and can be represented as functions from the set of statements to the power-set of *State*:

$$\mathcal{P}re, \mathcal{P}ost : Statement \longrightarrow \wp(State)$$

Let $s \in Statement$ and $\sigma \in State$

$$\begin{aligned} \mathcal{P}re(s) &= \{\sigma : \sigma \in dom(\mathcal{T}(s))\} \\ \mathcal{P}ost(s) &= \{\sigma' : \sigma' = \mathcal{T}(s)(\sigma), \sigma \in \mathcal{P}re(s)\} \end{aligned}$$

According to these definitions, the transformer of a statement is *applied* to the precondition of the statement to obtain the postcondition. This postcondition becomes the precondition of the following statement in a sequence. So the computation of the precondition (and the postcondition) is based on the transformer computation whose semantic function has already been described in Section 4.3.

4.4.2 Approximations of Preconditions

Like transformers, we are interested in the over-approximation of preconditions:

$$\begin{aligned} \overline{\mathcal{P}} : \quad & Statement \longrightarrow \wp(State) \\ & \mathcal{P}(s) \subseteq \overline{\mathcal{P}}(s) \end{aligned}$$

Over-approximated preconditions are computed for scalar integer variables and represented as systems of equalities and inequalities. They are propagated from the module entry point down to the abstract syntax tree leaves.

Initial Precondition

The precondition on the entry of the procedure represents what is known about the variables at the start of any execution. The initial precondition of a procedure in the intraprocedural analysis is derived from `DATA` or `PARAMETER` statements or, if no information is available, we use an empty precondition `P() {}` that represents all possible program states.

Elementary Statement

The precondition and transformer of a statement are used to compute the postcondition. The variables of the precondition and the initial values of the transformer are renamed to intermediate variables in order to merge the precondition's and the transformer's constraints. The intermediate variables are then projected to obtain the postcondition of the current statement. The postcondition for a `STOP` statement always is `P(){0==-1}`; for a single input statement `READ *,N`, is `P(N){}`.

Conditional Statement

As for transformers, the postcondition of a test statement is the convex hull of the two postconditions that have been propagated from the precondition of the statement, along the true and false branches.

Loop Statement

The loop body preconditions and the loop postcondition are derived directly using the loop fix point transformer. The accuracy is improved by detecting if the loop is always, never or sometimes entered, and by using the corresponding semantic equations.

Sequence of Statements

The transformer associated to each statement in a sequence, which is computed by a previous phase, is applied to the precondition of the statement to obtain the postcondition. This postcondition becomes the precondition of the following statement and we repeat the process until we reach the postcondition of the last statement.

Control Flow Graph

The postcondition is obtained by applying the transformer of the control flow graph to its precondition. This precondition is in fact a simple projection on all modified variables on every node in the control flow graph.

```
      PROGRAM MAIN
      COMMON /CT/ A(52,21,60)
      COMMON /CI/ K,L,M,N
      DATA NC /17/
C     P() {NC==17}
      READ *,K,L,N
C     P(K,L,N) {NC==17}
      IF (L.GE.1.AND.N.GE.1) THEN
C     P(K,L,N) {NC==17, 1<=L, 1<=N}
         NI = 4
C     P(K,L,N,NI) {NC==17, NI==4, 1<=L, 1<=N}
         L = L+1
C     P(K,L,N,NI) {NC==17, NI==4, 2<=L, 1<=N}
         M = 2*K+1
C     P(K,L,M,N,NI) {2K==M-1, NC==17, NI==4, 2<=L, 1<=N}
         K = K+1
C     P(K,L,M,N,NI) {2K==M+1, NC==17, NI==4, 2<=L, 1<=N}
         CALL EXTR(NI,NC)
      ENDIF
C     P(K,L,M,N,NI) {NC==17}
      END

C     P() {2K==M+1, NC==17, 2<=L}
      SUBROUTINE EXTR(NI,NC)
      COMMON /CT/ A(52,21,60)
      COMMON /CI/ K,L,M,N
C     P() {2K==M+1, NC==17, 2<=L}
      DO I = L,K
C     P(I,J) {2K==M+1, NC==17, L<=I, I<=K, 2<=L}
         A(I,1,NC) = I*NC
C     P(I,J) {2K==M+1, NC==17, L<=I, I<=K, 2<=L}
         A(I,1,NC+1) = I*(NC+1)
C     P(I,J) {2K==M+1, NC==17, L<=I, I<=K, 2<=L}
         J = L+M-I
C     P(I,J) {I+J==2K+L-1, 2K==M+1, NC==17, L<=I, I<=K, 2<=L}
         A(J,1,NC) = A(I,1,NC)
C     P(I,J) {I+J==2K+L-1, 2K==M+1, NC==17, L<=I, I<=K, 2<=L}
         A(J,1,NC+1) = A(I,1,NC+1)
      ENDDO
C     P(I,J) {2K==M+1, NC==17, K+1<=I, L<=I, 2<=L}
      END
```

Figure 4.4: Precondition example

4.4.3 Interprocedural Preconditions

The interprocedural analysis of preconditions is performed in the invocation order, which processes a procedure before all its callees (see Chapter 3, page 23).

Precondition Translation

Each time a procedure is invoked, the precondition of the current call site is available and is translated into the frame of the called procedure. The process is similar to transformer translation. Equations between actual and formal parameters are added. Local variables of the caller are eliminated which may cause some information loss. Global variables are renamed when possible.

Summary Precondition

The summary precondition of the called procedure is then replaced by its convex hull with the new translated precondition. This summary precondition, derived from the calling contexts, becomes the initial precondition of the procedure.

Precondition Example

Figure 4.4 shows the preconditions computed for the example in Figure 4.2. `{NC==17}` is the initial precondition of `MAIN`. The test condition `L.GT.1.AND.N.GE.1` is polyhedric and is added to the precondition of the true branch. This precondition is propagated to the call to `EXTR` and is translated to its frame. Since `NI` and `N` are not used by `EXTR`, constraints on them are not kept in the summary precondition of `EXTR`. The last precondition in `EXTR` is the loop postcondition. We have `{K+1<=I, L<=I}`, the convex hull of two predicates obtained when the loop is entered or not: `{I==K+1, L<=K}` and `{I==L, K+1<=L}`.

4.5 Convex Array Regions

Array region analysis collects information about the way array elements used and defined by programs. *Convex array regions* [TFI86, CI95, Cre95, CI96] are used throughout the dissertation.

4.5.1 Array Region Analyses

The region of an array A of n dimensions at a statement s is a function from the set of states to the power set of $\mathbb{Z}^n$:

$$\begin{aligned} \mathcal{R} : \mathit{Statement} \longrightarrow &\quad \mathit{State} \longrightarrow \wp(\mathbb{Z}^n) \\ \mathcal{R}(s) \longmapsto &\quad \lambda\sigma.(\{\phi = (\phi_1, \ldots, \phi_n) \in \mathbb{Z}^n : r(\phi, \sigma)\}) \end{aligned}$$

The semantic function $\mathcal{R}$ associates to each statement and to each program state a set of elements corresponding to the concerned array, described by the *region parameters*

$\phi = (\phi_1, \dots, \phi_n)$. The variable ϕ_i represents the i-th dimension parameter. r is the relationship existing between ϕ and the current program state.

Since array region analysis was introduced to support dependence analyses on array structures, two kinds of effects on array elements are used: `READ` region if they are used and `WRITE` region if they are defined. The difference between `READ` and `WRITE` regions is that `WRITE` regions are only defined on the domain of statements. Since expressions also have read effects, we have to define the semantic function of `READ` regions on the domain of expressions and that of statements:

$$\begin{aligned} \mathcal{R}_w &: \mathit{Statement} \longrightarrow \mathit{State} \longrightarrow \wp(\mathbb{Z}^n) \\ \mathcal{R}_r &: \mathit{Statement} \oplus \mathit{Expression} \longrightarrow \mathit{State} \longrightarrow \wp(\mathbb{Z}^n) \end{aligned}$$

To simplify the representation, we compute regions for each array independently. Array regions for all arrays in the program are the product of array regions for each array.

Suppose A is the current array with declarations $A(l_1 : u_1, \dots, l_n : u_n)$. Let $e \in \mathit{Expression}, s \in \mathit{Statement}$ and $\sigma \in \mathit{State}$. Then the semantic functions for `READ` and `WRITE` regions, $\mathcal{R}_r$ and $\mathcal{R}_w$ respectively, are defined as follows:

$$\begin{array}{ll} \mathcal{R}_r(e) := \lambda\sigma.(\text{case } e \text{ of} & \\ \mathit{Constant} & \longrightarrow \emptyset \\ \mathit{var} & \longrightarrow \emptyset \\ \mathit{var}(e_1, \dots, e_n) & \longrightarrow \cup_{i=1}^{n} \mathcal{R}_r(e_i)(\sigma) \cup (\text{if } (\mathit{var} = A) \text{ then } \{(\phi_1, \dots, \phi_n) : \phi_i = \mathcal{E}(e_i)(\sigma)\} \text{ else } \emptyset) \\ (\mathit{exp}) & \longrightarrow \mathcal{R}_r(\mathit{exp})(\sigma) \\ \mathit{unary\_op}\ \mathit{exp} & \longrightarrow \mathcal{R}_r(\mathit{exp})(\sigma) \\ \mathit{exp}_1\ \mathit{binary\_op}\ \mathit{exp}_2 & \longrightarrow \mathcal{R}_r(\mathit{exp}_1)(\sigma) \cup \mathcal{R}_r(\mathit{exp}_2)(\sigma)) \end{array}$$

$$\begin{array}{ll} \mathcal{R}_r(s) := \lambda\sigma.(\text{case } s \text{ of} & \\ \text{CONTINUE} & \longrightarrow \emptyset \\ \text{STOP} & \longrightarrow \emptyset \\ \text{READ } \mathit{var} & \longrightarrow \emptyset \\ \text{READ } \mathit{var}(e_1, \dots, e_n) & \longrightarrow \cup_{i=1}^{n} \mathcal{R}_r(e_i)(\sigma) \\ \text{WRITE } \mathit{exp} & \longrightarrow \mathcal{R}_r(\mathit{exp})(\sigma) \\ \mathit{var} = \mathit{exp} & \longrightarrow \mathcal{R}_r(\mathit{exp})(\sigma) \\ \mathit{var}(e_1, \dots, e_n) = \mathit{exp} & \longrightarrow \mathcal{R}_r(\mathit{exp})(\sigma) \cup (\cup_{i=1}^{n} \mathcal{R}_r(e_i)(\sigma)) \\ \text{IF } e \text{ THEN } s_1 \text{ ELSE } s_2 \text{ ENDIF} & \longrightarrow \mathcal{R}_r(e)(\sigma) \cup (\text{if } \mathcal{E}(e)(\sigma) \text{ then } \mathcal{R}_r(s_1)(\sigma) \text{ else } \mathcal{R}_r(s_2)(\sigma)) \\ \text{DO } e\ s_1 \text{ ENDDO} & \longrightarrow \mathcal{R}_r(e)(\sigma) \cup (\text{if } \mathcal{E}(e)(\sigma) \text{ then } \mathcal{R}_r(s_1)(\sigma) \cup \mathcal{R}_r(s)(\mathcal{T}(s_1)(\sigma)) \text{ else } \emptyset) \\ \text{CALL } p(e_1, \dots, e_n) & \longrightarrow \mathcal{R}_r(p.\mathit{body})(\mathcal{T}(p.\mathit{body})(\sigma \uplus \{(p.f_1, \mathcal{E}(e_1)(\sigma)), \dots, (p.f_n, \mathcal{E}(e_n)(\sigma))\})) \\ s_1; s_2; \dots; s_n; & \longrightarrow \mathcal{R}_r(s_1)(\sigma) \cup \mathcal{R}_r(s_2; \dots; s_n;)(\mathcal{T}(s_1)(\sigma))) \end{array}$$

$$
\begin{array}{ll}
\mathcal{R}_w(s) := \lambda\sigma.(\text{case } s \text{ of} & \\
\text{CONTINUE} & \longrightarrow \emptyset \\
\text{STOP} & \longrightarrow \emptyset \\
\text{READ } A & \longrightarrow \{(\phi_1, \dots, \phi_n) : \mathcal{E}(l_i)(\sigma) \leq \phi_i \leq \mathcal{E}(u_i)(\sigma), i = 1, \dots, n\} \\
\text{READ } A(e_1, \dots, e_n) & \longrightarrow \{(\phi_1, \dots, \phi_n) : \phi_i = \mathcal{E}(e_i)(\sigma), i = 1, \dots, n\} \\
\text{WRITE } exp & \longrightarrow \emptyset \\
A(e_1, \dots, e_n) = exp & \longrightarrow \{(\phi_1, \dots, \phi_n) : \phi_i = \mathcal{E}(e_i)(\sigma), i = 1, \dots, n\} \\
\text{IF } e \text{ THEN } s_1 \text{ ELSE } s_2 \text{ ENDIF} & \longrightarrow \text{if } \mathcal{E}(e)(\sigma) \text{ then } \mathcal{R}_w(s_1)(\sigma) \text{ else } \mathcal{R}_w(s_2)(\sigma) \\
\text{DO } e\ s_1 \text{ ENDDO} & \longrightarrow \text{if } \mathcal{E}(e)(\sigma) \text{ then } \mathcal{R}_w(s_1)(\sigma) \cup \mathcal{R}_w(s)(\mathcal{T}(s_1)(\sigma)) \text{ else } \emptyset \\
\text{CALL } p(e_1, \dots, e_n) & \longrightarrow \mathcal{R}_w(p.body)(\mathcal{T}(p.body)(\sigma \uplus \{(p.f_1, \mathcal{E}(e_1)(\sigma)), \dots, (p.f_n, \mathcal{E}(e_n)(\sigma))\})) \\
s_1; s_2; \dots; s_n; & \longrightarrow \mathcal{R}_w(s_1)(\sigma) \cup \mathcal{R}_w(s_2; \dots; s_n;)(\mathcal{T}(s_1)(\sigma)))
\end{array}
$$

Note that unlike the evaluation expression $\mathcal{E}$ and transformer $\mathcal{T}$ analyses (page 37), references to a scalar variable or an array element are distinguished by var and $var(e_1, \dots, e_n)$. The `READ` regions of an array A in the assignment $var(e_1, \dots, e_n) = exp$ consist of regions of the right hand side expression (exp) and that of the subscript expressions on the left side $(e_1, \dots, e_n)$. These regions are defined recursively by the semantic function of `READ` regions for an expression. The `WRITE` region of the input statement `READ A` is the whole array declaration.

4.5.2 Approximations of Array Regions

Unlike transformers and preconditions, both under- and over-approximations are computed for array regions. A region has the approximation `MUST` if every element in the region is accessed with certainty, and the approximation `MAY` if its elements are simply potentially accessed. The approximation of a region is `EXACT` if the region exactly represents the requested set of array elements.

$$
\begin{aligned}
\overline{\mathcal{R}} &: \mathit{Statement} \longrightarrow \mathit{State} \longrightarrow \wp(\mathbb{Z}^n) \\
\underline{\mathcal{R}} &: \mathit{Statement} \longrightarrow \mathit{State} \longrightarrow \wp(\mathbb{Z}^n) \\
\underline{\mathcal{R}}(s)(\sigma) &\subseteq \mathcal{R}(s)(\sigma) \subseteq \overline{\mathcal{R}}(s)(\sigma)
\end{aligned}
$$

The array region is approximated by a convex polyhedron containing affine equalities and inequalities. These constraints link the region parameters that represent the array dimensions to the value of the program integer scalar variables.

Regions are built bottom-up from the deepest nodes to the largest compound statement nodes in the hierarchical control flow graph. It means that at each meet point of a control flow graph, the region information from different control branches are merged with a convex hull operator. The approximation of regions is conservative. Intraprocedural region analysis has to deal with statements such as assignments, basic blocks, tests, loops and control flow graphs.

Elementary Statement

Each array reference in an assignment is converted to an elementary region. This region is `EXACT` if and only if the subscript expressions are affine functions of the program integer scalar variables. The effect depends on the reference position in the statement. To save space, regions of the same array with the same effect are merged by using the convex hull operator. If the resulting region contains array elements that do not belong to the original regions, it becomes a `MAY` region.

Conditional Statement

The array regions of a conditional statement contain the regions of the test condition, plus the unified regions of the true and false branches which are restrained by the test condition evaluation. `EXACT` regions may become `MAY` regions.

Loop Statement

The region corresponding to the body of a loop is a function of the loop index value. During the propagation of regions, we need to unify regions corresponding to different, but successive, instances of the loop body in order to get the summary region of the loop. This union operation is equivalent to eliminating the loop index i from the system which consists of the region predicate and the constraint on the iteration space: $lb \leq i \leq ub$. The operation is exact if the lower and upper bounds of the loop index are affine functions and the elimination of loop index is exact.

Sequence of Statements

Array regions at different program points correspond to different program states. To obtain array regions of a sequence of statements, we must translate array regions associated to different statements to a reference state, e.g. the state preceding the sequence. The translation uses transformers and consists of adding to the predicate of the region the constraints abstracting the effects of the statement between the two states. Then variables of the original state are eliminated. Only variables referring to the reference state are kept in the resulting polyhedron. The projection of variables from a convex polyhedron may introduce integer points that do not belong to the actual projection. In this case, we have a `MAY` region.

Control Flow Graph

PIPS handles control flow graphs in a very simple way: every variable modified at a node is projected from the regions of all other nodes. All approximations are decreased to `MAY`.

```
      PROGRAM MAIN
C   <A(PHI1,PHI2,PHI3)-READ-MAY-{PHI2==1, 2<=PHI1, NC<=PHI3, PHI3<=NC+1}>
C   <A(PHI1,PHI2,PHI3)-WRITE-MAY-{PHI2==1, 2<=PHI1, NC<=PHI3, PHI3<=NC+1}>
      READ *,K,L,N
C   <A(PHI1,PHI2,PHI3)-READ-MAY-{PHI2==1,1+L<=PHI1,PHI1<=1+K,NC<=PHI3,PHI3<=NC+1}>
C   <A(PHI1,PHI2,PHI3)-WRITE-MAY-{PHI2==1,1+L<=PHI1,PHI1<=1+2K,NC<=PHI3,PHI3<=NC+1}>
      IF (L.GE.1.AND.N.GE.1) THEN
C   <A(PHI1,PHI2,PHI3)-READ-EXACT-{PHI2==1,1+L<=PHI1,PHI1<=1+K,NC<=PHI3,PHI3<=NC+1}>
C   <A(PHI1,PHI2,PHI3)-WRITE-MAY-{PHI2==1,1+L<=PHI1,PHI1<=1+2K,NC<=PHI3,PHI3<=NC+1}>
C        BEGIN BLOCK
         NI = 4
         L = L+1
         M = 2*K+1
         K = K+1
C   <A(PHI1,PHI2,PHI3)-READ-EXACT-{PHI2==1,L<=PHI1,PHI1<=K,NC<=PHI3,PHI3<=NC+1}>
C   <A(PHI1,PHI2,PHI3)-WRITE-MAY-{PHI2==1,L<=PHI1,PHI1+1<=2K,NC<=PHI3,PHI3<=NC+1}>
         CALL EXTR(NI,NC)
C        END BLOCK
      ENDIF
      END

C   <A(PHI1,PHI2,PHI3)-READ-EXACT-{PHI2==1,L<=PHI1,PHI1<=K,NC<=PHI3,PHI3<=NC+1}>
C   <A(PHI1,PHI2,PHI3)-WRITE-MAY-{PHI2==1,L<=PHI1,PHI1+1<=2K,NC<=PHI3,PHI3<=NC+1}>
      SUBROUTINE EXTR(NI,NC)
C   <A(PHI1,PHI2,PHI3)-READ-EXACT-{PHI2==1,L<=PHI1,PHI1<=K,NC<=PHI3,PHI3<=NC+1}>
C   <A(PHI1,PHI2,PHI3)-WRITE-MAY-{PHI2==1,L<=PHI1,PHI1+1<=2K,NC<=PHI3,PHI3<=NC+1}>
      DO I = L,K
C   <A(PHI1,PHI2,PHI3)-READ-EXACT-{PHI1==I,PHI2==1,NC<=PHI3,PHI3<=NC+1}>
C   <A(PHI1,PHI2,PHI3)-WRITE-MAY-{PHI2==1,I<=PHI1,PHI1+1+I<=2K+L,NC<=PHI3,PHI3<=NC+1}>
C        BEGIN BLOCK
C   <A(PHI1,PHI2,PHI3)-WRITE-EXACT-{PHI1==I, PHI2==1, PHI3==NC}>
         A(I,1,NC) = I*NC
C   <A(PHI1,PHI2,PHI3)-WRITE-EXACT-{PHI1==I, PHI2==1, PHI3==NC+1}>
         A(I,1,NC+1) = I*(NC+1)
         J = L+M-I
C   <A(PHI1,PHI2,PHI3)-READ-EXACT-{PHI1==I, PHI2==1, PHI3==NC}>
C   <A(PHI1,PHI2,PHI3)-WRITE-EXACT-{PHI1==J, PHI2==1, PHI3==NC}>
         A(J,1,NC) = A(I,1,NC)
C   <A(PHI1,PHI2,PHI3)-READ-EXACT-{PHI1==I, PHI2==1, PHI3==NC+1}>
C   <A(PHI1,PHI2,PHI3)-WRITE-EXACT-{PHI1==J, PHI2==1, PHI3==NC+1}>
         A(J,1,NC+1) = A(I,1,NC+1)
C        END BLOCK
      ENDDO
      END
```

Figure 4.5: Array region example (declarations omitted)

4.5.3 Interprocedural Array Regions

The interprocedural propagation of `READ` and `WRITE` regions is a reverse invocation order traversal on the call graph. The callees are analyzed first. The procedure summary regions are then used at each call site, after the translation process, during the intraprocedural analysis of the callers.

Summary Region

The summary region of a procedure is computed by eliminating local effects from the region of the procedure body. Each analysis is performed once on each procedure in order to keep a reasonable analysis speed. The result is stored in the PIPS compiler database and is retrieved each time a procedure call is encountered.

Region Translation

At each call site, the summary regions of the called subroutine are translated from the callee's name space into the caller's name space, using the relationship between actual and formal parameters, and between the declarations of global variables in both routines. The translation process computes offsets of the formal parameters with respect to the actual parameters if we pass array elements in procedure calls. It also handles *array reshaping* (the number and size of dimensions in an actual argument array declaration are different from those in an associated formal argument array declaration) by building appropriate *translation systems* [CI95]. The translation consists of adding to the predicate of the region the equations giving the relationship between the subscript values of the arrays in terms of ϕ variables. The ϕ variables of the formal array are then eliminated. If this variable projection is not exact, we have a `MAY` region.

Region Example

Figure 4.5 illustrates the array region computation of the example used for transformers and preconditions. Each statement is preceded by its regions. Accesses to array `A` occur in the loop body of `EXTR`. The merged region of `READ` regions of `A` is `EXACT` while that of `WRITE` regions is `MAY` because unreferenced elements with `PHI1` in maybe non-empty range `[I+1,2K+L-I-2]` are added in the convex hull `{I<=PHI1, PHI1+1+I<=2K+L}`. These regions are unified over different iterations, propagated up to have the summary regions of `EXTR`. After the interprocedural translation, the regions before the call site to `EXTR` are propagated up, and the `READ` region becomes `MAY` because if the test condition is false, no array elements are used.

4.6 Conclusion

Much research on the semantic meanings of expressions and statements has been carried out within the theory of programming languages community (Gordon [Gor79, Gor88], Meyer [Mey90]). They can be described by using denotational semantics as for the evaluation of expressions and transformer analyses in this thesis. Meanwhile, preconditions and postconditions are indeed two kinds of assertions to express the program properties by axiomatizing the programming languages. They can be modeled using the denotational interpretation as described. A close approach was initially developed by Karr [Kar76], Cousot and Halbwachs [CH78] and Halbwachs [Hal79] that tries to find the affine relationship among variables in a program.

In fact, transformers and preconditions are powerful symbolic analyses that abstract the relationship between program states and polyhedra, and encompass most standard interprocedural constant propagation as well as interval analyses. The main advantage of the transformer and precondition analyses is that we can deduct from the program semantics information that is not stated explicitly.

The array accesses analysis is studied in [ABC$^+$88, CK88, Fea91, May92, MAL93, DGS93, HBCM92, TP95, Les96, BCF97]. There are different approximations to represent the set of elements of an accessed array such as convex polyhedra [TFI86, CI95, Cre95, CI96], regular section descriptors and guarded regular section descriptors [GLL95, NGL95, GL00, LGL01], symbolic array section [GMS00], data access descriptors and linear memory access descriptor [HPY01, PHP02], etc. The techniques differ in the amount of precision as well as the efficiency on storage and time. They are compared empirically by Hind et al. in [HBCM94]. For example, in Hall et al. [HMA$^+$95a, HMA$^+$95b], array element sets are represented by lists of polyhedra and there is no exact representation, only under- and over-approximations. In Gu et al. [GLL95, NGL95, GL00], array element sets is a list of RSDs (Regular Section Descriptor) with bounds and step, guarded by predicates derived from `IF` conditions. Lin and Padua [LP99] propose an analysis finding properties of arrays that appear in the subscripts of other arrays (indirection arrays). In Manjunathaiah and Nicole [MN97], precision is improved by keeping the complementary array sections when computing the envelope convex union. Feautrier [Fea01] gives a review of the *ad hoc* techniques which have been designed for the analysis of array programs, initially based on the data dependence concept. These techniques try to compute the *source function*, which gives, for each value in the program, the name of the write operation which created it. Exact and exhaustive array dataflow information is retrieved for *static control programs*, and approximate results are given for more general program models by using *fuzzy array dataflow analysis*. The implementation choice is really a trade-off between efficiency and precision. Our array region analyses have proved to be good enough to provide useful information for the analyses that are described in the following chapters.

Chapter 5

ARRAY RESIZING

This chapter describes *array resizing*, a process to associate automatically new array bounds to pointer-like array declarations. Section 5.1 discusses the problems related to array declarations in FORTRAN and other programming languages. Section 5.2 presents three different methods of solving these problems. The *argument association rules* approach uses the relationship between actual and formal arguments from parameter-passing rules. New array declarations in the called procedure are computed with respect to the declarations in the calling procedures. The second method is based on *array regions* that give information about the set of array elements accessed during any code execution. The *code instrumentation* approach completes the above two methods by adding new common variables to pass array descriptors to the called procedures. Experimental results with different approaches as well as their combinations are given in Section 5.3. Comparisons are given in Section 5.4 which concludes this chapter.

5.1 Array Declaration Problems

In FORTRAN 77, each array declaration is either a constant, an adjustable, or an assumed-size array declaration (Section 5.12 in [ANS83]). An *assumed-size array declaration* is a constant or an adjustable array declaration, except that the upper bound of the last dimension is an asterisk (*). This assumed-size array is only permitted in formal array declarations. Array argument declarations with an upper bound of 1 (*ugly assumed-size* array declaration by GNU terminology) instead of an asterisk are also encountered. They were used by programmers using older versions of the FORTRAN standard to declare implicitly array pointers.

```
SUBROUTINE SUB(A,N)                  SUBROUTINE SUB(A,N)
REAL A(100)                          REAL A(N)
Constant array declaration           Adjustable array declaration

SUBROUTINE SUB(A,N)                  SUBROUTINE SUB(A,N)
REAL A(*)                            REAL A(1)
Assumed-size array declaration       Ugly assumed-size array declaration
```

Declaring `1` as the last upper bound of the array declaration, although array references in the corresponding procedure are outside the defined extent of the array, is dangerous. This causes premature aborts due to bound violations when the programs are compiled with an array range checking option. At the very least, it should be replaced with assumed-size array declaration.

Since arrays in FORTRAN are allocated in column-major order, the computation of an array element address in the memory does not involve the upper bound of the last dimension. This is the reason why the last upper bound can be left unspecified. It is only needed to determine the maximum allowed value of the last index, and therefore the size of the entire array. In addition, it is sometimes impracticable to use either constant or adjustable array declarations in a procedure because the actual size of the array is unknown when the procedure starts executing. It is then the responsibility of the programmer to ensure that any reference to a formal array argument does not go past the end of the corresponding actual array, since any attempt to read from or write to a location outside the declared ranges is an error, and may result in unexpected results, security holes or failure.

The problem with assumed-size arrays is that the compiler does not know their logical size in the called procedure. Array bound checking as well as other relative checking which are critical for code safety and debugging cannot be applied. As reported by Appleyard [App01], many current compilers omit such checking on assumed-size arrays, which is not acceptable for software verification. Furthermore, assumed-size arrays may cause debugging tools to crash when diagnosing some input/output operations such as `PRINT *,A` where `A` is an assumed-size array. Since the assumed-size feature relies on the assumption of a linear memory model, it is also hard to handle specialized FORTRAN implementations.

When ugly and assumed-size array declarations are used by programmers, code understanding, one of the core software engineering activities, becomes more difficult. Program comprehension is really needed to maintain, reuse, re-engineer and enhance software systems. These *pointer-typed* array declarations should be dropped from the future versions of the language. In good programming practice, the formal array should be declared as an adjustable array and the actual size should be passed as an additional argument or by a common variable.

Our study of scientific applications from Linpack, PerfectClub [Hoc96] and SPEC95 CFP [DD98] benchmarks shows a large number of pointer-typed array argument declarations. The numbers of ugly and assumed-sized arrays given in Table 5.1 show that these kinds of declarations occur in 18 out of the 23 programs considered. The average percentage of these declarations is about 60% out of the total array argument declarations in the 18 programs. To ease code maintenance and debugging, an automatic phase is essential if we do not want to infer thousands of precise declarations manually.

Dimension declarations of formal array arguments are not always explicit in many other languages. In C and JAVA, the programmer can replace the first array dimension with `[]` in the array declaration, which prevents other program analyses. Array declarations can also raise problems in other class of programming languages that do not have variable declarations such as MATLAB and APL [MP99, JB01]. A large execution time overhead for dynamic array allocation and resizing in MATLAB is due to repeated reallocation and copying. As matrix and vector sizes are not declared by the programmer, the MATLAB interpreter allocates storage for these variables on demand, during program execution. An attempt to write into a matrix element outside the bounds of the matrix causes the system to reallocate storage for the entire matrix, copying over all elements from the old storage to newly allocated space. This becomes very time-consuming if these operations are done within loops. The same problem is encountered with the dynamic allocation of the Vector class in JAVA. In this case, an implicit appropriate preallocation, when possible, would be interesting to improve performance.

Table 5.1: Linpack, PerfectClub, SPEC95 CFP: number of total array argument declarations, number and percentage of ugly and assumed-size array declarations.

Benchmark	**Array**	**Ugly (1)**	**Assumed (*)**	**Total**	**Percentage**
linpack	16	13	3	16	100.00%
bdna	228	0	191	191	83.77%
dyfesm	81	0	19	19	23.46%
flo52	73	18	0	18	24.66%
mdg	40	29	0	29	72.50%
mg3d	84	68	0	68	80.95%
ocean	60	55	0	55	91.67%
qcd	32	0	11	11	34.38%
spc77	286	11	0	11	3.85%
spice	38	34	0	34	89.47%
trfd	22	0	13	13	59.09%
su2cor	47	0	5	5	10.64%
hydro2d	28	0	7	7	25.00%
applu	20	0	18	18	90.00%
turb3d	60	43	15	58	96.67%
apsi	633	145	292	437	69.04%
fpppp	26	19	0	19	73.08%
wave5	153	1	69	70	45.75%

A judicious use of declarations can significantly improve the quality of code that can be generated by an APL compiler as shown in [Bud88]. Precise array declarations are also important for data distribution in FORTRAN compilers for message-passing machines as in [KMM+95]. Finally, in the next chapters, we see that the normalization of array declarations is unavoidable for alias analysis, array range and uninitialized variable checking.

```
PROGRAM LINPACK
REAL A(201,200),B(200)
INTEGER IPVT(200)
LDA = 201
N = 100
CALL DGESL(A,LDA,N,IPVT,B)
CALL DGEFA(N,IPVT,B)
END

SUBROUTINE DGESL(A,LDA,N,IPVT,B)
INTEGER LDA,N,IPVT(1)
REAL A(LDA,1),B(1)
DO K = 1, N-1
   L = IPVT(K)
   CALL DAXPY(N-K,L,A(K+1,K),B(K+1))
ENDDO
END

SUBROUTINE DGEFA(N,IPVT,B)
INTEGER N,IPVT(1)
REAL B(1)
DO K = 1, N-1
   L = IPVT(K)
   CALL DAXPY(N-K,L,B,B(K+1))
ENDDO
END

SUBROUTINE DAXPY(M,DA,DX,DY)
REAL DX(*),DY(*),DA
DO I = 1,M
   DY(I) = DY(I) + DA*DX(I)
ENDDO
END
```

Figure 5.1: Array resizing example (excerpt from *Linpack*)

As a result of these experiments, we developed three methods to identify automatically a proper last upper bound for the ugly and assumed-size array declarations, a process we call *array resizing*. The first approach to array resizing is an invocation order analysis (see Chapter 3, page 23) on the call graph, based on the *association rules of arguments* in the FORTRAN standard. This method tries to find the exact array argument declarations in the called procedure with respect to the declarations in the calling procedures. The second approach is a reverse invocation order analysis (see Chapter 3, page 23), using array regions. New array sizes are computed from the sets of actually accessed array elements during the execution of the program. We do not necessarily need to use the main program and this approach can also be used for library maintenance. The third approach, code instrumentation, is somewhat similar to the first one, except that new array sizes are established by adding new common variables to communicate among different procedures. Since the first and second approach do not work for all pointer-typed arrays, this third approach is a complementary method to obtain 100% resized arrays. Note that FORTRAN aliasing rules are not used as another approach (see Chapter 9).

The example in Figure 5.1 is extracted from the Linpack benchmark to illustrate our different approaches to array resizing. It contains 7 ugly and assumed-size array declarations that should be resized such as `IPVT(1)`, `A(LDA,1)` in `DGESL` and `DX(*)`, `DY(*)` in `DAXPY`.

By propagating information down the call tree, the procedure call `CALL DGESL(A, LDA,N,IPVT,B)` in the main program shows that the actual array `A(201,200)` in the main program `LINPACK` (noted as `LINPACK:A(201,200)`) is associated to the formal array `A(LDA,1)` in subroutine `DGESL` (noted as `DGESL:A(LDA,1)`) and as we have `LDA = 201`, the second dimension of the array `DGESL:A` could be replaced with `[1:200]`. In the same way, we have new array declarations: `DGESL:B(200)`, `DGESL:IPVT(200)`, `DGEFA:B(200)` and `DGEFA:IPVT(200)`.

On the other hand, the array region analysis in subroutine `DAXPY` shows that array region `DX(PHI1){1<=PHI1,PHI1<=M}` must be read, and that `DY(PHI1){1<=PHI1,PHI1<=M}` must be read and written, assuming `M>=1`, which can be proven by using interprocedural information. So new declarations `REAL DX(M)` and `REAL DY(M)` can be induced by finding the real array accesses in this subroutine. More details about the above approaches are given in the following section.

5.2 Array Resizing Approaches

5.2.1 Argument Association Rules

This approach is based on the association rules of formal and actual arguments (the actual FORTRAN terminology is *dummy* and *actual* arguments) in Section 15.9.3.3 of the FORTRAN 77 standard [ANS83]. A formal array can be associated to an actual array or to an actual array element. In the first case, the size of the formal argument array must not exceed the size of the actual argument array. In the second case, the size of the formal argument array must not exceed the size of the actual argument array plus one minus the subscript value of the array element.

In a program, the number and size of dimensions in an actual argument array declaration may be different from those of dimensions in an associated formal argument array declaration. The application of the above conditions is not straightforward because of this array reshaping (see Chapter 4, page 50) and other practice features such as actual and formal arguments of different types. We use the following systems to represent the relationship between array variables. Let A and B be the corresponding actual and formal array arguments, respectively. The characteristics of both arrays are summarized in Table 5.2. The number of dimensions of an array in FORTRAN varies from 1 to 7. The

Table 5.2: Summarization of actual and formal arrays

array name	A	B
number of dimensions	n	m
lower bounds	$l_{a1}, \dots, l_{an}$	$l_{b1}, \dots, l_{bm}$
upper bounds	$u_{a1}, \dots, u_{an}$	$u_{b1}, \dots, u_{bm}$
element size	e_a	e_b
dimension size	$d_{ai} = u_{ai} - l_{ai} + 1$	$d_{bi} = u_{bi} - l_{bi} + 1$
array size	$\prod_{i=1}^{n} d_{ai}$	$\prod_{i=1}^{m} d_{bi}$
array element	$A(s_{a1}, \dots, s_{an})$	$B(s_{b1}, \dots, s_{bm})$
subscript value	$1 + \sum_{i=1}^{n} \left((s_{ai} - l_{ai}) \prod_{j=1}^{i-1} d_{aj} \right)$	$1 + \sum_{i=1}^{m} \left((s_{bi} - l_{bi}) \prod_{j=1}^{i-1} d_{bj} \right)$

size of an array element is represented in *numerical* or *character storage units*. The size of an array is equal to the number of elements in the array (Section 5.2 [ANS83]). As FORTRAN language allocates arrays in column-major order, the subscript value of an array reference does not involve the last upper bound (u_{an} and u_{bm}). Note that $\prod_{j=1}^{0} x_j = 1$.

Definition 5.2.1 A program respects the association rules of formal and actual arguments if the following conditions are satisfied:

- Case 1: If the whole array A is passed as an actual argument to the formal array argument B, then:

$$e_b . \prod_{i=1}^{m} d_{bi} \leq e_a . \prod_{i=1}^{n} d_{ai}$$

- Case 2: If the array element $A(s_{a1}, \dots, s_{an})$ is passed as an actual argument to the formal array argument B, then:

$$e_b . \prod_{i=1}^{m} d_{bi} \leq e_a . \left(\prod_{i=1}^{n} d_{ai} + 1 - \left(1 + \sum_{i=1}^{n} \left((s_{ai} - l_{ai}) \prod_{j=1}^{i-1} d_{aj} \right) \right) \right)$$

If B has an ugly or assumed-size array declaration, the value of u_{bm} would be 1 or *, and $l_{bm} = 1$. By using equations from the above inequalities, we can calculate the supposed new value of u_{bm}, represented by variable x, for Case 1 and 2:

$$x.e_b.\prod_{i=1}^{m-1} d_{bi} = e_a.\prod_{i=1}^{n} d_{ai} \tag{5.1}$$

$$x.e_b.\prod_{i=1}^{m-1} d_{bi} = e_a.\left(\prod_{i=1}^{n} d_{ai} - \sum_{i=1}^{n}\left((s_{ai} - l_{ai})\prod_{j=1}^{i-1} d_{aj}\right)\right) \tag{5.2}$$

Lemma 5.2.1 If both arrays have the same element size, that is $e_a = e_b$, we have the simplified equations:

$$x.\prod_{i=1}^{m-1} d_{bi} = \prod_{i=1}^{n} d_{ai} \text{ and } x.\prod_{i=1}^{m-1} d_{bi} = \prod_{i=1}^{n} d_{ai} - \sum_{i=1}^{n}\left((s_{ai} - l_{ai})\prod_{j=1}^{i-1} d_{aj}\right)$$

The proof of this lemma is straightforward.

Lemma 5.2.2 If there exists $k \in N, 1 \leq k \leq min(n, m-1)$ such that $\forall j = 1, \ldots, k$: $d_{aj} = d_{bj}$, Equation 5.1 is equivalent to:

$$x.e_b.\prod_{i=k+1}^{m-1} d_{bi} = e_a.\prod_{i=k+1}^{n} d_{ai}$$

Moreover, if A is passed as an array element whose first k subscripts are equal to their corresponding lower bounds, that is: $\forall j = 1, \ldots, k : s_{aj} = l_{aj}$, Equation 5.2 is equivalent to:

$$x.e_b.\prod_{i=k+1}^{m-1} d_{bi} = e_a.\left(\prod_{i=k+1}^{n} d_{ai} - \sum_{i=k+1}^{n}\left((s_{ai} - l_{ai})\prod_{j=k+1}^{i-1} d_{aj}\right)\right)$$

Proof: The first equivalence is obtained by dividing both sides by common term. The second one is proven. If $\exists k \in \mathbb{N}, 1 \leq k \leq min(n, m-1)$ such that $\forall j = 1, \ldots, k : d_{aj} = d_{bj}$ and $s_{aj} = l_{aj}$, Equation 5.2 is equivalent to:

$$x.e_b.\prod_{i=1}^{m-1} d_{bi} = e_a.\left(\prod_{i=1}^{n} d_{ai} - \sum_{i=1}^{k}\left((s_{ai} - l_{ai})\prod_{j=1}^{i-1} d_{aj}\right) - \sum_{i=k+1}^{n}\left((s_{ai} - l_{ai})\prod_{j=1}^{i-1} d_{aj}\right)\right)$$

$$x.e_b.\prod_{i=1}^{m-1} d_{bi} = e_a.\left(\prod_{i=1}^{n} d_{ai} - \sum_{i=k+1}^{n}\left((s_{ai} - l_{ai})\prod_{j=1}^{i-1} d_{aj}\right)\right) \text{(because } s_{ai} = l_{ai}, i = 1, k)$$

$$x.e_b.\prod_{i=1}^{k} d_{bi}.\prod_{i=k+1}^{m-1} d_{bi} = e_a.\left(\prod_{i=1}^{k} d_{ai}.\prod_{i=k+1}^{n} d_{ai} - \sum_{i=k+1}^{n}\left((s_{ai}-l_{ai})\prod_{j=1}^{k} d_{aj}.\prod_{j=k+1}^{i-1} d_{aj}\right)\right)$$

$$x.e_b.\prod_{i=k+1}^{m-1} d_{bi} = e_a.\left(\prod_{i=k+1}^{n} d_{ai} - \sum_{i=k+1}^{n}\left((s_{ai}-l_{ai})\prod_{j=k+1}^{i-1} d_{aj}\right)\right) (\text{as } d_{ai} = d_{bi}, i = 1, k)$$

□

The argument association rules approach is an invocation order traversal on the call graph. It begins with the main program, and the new sizes of array arguments in the called procedure are computed with respect to the actual declarations in the calling procedures, as given in Algorithm 5.2.1.

Algorithm 5.2.1

procedure Argument_Association_Rules_Array_Resizing(p)
    p : current procedure
begin
    for each ugly or assumed-size array b in p
        $V := \emptyset$ /* set of new values for the last upper bound */
        for each call site c to p
            $q :=$ corresponding_caller(c)
            $a :=$ corresponding_actual_array_argument(b, c)
            $k :=$ number_of_equal_dimensions(a, b, c, p, q)
            $s_1 :=$ size_subscript_value_of_actual_array(a, k, c, q)
            if s_1 can be translated to the frame of p **then**
                $s_1' :=$ translate_to_callee_frame(s_1, p, c, q)
                $s_2 :=$ size_of_formal_array_except_last_dimension(b, k, p)
                $v := (e_a * s_1')/(e_b * s_2)$
            else
                $v :=$ assumed-size
            endif
            $V := V \cup \{v\}$
        endfor
        if $V = \{v\}$ **then**
            last_upper_bound(b) $:= v$
        else
            last_upper_bound(b) $:=$ assumed-size
        endif
    endfor
end

Since there may be several calls to one procedure p, we may have different values for the last upper bound of array b. For each call site c, the $number\_of\_equal\_dimensions$ (a, b, c, p, q) function computes k, the number of first equal dimensions of the actual array a and the formal array b. When the actual argument is an array element, k is also the number of first subscripts that are equal to their corresponding lower bounds. This step simplifies the computation of the array sizes and the subscript value expressions. The size of the actual array in procedure q is calculated by using all dimensions but the first k ones. If the actual argument is an array element, its subscript value expression is evaluated only from subscript $k + 1$. The size of the formal array is computed in the same way, omitting the last dimension which has an ugly or assumed-size bound to be replaced.

The $translate\_to\_callee\_frame(s_1, p, c, q)$ function attempts to translate all variables in the size expression of the actual argument from the caller's name space into the callee's name space, by using the relationship between formal and actual arguments, and between the declarations of global variables in both routines. If the new values contain variables that cannot be translated in the scope of the callee, or if these values are different for different call sites, we have to keep an assumed-size declarator for the array. If not, we have the same values for different call sites and they are in the callee's frame, thus the array can be resized with this new upper bound value.

The precondition (see Chapter 4) of the current call site can be taken into account as it gives more information for the simplification and translation steps. Returning to the example in Figure 5.1, array element passing in `CALL DAXPY(N-K,L,A(K+1,K),B(K+1))` and `CALL DAXPY(N-K,L,B,B(K+1))` makes it more difficult to find the right sizes for arrays `DAXPY:DX` and `DAXPY:DY`.

For `DX`, the corresponding actual array size in the caller `DGESL` is `LDA*200+1-(1+(K+1-1)+(K-1)*201)` and in the caller `DGEFA` is `200`. By using the preconditions `DGESL:N==100` and `DGESL:LDA==201` shown in Figure 5.2, and the binding information `DGESL:N-DGESL:K==DAXPY:M`, after the translation to the scope of `DAXPY`, we have two different values: `20201+202*M` and `200`. So array `DX` still has an assumed-size array declaration.

However, all actual array sizes of array `DY` can be translated to `100+DAXPY:M` and the new declaration is `DY(100+M)`. 6 out of 7 arrays (Figure 5.2) have successfully been resized with appropriate bounds using our first algorithm.

5.2.2 Array Region Based Approach

The array region based approach has been designed to normalize programs where initial declarations are not explicit in the main program or where the main program is missing, i.e for numerical libraries. A typical example is a program where arrays are declared as pointers and dynamically allocated by *malloc*-like functions. In its subroutines, these arrays are passed as arguments but their actual sizes are not always in the argument list.

One solution is to analyze the program behavior and to try to extract information from the dynamic allocation function arguments in order to induce the actual array size

```
      PROGRAM LINPACK
      REAL A(201,200),B(200)
      INTEGER IPVT(200)
      LDA = 201
      N = 100
C     P(LDA,N) {LDA==201, N==100}
      CALL DGESL(A,LDA,N,IPVT,B)
      CALL DGEFA(N,IPVT,B)
      END

      SUBROUTINE DGESL(A,LDA,N,IPVT,B)
      INTEGER LDA,N,IPVT(200)
      REAL A(LDA,200),B(200)
      DO K = 1, N-1
         L = IPVT(K)
C     P(LDA,N,K) {LDA==201, N==100, 1<=K, K<=99}
         CALL DAXPY(N-K,L,A(K+1,K),B(K+1))
      ENDDO
      END

      SUBROUTINE DGEFA(N,IPVT,B)
      INTEGER N,IPVT(200)
      REAL B(200)
      DO K = 1, N-1
         L = IPVT(K)
C     P(N,K) {N==100, 1<=K, K<=99}
         CALL DAXPY(N-K,L,B,B(K+1))
      ENDDO
      END

      SUBROUTINE DAXPY(M,DA,DX,DY)
      REAL DX(*),DY(M+100),DA
      DO I = 1,M
         DY(I) = DY(I) + DA*DX(I)
      ENDDO
      END
```

Figure 5.2: New declarations obtained with the argument association rules approach for Linpack excerpt (6 bounds found, 1 failure)

and then use the argument association rules approach to propagate the information in the call tree. However, the allocation space computation often depends on the element and operator types so this solution seems unrealistic. Furthermore, when the main program itself is missing or when the programming languages support dynamic resizing, we need another method to find actual array accesses.

The array region approach computes, without knowledge of the initial array declarations, adequate declarations based on convex array region information. As described in Section 4.5, the region of an array A of n dimensions at a statement s is a function from the set of states to the power set of $\mathbb{Z}^n$:

$$\begin{aligned} \mathcal{R} : Statement \longrightarrow \quad & State \longrightarrow \wp(\mathbb{Z}^n) \\ \mathcal{R}(s) \longmapsto \quad & \lambda\sigma.(\{\phi = (\phi_1, \ldots, \phi_n) \in \mathbb{Z}^n : r(\phi, \sigma)\}) \end{aligned}$$

To find a new value for the last upper bound of an ugly or assumed-size array A, we need to compute $\max\{\phi_n : \phi = (\phi_1, \ldots, \phi_n) \in \mathcal{R}(s)(\sigma)\}$ where statement s is the body of the current procedure. The array regions of this statement are calculated incrementally from elementary statement nodes to the largest compound statement nodes in the hierarchical control flow graph. Algorithm 5.2.2 contains the different steps necessary to compute this maximum value.

Algorithm 5.2.2

procedure Array_Region_Based_Array_Resizing(p)
    p : current procedure
begin
    s := module statement of p
    for each ugly or assumed-size array a in p
        merge the read and write regions of a at s
        find upper bounds of ϕ_n in the merged region
        if there exists a maximum value of these upper bounds **then**
            last_upper_bound(b) := maximum value
        else
            last_upper_bound(b) := assumed-size
        endif
    endfor
end

If the system has several upper bounds and their relationships are known at compile time, preconditions are taken into account to find the maximum value. As in the first approach, the implementation returns the assumed-size * when the maximum operator fails. By applying this algorithm, array regions allow resizing 6 out of 7 arrays for the example in Figure 5.1. However, in subroutine `DGESL`, as array `A` is reshaped by the call to `DAXPY` (the

```
      SUBROUTINE DGESL(A,LDA,N,IPVT,B)
      INTEGER LDA,N,IPVT(99)
      REAL A(LDA,*),B(100)
C     <A(PHI1,PHI2)-READ-MAY-{1<=PHI1, PHI1<=201, 1<=PHI2}>
C     <B(PHI1)-READ-EXACT-{2<=PHI1, PHI1<=100}>
C     <B(PHI1)-WRITE-EXACT-{2<=PHI1, PHI1<=100}>
C     <IPVT(PHI1)-READ-EXACT-{1<=PHI1, PHI1<=99}>
      DO K = 1, N-1
         L = IPVT(K)
         CALL DAXPY(N-K,L,A(K+1,K),B(K+1))
      ENDDO
      END

      SUBROUTINE DGEFA(N,IPVT,B)
      INTEGER N,IPVT(99)
      REAL B(100)
C     <B(PHI1)-READ-MAY-{1<=PHI1, PHI1<=100}>
C     <B(PHI1)-WRITE-EXACT-{2<=PHI1, PHI1<=100}>
C     <IPVT(PHI1)-READ-EXACT-{1<=PHI1, PHI1<=99}>
      DO K = 1, N-1
         L = IPVT(K)
         CALL DAXPY(N-K,L,B,B(K+1))
      ENDDO
      END

      SUBROUTINE DAXPY(M,DA,DX,DY)
      REAL DX(M),DY(M),DA
C     <DX(PHI1)-READ-EXACT-{1<=PHI1, PHI1<=M}>
C     <DY(PHI1)-READ-EXACT-{1<=PHI1, PHI1<=M}>
C     <DY(PHI1)-WRITE-EXACT-{1<=PHI1, PHI1<=M}>
      DO I = 1,M
         DY(I) = DY(I) + DA*DX(I)
      ENDDO
      END
```

Figure 5.3: New declarations obtained with the array region based approach for Linpack excerpt (6 bounds found, 1 failure)

corresponding formal array `DX` is one dimension array), information about array regions is lost and we cannot compute a new value for the upper bound of `A`. The results are shown in Figure 5.3.

The efficiency of this approach depends on the accuracy of the array region analysis. The execution time is longer than with the first approach because of feasibility tests on linear systems to compute array regions. But if regions are computed before hand, execution times are not so different. The global execution time and the accuracy could be improved using more sophisticated implementation of array regions, such as dimension by dimension regions, guarded regions [GLL95, NGL95], list of regions [GL00] or linear memory access descriptors [PHP98, HPY01]. However, in most practical cases, the required time is acceptable and good results for array resizing using this second algorithm are reported in Section 5.3.

5.2.3 Code Instrumentation

As we have seen, not all assumed-size arrays are resized by the argument association rules or array region based approach. Code instrumentation is used to complete these two approaches. It instruments the initial code with new common variables for each array bound and assignments on these variables. In principle, the method is similar to the first one because it is also based on the relationship between actual and formal array arguments. The difference is that it uses common variables to communicate dynamically between the calling and called procedures. Derived from Equations 5.1 and 5.2 (page 59), the value assigned to the common variable is the right hand side expression:

$$\text{B\_PIPS} \; = e_a.\prod_{i=1}^{n} d_{ai} \text{ or } e_a.\left(\prod_{i=1}^{n} d_{ai} - \sum_{i=1}^{n}\left((s_{ai}-l_{ai})\prod_{j=1}^{i-1} d_{aj}\right)\right)$$

and the new value of the last upper bound of the ugly or assumed-size array B is:

$$\text{B\_PIPS} \;/\left(e_b.\prod_{i=1}^{m-1} d_{bi}\right)$$

Code instrumentation is more expensive than the other approaches because of increased code size and execution time. It should only be applied when both static approaches fail. However, there is always a trade-off among different methods, since the compilation time of the code instrumentation is much faster than the others, preconditions and array regions are not necessary. A general algorithm for the code instrumentation approach is given by Algorithm 5.2.3.

Algorithm 5.2.3

procedure Array_Resizing_by_Code_Instrumentation(p)

p : current procedure

begin
  for each ugly or assumed-size array b in p
    insert a declaration for new common variable b_pips in p
    s_2 := size_of_formal_array_except_last_dimension(b, p)
    insert the last upper bound $b\_pips/(e_b * s_2)$
    for each call site c to p
      q := corresponding_caller(c)
      insert a declaration for common variable b_pips in q
      a := corresponding_actual_array_argument(b, c)
      s_1 := size_subscript_value_of_actual_array(a, c, q)
      insert an assignment $b\_pips = e_a * s_1$ before c
    endfor
  endfor
end

This algorithm also traverses the call graph in the invocation order, new array sizes are propagated from the main program to the called procedures. All the pointer-typed arrays are resized, so program verifications such as array bound checking on these ugly and assumed-size arrays can be done. However, this is a naive instrumentation, since no available compile-time information is taken into account. The derived array declarations are symbolic expressions whose value is not known at compile-time. If combined with the other approaches, it becomes much more effective and 100% arrays are resized with better results. Figure 5.4 illustrates the result of combining code instrumentation with the argument association rules approach for the example in Figure 5.1.

5.2.4 Extension of Approaches

The three methods for array resizing have been implemented in PIPS. As explained, these methods compute a proper upper bound for the last dimension of all ugly and assumed-size formal arrays. However, new array bounds can be calculated not only for pointer-typed arrays but also for other formal array arguments. In addition, the array region based approach can compute new bounds for local array arguments. This is useful because sometimes array declarations do not represent the actual size of arrays exactly. Declaring local arrays as pointers and then allocating their memory spaces also happen in some existing FORTRAN 77 programs using proprietary language extensions. In the debugging or reverse engineering process, one might need to modify the code to have more appropriate declarations.

Furthermore, the main program is not needed in the second approach. In addition, for the argument association based approaches, one can decide to use information from the main program or not. If this information is used, modules that are never called by the main program are not taken into account. This is normal because ugly and assumed-size

```
SUBROUTINE DGESL(A,LDA,N,IPVT,B)
INTEGER DX_PIPS
COMMON /DX_PIPS_COMMON/ DX_PIPS
INTEGER LDA,N,IPVT(200)
REAL A(LDA,200),B(200)
DO K = 1, N-1
   L = IPVT(K)
   DX_PIPS = 200*LDA - 202*K + 201
   CALL DAXPY(N-K,L,A(K+1,K),B(K+1))
ENDDO
END

SUBROUTINE DGEFA(N,IPVT,B)
INTEGER DX_PIPS
COMMON /DX_PIPS_COMMON/ DX_PIPS
INTEGER N,IPVT(200)
REAL B(200)
DO K = 1, N-1
   L = IPVT(K)
   DX_PIPS = 200
   CALL DAXPY(N-K,L,B,B(K+1))
ENDDO
END

SUBROUTINE DAXPY(M,DA,DX,DY)
INTEGER DX_PIPS
COMMON /DX_PIPS_COMMON/ DX_PIPS
REAL DX(DX_PIPS),DY(M+100),DA
DO I = 1,M
   DY(I) = DY(I) + DA*DX(I)
ENDDO
END
```

Figure 5.4: New declarations with the argument association rules and code instrumentation approaches

array declarations in these modules do not affect analyses on program execution. The different options, *i*) choosing which kind of array to resize and *ii*) using the main program or not, are available in PIPS.

5.3 Experimental Results

The percentages of arrays resized by our methods for 18 benchmarks from Linpack, PerfectClub and SPEC95 CFP are shown in Table 5.3. The second column presents the total number of ugly and assumed-size array declarations. The third, fifth and seventh columns show, respectively the numbers of actual resized arrays by the argument association rules (AAR), array region based (ARB) and code instrumentation (CI) approaches.

Table 5.3: Linpack, PerfectClub, SPEC95 CFP: number and percentage of resized arrays by the argument association rules, the array region based, the code instrumentation and the combined approaches.

Benchmark		**AAR**		**ARB**		**CI**		**AAR + ARB**	
Name	**1+***	**Num**	**Per**	**Num**	**Per**	**Num**	**Per**	**Num**	**Per**
linpack	16	10	62.50%	11	68.75%	16	100%	14	87.50%
bdna	191	172	90.05%	46	24.08%	191	100%	186	97.38%
dyfesm	19	6	31.58%	1	5.26%	19	100%	6	31.58%
flo52	18	18	100.00%	3	16.67%	18	100%	18	100.00%
mdg	29	19	72.50%	12	41.38%	29	100%	21	72.41%
mg3d	68	54	79.41%	29	42.65%	68	100%	62	91.18%
ocean	55	47	85.45%	17	30.91%	55	100%	50	90.91%
qcd	11	5	45.45%	8	72.73%	11	100%	10	90.91%
spc77	11	8	72.73%	8	72.73%	11	100%	11	100.00%
spice	34	1	2.94%	19	56.00%	34	100%	20	58.82%
trfd	13	3	23.08%	5	38.46%	13	100%	6	46.15%
su2cor	5	5	100.00%	1	20.00%	5	100%	5	100.00%
hydro2d	7	7	100.00%	1	14.29%	7	100%	7	100.00%
applu	18	17	94.44%	13	72.22%	18	100%	18	100.00%
turb3d	58	45	77.59%	35	60.34%	58	100%	45	77.59%
apsi	437	8	1.83%	211	48.28%	437	100%	219	50.11%
fpppp	19	8	42.11%	3	15.97%	19	100%	8	42.11%
wave5	70	39	55.71%	26	37.14%	70	100%	62	88.57%

5.3.1 Approach Comparison

Since the results of the code instrumentation method, in spite of 100% resized arrays, are not very interesting, we compare only the first and the second static methods. Among

18 benchmarks, the first approach gives 12 better, 1 equal and 5 worse results. It is specially good for *bdna, flo52, ocean, su2cor, hydro2d* and *applu* with more than 80% resized arrays. In benchmarks *spice, trfd* and *apsi*, it is not whole arrays but array elements that are passed in most procedure calls. Some variables in the array size and subscript value expressions cannot be translated into the callee's space name. Or there are several calls to the same procedure from different callers and the computed values for the last upper bound are different, as in Figure 5.1 example. Furthermore, information is lost right after the beginning of the invocation order traversal on the call graph so the results for these three benchmarks are not so good. In fact, the variables cannot be translated into the callee's frame because they are neither in the argument list nor in the common blocks, and the preconditions for non-linear expressions and indirections are not available. This argument association rules approach could be improved by using more accurate precondition analyses. The average percentage of resized arrays is 62.80% with the current implementation.

The second approach to array resizing is more time-consuming and the average number of resized arrays is smaller than with the first one, 40.65%. The reasons and possible improvements are mentioned in Section 5.2. However, we have about 70% resized arrays for *linpack, qcd, spc77, applu* and much better results for *spice* and *apsi* where the first algorithm does not work. Furthermore, this approach is more effective on programs having non-explicit initial declarations and it is the only solution for libraries.

5.3.2 Combined Approaches

Since neither approach is always superior, it is desirable to combine these approaches to have as many proper array declarations as possible. Different combinations can be achieved by using implemented options in PIPS. If we do not want to use array regions, an expensive analysis, code instrumentation can be applied just after the argument association rules approach. New declarations are only computed for pointer-typed arrays where the association rules approach fails. This combined approach (AAR + CI) is efficient with 100% resized arrays.

Another composed solution (AAR + ARB) has also been implemented: the least expensive, argument association rules analysis, is used first, then the second approach is only applied for subroutines in which precise declarations have not been found. About 80% of ugly and assumed-size arrays are resized (two last columns in Table 5.3). Furthermore, code instrumentation can then be used to have 100% resized arrays (AAR + ARB + CI).

5.4 Conclusion

To our knowledge, there has not been much related work done on array resizing. Our methods have solved the problems of implicit array pointer declarations in FORTRAN scientific codes as discussed in the introduction section. In the worst case, the ugly assumed-size declarations `A(1)` that disable many program analyses and optimizations are replaced by the assumed-size array declarations in the first and second methods. The code instrumentation algorithm can be used to find out new declaration for the remaining assumed-size arrays.

In fact, the returned values of array bounds given by our approaches can be different because they are based on different assumptions. The first and third approaches are *interprocedural analyses* that use the association between two program units. The new array declarations in the called procedure are computed with respect to the declarations in the calling procedure. As a consequence, these new declared ranges can be too large for the actual array accesses or, in contrast, can detect *intraprocedural array bound violations* in the execution of the called procedure.

On the other hand, the second approach is rather an *intraprocedural analysis* because it is based on actual array references in one program unit. The interprocedural propagation of regions is only needed when an array is passed as an actual argument by nested subroutines, but no array elements are accessed in these subroutines except the last one. This method relies on the code execution and there are no more out-of-bound errors for array arguments in the current procedure. But we can always check for *interprocedural array bound violation* (see Chapter 6) in the whole program.

So if the considered program is supposed to be correct, there is no danger if either the first, the second or the third approach to array resizing is applied. Otherwise, if our purpose is to verify the standard conformance of programs, all approaches can be applied and they must be followed by the array bound checking phase (Chapter 6). The interactions between array resizing and array bound checking are discussed later, in Chapter 9.

The first and third approaches are appropriate for FORTRAN with its assumed-size array declarations and they are applicable to other programming languages with the same parameter passing mechanism. We get very good results for Linpack, PerfectClub and SPEC95 CFP benchmarks by using the first approach. A perfect percentage of resized arrays can be achieved by combining these approaches but there is a trade-off between the static and dynamic techniques. Static analyses have been improved in order to reduce as much as possible the overhead caused by code instrumentation.

However, when the array is allocated dynamically and its actual size is difficult to find, or when dealing with languages without variable declarations, the array region based approach is needed to find the explicit declarations. Furthermore, this method can also be applied to other programming languages such as MATLAB, APL and JAVA to improve performance by reducing the execution overhead of dynamic resizing. And, of course, other program analyses and verifications are facilitated, as will be shown in the next chapters.

Chapter 6

ARRAY BOUND CHECKING

Date: Wed, 26 Dec 2001 21:19:22 -0800
From: Henry Baker <hbaker1@pipeline.com>
Subject: "Buffer Overflow" security problems

"If I remove a stop sign and someone is killed in a car accident at that intersection, I can be sued and perhaps go to jail for contributing to that accident. If I lock an exit door in a crowded theater or restaurant that subsequently burns, I face lawsuits and jail time. If I remove or disable the fire extinguishers in a public building, I again face lawsuits and jail time. If I remove the shrouding from a gear train or a belt in a factory, I (and my company) face huge OSHA fines and lawsuits. If I remove **array bounds checks** *from my software, I will get a raise and additional stock options due to the improved "performance" and decreased number of calls from customer service. I will also be promoted, so I can then make sure that none of my reports will check array bounds, either."*

The chapter is organized as follows. Section 6.1 describes the problems of exceeding array bounds and the overhead encountered with bound checking. Section 6.2 discusses the related work on optimization of array range checking and explains what is missing in these results. Section 6.3 presents our first intraprocedural bound checking approach, the elimination of redundant tests, based on preconditions. Section 6.4 presents the second approach, insertion of unavoidable tests, based on array regions. Section 6.5 reports the experimental results and the comparison of these two intraprocedural checking. Section 6.6 introduces a new kind of interprocedural array bound checking that verifies whether the declared size of a formal array is consistent with the size of the corresponding actual parameter, as well as the obtained experimental results. Conclusions are given in Section 6.7.

6.1 Out-of-bound Problems

Array bound checking refers to determining whether all array references in a program are within their declared ranges. These array bound checks may be analyzed intraprocedurally or interprocedurally, depending on the need for safety. Intraprocedural checking refers to verifying array accesses within a procedure unit and the interprocedural one refers to checking the actual/formal array size mismatches, through procedure calls. Such checking is desirable for any program, regardless of the programming language used, since bound violations are among the most common programming errors. Subscripting arrays beyond their declared sizes may result in unexpected results, security holes or failures. For the safety of execution, some languages such as JAVA require that a program only be allowed to access elements of an array that are part of the defined extent of the array.

But on the other hand, bound checking can be very expensive because every array access must be guarded by two bound checks per dimension to assert the legality of the access. This increases the size of the executable file, the compilation time and the execution time. In fact, these two bound checks can be implemented as one unsigned comparison instruction, i.e *if ((unsigned) (ref-low) > up-low) stop*. If the reference is smaller than the lower bound, the unsigned subtraction underflows and creates a very large number, causing the test to fail. However, the overhead still remains, in part because other code transformations or optimizations are prevented by these bound checks.

Some compilers solve this problem by providing the user with a compile-time option to enable or disable the checking. The purpose of this option, as mentioned by Muchnick [Muc97], is to allow users to enable checking in the development and debugging processes, and then, once all the defects are supposedly found and fixed, to turn it off for the production version. However, bound checking is just as important for delivered versions of programs as for development versions because the production ones may have bugs that were not even observed during testing phases. Instead of providing a way to turn bound checking off, what is needed is to optimize it so that it has a minimal overall cost.

This chapter presents two approaches for limiting the number of array bound checks by using the preexisting analyses provided by PIPS. The first technique is based on the elimination of redundant tests, to provide very accurate information about faulty references during development and testing phases. The second one is based on the insertion of unavoidable tests to provide the smallest possible slowdown during the production phase. We also introduce in this thesis a new kind of error checking that verifies whether the declared size of a formal array is consistent with the size of the corresponding actual parameter, for languages with the same parameter passing rules as FORTRAN. This analysis is apparently only addressed in some Salford Fortran compilers and the static analyzer *forcheck*, not in other work, although this conformance test is as important as out-of-bound checking.

In addition, another objective of this work is to see if it is possible to perform efficient range checking by reusing interprocedural analysis techniques already implemented in commercial compilers. Many routines are written to manipulate arrays of arbitrary size, but are used in actual programs only on arrays whose sizes are determined in the main program. So interprocedural analyses that propagate information through procedure boundaries should allow us to eliminate more unnecessary bounds checking and may result in significant speedups. Moreover, interprocedural translation helps to prove the absence of interprocedural array bound violations or to detect them at either compile-time or run-time. If insufficient information is available, run-time checks are generated before each call site to guarantee the program correctness.

6.2 Related Work

The first approach to optimizing array bound checks was developed by Markstein et al. [MCM82] and then refined several times by Asuru [Asu92] and Gupta [Gup90, Gup93], Spezialetti and Gupta [SG95], Kolte and Wolfe [KW95]. They introduce algorithms to reduce the execution overhead of range checks through the elimination and propagation of bound checks by using data flow analysis. Their techniques became more and more sophisticated in order to improve results. In [Gup93], a bound check that is identical or subsumed by other bound checks is suppressed in a local elimination. In a global elimination, an algorithm first modifies bound checks to create additional redundant checks and then carries out the elimination of redundant checks by using the notions of *available* and *very busy* checks. These two algorithms use backward and forward data flow analyses to solve the problems. For the propagation of checks out of loops, R. Gupta identifies candidates for propagation which include invariants, loops with increment or decrement of one, and variables of increasing or decreasing values. Then he uses check hoisting to move checks out of loops. However, the results are not very convincing because of the small size of his examples and because the optimizations have only been applied by hand.

In [KW95], Kolte and Wolfe rely on a check implication graph where nodes are sets of range checks in canonical form and edges denote implications among these families. As in [Gup93], they also compute the *available* and *anticipatable* checks by solving forward and backward data flow problems. To create more redundant checks, there are five schemes to insert checks at safe and profitable program points: *no-insertion, safe-earliest, latest-not-isolated placement, check-strengthening and preheader insertion*. They use partial redundancy elimination after determining the most efficient places to move bound checks to. Their implementation in the FORTRAN compiler Nascent [KW95] with different techniques in range check optimization leads to experimental results that show the necessity of range check optimization, the effectiveness and cost of these optimizations. However, high percentages, even 99.99%, of eliminated tests do not always mean faster execution times. This article lacks comparisons between the execution times of codes with and without optimized bound checks to show the impact of removed checks. Furthermore, there are no mentions of bound violations in PerfectClub (*mdg, spc77, trfd*) and Riceps benchmarks (*linpackd*) which are caused by ugly assumed-size array declaration (see Chapter 5, page 53).

Suzuki and Ishihata [SI77] implemented a system that inserts logical assertions before array element accesses and then uses theorem proving techniques to verify the absence of array range violations. Such techniques are often expensive and are restricted to programs written in a structured manner, i.e without goto statements.

The abstract interpretation approach proposed by Cousots and Halbwachs [CC76, CH78] considers array range checking as an example of the automated verification of execution properties of programs. As in [Har77, Wel78, Hal79, SKL88, RR00], they use static data flow analysis information to prove at compile-time that an array bound violation cannot occur at run-time and that the test for this violation is unnecessary. Their algorithms for propagating and combining assertions depend on the different rules they use. Since the algorithms in the abstract interpretation and the program verification approaches do not perform any insertion of checks in the program to create more redundant checks, they could only take advantage of completely redundant checks. So the run-time overhead of the partial redundant checks that cannot be evaluated at compile-time still remains. For instance, Rugina and Rinard [RR00] proposed a novel framework for the symbolic bounds analysis that characterizes the regions of memory accessed by statements and procedures. If these regions fall within array bounds, unnecessary array bounds checks are eliminated. However, as said in the paper, they have not attempted to eliminate partially redundant checks or move checks to less frequently executed program points. Earlier in the literature are the work of Harrison [Har77] and Welsh [Wel78] that used value range propagation to eliminate redundant tests and verify the program correctness. Also present in this approach is the model checking group, Delzanno et al. [DJP00], which uses fix point acceleration techniques to help the automated verification of programs.

Pointer, string and array access errors in C are also studied in many papers [Ste92, HJ92, ABS94, JK97, PF97, NL98, Wag00, DRS01, KDA02, EL02]. Lightweight static analyses are used to detect buffer overflow vulnerabilities, but at the expense of soundness and completeness [Wag00, EL02]. These analyzers will sometimes generate false warnings, or even miss real problems. The overheads of run-time safety checking reported in some research papers are significant, ranging from 130% to 540% by [ABS94], about five times by Purify [HJ92], etc. In addition, Purify cannot detect if we access past the end of an array into the region of the next variable. The certifying compiler of Necula and Lee [NL98] does not success in eliminating bound checks when the size of the formal array in a called procedure is unknown. Kowshik et al. [KDA02] introduce a language called Control-C with key restrictions to ensure that memory safety of code can be verified entirely by static checking. However, their assumptions are restrictive, and are only suitable for a small class of specific purpose languages, i.e for real-time control systems.

Other articles of Midkiff, Moreira et al. [MMS98, MMG00, MMG$^+$01] describe another approach to optimize array reference checking in JAVA programs. It is based on code replication. Loop iteration spaces are partitioned into regions with different access violation characteristics. In unsafe regions, run-time tests are performed, whereas in other regions they are not necessary because all array indices are guaranteed to be within bounds. The optimizations differ on their level of refinement and practicality. These techniques are less complicated than the abstract interpretation approach while still being effective. However, they do not use any control-flow analysis to reduce code replication, and the optimizations here are mainly for JAVA applications because of its precise exception semantics. Another approach for JAVA, based on an extended Static Single-Assignment graph representation, the Eliminating Array Bound Checks on Demand by Bodik et al. [BGS00], can remove about 45% of dynamic bound checks of a representative set of JAVA programs. Aggarwal and Randall [AR01a] use the related field analysis to remove about 50% array bound checks that have already optimized by other simple optimizations. Qian et al. [QHV02] describe a bound check elimination algorithm which combines three analyses: variable constraint, array field and rectangular array analysis to prove the safety of array references. JAVA virtual machines can use this information to avoid emitting bound checks for these references.

Although there are many different techniques for array bound checking optimization, we can partition them into two main approaches. The first approach puts array bound checks at every array reference and removes a check if it is redundant [MCM82, Gup90, Asu92, Gup93, KW95]. In the second approach, array bound checks are put at places where it is not possible to prove them useless [CC76, SI77, Har77, CH78, Wel78, SKL88, MMS98, RR00]. The first approach attempts to reduce the dynamic and static numbers of bound tests and the overhead induced by a test even if it cannot be eliminated. This is done by determining if a test is subsumed by another test, so that it can be eliminated. Hoisting range checks out of loops is also applied when it is possible. The analyses are simple or sophisticated depending on each technique. In the second approach, by using data flow information, if it is proven that no array bound violation will occur at runtime in some region of code, tests are unnecessary for this region. If it is proven that an access violation might occur, tests are generated as needed. The number of generated tests is limited; range checks are put only where there might be bound violations. But the difficulty of this approach is that the information needed to prove that no violation will occur may not be available at compile-time. Then tests may remain inside inner loops.

The amount of information about the array bound violation is never discussed in the above papers. It is often reduced to *a violation occurred* with no information about the array accessed nor the statement where the array element was referenced. Especially with code hoisting, the information cannot always be preserved.

So both approaches have advantages and drawbacks when comparing the number of needed transformations and analyses as well as information about array violation. A goal of our work here is to compare the effectiveness and optimization costs of two different algorithms for intraprocedural array bound checking. The first one is based on test elimination without hoisting, and the second one is based on optimized test insertion without code replication. The amount of information preserved by each approach is also reported. The two algorithms for intraprocedural bound checking are described in the next two sections.

6.3 Elimination of Redundant Tests

Let $\mathcal{VC}$ be the semantic function that represents the array bound violation condition. It associates to each statement a set of program states that cause array bound violations.

$$\mathcal{VC} : Statement \longrightarrow \wp(State)$$

As defined in Chapter 4, the precondition of a statement and its over approximation are also functions from the set of statements to the power-set of the program state set:

$$\mathcal{P}, \overline{\mathcal{P}} : Statement \longrightarrow \wp(State)$$

We have the two following lemmas:

6.3.1 Static Safety and Error

Lemma 6.3.1 For any statement, if the intersection of the over-approximation precondition and the violation condition is empty then there is no bound violation caused by this statement.

Proof: Let s be the concerned statement, the hypotheses implies:

$$\overline{\mathcal{P}}(s) \cap \mathcal{VC}(s) = \emptyset \tag{6.1}$$

On the other hand, following the definition of the over-approximation precondition:

$$\mathcal{P}(s) \subseteq \overline{\mathcal{P}}(s) \tag{6.2}$$

From 6.1 and 6.2, we have:

$$\mathcal{P}(s) \cap \mathcal{VC}(s) = \emptyset$$

which means that any program state reaching s does not belong to the set of states causing bound violations at s. So there is no bound violation caused by this statement. □

Lemma 6.3.2 For any statement, if the intersection of the over-approximation precondition and the negation of the violation condition is empty then there is surely a bound violation caused by this statement.

Proof: Let s be the concerned statement, the hypotheses implies:

$$\overline{\mathcal{P}}(s) \cap \neg\mathcal{VC}(s) = \emptyset \Longrightarrow \overline{\mathcal{P}}(s) \subseteq \mathcal{VC}(s) \tag{6.3}$$

On the other hand:

$$\mathcal{P}(s) \subseteq \overline{\mathcal{P}}(s) \tag{6.4}$$

From 6.3 and 6.4, we have:

$$\mathcal{P}(s) \subseteq \mathcal{VC}(s)$$

which means that every program state reaching s is in the set of states that cause bound violations at s. So there is certainly a bound violation caused by this statement. □

6.3.2 Elimination Algorithm

Our first implementation of an intraprocedural range check optimizer consists of two phases: *bound checks generation* and *redundancy elimination*. Algorithm 6.3.1 describes the elimination of redundant tests, based on the two above lemmas.

Algorithm 6.3.1

procedure Elimination_of_Redundant_Tests(p)
  p : current procedure
begin
  for each statement s of p
    for each array reference $A(s_1, \dots, s_n)$ in s
      for each dimension i
        l_i = lower_bound_dimension(A, i)
        u_i = upper_bound_dimension(A, i)
        insert IF $(s_i < l_i)$ $STOP$ $message$ before s
        insert IF $(s_i > u_i)$ $STOP$ $message$ before s
      endfor
    endfor
  endfor
  compute preconditions for modified p
  for each inserted test statement t with condition e of p
    P = precondition(t)
    $sc_1 = P \cap \{e\}$
    $sc_2 = P \cap \{\neg e\}$
    switch
      case infeasible(sc_1) :
        no bound violation, remove t from p
      case infeasible(sc_2) :
        bound violation found at compile-time
      default : /* indeterminable, keep the dynamic check */
    endswitch
  endfor
end

In the generation phase, only non-trivial bound checks are generated. Trivial tests which are always false such as `2 < 1` or `N-1 > N` are not taken into account. Each bound check is accompanied with a stop message and if a bound violation is detected, the message tells the user in which array, on which dimension, which bound, and in which line the subscript is out of range.

When computing the preconditions for the new code, we have two options: intraprocedural and interprocedural analyses. Information provided by these preconditions is used to detect bound violations or eliminate redundant checks. The precondition of a given statement incorporates information propagated from the test condition of the bound checking statements inserted before this statement. For each bound check, we test the feasibility of the system built from it and the precondition. The feasibility test of a system of constraints is implemented in PIPS by using the Simplex and Fourier-Motzkin algorithms (see Section 4, page 35). There are three possibilities:

1. If the system is infeasible, the bound check is false and is removed from the test condition.

2. If the bound check is true with respect to the precondition (the system built from the precondition and the negation of the bound check is infeasible), a bound violation is detected at compile-time.

3. Otherwise, the bound check is preserved.

```
PROGRAM SHALOW
PARAMETER (N1=513, N2=513)
COMMON U(N1,N2)
COMMON /CONS/ M,N
CALL INITAL
MN = MIN(M,N)
UCHECK = 0.0
DO I = 1, MN
   DO J = 1, MN
      UCHECK = UCHECK + U(I,J)
   ENDDO
ENDDO
END
```

```
SUBROUTINE INITAL
PARAMETER (N1=513, N2=513)
COMMON U(N1,N2)
COMMON /CONS/ M,N
READ (5,*) M, N
IF (M.LE.512.AND.N.LE.512) THEN
   DO I = 1, M
      DO J = 1, N
         U(I,J) = I*J
      ENDDO
   ENDDO
   U(M+1,N+1) = U(1,1)
ELSE
   STOP
ENDIF
END
```

Figure 6.1: Array bound checking example (excerpt from *swim*, SPEC95 CFP)

Elimination Example

The running example in Figure 6.1 is used to illustrate our two intraprocedural bound check optimizers. This example is extracted from *swim*, a weather prediction program in the SPEC95 CFP benchmarks with procedure calls and array references, which are used to show the effect of interprocedural analyses.

After the bound check generation, the instrumented code with intraprocedural and interprocedural preconditions are represented, respectively in Figure 6.2 + 6.3, and Figure 6.5. The preconditions are in comment lines. Some trivial checks that are never true such as `1.LT.1` and `1.GT.513` are not generated by our bound checker. The codes after the redundancy elimination for the intraprocedural and interprocedural versions are given in Figure 6.4 and Figure 6.6, respectively. Since there is no difference for module `INITAL`, it is omitted in the interprocedural version (Figure 6.5 and Figure 6.6).

```
      PROGRAM SHALOW
C     P() {}
      CALL INITAL
C     P(M,N) {}
      MN = MIN(M, N)
C     P(M,MN,N) {MN<=M, MN<=N}
      UCHECK = 0.0
C     P(M,MN,N) {MN<=M, MN<=N}
      DO I = 1, MN
C     P(I,J,M,MN,N) {1<=I, I<=MN, N<=M, MN<=N}
         DO J = 1, MN
C     P(I,J,M,MN,N) {1<=I, I<=MN, 1<=J, J<=MN, MN<=M, MN<=N}
            IF (J.LT.1) STOP "Bound violation:array U,lower dimension 2,line 10"
C     P(I,J,M,MN,N) {1<=I, I<=MN, 1<=J, J<=MN, MN<=M, MN<=N}
            IF (J.GT.N2) STOP "Bound violation:array U,upper dimension 2,line 10"
C     P(I,J,M,MN,N) {1<=I, I<=MN, 1<=J, J<=513, J<=MN, MN<=M, MN<=N}
            IF (I.LT.1) STOP "Bound violation:array U,lower dimension 1,line 10"
C     P(I,J,M,MN,N) {1<=I, I<=MN, 1<=J, J<=513, J<=MN, MN<=M, MN<=N}
            IF (I.GT.N1) STOP "Bound violation:array U,upper dimension 1,line 10"
C     P(I,J,M,MN,N) {1<=I, I<=513, I<=MN, 1<=J, J<=513, J<=MN, MN<=M, MN<=N}
            UCHECK = UCHECK + U(I,J)
         ENDDO
      ENDDO
      END
```

Figure 6.2: Elimination of Redundant Tests with Intraprocedural Preconditions: code with generated bound checks and preconditions (declarations omitted)

In both versions, the preconditions `{1<=I, 1<=J}` after entering the loops in `SHALOW` and `INITAL` allow us to remove all lower bound checks. The difference between the in-

```
      SUBROUTINE INITAL
      READ (5,*) M, N
C     P(M,N) {}
      IF (M.LE.512.AND.N.LE.512) THEN
C     P(M,N) {M<=512, N<=512}
         DO I = 1, M
C     P(I,J,M,N) {1<=I, I<=M, M<=512, N<=512}
            DO J = 1, N
C     P(I,J,M,N) {1<=I, I<=M, 1<=J, J<=N, M<=512, N<=512}
               IF (J.LT.1) STOP "Bound violation:array U,lower dimension 2,line 22"
C     P(I,J,M,N) {1<=I, I<=M, 1<=J, J<=N, M<=512, N<=512}
               IF (J.GT.N2) STOP "Bound violation:array U,upper dimension 2,line 22"
C     P(I,J,M,N) {1<=I, I<=M, 1<=J, J<=513, J<=N, M<=512, N<=512}
               IF (I.LT.1) STOP "Bound violation:array U,lower dimension 1,line 22"
C     P(I,J,M,N) {1<=I, I<=M, 1<=J, J<=513, J<=N, M<=512, N<=512}
               IF (I.GT.N1) STOP "Bound violation:array U,upper dimension 1,line 22"
C     P(I,J,M,N) {1<=I, I<=513, I<=M, 1<=J, J<=513, J<=N, M<=512, N<=512}
               U(I,J) = I*J
            ENDDO
         ENDDO
C     P(I,J,M,N) {1<=I, I<=513, M+1<=I, N<=512}
         IF (N+1.LT.1) STOP "Bound violation:array U,lower dimension 2,line 25"
C     P(I,J,M,N) {1<=I, I<=513, M+1<=I, 0<=N}
         IF (N+1.GT.N2) STOP "Bound violation:array U,upper dimension 2,line 25"
C     P(I,J,M,N) {1<=I, I<=513, M+1<=I, 0<=N, N<=512}
         IF (M+1.LT.1) STOP "Bound violation:array U,lower dimension 1,line 25"
C     P(I,J,M,N) {1<=I, I<=513, M+1<=I, 0<=M, 0<=N, N<=512}
         IF (M+1.GT.N1) STOP "Bound violation:array U,upper dimension 1,line 25"
C     P(I,J,M,N) {1<=I, I<=513, M+1<=I, 0<=M, M<=512, 0<=N, N<=512}
         U(M+1,N+1) = U(1,1)
      ELSE
C     P(M,N) {}
         STOP
      ENDIF
      END
```

Figure 6.3: Elimination of Redundant Tests with Intraprocedural Preconditions: code with generated bound checks and preconditions (declarations omitted)

traprocedural and the interprocedural options of preconditions is that in Figure 6.2, after the call to INITAL, we have an empty postcondition while, in Figure 6.5, we have P(M,N) {0<=M, M<=512, 0<=N, N<=512}. This more precise postcondition helps to eliminate all upper bound checks (I.GT.N1 and J.GT.N2) in the nested loop in SHALOW (Figure 6.6). Here we see the strength of the interprocedural precondition analysis because in the intraprocedural option, bound checks are left inside loops (Figure 6.4), as long as code hoisting has not been applied. However, for the subroutine INITAL, since we have no information about the lower bound of variables M and N, the lower bound tests remain.

```
PROGRAM SHALOW
CALL INITAL
MN = MIN(M, N)
UCHECK = 0.0
DO I = 1, MN
   DO J = 1, MN
      IF (J.GT.N2) STOP "Bound violation:array U,upper dimension 2,line 10"
      IF (I.GT.N1) STOP "Bound violation:array U,upper dimension 1,line 10"
      UCHECK = UCHECK + U(I,J)
   ENDDO
ENDDO
END

SUBROUTINE INITAL
READ (5,*) M, N
IF (M.LE.512.AND.N.LE.512) THEN
   DO I = 1, M
      DO J = 1, N
         U(I,J) = I*J
      ENDDO
   ENDDO
   IF (N+1.LT.1) STOP "Bound violation:array U,lower dimension 2,line 25"
   IF (M+1.LT.1) STOP "Bound violation:array U,lower dimension 1,line 25"
   U(M+1,N+1) = U(1,1)
ELSE
   STOP
ENDIF
END
```

Figure 6.4: Elimination of Redundant Tests with Intraprocedural Preconditions: code after redundancy elimination (declarations omitted)

After the redundancy elimination transformation, the number of generated tests is greatly reduced. Programs are instrumented with bound checks and then compiled and executed using their standard input data sets to detect out-of-bound errors.

```
      PROGRAM SHALOW
C     P() {}
      CALL INITAL
C     P(M,N) {0<=M, M<=512, 0<=N, N<=512}
      MN = MIN(M, N)
C     P(M,MN,N) {0<=M, M<=512, MN<=M, MN<=N, 0<=N, N<=512}
      UCHECK = 0.0
C     P(M,MN,N) {0<=M, M<=512, MN<=M, MN<=N, 0<=N, N<=512}
      DO I = 1, MN
C     P(I,J,M,MN,N) {1<=I, I<=MN, 0<=M, M<=512, MN<=M, MN<=N, 0<=N, N<=512}
         DO J = 1, MN
C     P(I,J,M,MN,N) {1<=I,I<=MN,1<=J,J<=MN,0<=M,M<=512,MN<=M,MN<=N,0<=N,N<=512}
            IF (J.LT.1) STOP "Bound violation:array U,lower dimension 2,line 10"
C     P(I,J,M,MN,N) {1<=I, I<=MN, 1<=J, J<=MN, M<=512, MN<=M, MN<=N, N<=512}
            IF (J.GT.N2) STOP "Bound violation:array U,upper dimension 2,line 10"
C     P(I,J,M,MN,N) {1<=I, I<=MN, 1<=J, J<=MN, M<=512, MN<=M, MN<=N, N<=512}
            IF (I.LT.1) STOP "Bound violation:array U,lower dimension 1,line 10"
C     P(I,J,M,MN,N) {1<=I, I<=MN, 1<=J, J<=MN, M<=512, MN<=M, MN<=N, N<=512}
            IF (I.GT.N1) STOP "Bound violation:array U,upper dimension 1,line 10"
C     P(I,J,M,MN,N) {1<=I, I<=MN, 1<=J, J<=MN, M<=512, MN<=M, MN<=N, N<=512}
            UCHECK = UCHECK + U(I,J)
         ENDDO
      ENDDO
      END
```

Figure 6.5: Elimination of Redundant Tests with Interprocedural Preconditions: code with generated bound checks and preconditions (declarations omitted)

```
      PROGRAM SHALOW
      CALL INITAL
      MN = MIN(M, N)
      UCHECK = 0.0
      DO I = 1, MN
         DO J = 1, MN
            UCHECK = UCHECK + U(I,J)
         ENDDO
      ENDDO
      END
```

Figure 6.6: Elimination of Redundant Tests with Interprocedural Preconditions: code after redundancy elimination (declarations omitted)

6.4 Insertion of Unavoidable Tests

The second intraprocedural array bound checker is based on the convex array region analysis. As defined in Section 4.5, the region of an array A of n dimensions at a statement s is a function from the set of states to the power-set of $\mathbb{Z}^n$:

$$\begin{aligned} \mathcal{R} : \mathit{Statement} \longrightarrow &\quad \mathit{State} \longrightarrow \wp(\mathbb{Z}^n) \\ \mathcal{R}(s) \longmapsto &\quad \lambda\sigma.(\{\phi = (\phi_1, \dots, \phi_n) \in \mathbb{Z}^n : r(\phi, \sigma)\}) \end{aligned}$$

and we introduce the `MAY` and `MUST` approximations:

$$\begin{aligned} \overline{\mathcal{R}} &: \mathit{Statement} \longrightarrow \mathit{State} \longrightarrow \wp(\mathbb{Z}^n) \\ \underline{\mathcal{R}} &: \mathit{Statement} \longrightarrow \mathit{State} \longrightarrow \wp(\mathbb{Z}^n) \\ \underline{\mathcal{R}}(s)(\sigma) &\subseteq \mathcal{R}(s)(\sigma) \subseteq \overline{\mathcal{R}}(s)(\sigma) \end{aligned}$$

The violation condition of an array A associates to each statement a set of program states that cause bound violations of this array:

$$\mathcal{VC} : \mathit{Statement} \longrightarrow \wp(\mathit{State})$$

In order to associate the two above analyses, we define $\mathcal{VC}$ in more detail by using the region vector ϕ:

$$\mathcal{VC}(s) = \{\sigma : \exists\phi = (\phi_1, \dots, \phi_n) \in \mathbb{Z}^n\ \exists i = 1, n\ \ r(\phi, \sigma) \wedge (\phi_i < \mathcal{E}(l_i)(\sigma) \vee \phi_i > \mathcal{E}(u_i)(\sigma))\}$$

r is the relationship existing between ϕ and the current program state (see Chapter 4, page 45), defined by array region. l_i and u_i are, respectively, the lower and upper bound of the dimension i of array A. $\mathcal{E}(e)$ is the function returning the value of an expression e in a program state σ (see Chapter 4, page 37).

Given a procedure, the declaration of an array can be defined as:

$$\begin{aligned} \mathcal{DEC} : \mathit{State} &\longrightarrow \wp(\mathbb{Z}^n) \\ \sigma &\longmapsto \{\phi = (\phi_1, \dots, \phi_n) \in \mathbb{Z}^n : \forall i = 1, n\ \ \mathcal{E}(l_i)(\sigma) \le \phi_i \le \mathcal{E}(u_i)(\sigma)\} \end{aligned}$$

We have the two following lemmas:

6.4.1 Static Safety and Error

Lemma 6.4.1 For any statement, if the over-approximated region of an array is included in the declared dimensions of the array then there is no bound violation of the array caused by this statement.

Proof: Let s be the statement and A the array. For any program state σ reaching s, we have:

$$\overline{\mathcal{R}}(s)(\sigma) \subseteq \mathcal{DEC}(\sigma) \tag{6.5}$$

On the other hand, following the definition of the over-approximation region:

$$\mathcal{R}(s)(\sigma) \subseteq \overline{\mathcal{R}}(s)(\sigma) \tag{6.6}$$

From 6.5 and 6.6, we have:

$$\begin{gathered}\mathcal{R}(s)(\sigma) \subseteq \mathcal{DEC}(\sigma)\\ \text{or } \forall\sigma\ \forall\phi = (\phi_1, \dots, \phi_n) \in \mathbb{Z}^n : r(\phi, \sigma) \Longrightarrow \forall i = 1, n\ \ \mathcal{E}(l_i)(\sigma) \le \phi_i \le \mathcal{E}(u_i)(\sigma)\end{gathered}$$

that leads to $\mathcal{VC}(s) = \emptyset$. There is no array bound violation caused by s. □

Lemma 6.4.2 For any statement, if the under-approximated region of an array contains elements which are outside the declared dimensions of the array then there is certainly a bound violation of the array caused by this statement.

Proof: Let s be the statement and A the array. There exists a program state σ reaching s such that:

$$\underline{\mathcal{R}}(s)(\sigma) \cap \mathcal{DEC}(\sigma) \neq \emptyset \tag{6.7}$$

On the other hand, following the definition of the under-approximation region:

$$\underline{\mathcal{R}}(s)(\sigma) \subseteq \mathcal{R}(s)(\sigma) \tag{6.8}$$

From 6.7 and 6.8, we have:

$$\begin{gathered}\mathcal{R}(s)(\sigma) \cap \mathcal{DEC}(\sigma) \neq \emptyset\\ \text{or } \exists\sigma\ \exists\phi = (\phi_1, \dots, \phi_n) \in \mathbb{Z}^n\ \exists i = 1, n : r(\phi, \sigma) \wedge (\phi_i < \mathcal{E}(l_i)(\sigma) \vee \phi_i > \mathcal{E}(u_i)(\sigma))\end{gathered}$$

which leads to $\mathcal{VC}(s) \neq \emptyset$ by definition. There is an array bound violation caused by this statement. □

6.4.2 Insertion Algorithm

Array regions are built bottom-up, from the elementary statements to the compound statements. Our analysis is a top-down analysis: it begins with the largest compound statement and if we have a global answer about array element accesses for this statement, we do not have to go down into its sub-statements. The analysis can stop here, and bound checks can be inserted at the very beginning of the module entry and outside loops if we have sufficient information.

However, array regions computed for the input code may not be safe, since the semantics of an incorrect code is undefined. The correct code assumption for preexisting analyses is described in detail in Chapter 9. The example in Figure 6.7 shows how the array bound violation can lead to an unsafe propagation of array `A` regions.

```
      PROGRAM REGION
      COMMON ITAB(10),J
      REAL A(10)
      READ *,M
      J = 11
      DO I = 1,M
         ITAB(I) = 1
      ENDDO
      A(J) = 0
      END
```

Figure 6.7: Example for correct code assumption

If every access to `ITAB` is within its bounds, the propagated regions of array `A` before the `DO` loop, the assignment `J = 11` and `READ *,M` are correct. Otherwise, when `M >= 11`, the bound violation in `ITAB` modifies the value of `J` (recall that FORTRAN stores `COMMON` data contiguously) and these regions are not correct any more (Figure 6.8). There is no overflow in array `A` but in `ITAB`.

```
      PROGRAM REGION
      COMMON ITAB(10),J
      REAL A(10)
C     <ITAB(PHI1)-WRITE-MAY-{1<=PHI1}>
C     <A(PHI1)-WRITE-EXACT-{PHI1==11}>
      READ *,M
C     <ITAB(PHI1)-WRITE-EXACT-{1<=PHI1, PHI1<=M}>
C     <A(PHI1)-WRITE-EXACT-{PHI1==11}>
      J = 11
C     <ITAB(PHI1)-WRITE-EXACT-{1<=PHI1, PHI1<=M}>
C     <A(PHI1)-WRITE-EXACT-{PHI1==J}>
      DO I = 1,M
         ITAB(I) = 1
      ENDDO
C     <A(PHI1)-WRITE-EXACT-{PHI1==J}>
      A(J) = 0
      END
```

Figure 6.8: Example with incorrect regions if $M > 10$

To cope with this problem, our analysis is based on the insight that it is safe to propagate an array region from a program point p_2 up to an earlier point p_1 if and only if on every execution path from p_1 to p_2, any written reference to any array is inside the declared range. In other words, an array region at point p_1 is *safe to be used* if and only if all written array references before p_2 are checked. Only written references are taken into account because read references do not modify memory locations, so they have no effects on the correctness of the array region computation. The *write order* is used to decide which array region is checked first. We know that the propagated regions for array `A` become false only when an element outside the declared range of `ITAB` is written. So if bound checks for `ITAB` are generated before testing regions of `A`, there is no problem. The correct code with unavoidable tests is shown in Figure 6.9.

```
PROGRAM REGION
COMMON ITAB(10),J
REAL A(10)
READ *,M
IF (11.LE.M) STOP "Bound violation:  array ITAB, 1st dimension"
STOP "Bound violation:  array A, 1st dimension"
J = 11
DO I = 1,M
   ITAB(I) = 1
ENDDO
A(J) = 0
END
```

Figure 6.9: Example with instrumented code

This is very similar to code hoisting; we are only allowed to hoist code up to an earlier point if it is safe to do so. The regions of an array are considered only when every array whose element assignment occurs at least once before the concerned array assignment has been already checked. In case the write order cannot be established for a compound statement, we have to go down to the sub-statements of the current statement. For instance, with the following sequence of statements:

```
s1     A(I) = I
s2     B(M,N) = M + N
s3     A(J) = J
```

not every element of `A` is always written before elements of `B`, we have to go down and check the array region for `A` at s1, array region for `B` and then for `A` at the beginning of sequence s2; s3.

The algorithm consists of two phases: *array region computation* and *insertion of unavoidable tests*. Since transformers are used in the array region equations to model the effects of state transitions (see Chapter 4, page 47), we have two options to compute array regions: intraprocedural and interprocedural transformer analyses. The Insertion of Unavoidable Tests is given by Algorithm 6.4.1.

Algorithm 6.4.1

```
procedure Insertion_of_Unavoidable_Tests(p)
    p : current procedure
begin
    s = module_statement(p)
    insertion_of_unavoidable_tests_at_statement(s)
end
procedure insertion_of_unavoidable_tests_at_statement(s)
    while the first written array (A) is computable
        for each region R of A in the list of array regions at s
            for each bound check e (φ_i < l_i or φ_i > u_i) at dimension i
                sc_1 = R ∩ {e}
                sc_2 = R ∩ {¬e}
                switch
                    case R = MAY or EXACT and infeasible(sc_1) :
                        no bound violation for A at s
                    case R = MUST or EXACT and infeasible(sc_2):
                        bound violation
                    case R = EXACT:
                        sc = project_phi_variables(sc_1)
                        if the projection is exact then
                            insert IF (sc) STOP message before s
                        else
                            for each sub-statement ss of s
                                insertion_of_unavoidable_tests_at_statement(ss)
                            endfor
                        endif
                    default:
                        for each sub-statement ss of s
                            insertion_of_unavoidable_tests_at_statement(ss)
                        endfor
            endfor
        endfor
    endwhile
    if there are unchecked arrays then
        for each sub-statement ss of s
            insertion_of_unavoidable_tests_at_statement(ss)
        endfor
    endif
end
```

At a compound statement, the read and write regions of each array are used to test the feasibility of the corresponding array bound checks:

1. If the region is a `MAY` or `EXACT` region included in the declared dimensions of the array, no bound check is needed for the compound statement and we stop the process for the array here;

2. If the region is a `MUST` or `EXACT` region that contains elements outside the declared dimensions of the array, there is certainly a bound violation. An error is detected at compile-time;

3. If the region is an `EXACT` region and it is possible to project all unnecessary variables ϕ from the system sc_1, we have unavoidable tests to insert before the compound statement. Each bound check is accompanied with a stop message that tells the user in which array and on which dimension the subscript is out of range. The process stops here for the array;

4. Otherwise, we go down to the sub-statements of the current compound statement, take the regions of the concerning array and repeat the above steps. We have global variables to store information that the lower or upper bound of a dimension of an array has already been checked or not. If so, we do not have to check for this bound when going down to the sub-statements.

The algorithm terminates because we can always generate bound checks directly for array references of elementary statements in the control flow graph.

Insertion Example

Figures 6.10 and 6.11, and Figures 6.12 and 6.13 show the running example with the Insertion of unavoidable tests approach, respectively for the intraprocedural and interprocedural transformer options. The result is the same for `INITAL` so it is omitted in the interprocedural option (Figures 6.12 and 6.13).

For the procedure `INITAL`, we have two kinds of regions: `READ` and `WRITE` for array `U` and these regions are treated separately. Lower bound checks remain for the write region `U(PHI1,PHI2)-WRITE-MAY-{PHI1<=513, PHI2<=513}` and unavoidable tests are inserted before statement `U(M+1,N+1) = U(1,1)`. For the procedure `SHALOW` in the intraprocedural version, the cumulated region of the nested loop allows us to generate an upper bound check that is outside the loop. Compared to the intraprocedural version of the first approach, this is a point in favor of the second approach because it may lift tests out of loops automatically while the former does not.

In the interprocedural version, the interprocedural transformer `{M<=512, N<=512}` of `INITAL` integrated in the cumulated region before the call to `INITAL` decides that no check is needed in `SHALOW`. We have the same result for both approaches.

```
      PROGRAM SHALOW
C     <U(PHI1,PHI2)-READ-MAY-{1<=PHI1, 1<=PHI2}>
      CALL INITAL
C     <U(PHI1,PHI2)-READ-MAY-{1<=PHI1, PHI1<=M, PHI1<=N, 1<=PHI2, PHI2<=M, PHI2<=N}>
      MN = MIN(M, N)
C     <U(PHI1,PHI2)-READ-EXACT-{1<=PHI1, PHI1<=MN, 1<=PHI2, PHI2<=MN}>
      UCHECK = 0.0
C     <U(PHI1,PHI2)-READ-EXACT-{1<=PHI1, PHI1<=MN, 1<=PHI2, PHI2<=MN}>
      DO I = 1, MN
C     <U(PHI1,PHI2)-READ-EXACT-{PHI1==I, 1<=PHI2, PHI2<=MN}>
         DO J = 1, MN
C     <U(PHI1,PHI2)-READ-EXACT-{PHI1==I, PHI2==J}>
            UCHECK = UCHECK + U(I,J)
         ENDDO
      ENDDO
      END

      SUBROUTINE INITAL
C     <U(PHI1,PHI2)-READ-MAY-{PHI1==1, PHI2==1}>
C     <U(PHI1,PHI2)-WRITE-MAY-{PHI1<=513, PHI2<=513}>
      READ (5,*) M, N
C     <U(PHI1,PHI2)-READ-EXACT-{PHI1==1, PHI2==1, M<=512, N<=512}>
C     <U(PHI1,PHI2)-WRITE-MAY-{PHI1<=M+1, PHI2<=N+1, M<=512, N<=512}>
      IF (M.LE.512.AND.N.LE.512) THEN
C     <U(PHI1,PHI2)-WRITE-EXACT-{1<=PHI1, PHI1<=M, 1<=PHI2, PHI2<=N}>
         DO I = 1, M
C     <U(PHI1,PHI2)-WRITE-EXACT-{PHI1==I, 1<=PHI2, PHI2<=N}>
            DO J = 1, N
C     <U(PHI1,PHI2)-WRITE-EXACT-{PHI1==I, PHI2==J}>
               U(I,J) = I*J
            ENDDO
         ENDDO
C     <U(PHI1,PHI2)-READ-EXACT-{PHI1==1, PHI2==1}>
C     <U(PHI1,PHI2)-WRITE-EXACT-{PHI1==M+1, PHI2==N+1}>
         U(M+1,N+1) = U(1,1)
      ELSE
         STOP
      ENDIF
      END
```

Figure 6.10: Insertion of Unavoidable Tests with Intraprocedural Transformers: code with array regions (declarations omitted)

The purpose of Insertion of unavoidable tests is to generate a minimum number of bound checks using the available information from array regions. Bound checks are inserted outside loops and at the beginning of the program. The other advantage of this algorithm is that it detects the sure bound violations or indicates that there is certainly no bound violation as early as possible, thanks to the context given by the top-down analysis of insertion of tests. That is the goal of the second approach group as explained in Section 6.2. Our region based algorithm can be parameterized with respect to different notions of array regions, not only convex polyhedra region. Guarded regions, list of regions [GL00] or dimension per dimension regions could be used to improve the computation time of convex regions. Furthermore, we can merge the read and write regions of the same array, or detect arrays that have the same declarations and same regions in order to reduce redundant checks.

```
PROGRAM SHALOW
CALL INITAL
MN = MIN(M, N)
IF (514.LE.MN) STOP "Bound violation:  array U, 1st dimension"
UCHECK = 0.0
DO I = 1, MN
   DO J = 1, MN
      UCHECK = UCHECK + U(I,J)
   ENDDO
ENDDO
END

SUBROUTINE INITAL
READ (5,*) M, N
IF (M.LE.512.AND.N.LE.512) THEN
   DO I = 1, M
      DO J = 1, N
         U(I,J) = I*J
      ENDDO
   ENDDO
   IF (N+1.LT.1) STOP "Bound violation:  array U, 2nd dimension"
   IF (M+1.LT.1) STOP "Bound violation:  array U, 1st dimension"
   U(M+1,N+1) = U(1,1)
ELSE
   STOP
ENDIF
END
```

Figure 6.11: Insertion of Unavoidable Tests with Intraprocedural Transformers: code with unavoidable tests (declarations omitted)

```
      PROGRAM SHALOW
C     <U(PHI1,PHI2)-READ-MAY-{1<=PHI1, PHI1<=512, 1<=PHI2, PHI2<=512}>
      CALL INITAL
C     <U(PHI1,PHI2)-READ-MAY-{1<=PHI1,PHI1<=M,PHI1<=N,1<=PHI2,PHI2<=M,PHI2<=N}>
      MN = MIN(M, N)
C     <U(PHI1,PHI2)-READ-EXACT-{1<=PHI1, PHI1<=MN, 1<=PHI2, PHI2<=MN}>
      UCHECK = 0.0
C     <U(PHI1,PHI2)-READ-EXACT-{1<=PHI1, PHI1<=MN, 1<=PHI2, PHI2<=MN}>
      DO I = 1, MN
C     <U(PHI1,PHI2)-READ-EXACT-{PHI1==I, 1<=PHI2, PHI2<=MN}>
         DO J = 1, MN
C     <U(PHI1,PHI2)-READ-EXACT-{PHI1==I, PHI2==J}>
            UCHECK = UCHECK + U(I,J)
         ENDDO
      ENDDO
      END
```

Figure 6.12: Insertion of Unavoidable Tests with Interprocedural Transformers: code with array regions (declarations omitted)

```
      PROGRAM SHALOW
      CALL INITAL
      MN = MIN(M, N)
      UCHECK = 0.0
      DO I = 1, MN
         DO J = 1, MN
            UCHECK = UCHECK + U(I,J)
         ENDDO
      ENDDO
      END
```

Figure 6.13: Insertion of Unavoidable Tests with Interprocedural Transformers: code with unavoidable tests (declarations omitted)

6.5 Intraprocedural Experimental Results

Ten benchmarks in SPEC95 CFP that contain a large amount of subscripted references to arrays are used in our experiments. The codes are instrumented and then executed using the standard input data to compute the number of dynamic range checks. Table 6.1 provides the number of compile-time and run-time array bound checks for each benchmark in SPEC95 CFP.

Table 6.1: SPEC95 CFP: number of unoptimized compile-time and run-time checks, array resizing

Benchmark	Compile-time Check	Run-time Check	Array Resizing
tomcatv	304	4933×10^7	
swim	772	6991×10^7	
su2cor	4460	4810×10^7	[*]
hydro2d	2016	6530×10^7	[*]
mgrid	1162	17608×10^7	
applu	9562	11562×10^7	[*]
turb3d	1852	6081×10^7	*
apsi	7172	3936×10^7	*
fpppp	2894	2928×10^7	*
wave5	10546	3404×10^7	[*]

6.5.1 Preprocessing

Three benchmarks in SPEC95 CFP (*turb3d, apsi* and *fpppp*) do not meet the FORTRAN standard for array declaration and reference: they have ugly assumed-size declarations (see Chapter 5, page 53), although array references in the corresponding procedures are outside the defined extent of the array. In addition, assumed-size array declarations also exist in these three benchmarks and four other benchmarks (column 4, Table 6.1).

Ugly assumed-size arrays must be treated at least as assumed-size arrays to avoid premature aborts due to bound violations. Then array resizing (Chapter 5) can be applied to these seven benchmarks. The combined approach (AAR + AR + CI) can be used to obtain all pointer-typed arrays resized. To compare with other commercial compilers that do not check bound violations for assumed-size arrays, in this experiment set, we only replace ugly assumed-size arrays in the three benchmarks with assumed-size arrays. Except the spurious violations caused by the ugly assumed-size declarations, there are no intraprocedural out-of-bound errors with the standard data input of SPEC95 CFP.

6.5.2 Removed Checks

Table 6.2 shows the percentages of bound checks removed by the two approaches for intraprocedural array bound checking: Elimination of Redundant Tests and Insertion of Unavoidable Tests. For both approach, we used the intraprocedural option for transformers and preconditions analyses, which is faster but less accurate and the interprocedural option, which is slower but improves the accuracy.

Table 6.2: SPEC95 CFP - Intraprocedural Array Bound Checking: percentage of removed compile-time and run-time checks

	Elimination of Redundant Tests				**Insertion of Unavoidable Tests**			
	Intra		**Inter**		**Intra**		**Inter**	
Benchmark	**Comp.**	**Run**	**Comp.**	**Run**	**Comp.**	**Run**	**Comp.**	**Run**
tomcatv	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
swim	97.00	97.00	98.45	99.99	84.60	99.99	84.72	99.99
su2cor	94.52	95.20	96.59	97.54	92.12	96.60	94.48	97.66
hydro2d	95.13	93.50	96.13	94.14	90.02	97.70	93.85	99.46
mgrid	91.32	99.60	94.92	99.60	96.61	99.50	97.93	99.66
applu	98.54	96.75	99.62	97.09	96.38	99.80	96.41	99.87
turb3d	92.23	56.18	97.62	65.00	87.03	76.78	98.97	85.57
apsi	97.08	99.20	98.02	99.90	98,70	99.31	99.79	99.99
fpppp	94.12	96.48	94.61	97.02	92.18	97.23	95.82	97.40
wave5	94.52	86.26	94.66	86.86	91.12	89.83	94.01	91.29

For each combination of approach and option, we measured the percentages of compile-time and run-time checks removed. The number of eliminated compile-time checks may be high, but if remaining checks are inside some frequently executed blocks of code, we do not have much speed-up. Run-time checks are more interesting because they have direct effects on execution time. So to compare between intraprocedural and interprocedural analysis options, between the Elimination of redundant tests and Insertion of unavoidable tests approaches, we use only run-time checks results.

In both approaches, we do not have much gain with interprocedural analysis if the intraprocedural one has already done a good job (*tomcatv, mgrid, applu*). There is more room left to improve the number of removed tests with *turb3d* (8.82%, 8.75%), *su2cor* (2.34%, 1.06%), *swim* (2.99% with Elimination approach), *hydro2d* and *wave5* (respectively, 1.76% and 1.36% with Insertion approach) because of the smaller percentages of removed tests. For *tomcatv*, we can statically prove that there is no array bound violation, which is an interesting result for verification purposes.

Comparing the two approaches, we see that the second one works uniformly better. We have almost no gain for *tomcatv, mgrid* and *apsi*, but there are very big gaps between Insertion of unavoidable tests and Elimination of redundant tests for *turb3d* (about 20.0%

with both options), *hydro2d* (4.20% with intraprocedural option, 5.32% with interprocedural option) and *wave5* (3.57% with intraprocedural option, 4.43% with interprocedural option).

The percentage of removed tests varies for different benchmarks, approaches and options. It is not very high for *turb3d* and *wave5*, since there are a lot of non-linear expressions in array bound checks that have not yet been handled by PIPS.

6.5.3 Debugging Information

The amount of information given when a bound violation occurred differs among compilers. This information is shown in Table 6.3 by experiments with three original benchmarks violating the standard for array references: *turb3d*, *apsi* and *fpppp*. Experiments have been performed with three commercial compilers: SUN Workshop F77 version 5.0, SGI MIPSpro F90 version 7.3 and IBM XLF F77 version 7.1.0.0. There is no range checking option for SGI F77 and GNU G77 compilers and we had to leave them out.

Table 6.3: Debugging information provided by different compilers

	SUN	**SGI**	**IBM**	**PIPS Elimination**	**PIPS Insertion**
Line	x			x	
Array	x			x	x
Dimension	x			x	x
Bound				x	x

The SUN compiler -C option provides the lines of code, the arrays and the dimensions associated to the violations. The SGI -C option does not provide any information, which is particularly clear since the programs did not stop on the system we used when they reached an out-of-bound trap. The IBM -C option spots errors in programs with the *Trace/BPT trap(coredump)* message. Meanwhile, our first array bound checker provides full information about the location of the violations and the second one gives information about the array and the dimension whose bounds are violated. This information is very important for the debugging process. Other experiments with more than 100000 line industrial codes show that the supplement information such as the module, the lower or upper bound, the values of concerning variables given by our first approach is very useful for tracing the origin of the error. In addition, to detect a maximum number of errors in one pass, we have another option to replace the `STOP` message by the `PRINT` message with all available information. The repeated compilation and execution are avoided.

6.5.4 Compilation Times

The compilation speeds, expressed in source lines per second, obtained with PIPS to parse, analyze (transformers, preconditions, array regions), optimize (array bound check) and generate FORTRAN code with its own range checking for SPEC95 CFP are shown in Column 2 and 6, Table 6.4. The speeds are measured with interprocedural analyses for transformers and preconditions, which are slower than the intraprocedural ones. Comment lines are not taken into account. The ten benchmarks, with 20644 lines of code and 374 subroutines, are processed at an average speed of 66.07 lines per second for the Elimination of redundant tests and 17.24 lines per second for the Insertion of unavoidable tests. The range check optimization phase only takes a very small fraction of this compilation time but we have not attempted to measure it because only the total time matters to the user.

Table 6.4: SPEC95 CFP - Intraprocedural Array Bound Checking: compilation time (in minutes and seconds) - Ultra SPARC 360MHz - Optimized code (f77 -fast -xarch=v8plusa -fsimple=2 -xprefetch)

	Elimination				**Insertion**				**SUN F77**	
Bench	**Speed**	**PIPS**	**SUN**	**Total**	**Speed**	**PIPS**	**SUN**	**Total**	**Wo C**	**+ C**
tomcatv	55.5	0:02	0:02	0:04	22.2	0:05	0:02	0:07	0:03	0:13
swim	70.7	0:04	0:02	0:06	23.5	0:12	0:05	0:17	0:04	0:20
su2cor	39.4	0:40	0:25	1:05	10.5	2:28	0:33	3:01	0:33	2:23
hydro2d	107.0	0:16	0:19	0:35	39.8	0:43	0:32	1:15	0:32	1:03
mgrid	85.2	0:04	0:07	0:11	8.9	0:38	0:06	0:44	0:08	0:37
applu	74.9	0:33	0:25	0:58	20.2	2:02	0:54	2:56	0:57	11:12
turb3d	86.0	0:15	0:16	0:31	19.8	1:05	0:17	1:22	0:17	0:44
apsi	55.8	1:16	1:02	2:18	9.1	7:42	1:03	8:45	1:16	3:15
fpppp	50.6	0:42	0:45	1:27	8.8	4:01	0:56	4:57	0:54	1:22
wave5	35.2	5:03	1:25	6:28	8.9	12:04	1:50	13:54	2:04	6:20

The compilation times (in minutes and seconds) for the second approach are longer, especially for *mgrid, apsi, fpppp* and *wave5*. It is due to the satisfiability test used in PIPS to compute array regions. This could be improved by a more sophisticated implementation of array regions. As shown in Table 6.2, the percentage of removed checks of Insertion of unavoidable tests is high enough to pay for this tradeoff.

Since PIPS is a FORTRAN *source-to-source* compiler, the code generated by PIPS with its own range checking is then compiled by other compilers. We measured the compilation times taken by PIPS as a preprocessor and by SUN Workshop F77 5.0 compiler for PIPS generated codes. The original codes of SPEC95 CFP are also compiled with and without the array range checking option of SUN. Experimental results show shorter times for the two implementations of PIPS than for SUN (see columns Total of Elimination of redundant tests and of Insertion of unavoidable tests and column + C of SUN F77).

6.5.5 Execution Times

The execution times of SPEC95 CFP are measured on different platforms to see the relationship between the percentage of eliminated checks and the slowdown. This set of experiments is reported with the optimizing options turned on, using the SPEC95 CFP measurement guidelines. The code generated by PIPS with its own range checking using the interprocedural option for transformers and preconditions is compiled by other compilers (SUN, SGI and IBM) to generate executable files. For IBM, because an internal compiler error occurred when compiling the FORTRAN code with options -O5 and -C together, we used -O3. In addition, there is an input/output error for *apsi* so we do not have results for this benchmark on the IBM machine. The execution times of codes obtained with and without the bound checking option of these compilers and with the PIPS versions are provided in Figure 6.14, Figure 6.15 and Figure 6.16.

We can see the overheads of range checking in *mgrid* and *applu* for SUN (Figure 6.14), *mgrid* and *swim* for SGI (Figure 6.15) and *tomcatv* and *turb3d* for IBM (Figure 6.16). PIPS optimizing array bound checkers work very well for *tomcatv, swim, mgrid* and *applu*. These benchmarks have more dynamic bound checks than others, as shown in Table 6.1.

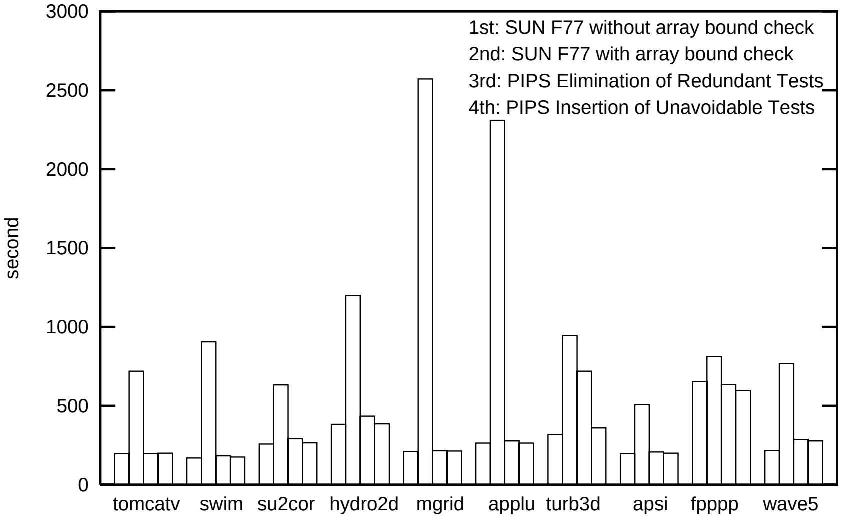

Figure 6.14: Execution time: SUN F77 and PIPS - SUN Workshop F77 5.0 - Ultra SPARC 360MHz - Optimized code (f77 -fast -xarch=v8plusa -fsimple=2 -xprefetch)

As the range checking of the IBM compiler is already optimized, the PIPS versions work better than IBM in general but worse for *turb3d* benchmark. The reason is that analyses of non-linear expressions are not implemented yet in PIPS.

Comparing the execution time of PIPS codes with that of other bound checked codes, on average, PIPS Elimination of redundant tests is about 3.94 times faster than SUN, 1.88 times faster than SGI and 1.03 times faster than IBM. PIPS Insertion of unavoidable tests is about 4.37 times faster than SUN, 2.02 times faster than SGI and 1.07 times faster than IBM. The execution times of programs with range checking added by PIPS are slightly

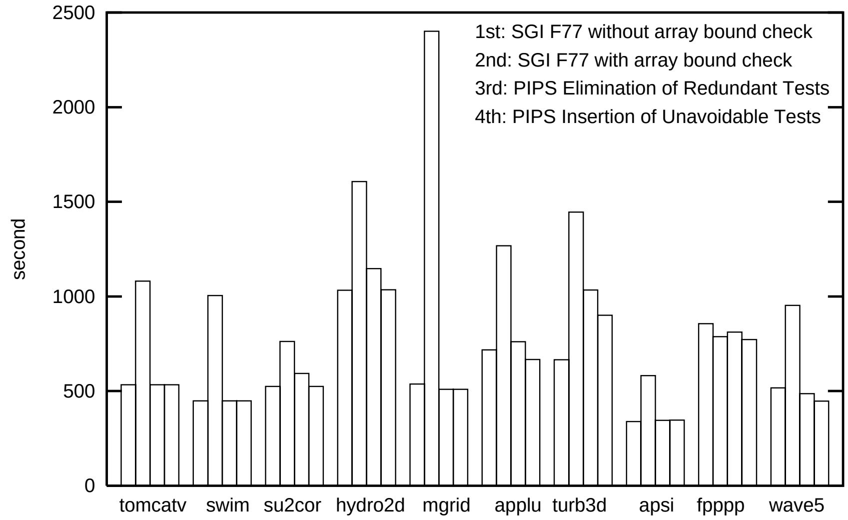

Figure 6.15: Execution time: SGI F90 and PIPS - SGI MIPSpro F90 7.3 - O2 R5000 195MHz, IRIX 6.3 - Optimized code (f90 -Ofast=ip32_5k)

longer than that of the unsafe programs without bound checks. On average, these times for PIPS Elimination of redundant tests are about 19.29% longer for SUN, 7.05% longer for SGI and 16.59% longer for IBM. For PIPS Insertion of unavoidable tests, they are about 5.33% longer for SUN, 0.61% longer for SGI and 10.62% longer for IBM.

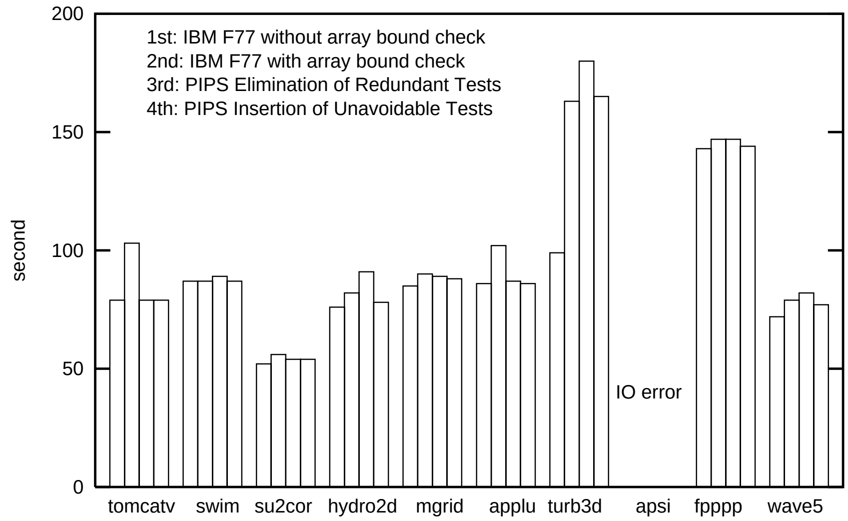

Figure 6.16: Execution time: IBM F77 and PIPS - IBM XL F77 7.1 - RS/6000 44P-270 375MHz 4 CPU, AIX 4.3 - Optimized code (f77 -O3 -lmass)

6.6 Interprocedural Array Bound Checking

Interprocedural array bound checking refers to checking the declared size of a formal array with respect to the range declaration of its corresponding actual argument. The relationship between the size of the formal and the actual arrays is defined by the argument association rules, presented in Definition 5.2.1 (see Chapter 5, page 58).

6.6.1 Actual/Formal Array Sizes

Within a program unit, the array declaration given for an array provides all range information needed for the array in an execution of the program unit. But in the whole program, when a formal array argument is associated with an actual array argument, we also have to ensure that there is no bound violation in every array access in the called procedure with respect to the array declarations in the calling procedure. If not, we cannot know what happens when accessing the memory beyond the allocated regions. The example in Figure 6.17 illustrates a simple interprocedural bound violation found in a real application.

```
PROGRAM VIOLATION
COMMON /FOO/ Z1(10,10), Z2(10,10)
CALL ZERO(Z1,10,20)
PRINT *, Z1
PRINT *, Z2
END
```

```
SUBROUTINE ZERO(X,N,M)
REAL X(N,M)
DO I = 1,N
   DO J = 1,M
      X(I,J) = 1.
   ENDDO
ENDDO
END
```

Figure 6.17: Interprocedural array bound violation example

Although exceeding the size of an actual argument array is strictly forbidden in the FORTRAN standard [ANS83], commercial compilers such as SUN Workshop F77 version 5.0, SGI MIPSpro F90 version 7.3 and IBM XLF F77 version 7.1.0.0 do not check it. One can argue that this kind of violation is rare in practice or conversely, that is used voluntarily as in Figure 6.17, but our implementation found an interprocedural bound violation in one out of the ten benchmarks from SPEC95 CFP. Furthermore, the fact that bugs related to interprocedural checking are much more difficult to track than to intraprocedural one is a good reason to provide the actual/formal array size checking.

Our analysis traverses the call graph of programs in the invocation order (see Chapter 3, page 23). By using Lemma 5.2.1, Lemma 5.2.2 and the notation in Table 5.2 (see Chapter 5, page 58), the interprocedural array bound checking is given by Algorithm 6.6.1. It consists of two steps: *interprocedural bound check generation* and *redundancy elimination*, using preconditions.

Algorithm 6.6.1

procedure Interprocedural_Array_Bound_Checking(p)
    p : current procedure
begin
    for each call site c of p
        q = corresponding_callee(c)
        for each actual array argument a of c
            b = corresponding_formal_parameter(a, q)
            if b is an array variable **then**
                k := number_of_equal_dimensions(a, b, c, p, q)
                s_2 := size_of_formal_array(b, k, q)
                s_2' := translate_to_caller_frame(s_2, p, c, q)
                s_1 := size_subscript_value_of_actual_array(a, k, c, q)
                $e := e_a.s_1 < e_b.s_2'$
                insert IF (e) $STOP$ $message$ before c
            endif
        endfor
    endfor
    compute preconditions for modified p
    for each inserted test statement t with condition e of p
        P = precondition(t)
        $sc_1 = P \cap \{e\}$
        $sc_2 = P \cap \{\neg e\}$
        switch
            case infeasible(sc_1) :
                no bound violation, remove t from p
            case infeasible(sc_2) :
                bound violation found at compile-time
            default: /* indeterminable, keep the dynamic check */
        endswitch
    endfor
end

For each corresponding formal array parameter at each call site, we generate a test to check whether the size of the formal array exceeds that of the actual array. As for array resizing, we try to treat dimensions independently by computing k, the number of equal values among the first dimensions of the actual and formal arrays. When the actual argument is an array element, k is also the number of first subscripts that are equal to their corresponding lower bounds. This step facilitates the computation of array sizes and subscript value expressions. The inequality between the sizes of the actual and formal arrays can be simplified, and thus the feasibility test can also be simplified. For instance, in the example in Figure 6.17, by knowing `N == 10`, we only have to check `10 < M` instead of introducing the non linear expression `10*10 < N*M`. Each bound check is accompanied with a stop message and if an bound violation is detected, the message tells the user which call site creates the violation in which array.

Information about global variables and calling contexts, such as the relationship between actual and formal arguments, are used to improve the translation process and simplify the inequality characterizing the bound violation. The size of the formal array can always be translated to the frame of the caller because the dimension bound expressions are integer constant expressions or contain only formal parameters.

By using preconditions, a test condition is checked feasible or not. If it is false with respect to the precondition, the test is removed from the program. If it is true, an interprocedural violation is detected at compile-time. Otherwise, it is preserved in the program.

6.6.2 Experimental Results

Table 6.5 shows the number of compile-time and run-time checks added, as well as the compilation time (in minutes and seconds) and the slowdown caused by the interprocedural array bound checking for the SPEC95 CFP benchmarks. Although ugly and assumed-size arrays are supposed not to exceed the size of the corresponding actual arrays, they still raise problems for interprocedural array bound checking. If the formal array is declared correctly but the actual one has an ugly or assumed-size, we cannot compare them. So we have to apply array resizing to seven benchmarks in SPEC95 CFP, with the argument association rules and code instrumentation approaches (Chapter 5).

Table 6.5: SPEC95 CFP - Interprocedural Array Bound Checking: number of added compile-time and run-time checks, compilation time (in minutes and seconds) and execution slowdown

Benchmark	**Compile Checks**	**Run Checks**	**Compilation**	**Slowdown**
tomcatv	0	0	0:01	0.00%
swim	0	0	0:03	0.00%
su2cor	0	0	0:59	0.00%
hydro2d	0	0	0:21	0.00%
mgrid	24	40151	0:05	0.15%
applu	0	0	0:17	0.00%
turb3d	29	281417	0:42	2.12%
apsi	6	240127	1:25	1.67%
fpppp	2	727917	2:29	0.79%
wave5	34	Bound violation	2:36	Bound violation

By using static analyses, our checking has proved that there is no interprocedural bound violation in five out of ten benchmarks. Other bound checks are added before some procedure calls in the five remaining benchmarks. We cannot compare the effectiveness of our approach to some rare compilers that do this checking, but the cost here is small enough. The maximum slowdown is only 2.12% for *turb3d*.

A bound violation is detected in *wave5*, an electro-magnetic particle simulation program. Figure 6.18 contains the piece of code that causes a bound violation for array `TMP` when passing it as argument in procedure calls. The size of array `TMP(NXD,NY,2)` in subroutine `SLV2XY` must be less than or equal to the size of array `TMP(NX2,78885/NX2)` in subroutine `SOLV2Y`. This array was declared as `TMP(NX2,*)` in the original code, before the array resizing phase. With the argument association rules, we infer its new declaration, based on the size of the actual array `BX(NC1)` in subroutine `FIELD`. By using binding information between formal and actual arguments and preconditions, we have `SLV2XY:NXD = SOLV2Y:NX2 = FIELD:NX2 = FIELD:NX+2` and `SLV2XY:NY = SOLV2Y:NY2 -2 = FIELD:NY2-2 = FIELD:NY`. So the size of array `TMP` in `SLV2XY` is translated into the frame of `SOLV2Y`, from `NXD*NY*2` to `NX2*(NY2-2)*2`. The interprocedural check on array sizes in the frame of `SOLV2Y` is `78885.LT.NX2*(NY2-2)*2`. When executing the instrumented code with its standard input data where the grid size `NX = 1250` and `NY = 60`, we have `78885 < 1252*60*2` is true, so there is a bound violation here.

```
SUBROUTINE FIELD
PARAMETER (NC1 = 78885)
COMMON /EFIELD/ EX(NC1),EY(NC1),EZ(NC1),BX(NC1),BY(NC1),BZ(NC1)
NX2 = NX + 2
NY2 = NY + 2
CALL SOLV2Y(NX2,NY2,HX,HY,0.0D0,V,BC,BDY1,BX)
END

SUBROUTINE SOLV2Y(NX2,NY2,HX,HY,DD,Q,BX,BY,TMP)
DIMENSION BX(4),TMP(NX2,78885/NX2),BY(4)
IF (78885.LT.NX2*(NY2-2)*2) STOP "Interprocedural violation:  array TMP"
CALL SLV2XY(NX2-2,NY2-2,NX2,HX,HY,DUMMY,DD,Q,BX,BY,TMP,0)
END

SUBROUTINE SLV2XY(NX,NY,NXD,HX,HY,GX,DD,Q,BX,BY,TMP,IGXSW)
DIMENSION TMP(NXD,NY,2),BX(4),GX(NXD),BY(4)
CALL VSLV1P(NX,NY,NXD,HX,GX,DIAG,Q(1,2),TMP,TMP(1,1,2),IGXSW,ISING)
END
```

Figure 6.18: Interprocedural array bound violation (excerpt from *wave5*, SPEC95 CFP)

Because `TMP`, associated to `BX`, is accessed outside its declared range by procedure calls and `BY` is allocated just after `BX` in memory, the two arrays `TMP` and `BY` share some memory locations if bound violations are not checked. So if interprocedural array bound checking is omitted, bound violation makes other analyses such as alias analysis (see Chapter 7) impossible.

6.7 Conclusion

We designed and experimented two intraprocedural and one interprocedural algorithms for array bound checking. The number of removed/added checks, compile-time and run-time performances as well as the debugging capability of our techniques compare favorably with the current commercial implementations and related work.

The study of Richardson and Ganapathi [RG89] suggests that interprocedural analyses give little benefit in optimization and are too expensive to be worthwhile. However, our implementations show that with powerful and efficient interprocedural analysis techniques, more redundant checks are removed and we can even prove the absence of bound violations in some programs. Once again, it is still a question of tradeoff between speed and accuracy, but in the domain of verification, proving or checking the correctness of code is the most important criteria.

The experimental results show the effectiveness and the limited optimization cost of our two intraprocedural array bound check approaches: Elimination of redundant tests and Insertion of unavoidable tests. The first one puts array bound checks everywhere and then removes the redundant ones. This approach is simple and the number of eliminated tests depends on the strength of data flow analyses, such as predicates over scalar integer variable values, used to perform the elimination. The second implementation inserts useful checks directly by using array region analyses. It produces better results with a higher number of removed dynamic checks and faster execution times. For a small program like *tomcatv*, the differences between the two approaches are limited, but for large programs with more than 2000 lines of code, there are clear differences. The maximum improvement in removed dynamic bound checks is 20.57% for *turb3d*. The main advantage of this top-down analysis approach is that it detects the sure bound violations or indicates that there is certainly no bound violation as soon as possible.

Within the SUN environment, we measured shorter compilation times using our two intraprocedural source-to-source array bound checkers followed by F77 than using F77 alone with its -C option. The compilation time speedups are 3.03 for Elimination of redundant tests and 1.1 for Insertion of unavoidable tests. The PIPS Elimination of redundant tests has shorter compilation time and execution time than the SUN compiler while preserving the same diagnostic capabilities.

At run-time, the slowdowns measured for SUN and SGI compilers are large enough to make improvement easy. We mostly obtained speedups by adding array bound checks of the Insertion of unavoidable tests version, in comparison with the execution time with no bound checks of the SGI compiler. This is not the case with IBM XLF compiler 7.1.0.0, which nevertheless is not uniformly efficient and which breaks down with an internal error when combining options -C and -O5. However, our execution times are in the same range as IBM's, 7 times out of 9 slightly better. Furthermore, as does the SUN compiler, our array bound checkers provide programmers useful and precise information about the dimension and name of the array experiencing a bound violation, even the line of code with the first approach, while the IBM compiler only indicates that an overflow occurred somewhere. As in code hoisting methods, PIPS Insertion of unavoidable tests propagates bound checks outside loops and into the beginning of the program so the precise location of the violation is lost. But compared to IBM's compiler, which is in the same performance range, the array and the dimension improperly accessed are still known. This shows that a simple comparison of techniques is not possible since the amount of information produced differs. Without good diagnostic capability, many hours can be wasted in debugging the cause of an array bound violation message. It should not be inferred from Figure 6.14 and Figure 6.16 that the SUN compiler is not as good as the IBM's one.

Among the related work, the article *Elimination of Redundant Array Subscript Range Checks* [KW95] contains most experimental results. They implemented in their Nascent research compiler different techniques for range checking optimizations based on different check placement schemes. Their experimental results show that simple optimizations such as preheader insertion with loop-limit-substitution of linear checks greatly reduce the number of dynamic checks removed. The set of experiments is ten FORTRAN programs from the Perfect, Riceps and Mendez benchmarks. However, we could not directly compare our results with theirs because the authors do not include execution times. We observe that the percentage of removed checks is not an accurate predictor of slowdown. These two numbers are not proportional because they may depend on architectures and compilers. For example, the execution time of code without bound checks for *swim* is only 92.35% although 99.99% of dynamic checks are removed (see Table 6.2 and Figure 6.14). So, to evaluate array bound checking optimization, it is necessary to compare the execution times of generated codes, which are missing in this paper. Furthermore, no information about the origin of violation is preserved in their approach. Problems related to ugly and assumed-size arrays in Perfect Club and Riceps benchmarks are not mentioned. Out-of-bound errors occur in 4 benchmarks: *mdg, spc77, trfd* from PerfectClub and *linpackd* from Riceps. These errors are caused by violating standard declaration of arrays in FORTRAN and there is no possibility to obtain comparable results for these programs. The percentage of dynamic checks removed is compared in Table 6.6 for the remaining benchmarks. Their best figures are in the very same range as ours and it is very interesting to see that specific techniques do not work better than re-used techniques.

Result analysis shows the importance of code quality. Proper array declarations are needed to avoid out-of-bound errors caused by standard violation. Assumed-size array declarations prevent a complete range checking so we have used array resizing to overcome this problem. Furthermore, some benchmarks require more sophisticated techniques or modifications such as cloning, parameter checking, scalarization for dealing with indi-

Table 6.6: PerfectClub, Riceps: percentage of dynamic checks eliminated

Benchmark	**PIPS Elimination**	**PIPS Insertion**	**Nascent Best result (ALL)**
arc2d	97.66%	97.69%	99.96%
bdna	95.90%	97.75%	98.44%
dyfesm	97.56%	99.51%	98.73%
qcd	96.12%	99.50%	97.06%
simple	99.96%	99.98%	99.97%

rections, scalarization for loop bounds, non-unit loop increments, etc. For example, we improve the percentage of removed checks in Elimination of redundant tests from 94.14% to 99.50% for *hydro2d* by cloning the subroutine `ADLEN` which has two totally different behaviors for two parameter values: "half" and "full" steps. The execution time on SUN is reduced by 10%. The elimination percentage goes up to 100% from 97.09% for *applu* by adding one `STOP` statement after the parameter checking that is performed for lower bound tests but not for upper bound test of read variables (`NX,NY,NZ` in the main program `APPLU`). A 5.4% decrease of the execution time is then measured on SUN. Poor code quality can make static analysis insufficient; run-time checks remain and run-time failures cannot be eliminated.

Another conclusion is retrieved from experiencing with codes that have out-of-bound violations. Figure 6.19 illustrates a kind of violation that exists in many real applications.

```
COMMON /FOO/ A(30,3), B(30,3), C(30,3)
DO I = 1, 30
   DO J = 1, 9
      A(I,J) = I*J
   ENDDO
ENDDO
```

Figure 6.19: Upper bound violation example

Programmers initialize the whole common block by the first array `A` that causes upper bound violations. To cope with this style of programming, we developed the *area bound checking* that compares the array reference to the size of the corresponding common block, not to the upper bound in the array declaration. So in fact, a lower bound violation seems rather be a real bug than the upper one.

Some analyses for non-linear expressions that have direct impact on array range checking have not been implemented in PIPS, so we do not have very good results for *turb3d* and *wave5* in Elimination of redundant tests. For the array region-based version, the number of bound checks could be reduced by replicating code as in [MMS98, MMG00] when `MAY` regions give necessary but not sufficient conditions for a bound violation to occur. However, the code size increase may raise problems that go beyond a simple reuse of existing techniques.

Actual/formal array size checking is as important as the normal array bound checking because it allows us to detect complicated bugs. We developed an efficient algorithm that guarantees the safety of code. This can be a complementary phase when doing array range checking for whole programs.

These implementations suggest that commercial products with automatic analyses could easily be improved to perform efficient array bound checking without sacrificing information about the location of the violation. Less than 1600 additional lines of C code (comments included) are sufficient to implement both intraprocedural and interprocedural checking in PIPS. The execution overhead is small enough to consider the use of safe versions of programs for production activities. These array bound checkers could possibly be a source-to-source preprocessor for GNU g77, since it does not have a range checking option.

Our approaches to optimizing bound checking could also be applied to other imperative languages for scientific applications that require software verification such as ADA, C, JAVA, etc but they should be extended to cope with functional and data structure recursion. Some additional analysis such as array resizing can help to propagate the array size when arrays or pointers are passed as formal parameters and enable bound checking on these data structures.

Chapter 7

ALIAS ANALYSIS

Fortran purity test
By: burley@geech.gnu.ai.mit.edu (Craig Burley)
Organization: Free Software Foundation 545 Tech Square Cambridge, MA 02139

"Question 5: Recent and future Fortran standards disallow multiple-dummy and/or dummy/common **aliasing** *of any variable when the called procedure modifies any variable involved in the aliasing because:*

a) It will give compiler vendors less work to do and faster products to sell

b) There should really be only one way to skin a cat

c) Fortran programming is for people who have nothing else to do but remember obscure rules like this

d) It makes it easier for people to write useful Fortran programs

d) I don't know what a dummy is

Here are my proposed "points" for the above questions, and the reasons why:

a) 10 points – A longer answer is possible, but a waste of time

b) 9 points – Not quite as eloquently expressed as a)

c) 8 points – Mostly true, but if we can get others to try it as well, compiler vendors can sell more compilers

d) 4 points – Only if you replace "easier" and "useful" with "possible" and "fast"

e) 10 points – You are likely to be excellent in X3J3 diplomacy"

One reason for making FORTRAN programs run faster than equivalent programs written in other languages is the no aliasing assumption. The FORTRAN standard guarantees that aliased formal and common variables must not be modified in a subroutine. It does not define the behavior for the non-conforming programs, so if the alias restriction is not checked, anything may happen.

We develop in this chapter the alias restriction verification. The chapter is organized as follows. Section 7.1 describes the problems encountered with dynamic aliasing. Section 7.2 discusses some related work on alias analysis. Section 7.3 presents the interprocedural alias propagation, Section 7.4 the interprocedural alias verification and Section 7.5 the impact of alias violations on program transformations. Section 7.6 studies experimental results on alias checking on SPEC95 CFP benchmarks. Finally, Section 7.7 presents conclusions.

7.1 Dummy Aliasing Problems

Aliasing occurs when two or more variables refer to the same storage location at the same program point. Alias analysis is critical for performing most optimizations correctly because all the ways a location, or the value of a variable, may or must be used or defined must be taken into account. Compile-time alias information is key to data dependence analysis and hence is important for program analyses, transformation, parallelization, verification, debugging and understanding. For example, a good precision of alias information increases the degree of instruction-level parallelism by a factor of 4 to 8 times with respect to the conservative alias information [Wal91].

The sources of aliases vary from language to language. *Intraprocedural aliases* occur due to pointers in languages like LISP, C, C++ or FORTRAN 90, union constructs in C or `EQUIVALENCE` in FORTRAN. *Interprocedural aliases* are generally created by parameter passing and by access to global variables, which propagate intraprocedural aliases across procedures and introduce new aliases. Alias analysis can be classified by its formal characterization: *may* versus *must* information, *flow-insensitive* versus *flow-sensitive* and *context-insensitive* versus *context-sensitive*.

In FORTRAN, parameters are passed by reference and, as long as the actual argument is associated to a named storage location, the called subprogram can change the value of the actual argument by assigning a value to the corresponding formal parameter. So new aliases are created among formal parameters if the same actual argument is passed to two or more formal parameters, or between a formal parameter and a common variable if an actual argument is allocated in a common block which is also visible in the called subprogram or other subprograms in the call chain below it. Restrictions on association of entities in FORTRAN 77 (Section 15.9.3.6 [ANS83]) state that *neither aliased formal parameters nor variables in the common blocks may become defined during execution of the called subprogram or other subprograms in the call chain.* If these rules were enforced by compilers, aliases would be created only in a few ways and be pin-pointed at compile-time. Mostly, they would not impact on data dependence analysis and the optimizations based on it, nor on maintenance and re-use.

However, established programming practice often violates the FORTRAN 77 standard [ZC90]. Compilers, assuming that programs are conform to the standard, consider that an assignment to one variable will not change the value of any other variable, allowing optimizations involving the aliased variables. Higher performance results, but anything may happen: the program may appear to run normally, or produce incorrect answers, or behave unpredictably.

In effect, compilers should follow practice at least to some degree so as not to place the burden of alias analysis on the programmer. This can cause programs to produce results depending on optimization levels and programmers to end up using different optimization levels for each module of an application. A contrived example of such aliasing is given in Figure 7.1.

```
PROGRAM ALIAS
INTEGER I,A(5)
DO I = 1, 5
   A(I) = 2
ENDDO
CALL SUB(A,A(1))
PRINT *,A
END
```

```
SUBROUTINE SUB(X,Y)
INTEGER I,X(5),Y
DO I = 1, 5
   X(I) = Y*X(I)
ENDDO
END
```

Figure 7.1: Formal parameter aliasing example

The assignment to `X(1)` in the first loop iteration modifies `Y` which is loop invariant and stored in a register. With the Sun WorkShop 6 FORTRAN 77 5.1 compiler, the output is 4 4 4 4 4 instead of 4 8 8 8 8 if the optimization level is greater than 2 and if the modules are compiled separately. Some FORTRAN compilers for OpenVMS, DEC Unix, Ultrix and AIX from IBM have an option to assert the presence of aliases among *dummy* arguments (FORTRAN terminology for formal parameters). If this option is selected, program semantics requires frequent recomputation on dummy arguments and common variables that insures correct results, but optimizations are inhibited. By default, no aliases between dummy arguments and common variables are assumed to exist. One called module can be compiled with the dummy aliasing assumption and the other modules with the opposite setting to improve performance. The no-alias assumption should only be used for source programs that strictly obey FORTRAN 77 rules for the association of variables. But how can the programmer know for sure if there are aliases between dummy arguments and common variables or not? As mentioned in a study comparing the diagnostic capabilities of FORTRAN compilers by Appleyard [App01], no compiler provides this standard violation check, one of the most common FORTRAN pitfalls. Only Forcheck (*http://www.forcheck.nl*), a commercial FORTRAN verifier, spots violation only on aliased scalar dummy arguments and only at run-time.

The most difficult problem in FORTRAN alias analysis is to compute the exact overlapping memory locations between arrays. Overlapping for arrays in an `EQUIVALENCE` statement is known at compile-time because the subscript expressions are integer constant expressions. In parameter passing cases, such as in Figure 7.2, the worst-case assumption is: the whole arrays `V1` and `V2` are aliased. But, if two intervals `[I,I+M1-1]` and `[J,J+M2-1]` can be proved disjoint, `V1` and `V2` are not aliased, and so optimizations can be applied in `SUB2`, if this is the only call to `SUB2`. Furthermore, `V1` and `V2` can overlap, but if all the written array elements in `SUB2` are proved not to be in the overlapping section, the restrictions on association of entities in FORTRAN are not violated.

```
SUBROUTINE SUB1                         SUBROUTINE SUB2(V1,V2,N1,N2)
REAL A(100)                             REAL V1(N1), V2(N2)
READ *,I,J,M1,M2                        ...
CALL SUB2(A(I),A(J),M1,M2)              END
END
```

Figure 7.2: Array overlapping example

Programmers sometimes pass overlapping array regions purposefully, thinking that it will be safe because they know in which order the subroutine deals with the data. However, bugs related to this programming practice are very difficult to track, since compilers can reorder the subroutine's code, which results in incorrect output.

Our alias verification is broken down into three steps. Firstly, interprocedural aliases are computed for the whole program. Then, this information is used to decide statically if the program violates the standard restrictions on alias or not. When the information is not known at compile-time, tests are inserted to check violations at execution time. Finally, to avoid false alarms, the impact of alias violations is studied. Before going into details of these steps, we describe in the next section some related work on alias analysis.

7.2 Related Work

A lot of work about alias analysis has been carried out over the past 25 years. Alias computation is usually divided into two parts Muchnick [Muc97]: *alias gatherer* and *alias propagator*. Since the sources of aliases vary from language to language, the alias gatherer is a language-specific component which is provided by the compiler front end. Meanwhile, the alias propagator is a common component which performs a data-flow analysis using the aliases discovered by the alias gatherer to combine aliasing information at join points in a procedure and to propagate it to where it is needed. The various alias analyses offer different trade-offs between computational complexity and accuracy.

Pointer alias analysis algorithms use varying degrees of flow-sensitivity, calling-context and alias representation and have empirically been studied in a lot of research papers [Cou86, LR92, CBC93, Deu94, BCCH95, Ruf95, WL95, ZRL96, Ste96, SH97, HBCC99, LH01, LPH01, CRL01, DLFR01, GLS01, HT01a, HT01b, HP01, Hin01, MDCE01, RLS+01]. Ruf [Ruf95] compares the precision of a simple, efficient, context-insensitive points-to analysis for the C programming language with that of a maximally context-sensitive version of the same analysis. He found that context-insensitive techniques can be implemented quite efficiently, and can produce information that is surprisingly precise, as well as useful in improving the efficiency of context-sensitive techniques. Hind and Pioli [HP01] provide the comparison of the effectiveness (compile-time efficiency and precision) of six pointer analysis algorithms: four flow-insensitive, one flow-sensitive, one flow-insensitive but uses precomputed flow-sensitive information. The results of their paper suggest that the Steensgaard's flow-insensitive algorithm [Ste96] computing one solution set for the entire program and using a union-find data structure to avoid iteration is usable in production compilers. If better precision is required, the analyses of Andersen [And94] or Burke et al. [BCCH95] can be used. Further precision may be obtained by the interprocedural analyses in Choi et al. [CBC93] and Hind et al. [HBCC99]. The efficiency of these analyses is improved by identifying realizable execution paths and by identifying different instances of dynamically allocated objects created along different call paths. Extensions of points-to analysis algorithms for object-oriented languages such as JAVA and C++ are studied in [LPH01, CRL01, WFPS02].

However, pointer analysis is not within the scope of our work. Alias analyses for programming languages without pointers such as FORTRAN 77 are discussed earlier in the literature [Bar77, Ban79, ABC+88, Coo84, CK89, CK88, MW93]. These analyses deal with aliases arising from the renaming effects at call sites in languages with call-by-reference parameter passing. They are formulated as a data-flow analysis problem. The static call graph of a program is built and used to find the potential aliases at every procedure entry point. Banning [Ban79] presents an aliasing analysis that follows parameter binding chains through the program in a depth first fashion to compute all possible aliases. Cooper and Kennedy [CK89] improve the alias analysis computation time based on the fundamental insight that significant advantages can be achieved by separating the treatment of reference formal parameters from the treatment of global variables. Their algorithm requires $\mathtt{O}(\mathtt{N}^2+\mathtt{NE})$ steps, where $\mathtt{N}$ and $\mathtt{E}$ are the number of nodes and edges of the program's call graph, respectively. An alias detection analysis is implemented by Mayer and Wolfe in the Nascent compiler [MW93], by improving Cooper and Kennedy's algorithm. The number and cardinality of maybe aliased variable sets as well as their relationship with the number of global variables and formal parameters per procedure on PerfectClub and Riceps benchmarks are reported in the paper. They also study the interaction of dependence analysis and aliasing by considering the parallelization of loops with formal array arguments. However, these methods treat arrays as atomic objects. This granularity is not fine enough and imposes alias restrictions too strict to be useful. Moreover, the case where an actual array element can be associated to a formal array parameter seems omitted in [MW93], so aliases between two formal array parameters may go undetected. A more sophisticated analysis of arrays might produce useful information about the offsets and patterns of overlap in a program. Such an analysis is implemented in the PTRAN compiler (Allen et al., Hind et al. [ABC+88, HBCM92, HBCM94]). When an array is involved in an alias, it determines, when possible, the difference in starting address between the aliased variables. These offsets are used in dependence analysis by linearization between aliased arrays. In [HBCM94], a precise interprocedural array analysis (FIDA) is used to avoid the conservative aliasing assumption and to improve the number of parallelized loops.

The parameter-induced aliasing in ADA has recently been studied by Gellerich and Plodereder in [GP01]. They implement the alias analysis developed by [Coo84] and examine manually the introduced alias pairs. The experimental results reveal a rare occurrence of aliasing that may explain the reliability of ADA programs.

However, to our knowledge, no work has been done on the verification of restrictions on alias use, especially for array variables. This verification is critical for code safety, debugging and maintainability (referential transparency) because allowing writing on aliased variables may result in unpredictable behaviors and make optimizations impossible. Current compilers could detect the violations dynamically but run-time checks are still an overhead and they can catch only those violations that actually happen during a particular run. The objective of this work is to check the whole program, to generate a minimum number of tests by using precise alias information for both kinds of variables, scalar and array, and then to study the impact of alias violation on other program analyses and transformations, such as dependence analysis.

7.3 Interprocedural Alias Propagation

FORTRAN 77 standard [ANS83] defines several ways to create aliases and they are mostly detectable exactly during compilation. The `EQUIVALENCE` statement is used to specify statically how two or more entities in the same program unit do share storage units. The effects of aliases created by `EQUIVALENCE` statements are purely local and known at compile-time, as long as the equivalenced variables are not in common storage. The `COMMON` statement associates different variables in different subprograms to the same storage. Determining the full effects of variables in common storage requires interprocedural analysis as for aliases created by parameter passing. When a procedure is called, an association is established between the actual argument and the corresponding formal parameter in the called procedure. The formal parameter has the storage location of the actual argument for this invocation.

Formal parameters may become aliased in several ways. Two formal parameters are aliased to one another if the same actual argument is passed to both of them. Also, if a global variable is used as an actual argument and a variable aliased to the global variable is passed as another actual argument, the two corresponding formal parameters are aliased. In addition, formal parameter aliases can be passed through a call chain, creating more aliased formal parameters. A global variable can only become aliased to a formal parameter in a routine in which it is visible and only by its being passed as an actual argument to that formal parameter.

7.3.1 Storage Sequence Association

In our approach, arrays are not treated as atomic variables. To compute the overlap between arrays, the following definitions are needed to describe relationships that exist among variables, based on Section 17.1 [ANS83].

Definition 7.3.1 A *storage sequence* of a variable is a sequence of storage units that the variable represents. The *size* of a storage sequence is the number of storage units in the storage sequence.

For example, an integer type scalar variable or array element has a storage sequence of one storage unit; a double precision type has a storage sequence of two storage units. An array has a storage sequence consisting of the storage sequences of the array elements in the order determined by the array element ordering. The storage sequence of a common block consists of the storage sequences of all variables in the block in the order of their appearance.

Definition 7.3.2 Two storage sequences are *associated* if they share some storage units.

Definition 7.3.3 In a procedure, *two formal parameters are aliased by a call chain* if and only if their corresponding storage sequences for this call chain are associated.

Definition 7.3.4 In a procedure, *a formal parameter and a common variable are aliased by a call chain* if and only if their corresponding storage sequences for this call chain are associated.

7.3.2 Memory Location Naming

To decide when two storage sequences can be associated, we introduce a *memory location naming* technique that allows a compact representation of variable addresses. It provides all information required to locate a variable in the memory.

Actual Variable

Actual variables are variables that have a known address in some memory space. They include common and local variables. Since the storage sequences of two common blocks in a same program unit cannot share storage units, a unique name is given to each common block. This name is called an *area*. A local variable is stored in a static or dynamic area of the procedure where the variable is declared.

The *offset* of a variable is used to locate the variable in its area. The size of the storage sequence of this variable defines the offset of the next variable in the same area. If several variables are involved in an `EQUIVALENCE`, their offsets in the area are computed with respect to the association of the storage sequences. Since the subscript expressions in an `EQUIVALENCE` statement are known at compile-time, the computation of this information is straightforward.

Formal Variable

A *formal variable* does not have its own address. But when it is associated to an actual argument, it will have the address of the actual argument. This actual argument in turn may be a formal variable of the current caller and in this case, we have to go up the call chain until we reach an actual argument which is an actual variable. So, depending on the call path, different addresses may be associated to one formal variable.

When an array is passed in a `CALL` statement, the starting address of the formal array argument is computed using the offset of the actual array argument, and the subscript expression if an array element is passed. The subscript value expression of an array element determines the order of that element in the array, as already defined in Chapter 5. One important assumption for alias analysis is the absence of bound violations in the program, i.e to guarantee that all subscript values are bounded. This is discussed in more detail in Chapter 9.

Definition 7.3.5 The relationship between the offsets of the formal and actual array arguments is expressed by the equation:

offset(formal_array) = offset(actual_array) + element_size*(subscript_value-1)

Let A and B be, respectively the corresponding actual and formal arrays and by using the notations in Table 5.2 (see Chapter 5, page 58), we have:

$$\texttt{offset}(B) = \texttt{offset}(A) + e_a \times \sum_{i=1}^{n} \left((s_{ai} - l_{ai}) \prod_{j=1}^{i-1} d_{aj} \right)$$

The next two theorems specify the necessary conditions for an alias between a formal parameter and a common variable or two formal parameters in a procedure to exist. The treatments of formal parameters and common variables are separated.

Theorem 7.3.1 In a procedure, if a formal parameter and a common variable are aliased by one call chain, then one of the following conditions must be satisfied:

1. The corresponding actual argument is an actual variable whose area is that of the common variable. This alias is created by only one call site, regardless of the call chain.

2. The corresponding actual argument is a formal variable of the caller and following the call chain, it is located in the area of the common variable.

Proof: We prove that if the corresponding storage sequences of the formal parameter and the common variable are associated, either Condition 1 or 2 must be satisfied.

If the corresponding actual argument is an actual variable, it cannot be a local variable of the caller because a local variable of one procedure cannot be associated to a common variable of the other. This actual variable must be a common variable, and since the two common variables share the same storage units, they must be in the same common block, declared in two different procedures. So they are in the same area. The call chain has only one element, that is the direct call site.

If the corresponding actual argument is a formal variable of the caller, following the call chain, we reach an actual variable. Reasoning as in the first case, we come to the same conclusion. □

Theorem 7.3.2 In a procedure, if two formal parameters are aliased by a call chain then one of the following conditions must be satisfied:

1. The same actual argument is passed to them. This alias is created by only one call site.

2. The corresponding actual arguments are in an `EQUIVALENCE` statement of the caller. This alias is also created by only one call site.

3. One actual argument is a common variable and the other actual argument is a formal variable of the caller, and following the call chain, the address of this formal variable has the same area as the common variable.

4. The corresponding actual arguments are formal variables of the caller and following the call chain, they have addresses which share the same area.

Proof: Supposing that two formal parameters are aliased by a call chain, we have three possibilities:

1. Two corresponding actual arguments are actual variables of the caller. If they are different variables, the only way to associate their storage sequences within a program unit is to declare them in an `EQUIVALENCE` statement. So in this case, either Condition 1 or Condition 2 must be satisfied.

2. One actual argument is a formal variable and the other is an actual variable of the caller. This actual variable cannot be a local variable because the storage sequence of a local variable cannot be shared with that of another variable of another procedure. So the actual variable must be a common variable. Following the call chain, the formal variable will reach an actual variable. This actual variable in turn must be a

common variable. And since the storage sequence of two common variables in two different subprograms are associated, they must be in the same common block, and therefore the same area. Condition 3 is satisfied.

3. Both actual arguments are formal variables of the caller. The simplest case is when they are the same variables (Condition 1) and the alias is created by only one call, which is the direct caller. These actual arguments cannot be equivalenced by the caller because they are formal parameters. If not, following the call chain, we will reach, not necessarily at the same time, the two corresponding actual variables. We have two sub-cases:

   - We reach the two actual variables at the same call site. As in the first case, these actual variables are the same variables or in an `EQUIVALENCE` statement of the current call site procedure. Their addresses must share the same area.

   - If one actual variable is reached before the other, which means that they are actual variables of two different subprograms and since their storage sequences are associated, they must be common variables in the same common block and therefore in the same area.

   Condition 4 is satisfied by these sub-cases.

The theorem is proved by enumeration. □

7.3.3 Alias Propagation Algorithm

The basic idea for computing interprocedural aliases is to follow all the possible chains of parameter bindings at all call sites. The acyclic call graph is traversed in the invocation order (see Chapter 3, page 23) and alias information is accumulated incrementally from the main program. For each procedure, symbolic addresses of formal parameters of its callers are available and used to compute the addresses of its formal parameters. Our analysis is context-sensitive since it distinguishes the different calls to a procedure by storing the call path that produces the alias. Algorithm 7.3.1 shows the interprocedural alias propagation, based on Theorem 7.3.1 and Theorem 7.3.2. This propagation computes in fact an over-approximated set of aliases in a procedure.

By dividing the list of actual arguments into groups of identical or equivalenced variables, all formal parameters in a same group have the same area. If a same variable is passed to different formal parameters, a special area called `ALIAS_AREA_g` and a call chain of one element are associated to the formal parameters (Condition 1 and 2 of Theorem 7.3.2). The advantage is that although the actual argument can be a formal parameter of the caller, we do not need to keep the whole call chain in which an actual variable is reached. If the actual argument a is a common variable, an alias may exist under Condition 1 of Theorem 7.3.1 if a is visible in the called procedure and Condition 3 of Theorem 7.3.2

if there exists another actual argument that is a formal variable of the caller and one of its addresses has the same area as a.

Algorithm 7.3.1

procedure Interprocedural_Alias_Propagation (p)
  p : current procedure
begin
  for each call site c to p
    q := corresponding_caller(c)
    partition the actual argument list into groups of same and equivalenced variables
    for each group g
      for each actual argument a in g
        f = corresponding_formal_argument(a, c)
        if g is a group of same variables **then**
          area(f) = ALIAS_AREA_g
        else
          area(f) = area(a)
        endif
        call_chain(f) = $\{c\}$
        offset(f) = compute_and_translate_offset(a, c, p, q)
      endfor
    endfor
    for each actual argument a in the argument list
      if a is a common variable **then**
        f = corresponding_formal_argument(a, c)
        area(f) = area(a)
        call_chain(f) = $\{c\}$
        offset(f) = compute_and_translate_offset(a, c, p, q)
      endif
      if a is a formal variable and one of its addresses has a common area or the same area as an address of another actual argument that is a formal variable **then**
        f = corresponding_formal_argument(a, c)
        area(f) = corresponding_area(a)
        call_chain(f) = corresponding_call_chain(a) $\cup \{c\}$
        offset(f) = compute_and_translate_offset(a, c, p, q)
      endif
    endfor
  endfor
end

Algorithm 7.3.2

function compute_and_translate_offset(a, c, p, q)
begin
  off = offset(a) + subscript_value(a, c)
  if off can be translated to the frame of p **then**
    off = translate_to_callee_frame(off,c, p, q)
  else
    off = unknown
  endif
  return off
end

If a is a formal variable and one of its addresses has a common area which is visible in p, there may be an alias according to Condition 2 of Theorem 7.3.1. If this address has the same area as another common actual argument, an alias may exist under Condition 3 of Theorem 7.3.2. In the last case, if this address has the same area as an address of another actual argument that is a formal variable, an alias may happen following Condition 4 of Theorem 7.3.2.

The *compute_and_translate_offset* function in Algorithm 7.3.2 computes the offset of the formal parameter with respect to that of the actual argument and the subscript value. The offset is translated to the frame of the procedure p by using global variables information and the bindings between actual and formal parameters. If the translation is not possible, an unknown expression is returned and the alias verification phase treats this case differently, as discussed in the next section.

Alias Propagation Example

Figure 7.3 illustrates the alias propagation. Each formal parameter of subroutine `SUB2` has two addresses corresponding to two different call paths: `{MAIN:CALL SUB2(W,B,50)}` and `{MAIN:CALL SUB1(A,100), SUB1:CALL SUB2(V,V(M+1),M)}`.

There is no possible alias for `SUB1`. The call site `CALL SUB2(W,B,50)` in the main program and `CALL SUB2(V,V(M+1),M)` in procedure `SUB1` may cause aliases for `SUB2`.

In the first call, the common variable `W` that is visible in `SUB2` is passed as an actual argument. An address is added for `V1` by the alias propagation. The area of this address is that of `W`, the offset is computed from zero and the call chain is this call site.

- area(V1) = COMMON:COM
- offset(V1) = 0
- call_chain(V1) = {MAIN:CALL SUB2(W,B,50)}

In the second call site, array `V` is associated to two different formal parameters `V1` and `V2`. The offset of `V1` and `V2` is computed with respect to that of `V` and the subscript value in the actual argument list. A special area is used and we only need to keep the direct caller of SUB2. The element size of real variable is 4.

- area(V1) = area(V2) = ALIAS_AREA_1
- offset(V1) = 0, offset(V2) = 4*L
- call_chain(V1) = call_chain(V2) = {SUB1:CALL SUB2(V,V(M+1),M)}

So for each procedure, the propagation phase produces compact and precise alias information that is used in the next phase to check the restrictions on aliased variables.

```
PROGRAM MAIN
COMMON /COM/ W(50)
REAL A(100),B(50)
CALL SUB2(W,B,50)
CALL SUB1(A,100)
END

SUBROUTINE SUB1(V,N)
REAL V(N)
READ *,M
IF (2*M.LE.N) THEN
   CALL SUB2(V,V(M+1),M)
ENDIF
END

SUBROUTINE SUB2(V1,V2,L)
COMMON /COM/ W(50)
REAL V1(L),V2(L)
DO I = 1,L
   V1(I) = V2(I)
ENDDO
END
```

Figure 7.3: Interprocedural alias propagation example

7.4 Interprocedural Alias Verification

We enforce the FORTRAN standard about the restrictions on association of entities (Section 15.9.3.6 in [ANS83]). However, the restrictions on association of array variables are not so clear. Arrays can be treated as units, that is, if two formal array parameters or one formal array parameter and one global array variable are aliased, assignment on any element of any array is forbidden. This granularity is not fine enough because we may not write on the aliased elements, which identify the same storage units.

Arrays can also be treated element by element. Each array element is considered as a scalar variable. This interpretation is the most complicated because we must compare between all different referenced array elements during program execution, which is not often performed efficiently using static analysis. So we give other definitions of the so called writing violation on aliased variables in a subprogram, which is finer than the first interpretation but less complicated than the last one.

7.4.1 Alias Violation Condition

Definition 7.4.1 A call chain causes an *alias violation of formal parameter* (*of common variable*) in a subprogram if, following this call chain, there exists two different formal parameters (a common variable and a formal parameter) sharing storage units that are defined during the execution of the subprogram.

Note that if the storage sequences of variables concerned are disjoint, they share no storage unit. Therefore, we have:

Lemma 7.4.1 In a subprogram, if the storage sequences of all formal parameters (if the storage sequences of all formal parameters and those of all common variables) allocated by the execution of a call chain are disjoint, there is no alias violation of formal parameter (of common variable) caused by this call chain.

The next two lemmas show that the alias violation depends on writing or not on the overlapping section of two aliased variables. The sequence of storage units that are defined during execution of the subprogram is called *defining sequence*.

Lemma 7.4.2 In a subprogram, if the defining sequence of every formal parameter (every common variable) is disjoint with the storage sequence of all other formal parameters allocated by the execution of a call chain, there is no alias violation of formal parameter (of common variable) caused by this call chain.

Lemma 7.4.3 In a subprogram, if there exists a formal parameter (a common variable) whose defining sequence is not disjoint with the storage sequence of another formal parameter allocated by the execution of a call chain, there is an alias violation of formal parameter (of common variable) caused by this call chain.

The alias verification on each procedure of the program is defined by these three lemmas. Procedures with no formal parameter and procedures with formal parameters but no possible alias (calculated during the propagation phase) are excluded. To compute the defining sequence of a variable in a procedure, we need information about write effects of statements on this variable. In general, the exact effect is only known at elementary statements such as assignments because of different control paths. However, to reduce the number of analyzed pairs of aliased variables, we can use the *summary effect* [KAC+96] of a procedure on its formal and common variables. This is an over approximation of the exact effects. By using this information, if a variable is never defined during the procedure execution, we do not have to treat it.

7.4.2 Alias Verification Algorithm

A necessary condition for two formal parameters or one formal parameter and one global variable to be aliased by a call chain is that their addresses have the same area. The alias verification applied for each procedure having at least one formal parameter and possible alias is given by Algorithm 7.4.1.

Algorithm 7.4.1

procedure Interprocedural_Alias_Verification (p)
    p : current procedure
begin
    for each maybe_aliased_variables_by_call_chain(v_1, v_2, cc)
        o_1 = corresponding_offset(v_1)
        o_2 = corresponding_offset(v_2)
        if $o_1 \neq$ unknown and $o_2 \neq$ unknown **then**
            alias_check_in_callee(v_1, v_2, o_1, o_2, p)
        else
            c = direct_call_site(cc)
            q = direct_caller(cc)
            o'_1 = offset_in_caller(v_1, c, q)
            o'_2 = offset_in_caller(v_2, c, q)
            if $o'_1 \neq$ unknown and $o'_2 \neq$ unknown **then**
                alias_check_in_caller(v_1, v_2, o'_1, o'_2, q)
            else
                direct_alias_check(v_1, v_2, p)
            endif
        endif
    endfor
end

For each maybe aliased pair of variables where at least one of them is present in the summary write effects of the procedure, the verification works as follow. If the offsets of both variables are known, depending on the type of variable, scalar or array, different check cases are performed in the frame of the current procedure (Algorithm 7.4.2). As a matter of space, we only describe the most complicated case, when both variables are array variables. In the cases of Lemma 7.4.1 and Lemma 7.4.2, there is no alias violation between the two arrays. Otherwise, a test is inserted before each statement that writes on one variable. Flags are inserted before each call site in the call chain to mark if the execution reaches the current statement or not. If one clause $o_j \leq r_i$ or $r_i \leq o_j + s_j$ $(i, j = 1, 2\ i \neq j)$ is known to be true at compile-time, it can be removed from the test condition.

Algorithm 7.4.2

procedure Alias_check_in_callee (v_1, v_2, o_1, o_2, p)
begin
  ... both variables are array variables ...
  s_1 = size_of_storage_sequence(v_1)
  s_2 = size_of_storage_sequence(v_2)
  if $(o_2 + s_2 \leq o_1)$ or $(o_1 + s_1 \leq o_2)$ is true **then**
    no alias violation
  else
    for each statement s in p that writes on $v_i, i = 1, 2$
      $r_i = o_i$ + element_size(v_i) * (subscript_value(v_i, s) - 1)
      if $r_i < o_j$ or $r_i > o_j + s_j$ is true **then**
        no alias violation
      else
        insert $flag_k = .true.$ before each call site k in cc $(k = 1, n)$
        insert $IF\ (flag_1\ .and.\ ...\ flag_n\ .and.\ (o_j \leq r_i)\ .and.\ (r_i \leq o_j + s_j))$
          $STOP\ message$ before s
      endif
    endfor
  endif
end

If at least one offset is unknown because it cannot be translated to the frame of the called procedure in the propagation phase, we try to check for alias violation in the frame of the direct caller of the current procedure (Algorithm 7.4.3). The new offset of each variable is computed in the caller's frame by using function *offset_in_caller*, described as follows:

- If the variable is a common variable, its offset in the common block does not change and is returned if the variable is visible in the caller. Otherwise, an unknown expression is returned.
- If the variable is a formal parameter, we return the left hand-side expression in Definition 7.3.5 (page 114). In fact, this is the value before the translation step in the propagation phase. If the offset of the actual argument is unknown because the actual argument is a formal parameter of the caller and information is lost somewhere earlier in the call chain, we return an unknown expression.

If both new offsets can be computed in the caller's frame, we repeat the alias violation checking between two variables as above, but in the frame of the caller. Each time the size of the storage sequence or the subscript value is needed, we have to translate it to the caller's frame, by using global variables information and the bindings between actual and formal parameters. Tests inserted before each statement defining a variable in the current procedure and before the direct call site are different from those in the first case.

Algorithm 7.4.3

procedure Alias_check_in_caller(v_1, v_2, o'_1, o'_2, q)
begin
    ... both variables are array variables ...
    s_1 = size_of_storage_sequence(v_1)
    s_2 = size_of_storage_sequence(v_2)
    if s_1 and s_2 can be translated to the frame of q **then**
        s'_1 = translate_to_caller_frame(s_1, c, p, q)
        s'_2 = translate_to_caller_frame(s_2, c, p, q)
        if $(o'_2 + s'_2 \leq o'_1)$ or $(o'_1 + s'_1 \leq o'_2)$ is true **then**
            no alias violation
        else
            for each statement s in p that writes on $v_i, i = 1, 2$
                $r_i = o_i$ + element_size(v_i) * (subscript_value(v_i, s) - 1)
                if r_i can be translated to the frame of q **then**
                    r'_i = translate_to_caller_frame(r_i, c, p, q)
                    if $r'_i < o'_j$ or $r'_i > o'_j + s'_j$ is true **then**
                        no alias violation
                    else
                        insert $flag_k = .true.$ before each call site k in cc $(k = 1, n-1)$
                        insert $IF\ (o'_j \leq r'_i).and.(r'_i \leq o'_j + s'_j)\ flag_n = .true.$ before c
                        insert $IF\ (flag_1\ .and.\ ...\ flag_n)\ STOP\ message$ before s
                    endif
                else
                    direct_alias_check(v_1, v_2, p)
                endif
            endfor
        endif
    else
        direct_alias_check(v_1, v_2, p)
    endif
end

If the translation is not possible, or one offset is unknown, we have to use the direct alias check version that inserts, before each statement defining a variable, for example v_1, the procedure call CALL ALIAS_CHECK(message, $\mathtt{v}_1, \mathtt{r}_1, \mathtt{v}_2, \mathtt{s}_2$) where $\mathtt{r}_1$ is computed as in Algorithm 7.4.2 and $\mathtt{s}_2$ is the size of the storage sequence of $\mathtt{v}_2$. ALIAS_CHECK(char $*$message, void$*$ $\mathtt{p}_1$, int $*$ $\mathtt{i}_1$, void $*$ $\mathtt{p}_2$, int $*$ $\mathtt{i}_2$) is a C function that takes the base address in the memory of two variables, the referenced element of one variable and the size of the other to check if there is alias violation on the referenced element or not. This check is expensive because no static analysis is exploited.

Alias Verification Example

In the example in Section 7.3 (page 119), no alias violations are caused by CALL SUB2(V,V(M+1),M) because the intersection between [0:4*L-1] and [4*L:8*L-1] is empty (the storage sequence size of each array is 4*L). However, alias violations occur when writing on V1(I) which is aliased with an element in W. The instrumented code is shown in Figure 7.4.

```
BLOCK DATA ALIAS
COMMON /ALIAS_FLAGS/ ALIAS_FLAG(1)
LOGICAL ALIAS_FLAG(1)
DATA ALIAS_FLAG /1*.FALSE./
END

PROGRAM MAIN
COMMON /COM/ W(50)
REAL A(100),B(50)
COMMON /ALIAS_FLAGS/ ALIAS_FLAG(1)
LOGICAL ALIAS_FLAG(1)
ALIAS_FLAG(1) = .TRUE.
CALL SUB2(W,B,50)
CALL SUB1(A,100)
END

SUBROUTINE SUB2(V1,V2,L)
COMMON /COM/ W(50)
REAL V1(L),V2(L)
COMMON /ALIAS_FLAGS/ ALIAS_FLAG(1)
LOGICAL ALIAS_FLAG(1)
DO I = 1,L
   IF (I.GE.1.AND.I.LE.50.AND.ALIAS_FLAG(1)) STOP "Alias violation:
   write on V1, aliased with W by CALL SUB2(W,B,50) in MAIN"
   V1(I) = V2(I)
ENDDO
END
```

Figure 7.4: Interprocedural alias verification example

The generated alias checks are expensive because they are left inside loops. They can be optimized by safely applying code hoisting since our instrumented code does not violate the standard anymore.

7.5 Alias Violation Impact

Although modifying an aliased variable may let the program produce incorrect results or behave unpredictably when using optimization, this violation on alias restrictions occurs occasionally in established programming practice. This section studies the impact of these violations on program transformations and optimizations.

As explained by Allen and Kennedy in [AK02], a transformation preserves the semantics of the program if the control and data dependences are respected. Any ordering-based optimization that does not change the dependences of a program is guaranteed not to change the results of the program.

```
      SUBROUTINE FCT(UNEW,UTRA,UOLD)
      PARAMETER (MP = 402, NP = 160)
      DIMENSION UNEW(MP,NP), UTRA(MP,NP), UOLD(MP,NP)
      DO I = 1,MQ
         DO J = 1,NQ
s1          DZ1(I,J) = UOLD(I+1,J) - UOLD(I,J)
s2          AZ1(I,J) = UTRA(I+1,J) - UTRA(I,J)
         ENDDO
      ENDDO
      DO I = 1,MQ
         DO J = 1,NQ
s3          UTRA(I,J) = UTRA(I,J) + DZ1(I,J) - DZ1(I-1,J)
         ENDDO
      ENDDO
      DO I = 1,MQ
         DO J = 1,NQ
s4          UNEW(I,J) = UTRA(I,J) - AZ1(I,J) - AZ1(I-1,J)
         ENDDO
      ENDDO
      END
```

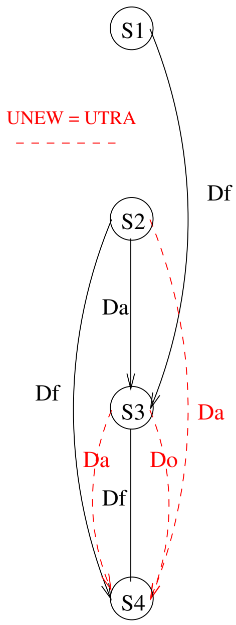

Figure 7.5: A code fragment excerpt from *hydro2d*, SPEC95 CFP

So the effect of new dependence arcs created by aliasing violation on the data dependence graph is important because it decides whether other program transformations on the procedure are safe or not. If these new data dependence arcs are redundant with existing paths in the data dependence graph, the aliases have no impact on ordering-based optimizations. Take as an example a piece of code from the benchmark *hydro2d* in SPEC95 CFP where calls to subroutine `FCT` make two formal parameters `UNEW` and `UTRA` aliased and they are defined by the called module. As shown in Figure 7.5, the new data dependence arcs (dashed arcs) created by the alias between `UNEW` and `UTRA` are redundant with existing paths in the dependence graph. So we can prove that all ordering-based optimizations for `FCT` are safe.

This phase has been implemented in PIPS but only for scalar variables. The effect of array alias violation is studied by hand with the SPEC95 CFP benchmarks, that is shown in the next section.

7.6 Experimental Results

We use the SPEC95 CFP benchmarks that contain ten existing applications written in FORTRAN 77. We are not interested in *tomcatv* which is a single procedure program. For the other nine benchmarks, the number of routines in a program varies from 6 to 105; the number of procedure and function calls is in the range from 5 to 243. Table 7.1 summarizes relevant information for all benchmarks in SPEC95 CFP.

Table 7.1: SPEC95 CFP: numbers of lines, modules, calls and array resizing (AAR: argument association rules, AR: array region, CI: code instrumentation)

Benchmark	**Line**	**Module**	**Call**	**AAR + CI**	**AR + CI**
tomcatv	190	1	0		
swim	429	6	5		
su2cor	2332	35	166		
hydro2d	4292	42	98		
mgrid	484	12	23		
applu	3868	16	27	*	
turb3d	2101	23	111	*	
apsi	7361	96	190	*	*
fpppp	2784	38	27	*	
wave5	7764	105	243	*	*

Some array declarations conflict with alias analysis. The ugly and assumed-size array declarations (see Chapter 5, page 53) prevent us from knowing the storage sequence size of some arrays in the called procedures. Unlike other compilers that ignore these arrays, we deal with this by applying array resizing (Chapter 5) to five benchmarks: *applu, turb3d, apsi, fpppp* and *wave5*. Assumed-size declarations also occur in two other benchmarks, *su2cor* and *hydro2d*, but the corresponding arrays are not involved in aliases so array resizing is not needed for these benchmarks. Two combinations of array resizing are applied: argument association rules with code instrumentation (AAR + CI) and the array region based approach with code instrumentation (AR + CI). Code instrumentation is used to end up with 100% resized arrays.

The numbers of inserted flags, tests, direct alias checks, modules with possible aliases and the compilation times (in minutes and seconds) are reported in Table 7.2. Compilation times for both alias propagation and verification phases are measured on an UltraSparc II 440MHz 256Mo RAM. Codes with generated checks are then compiled and executed using the standard input data for SPEC95 CFP benchmarks. Six out of ten benchmarks violate alias rules: *hydro2d, mgrid, applu, turb3d, apsi* and *wave5*. As *tomcatv* has only one routine, there are certainly no aliases for this benchmark. *swim* and *su2cor* are proved to be free of dummy aliased variables by our analysis. *fpppp* is instrumented with 250 checks, generating 2% execution slowdown and has no alias violations for its standard input files. Runtime slowdown cannot be measured for the other benchmarks because the execution is interrupted at the first violation.

Table 7.2: SPEC95 CFP: number of inserted flags, tests, direct alias checks and modules with possible aliases; compilation time (in minutes and seconds) and alias violation result (AR: array resizing, DG: dependence graph)

Benchmark	**Flag**	**Test**	**Check**	**Mod**	**Compilation**	**Violation**	**AR**	**DG**
tomcatv	0	0	0	0	0:00	No		
swim	0	0	0	0	0:01	No		
su2cor	0	0	0	0	0:14	No		
hydro2d	8	12	0	2	0:09	Yes		*
mgrid	0	0	10	3	0:01	Yes		*
applu	6	6	0	2	0:08	Yes		*
turb3d	60	156	13	12	0:10	Yes		*
apsi	23	194	2945	66	1:32	Yes	*	
fpppp	0	0	250	1	0:30	No		
wave5	36	334	495	24	1:38	Yes	*	

Column 5 (Table 7.2) shows the number of modules where there may be aliases between its formal parameters and common variables. It is important to see that, in comparison to Column 3 (Table 7.1), this number only takes a small percentage. For example, only one called module in *fpppp* must be compiled with the dummy aliasing assumption to assert the presence of aliases. Meanwhile, the total number of called modules in this benchmark is 37. The no alias assumption can be safely applied to the remaining modules in the program; hence the execution performance is improved.

Two kinds of alias violations appear in six benchmarks. The first category exists in *apsi* and *wave5*: assumed-size arrays are resized with new dimensions that are too large with respect to actual array accesses. Figure 7.6 gives an illustrating example.

With the argument association rule approach to array resizing, we infer new declarations: `DIMENSION X(1000),Y(1000-L),Z(1000-2*L)`. These declarations are correct with respect to the standard but they may be too large for actual array accesses in the called

```
PROGRAM ALIAS                                SUBROUTINE RUN(X,Y,Z,L)
REAL WORK(1000)                              DIMENSION X(*),Y(*),Z(*)
CALL RUN(WORK,WORK(L+1),WORK(2*L+1),L)       ...
END                                          END
```

Figure 7.6: Assumed-size array declaration example

routines. The intersection between the storage sequence of array `X`, computed from the array size, and the defining sequence of array `Y` or `Z` are not empty. So alias alarms are raised because the actual accesses of array `X` are not taken into account. To cope with this problem, we use the array region based approach to array resizing. With this approach, array declarations are more precise and greatly reduce the number of alias violations. However, it is not always possible to compute the array regions, due to non-linear expressions, indirection arrays, the lack of structure of programs, etc. Code instrumentation is used for the remaining arrays where we cannot find a proper upper bound of the last dimension by using array regions. However, the principle of code instrumentation is the same as the argument association rules approach, arrays are resized with too large dimensions so alias violations still exist in these two benchmarks. A good programming practice is to pass disjoint array sections to the called routine, i.e `DIMENSION X(L),Y(L),Z(L)`. This motivates us to use alias information to derive proper array declarations, a problem not only in FORTRAN but also in other programming languages such as C, MATLAB and APL. This prospective solution is described in Chapter 9.

The second category of alias violation includes real violations because two scalar variables or two array elements share the same storage units and one of them is written; but the legal schedules are not changed by these aliases. Example on *hydro2d* in Section 7.5 illustrates this kind of violation. Aliases in *mgrid, applu* and *turb3d* are simpler than in *hydro2d* because one aliased variable is defined only once in the referenced procedure, and the other aliased variables are not used in this procedure.

To sum up, although the standard specification is violated by some programs, we did not find any aliasing problem in the SPEC95 CFP benchmarks. This is no surprise since these benchmarks are well-debugged. We applied the analysis to a large scale industrial application, about 100.000 lines of code, and found several potential bugs. In addition, to catch different errors in one run, the `STOP` is replaced by the `PRINT` message with all available information to help debugging.

7.7 Conclusion

Our alias analysis is flow- and context- sensitive and gives efficient and precise alias information. This information lets us avoid the worst-case assumption about aliases and perform program analyses under less restrict assumptions about the program quality. For example, without alias information, the used-before-set analysis (see Chapter 8) can give false results because some variables may be initialized implicitly by aliasing. Other optimizations such as load and store reordering, partial redundancy elimination, constant propagation and copy propagation are hard or impossible to carry out because of aliasing. Our analysis can be applied to other programming languages with the call-by-reference mechanism.

We developed algorithms to check for violation of aliasing rules in FORTRAN, an option which is missing in most compilers. Violations are detected not only for scalar but also for array variables. This standard checking is useful to debug code, to help programmers to correct errors in order to gain performance by applying optimizations safely. Once alias checks are generated, the instrumented code respects the standard about aliasing. Other techniques such as code hoisting, partial redundancy elimination, induction variable optimization can be safely used to optimize the generated code. On the other hand, a program can be guaranteed to be free of dynamic alias errors by our static analysis.

The experimental results show that the SPEC95 CFP programs do not suffer from effective aliasing errors. They also show that accurate alias detection depends on array resizing and dependence graph analysis, and that alias analysis cannot be fully evaluated with standard benchmarks. Experiments on less well debugged codes are necessary.

Chapter 8

UNINITIALIZED VARIABLES

Another common programming error is the use of a variable before its definition. As well as an out-of-bound violation, this undefined value may produce incorrect results, memory violations, unpredictable behaviors and program failure. To detect this kind of error, two approaches can be used: compile-time analysis and run-time checking. However, compile-time analysis is far from perfect because of complicated data and control flows as well as arrays with non-linear, indirection subscripts, etc. On the other hand, dynamic checking, although supported by hardware and compiler techniques, is costly due to heavy code instrumentation while information available at compile-time is not taken into account. We develop in this chapter a combination of an advanced compile-time analysis and a source code instrumentation to detect errors due to a forgotten initialization.

The chapter is organized as follows. Section 8.1 describes the problems encountered with the initialization checking. Section 8.2 presents some related work in this area. Section 8.3 explains the imported array regions analysis, initially developed in [CI95, Cre95, Cre96]. Our used-before-set verification method based on these array regions is presented in Section 8.4. In Section 8.5, the experimental results obtained with the SPEC95 CFP benchmark suite are given. Conclusions are drawn in Section 8.6.

8.1 Used-before-set Problems

Used-before-set refers to the error occurring when a program uses a variable which has not been assigned a value. This uninitialized variable, once used in a calculation, can be quickly propagated throughout the entire program and anything may happen. The program may produce different results each time it runs, or may crash for no apparent reason, or may behave unpredictably. This is also a known problem for embedded software. To detect this kind of error, two approaches can be used: compile-time analysis and run-time checking. However, compile-time analysis is far from perfect because complicated data and control flows result in a very large imprecision. Furthermore, the use of global variables and arrays with non-linear, indirection subscripts, etc sometimes makes static checking completely ineffective, leading to many spurious warnings. In addition, some other program analyses such as points-to analysis [Ste96], array bound checking (Chapter 6) and alias analysis (Chapter 7) are prerequisite for the detection of uninitialized variable uses.

On the other hand, pure dynamic checking is costly due to heavy code instrumentation while information available at compile-time is not taken into account. The slowdown between instrumented and uninstrumented codes has been measured to be up to 130 times by Loginov et al. [LYHR01]. Dynamic checking is not so effective that, as shown in a report comparing FORTRAN compilers Appleyard [App01], only the Lahey/Fujitsu and Salford compilers offer run-time checking for all kinds of variables. The other compilers such as APF (Absoft Pro Fortran) version 7.5, G77 version 0.5.26, NAS (N.A.Software FortranPlus) version 2.2 and PGI (Portland Group HPF Compiler) version 3.2-4 do not have this option. Intel Fortran Compiler version 6.0 and NAGWare F95 version 4.1 only check for local and formal scalar variables; array and global variables are omitted. The code instrumentation degrades the execution performance so it can only be used to create a test version of the program, not a production version. In addition, run-time checking only validates the code for a specific input.

With the growth of hardware performance, software systems have become more and more complicated to solve better real application problems. Debugging several million lines of code becomes more difficult and time-consuming. Execution time overheads of dynamic checking or a large number of possibly undefined variable warnings issued by static checking are not highly appreciated. Efficient compile-time analysis to prove the safety of programs, to detect statically program errors, or to reduce the number of run-time checks is necessary.

The question is, by using advanced interprocedural analyses, i.e interprocedural array analysis, can static analysis be an adequate answer to the used-before-set problem for scientific codes? If not, can a combination of static and dynamic analyses reduce the cost of uninitialized variable checking? The goal of our research is to provide a more precise and efficient static analysis to detect uninitialized variables, and if sufficient information is not available, minimal run-time checks are added to guarantee the program correctness.

8.2 Related Work

To cope with the used-before-set problem, most compilers silently initialize variables to a predefined value such as zero, so that programs work consistently, but give incorrect results. Other compilers provide a run-time check option to spot uses of undefined values. There are different solutions.

8.2.1 Hardware Support and Instrumentation

The run-time error detection can be done by initializing each variable with a special value, depending on the variable type. If this value is encountered in a computation, a trap is activated. This technique was pioneered by Watfor, a FORTRAN debugging environment for IBM mainframes in the 70's, and then used in Salford, SGI and Cray compilers.

For example, the option *trap_uninitialized* of SGI compilers forces all real variables to be initialized with a NaN value (Not a Number - IEEE Standard 754 Floating Point Numbers) and when this value is involved in a floating-point calculation, it causes a floating-point trap. This approach raise several problems. Exception handler functions or compiler debugging options can be used to find the location of the exception but they are platform- and compiler-dependent. Furthermore, the IEEE invalid exception can be trapped for other reasons, not necessarily an uninitialized variable. In addition, when no floating-point calculation is done, e.g in the assignment `X = Y`, no used-before-set error is detected which makes tracking the origin of an error detected later difficult. Other kinds of variables such as integer, logical, etc are not checked for uninitialization. In conclusion, the execution overhead of this technique is low but the used-before-set debugging is almost impossible.

Other compilers such as Lahey/Fujitsu compilers, SUN dbx debugger, etc use a *memory coloring algorithm* to detect run-time errors. For example, Valgrind, an open-source memory debugger (*http://devel-home.kde.org/sewardj*) tracks each byte of memory in the original program with nine status bits, one of which tracks the addressability of that byte, while the other eight track the validity of the byte. As a result, it can detect the use of single uninitialized bits, and does not report spurious errors on bit-field operations. The object code instrumentation method is used in Purify, a commercial memory access checking tool. Each memory access is intercepted and monitored. The advantage of this method is that it does not require recompilation and it supports libraries. However, the method is instruction set and operating system dependent. The memory usage is bigger than other methods. Furthermore, the program semantics is lost. An average slowdown of 5.5 times between the generated code and the initial code is reported for Purify in [HJ92].

The plusFORT toolkit (*http://www.polyhedron.com*) instruments source code with probe routines so that uninitialized data can be spotted at run-time using any compiler and platform. There are functions to set variables to undefined, and functions to verify if a data item is defined or not. Variables of all types are checked. The amount of information about the violation provided by plusFORT is precise and useful for debugging. The name of the subprogram, the line where the reference to an uninitialized variable occurred is reported in a log-file. However, the instrumentation is far more effective because of inserted code. Figure 8.1 shows that plusFORT can detect bugs which depend on external data, but the execution time is greatly increased with such dynamic tests.

8.2.2 Static Analyzers

To reduce the execution cost, illegal uses of an uninitialized variable can be detected at compile-time by some compilers and static analyzers. LCLint [EGHT94] is an advanced C static checker that uses formal specification written in the LCL language to detect

```
      SUBROUTINE QKNUM(IVAL, POS)
      INTEGER IVAL, POS, VALS(50), IOS
      CALL SB$ENT('QKNUM', 'DYNBEF.FOR')
      CALL UD$I4(IOS)
      CALL UD$AI4(50, VALS)
      READ (11, *, IOSTAT = IOS) VALS
      CALL QD$I4('IOS', IOS, 4)
      IF (IOS .EQ. 0) THEN
         DO POS = 1,50
            CALL QD$I4('VALS(POS)', VALS(POS), 6)
            CALL QD$I4('IVAL', IVAL, 6)
            IF (IVAL .EQ. VALS(POS)) GOTO 10
         ENDDO
      ENDIF
      POS = 0
10    CALL SB$EXI
      END
```

Figure 8.1: Example with probe routines from plusFORT

instances where the value of a location may be used before it is defined. Although few spurious warnings are generated, there are cases where LCLint cannot determine if a use-before-definition error is present, so a message may be issued for a non-existing problem. In other cases, a real problem may go undetected because of some simplified assumptions.

The static analyzer ftnchek (*http://www.dsm.fordham.edu/ftnchek*) gives efficient warnings about possible uninitialized variables, but the analysis is not complete. Warnings about common variables are only given for cases in which a variable is used in some routine but not set in any other routine. It also has the same problems as LCLint about non-existing or undetected errors, because of, for example the simplified rule about equivalenced arrays.

Reps et al. [RHS95] consider the possibly uninitialized variable as an IFDS (*interprocedural, finite, distributive, subsets*) problem. A precise interprocedural data flow analysis via graph reachability is implemented with the Tabulation Algorithm to report the uses of possibly uninitialized variables. They compare the accuracy and time requirement of the Tabulation Algorithm with a naive algorithm that considers all execution paths, not only interprocedurally realizable ones. The number of possibly uninitialized variables detected by their algorithm ranges from 9% to 99% of that detected by the naive one. However, this is only an over-approximation that does not give an exact answer if there are really use-before-set errors in the program or not. The number of possibly undefined variables is rather high, 543 variables for a 897 line program, 894 variables for a 1345 line program.

PolySpace technologies (*http://www.polyspace.com*) apply abstract interpretation, the theory of semantic language approximations, to detect automatically read accesses to non-initialized data. This technique predicts efficiently run-time errors and information about maybe non-initialized variables can be useful for the debugging process of C and Ada programs. But no data is given by the PolySpace group so we cannot compare with them.

Another related work is that of Feautrier [Fea91]. He proposes to compute for each use of a scalar or array cell its *source function*, which gives the statement that is the source of the value contained therein at a given instant of the program execution. The knowledge of the source statement allows to solve many problems which include automatic single assignment form (SSA) translation or program optimizations such as array expansion, dead code elimination, etc. Uninitialized variable checking can be done by verifying in the source the presence of the sign $\perp$, which indicates access to an undefined memory cell. Unfortunately, the input language in this paper is restricted to assignment statements, `FOR` loops, affine indices and loop limits.

The main difficulties encountered by static analysis for the used-before-set verification are complicated data and control flows with different kinds of variables. This explains why only a small set of variables such as scalar and local variables is checked by some static analyzers and compilers. Our motivation is to develop a more efficient program analysis to the used-before-set problem by using imported array regions.

8.3 Imported Array Region Analysis

As described in Chapter 4, Section 4.5, array region analyses collect information about the way array elements used and defined by programs. `READ` and `WRITE` regions summarize the effects of statements and procedures upon array elements. However, they do not represent the order in which references to array elements are executed, or in other words, array data flow information, which is essential for some program optimizations.

8.3.1 Array Data Flow

`IN` array regions are introduced in [CI95, Cre95, Cre96] to summarize the set of array elements whose values are *imported* (or *locally upward exposed*) by the current piece of code. They differ from `READ` regions because the order in which array elements are defined is taken into account. One array element may be imported by a fragment of code if there exists at least one use of the variable whose value has not been defined earlier along one path in the fragment itself. For instance, in the illustrative example in Figure 8.2, the element `B(J,K)` in the second J loop is read but its value is not imported by the loop body because it is previously defined in the same iteration. On the contrary, the element `B(J,K-1)` is imported from the first J loop.

Like `READ` and `WRITE` regions, the imported region of an array A of n dimensions at a statement s is defined as a function from the set of states to the power-set of $\mathbb{Z}^n$:

```
K = FOO()
DO I = 1,N
   DO J = 1,N
      B(J,K) = J + K
   ENDDO
   K = K + 1
   DO J = 1,N
      B(J,K) = J*J - K*K
      A(I) = A(I) + B(J,K) + B(J,K-1)
   ENDDO
ENDDO
```

Figure 8.2: Imported array region example

$$\begin{aligned} \mathcal{R}_{in} : Statement \longrightarrow \quad & State \longrightarrow \wp(\mathbb{Z}^n) \\ \mathcal{R}_{in}(s) \longmapsto \quad & \lambda\sigma.(\{\phi = (\phi_1, \ldots, \phi_n) \in \mathbb{Z}^n : r(\phi, \sigma)\}) \end{aligned}$$

where r is the relation between the imported array element ϕ and the current program state σ. Let $e \in Expression, s \in Statement$ and $\sigma \in State$. The semantic function of `IN` array region is defined as follows:

$$\begin{array}{lll}
\mathcal{R}_{in}(s) \triangleq \lambda\sigma.(\text{case } s \text{ of} & & \\
\text{CONTINUE} & \longrightarrow & \emptyset \\
\text{STOP} & \longrightarrow & \emptyset \\
\text{READ } var & \longrightarrow & \emptyset \\
\text{READ } var(e_1, \ldots, e_n) & \longrightarrow & \cup_{i=1}^{n} \mathcal{R}_r(e_i)(\sigma) \\
\text{WRITE } exp & \longrightarrow & \mathcal{R}_r(exp)(\sigma) \\
var = exp & \longrightarrow & \mathcal{R}_r(exp)(\sigma) \\
var(e_1, \ldots, e_n) = exp & \longrightarrow & \mathcal{R}_r(exp)(\sigma) \cup (\cup_{i=1}^{n} \mathcal{R}_r(e_i)(\sigma)) \\
\text{IF } e \text{ THEN } s_1 \text{ ELSE } s_2 \text{ ENDIF} & \longrightarrow & \mathcal{R}_r(e)(\sigma) \cup (\text{if } \mathcal{E}(e)(\sigma) \text{ then } \mathcal{R}_{in}(s_1)(\sigma) \text{ else } \mathcal{R}_{in}(s_2)(\sigma)) \\
\text{DO } e\ s_1 \text{ ENDDO} & \longrightarrow & \mathcal{R}_r(e)(\sigma) \cup (\text{if } \mathcal{E}(e)(\sigma) \text{ then} \\
& & \mathcal{R}_{in}(s_1)(\sigma) \cup (\mathcal{R}_{in}(s)(\mathcal{T}(s_1)(\sigma)) \ominus \mathcal{R}_w(s1)(\sigma)) \text{ else } \emptyset) \\
\text{CALL } p(e_1, \ldots, e_n) & \longrightarrow & \mathcal{R}_{in}(p.body)(\mathcal{T}(p.body) \\
& & (\sigma \uplus \{(p.f_1, \mathcal{E}(e_1)(\sigma)), \ldots, (p.f_n, \mathcal{E}(e_n)(\sigma))\})) \\
s_1; s_2; \ldots; s_n; & \longrightarrow & \mathcal{R}_{in}(s_1)(\sigma) \cup (\mathcal{R}_{in}(s_2; \ldots; s_n;)(\mathcal{T}(s_1)(\sigma)) \ominus \mathcal{R}_w(s_1)(\sigma))
\end{array}$$

$\ominus$ is the set difference operator (see Chapter 4, page 36) that models the exclusion of written array elements. A more detailed explanation of the above cases is given in the next subsection.

8.3.2 Approximations of IN Array Regions

As `READ` and `WRITE` regions, the `IN` array regions also have its under and over approximations. They are abstracted by convex polyhedra. The propagation of `IN` regions follows the same principle as for `READ` and `WRITE` regions: it begins from the elementary statements, and proceeds towards the surrounding constructs and through procedure calls. Some modifications have been made with respect to [Cre96] and are specified as such below.

Elementary Statement

The `IN` regions of an assignment are identical to its `READ` regions because the values of the referenced elements cannot come from the assignment itself. Each array reference on the right hand side of the assignment is converted to an elementary region. Array references in the subscript expressions of the left hand side reference are also taken into account. These regions are `EXACT` if and only if the subscripts are affine functions of the program variables. Regions of the same array are merged by using the union operator.

The `IN` regions of an input/output statement are more complicated. The input/output status, error and end-of-file specifiers are handled with respect to the FORTRAN standard [ANS83]. These features are not expressed by the syntax of our toy language, but in order to handle real applications, they must be taken into account. In addition, the order of variable occurrences in the input list is used to compute correctly the `IN` regions of an input statement. For example, in the input statement `READ *,N,(A(I),I=1,N)`, `N` is not imported since it is written before being referenced in the implied-DO expression `(A(I),I=1,N)`. This is an improvement with respect to [Cre96].

Conditional Statement

The `IN` regions of a conditional statement contain the `READ` regions of the test condition, plus the `IN` regions of the true branch if the test condition is evaluated true, or the `IN` regions of the false branch if the test condition is evaluated false. Since the test condition value is not always known at compile-time, the `IN` regions of the true and false branches, combined with the test condition, are unified in the over-approximated regions.

Loop Statement

The `IN` regions of a loop contain array elements imported by each iteration but not previously written by the preceding iterations. Given the `IN` and `WRITE` regions of the loop body, the loop `IN` regions contain the imported array elements of the loop condition, plus the imported elements of the loop body if this condition is evaluated true. Then, when the loop is executed again, in the program state resulting from the execution of the loop body, they are added to the set of loop imported array elements in which all elements written by the previous execution are excluded.

Sequence of Statements

Let s be the sequence of instructions $s_1; s_2; \ldots ; s_n;$. The `IN` regions of the sequence contain all elements imported by the first statement s_1, plus the elements imported by $s_2; \ldots ; s_n;$ after the execution of s_1, but not written by the latter.

Control Flow Graph

Creusillet [Cre96] handles control flow graphs in a very straightforward fashion: the IN regions of the whole graph are equal to the union of the IN regions imported by all the nodes in the graph. Every variable modified at a node is projected from the regions of all other nodes. All approximations are decreased to MAY.

Interprocedural IN Array Region

The interprocedural propagation of IN regions is performed by a reverse invocation order traversal on the program call graph (see Chapter 3, page 23). As for READ and WRITE regions, the summary IN regions of each procedure are computed by preserving information about formal parameters, global and static variables in the IN regions of the procedure body. The IN regions of a call site are then calculated by translating the summary IN regions of the called procedure into the frame of the current procedure.

IN Region Example

```
      K = FOO()
C     <A(PHI1)-IN-EXACT-{1<=PHI1, PHI1<=N}>
      DO I = 1,N
C     <A(PHI1)-IN-EXACT-{PHI1==I}>
C     <B(PHI1,PHI2)-WRITE-EXACT-{1<=PHI1, PHI1<=N, PHI2==K}>
         DO J = 1,N
            B(J,K) = J + K
         ENDDO
C     <B(PHI1,PHI2)-IN-EXACT-{1<=PHI1, PHI1<=N, PHI2==K}>
C     <A(PHI1)-IN-EXACT-{PHI1==I}>
         K = K + 1
C     <B(PHI1,PHI2)-IN-EXACT-{1<=PHI1, PHI1<=N, PHI2==K-1}>
C     <A(PHI1)-IN-EXACT-{PHI1==I}>
         DO J = 1,N
C     <B(PHI1,PHI2)-IN-EXACT-{PHI1==J, PHI2==K-1}>
C     <A(PHI1)-IN-EXACT-{PHI1==I}>
C     <B(PHI1,PHI2)-WRITE-EXACT-{PHI1==J, PHI2==K}>
            B(J,K) = J*J - K*K
C     <B(PHI1,PHI2)-IN-EXACT-{PHI1==J, K-1<=PHI2, PHI2<=K}>
C     <A(PHI1)-IN-EXACT-{PHI1==I}>
            A(I) = A(I) + B(J,K) + B(J,K-1)
         ENDDO
      ENDDO
```

Figure 8.3: Example with IN and WRITE regions

Figure 8.3 shows computed `IN` regions for the example in Figure 8.2. In the body of the second `J` loop, array elements `A(I)`, `B(J,K)` and `B(J,K-1)` are imported by the second statement. Since `B(J,K)` is defined by the first statement, only `A(I)` and `B(J,K-1)` are imported by the loop body. The `IN` regions of the second `J` loop are `<B(PHI1,PHI2)-IN-EXACT -{1<=PHI1, PHI1<=N, PHI2==K-1}>` and `<A(PHI1)-IN-EXACT -{PHI1==I}>`. The transformer corresponds to the assignment `K = K+1` is `T(K) {K==K#init+1}`, so after applying this transformer to the above `IN` regions, the region of array `B` becomes `<B(PHI1,PHI2)-IN -EXACT-{1<=PHI1, PHI1<=N, PHI2==K}>`. Once again, all array elements in this region are defined by the first `J` loop, so only array elements of `A` are imported by the code fragment in this example.

8.3.3 Other Imported Array Analyses

Imported array analysis is also studied in many papers [HMA+95a, HMA+95b, GLL95, NGL95, TP95, Les96]. The convex array regions implemented in PIPS are based on the Regions method [TFI86] where `IN` array region, the set of imported array elements of a statement, is somewhat similar to *ExposedRead* region in [HMA+95a, HMA+95b], *UE* set in [GLL95, NGL95], *USE(s)* in [TP95] and *input effect* in [Les96].

However, our `IN` region and the others differ. For example, in [HMA+95a, HMA+95b], array element sets are represented by lists of polyhedra and there is no exact representation, only under- and over-approximations. The ExposedRead sets contain array elements which are used in the continuation of the whole program before being defined, while `IN` regions are restricted to the current level in the hierarchical control flow graph. In [GLL95, NGL95], an array element set is a list of RSDs (Regular Section Descriptor) with bounds and step, guarded by predicates derived from IF conditions. When insufficient information is available, our `MAY` regions should be more accurate because we can keep more information about ϕ variables. Input effects [Les96] give for each array element its first use in the considered fragment of code. This is similar to our `IN` regions but the precise statement instance in which the reference is performed is kept in the summary. Comparing between different approaches of imported array analysis, the implementation choice really is the trade-off between efficiency and precision. The `IN` regions developed by [Cre96] are used by our thesis, with some improvements.

8.4 Used-before-set Analysis

Our used-before-set analysis is directly based on the `IN` array regions. These regions are computed for arrays, but scalar variables also carry the same kind of information, which is cheaper to compute. In fact, the region of a scalar has an empty predicate, i.e `<V-IN-EXACT-{}>`. Information about imported array elements and scalar variables are propagated bottom-up, interprocedurally, from the elementary statements to the compound statements.

8.4.1 Used-before-set Algorithm

We traverse the program call graph in the invocation order (see Chapter 3, page 23). Algorithm 8.4.1 describes the used-before-set verification for each module.

Algorithm 8.4.1
procedure Used_Before_Set_Analysis(p)
    p : current procedure
begin
    s := module statement
    l := list of variables having IN region at s
    for each $v \in l$
        if local_variable(v) or global_variable_in_main_program(v, p) **then**
            if the `IN` region of v at s is `MUST` or `EXACT` **then**
                error: "Variable v is used before set"
            else /* `MAY IN` region */
                insert an initialization on v before s
                go_down_and_verify(v, s)
            endif
        else /* v is a formal parameter or a global variable in a called procedure*/
            if must_be_checked(v, p) **then**
                go_down_and_verify(v, s)
            endif
        endif
    endfor
end
procedure go_down_and_verify(v, s)
begin
    for each sub-statement s_i of s
        if the `IN` region of v at s_i is `EXACT` **then**
            insert a verification on v before s_i
        else /* `MAY IN` region*/
            if call_statement(s_i) **then**
                mark must_be_checked for the corresponding
                formal or global variables in the called procedure
            else
                go_down_and_verify(v, s_i)
            endif
        endif
    endfor
end

The list of `IN` regions at the module statement gives us the set of all possibly undefined variables of the module, and vice-versa, only variables in this list may be used before set. So at the module statement, if the list of `IN` regions is empty, there is no used-before-set error in this module. Otherwise, each variable in the list is checked.

In FORTRAN 77, the scope of a variable is always a module. The scope of a global variable declared in a module is that module but the scope of the common block where the variable is located is the whole program. So the `IN` regions of all global variables are propagated to the main program, although the variables are not declared in it. Depending on the variable type (local, formal or global) and the current module, we have two cases:

Case 1: Local or global variable in the main program. Depending on the region approximation, we have two sub-cases:

- If the `IN` region has the approximation `MUST` or `EXACT`, the variable must be used somewhere in the module before being defined and an error is detected.

- Otherwise, the region is `MAY`; we instrument the code by inserting an initialization function at the module entry and go down to the sub-statements where the `IN` region is propagated from. Before each statement where we know with certainty that the variable must be imported, a verification function is inserted. We continue to go down for each statement with `MAY` regions. If this statement is a procedure call, information is added to mark that the corresponding formal parameters of the actual variable, or the corresponding global variables must be checked at the callee's level. To help the debugging process, information about the call path is added to locate the run-time error.

Case 2: Formal parameter or global variable in a called procedure. If this variable is marked as *must be checked*, we repeat the process as for local variables, but no initialization is needed since it has been performed earlier in one of the callers.

The efficiency of our analysis depends on the accuracy of array region analyses. The more precise the imported array regions are, the smaller the number of variables to be checked is. Array regions may be approximated due to different causes. A major source of inexactness comes from not well-structured programs. Another limitation is due to nonlinear expressions that are not handled by PIPS. In addition, the choice of convex polyhedra for array accesses permits a compact but less precise representation because of approximative operators. For example, the union and difference of two regions cannot always be performed precisely. However, our `MAY` regions may still carry sufficient information to reduce significantly the initializations and verifications as is shown in the experimentation section.

8.4.2 Initialization and Verification

To trap premature usage, the initialization is implemented by assigning a special value to the maybe uninitialized scalar variable or array element. However, there are some implementation problems related to the special value and the type of variable. We can use a `SNaN` (Signaling Not a Number) for floating variables but it is not evident for integer and logical variables. Currently, we choose the value `MAXINT` (integer maximal representation) for integer variables, a value different from 0 and 1 for logical variables. The PIPS verification checks whether the variable value is equal to this special value, and reports the error if one has occurred. This may raise false positive warnings when program computations really involve these special values. This is also a problem for other compiler implementations. Some memory coloring techniques can be used to avoid this problem, but at the expense of memory usage.

If the variable is an array, the instrumented code is a `DO` loop associated to the corresponding `IN` region, whereas it is simple assignment and test for scalar variables. All array elements in the `MAY IN` region are marked uninitialized and all array elements in the `EXACT IN` region are checked. We use the algorithm described in [AI91] to compute loop bounds from the region predicate, which is a polyhedron. This algorithm scans the polyhedron and uses the Fourier pairwise elimination [Fou27] to find loop bounds for each dimension. The generated loops have the following forms:

```
IF (condition) THEN
   DO PHI1 = LOWER1, UPPER1
      DO PHI2 = LOWER2, UPPER2
         A(PHI1,PHI2) = special_value
      ENDDO
   ENDDO
ENDIF
```

a. Marking initialization of array A

```
IF (condition) THEN
   DO PHI1 = LOWER1, UPPER1
      DO PHI2 = LOWER2, UPPER2
         IF (A(PHI1,PHI2).EQ.special_value)
         STOP 'A is undefined in module ...'
      ENDDO
   ENDDO
ENDIF
```

b. Initialization verification

In order to reduce the execution time overheads, we can follow the strategy of the first compiler group (page 132). Verifications on real variables can be replaced by floating point traps. The FORTRAN code must then be linked to appropriate exception handling functions in order to catch the exceptions. However, the origin of the uninitialized errors is not as easy to locate with this method.

By using `IN` regions, we limit the number of variables to be checked, and code instrumentation is only used when we do not have enough information. Only array elements in the `MAY IN` regions are initialized and checked. PIPS initialization and verification statements are inserted in the source code and the program is then compiled and executed normally. The executable code appears to the user to operate as the original, but if a used-before-set error is detected, the program is stopped with a message to indicate the name of the variable, the module and the call path where this error occurred. Another implemented option is that when an use before definition takes place, we do not stop the program but write details to a log file for later analysis in order to catch several bugs in one run.

Used-before-set Example

To illustrate the used-before-set analysis, we use the example of plusFORT (page 134). Figure 8.4 shows the `IN` and `WRITE` regions computed for module `QKNUM`. At the module entry, there is one `IN` region for variable `IVAL` which means that only this formal variable may be used before set in the module. The variable `POS` is not imported by the loop because it is defined as the loop index. Neither is `IOS` imported by the module because its value has been defined by the `READ` statement, before being used in the test condition.

```
      SUBROUTINE QKNUM(IVAL, POS)
      INTEGER IVAL, POS, VALS(50), IOS
C     <IVAL-IN-MAY-{}>
C     <IOS-WRITE-EXACT-{}>
C     <VALS(PHI1)-WRITE-EXACT-{1<=PHI1, PHI1<=50, IOS==0}>
      READ (11, *, IOSTAT = IOS) VALS
C     <IOS-IN-EXACT-{}>
C     <IVAL-IN-MAY-{}>
C     <VALS(PHI1)-IN-MAY-{1<=PHI1, PHI1<=50, IOS==0}>
      IF (IOS .EQ. 0) THEN
C     IF (IVAL.EQ.MAXINT) STOP "IVAL is undefined in module QKNUM ..."
C     <IVAL-IN-EXACT-{}>
C     <VALS(PHI1)-IN-MAY-{1<=PHI1, PHI1<=50}>
C     <POS-WRITE-EXACT-{}>
         DO POS = 1,50
C     <IVAL-IN-EXACT-{}>
C     <POS-IN-EXACT-{}>
C     <VALS(PHI1)-IN-EXACT-{PHI1==POS}>
            IF (IVAL .EQ. VALS(POS)) GOTO 10
         ENDDO
      ENDIF
      POS = 0
10    END
```

Figure 8.4: Used-before-set analysis example

Array regions of the input statement are computed by taking into account the FORTRAN standard [ANS83]: if the input/output status equals to zero, neither an error condition nor an end-of-file condition is encountered by the processor, all data in the input/output list are transfered. If the input/output status is not equal to zero and there is no error or end-of-file specifier, execution of the executable program is terminated, no array element is defined. This is a language implementation feature but it must be respected to detect the uninitialized error correctly. The `WRITE` region of array `VALS` at the input statement is exactly:

```
<VALS(PHI1)-WRITE-EXACT-{1<=PHI1, PHI1<=50, IOS==0}>
```

and when propagating the `IN` region of array `VALS` backward, we have:

```
  <VALS(PHI1)-IN-MAY-{1<=PHI1, PHI1<=50, IOS==0}>
⊖ <VALS(PHI1)-WRITE-EXACT-{1<=PHI1, PHI1<=50, IOS==0}>
= <VALS(PHI1)-IN-EXACT-{}>
```

So all the elements of array `VALS` are well defined before they are used. There is no used-before-set error for the local variables in this module. By using static analysis, we prove that no instrumentation is needed, which is a big advantage with respect to the code generated by plusFORT (Figure 8.1). Since we do not have any calling context, it is not possible to conclude whether the variable `IVAL` is already initialized by the callers of `QKNUM`. If the whole program was given, and following some call paths, `IVAL` may not be initialized, a verification would be inserted before the loop and inside the conditional statement, as shown in the seventh comment line in Figure 8.4.

8.5 Experimental Results

This section shows the experimental results of our analysis technique on the SPEC95 CFP benchmarks. The ten benchmarks contain all kinds of variables: scalar and array, local, formal and global, with complicated data and control flow graphs. The experiments consist of two steps: `IN` *array region computation* and *used-before-set analysis*. Table 8.1 summarizes relevant information for each benchmark in SPEC95 CFP.

Table 8.1: SPEC95 CFP: number of lines, modules, variables (local and global), uninitialized and maybe uninitialized variables, percentage of maybe uninitialized variables, inserted verifications, compilation time (in minutes and seconds) and execution overhead.

Benchmark	Line	Mod	Var	Must	May	Percen	Verif	Comp	Overhead
tomcatv	190	1	38	0	8	21.05%	18	0:08	4.42
swim	429	6	88	0	13	14.77%	75	0:15	4.19
su2cor	2332	35	610	0	75	12.30%	274	5:35	5.43
hydro2d	4292	42	561	0	18	3.21%	302	1:47	5.07
mgrid	484	12	109	0	4	3.67%	26	1:30	3.77
applu	3868	16	354	0	11	3.11%	169	20:48	6.06
turb3d	2101	23	399	0	45	11.28%	122	2:03	6.41
apsi	7361	96	1480	0	136	9.19%	265	21:05	1.06
fpppp	2784	38	1107	0	357	32.25%	885	6:39	12.92
wave5	7764	105	2098	0	107	5.10%	431	26:46	UBS

Column 4 shows the number of local and global variables in each benchmark. Columns 5 and 6 present, respectively the number of uninitialized and maybe uninitialized variables, for both scalar and array. No used-before-set error is detected at compile-time (Column 5), which is expected for benchmarks. On average, the number of maybe uninitialized variables to be checked at run-time takes 11.6% of the total number of variables in the program (Column 7). Less than 4% of variables are checked for initialization in *hydro2d, mgrid* and *applu*. One initialization and several verifications are added for each maybe uninitialized variable. The number of instrumented verifications (single tests for scalar variables and loops for arrays) is reported in Column 8.

Column 9 shows the total compilation time (in minutes and seconds) required by PIPS to parse, compute imported array regions, analyze and generate code with used-before-set checks. This time is measured on a UltraSparc II 440MHz, 256 Mo RAM. The code instrumented with PIPS initializations and verifications is compiled with the SUN Workshop F77 version 5.0 compiler to generate executable files. This experiment is reported with the optimizing options turned on, using the SPEC95 CFP measurement guidelines (f77 -fast -xarch=v8plusa -fsimple=2 -xprefetch). Uninitialized variables are detected in *wave5*. In subroutine `PARTBL`, the local and static variables `LCMMAX` and `LCMR` are used before initialization.

The execution time overheads with the standard input data for other SPEC95 CFP benchmarks are shown in the last column. The floating-point trap is not used here. On average, it is 5.48 times slower than the initial code. There is only a 6% slowdown for *apsi*. The overhead is rather high for *fpppp* (about 13 times) because of irreducible control flow graphs in this benchmark. Information is lost and the approximation of array regions becomes `MAY`. It corresponds to 32.25% of must be checked variables. To improve the results, we should use a more sophisticated treatment on irreducible control flow graphs when computing array regions. On the other hand, some program transformations are needed for several benchmarks. For instance, we can reduce the number of variables to be checked from 18 to 14, added verifications from 302 to 288 for *hydro2d* by cloning the subroutine `ADLEN` which has two totally different behaviors for two parameter values: "half" and "full" steps. This optimization makes the array region analysis more precise, since interprocedural information helps to narrow down the scope of possible effects of the called procedure.

As mentioned in Section 8.4, there exists another solutions used in different compilers such as the undefined bit pattern or the NaN floating point trap, which can reduce greatly the execution time overheads. The verification on real variables can be omitted, and replaced by the NaN exception trap. However, the origin of the uninitialized errors is not easy to locate with this method. Since the primary objective of this work is to reduce the number of possibly undefined variables by using static analysis, we do not intend to implement these techniques at the moment.

Experimental results show that SPEC95 CFP are in general well debugged programs.

Used-before-set errors have been found in only one benchmark with its given input.

8.6 Conclusion

Static and dynamic analyses complement each other. Static analysis can discover automatically run-time errors and reduce the instrumentation or debugging cost. Dynamic checking takes into account program control flows and real input data that sometimes make static checking completely ineffective. Our used-before-set analysis combines these two approaches in order to reduce the overall cost while assuring the correctness of program.

PIPS is a source-to-source compiler that can be used for program analyses, transformation, parallelization and verification. By reusing advanced interprocedural analyses, the verification task becomes more efficient. `READ` and `WRITE` regions are already used to analyze program dependencies. `IN` regions are exploited for program optimizations such as array privatization, compile-time optimization of local memory or cache behavior in hierarchical memory machines, etc. Their precision is improved to target the verification. Only about 600 additional lines of C code are needed to implement the used-before-set analysis phase.

By using the `IN` region analysis, the static phase can improve one's confidence on program correctness by showing that the program is free from used-before-set errors. Or, an error can be detected statically and the bug can be fixed right after the analysis. A small number of maybe uninitialized variables pointed by our compile-time analysis can help the testing and validation process to save debugging time. Run-time checks are generated only when information is not available to monitor the verification process. When executing the code, if a used-before-set error happens, the message error provides information about what occurred prior to the error, which can be of great help when trying to identify the fragment of code that actually caused the error. Furthermore, we could have an appropriate exception handling mechanism which is very important for safety-critical systems. Our method could be applied to other imperative programming languages.

Experimental results also show that poor code quality can make static analysis insufficient; run-time checks remain and run-time failures cannot be eliminated. In addition to SPEC95 CFP suite, our analysis is applied to an industrial program with more than 100.000 lines of code to enhance the debugging process. We have encountered other problems with used-before-set checking, e.g the type mismatch, another standard violation. The actual variable is of integer type and the corresponding formal variable is declared of real type. In the called procedure, we verify if the formal variable is initialized by a NaN value check, which is in fact a check on integer variable and this may give false results. In addition, the compiler and platform dependences of initialization and verification functions are also implementation problems.

To improve the analysis, array regions could be enforced by using other approaches cited in Section 8.3. For example, input effects of Leservot [Les96] can give the exact statement where the used-before-set error occurs. Or, as in [MN97], the list of complementary array sections can be kept when performing some convex operators on array regions, in order to have more precise analysis on array usage. Improvements can be achieved by using more precise transformers of control flow graphs to improve `IN` array regions. In addition, other approaches of array data flow analysis such as dependence graph, source function [Fea91], etc can also be applied to the used-before-set checking problem. The question is to study the trade-off between precision and summarization, as well as the complexity in space and time.

Chapter 9

PUTTING IT ALL TOGETHER

Let's assume that we have to validate a whole industrial application. What should we do first? Check the array overflows assuming that no alias violation occurs? How can we be sure not to assume some property to check another one? Should we start with the main module or with leaves of the call tree? In the previous chapters we mostly ignore this ordering and global correctness issue.

The chapter is organized as follows. Section 9.1 describes the relationship among different program analyses and verifications, that is partly mentioned in the above chapters. Section 9.2 gives a synthesis about the constraints existing between the verifications and figures out the order of applications that ensures the correctness of an overall verification. This correctness is partial because we do not handle all the standard restrictions. A step-by-step example is given to illustrate the verification order. Conclusions are given in Section 9.3.

9.1 Interactions between Analyses and Verifications

Until now, we have reasoned as if each program analysis is performed under the assumption that no other standard violation occurs. For example, preexisting analyses presented in Chapter 4 such as transformers, preconditions and array regions are correct only if the program respects the standard about alias restriction and array element reference. The aliasing information in Chapter 7 is reliable if there is no array bound violation. The detection of uninitialized variables in Chapter 8 is trustworthy only if aliases among different variables have been taken into account, etc. All the constraints existing among the different analyses are studied in this section.

9.1.1 Preexisting Analyses

Compilers often assume that the source program is standard conforming. So program analysis information is dependable as long as no standard violation is revealed by its input. Our preexisting analyses are only defined on standard conforming programs. We develop some conditions to guarantee the correctness of these analyses: the absence of array bound violation and alias violation. Of course, they are necessary but not sufficient conditions because there are other standard violations that we do not handle.

Figure 9.1 illustrates how an array bound violation makes the preexisting analyses unreliable. Writing on an out-of-bound element of array `ITAB` modifies also the value of variable `J`, which is not handled by our transformers.

```
      PROGRAM ABC_TRANSFORMER
      COMMON ITAB(10),J
      J = 0
C     T() {}
      ITAB(11) = 1
      J = J + 1
      PRINT *,'J = ',J
      END
```

Figure 9.1: Transformers with array bound violation: the effect on J is not detected

Without the standard conformance assumption, the exact semantics of any given program is undefined, since no referential transparency is available. For example, if array `ITAB` were not in the same common block as `J`, any memory location after `ITAB(10)` could be modified by the out-of-range reference and it is impossible to analyze the program behavior. The incorrect preconditions computed for this example are shown in Figure 9.2.

```
      PROGRAM ABC_TRANSFORMER
      COMMON ITAB(10),J
C     P() {}
      J = 0
C     P() {J == 0}
      ITAB(11) = 1
C     P() {J == 0}
      J = J + 1
C     P() {J == 1}
      PRINT *,'J = ',J
      END
```

Figure 9.2: Preconditions with array bound violation: J is predicted equal to 1 but its effective value is 2

It is also noted that this program gives the output `J = 1` if it is compiled with option O3 of the SUN Workshop FORTRAN version 5.0 compiler. With array regions, the regions of expressions and elementary statements are always correct because they are associated with the current program state. The regions of composed statements may not be safe since unsafe transformers are used to translate regions referring to different program states to regions related to the unique entry state. This unsafe propagation is shown by the example in Figure 6.7, Chapter 6.

We also have the same problem with aliasing. The preexisting analyses are not correct if the program violates the restriction about aliasing. Figure 9.3 shows an example where writing on array elements `Y(1)` and `Y(2)` in procedure `SUB` is not detected, although actual/formal array size rule is respected.

```
      PROGRAM ALIAS_REGION
      INTEGER A(10)
C     <A(PHI1)-WRITE-EXACT-{1<=PHI1<=5}>
      CALL SUB(A,A(4))
      END
```

```
C     <X(PHI1)-WRITE-EXACT-{1<=PHI1<=5}>
      SUBROUTINE SUB(X,Y)
      INTEGER X(5),Y(5)
      DO I = 1, 5
         X(I) = I
      ENDDO
      END
```

Figure 9.3: Array regions with alias violation

Beside array bound and alias restriction violations, other FORTRAN standard violations such as side effects also impact the preexisting analyses. However, other kinds of errors such as the use of uninitialized variables, division by zero, over- and under-flows do not raise problems because memory locations that are defined are still known. Only memory violations such as writing on an out-of-bound array element or modifying a formal or common variable that is aliased with another variable may cause incorrect preexisting analyses.

Array declaration is a problem for programming languages such as C, APL and MATLAB when it is implicit or hidden by pointers (Chapter 5). With FORTRAN, the ugly and assumed-size array declarations are implicit pointer declarations and must be taken into account when computing array regions. The ugly assumed-size array must be considered to be an assumed-size array and information about last upper bounds is lost.

9.1.2 Array Resizing

Array resizing is a program declaration transformation that computes new upper bound for the ugly and assumed-size array declarations. Three approaches were discussed in Chapter 5: *i*) argument association rules, *ii*) array regions based and *iii*) code instrumentation.

The array region based approach uses array regions, so it may not be safe if the program violates the standard, as discussed in Subsection 9.1.1. The argument association rules without preconditions and the code instrumentation approaches are dependable because no preexisting analysis is performed. If we want to preprocess code by array resizing before applying other program verifications, these methods can be used. However, experimental results in Chapter 7 show that the inferred size may be too large with respect to the real array accesses in the called procedure and creates aliasing. So we may need finer alias violation conditions or we can use the non-aliasing conditions, described in Subsection 9.1.4, to infer new array declarations.

Another problem with array resizing is that the new array declaration may create bound violations. With the first and third approaches, there are no more interprocedural bound violations with the resized array because they are based on the same principle; the size of the formal array does not exceed that of the actual array. However, intraprocedural violations may occur because of the new size. In contrast, the second approach may create interprocedural array bound violations but no intraprocedural ones, because it relies on the actually accessed array elements in the procedure.

9.1.3 Array Bound Checking

In Chapter 6, two techniques for intraprocedural and one for interprocedural array bound checking are introduced. In the first approach to intraprocedural array bound checking, Elimination of redundant tests, array bound checks are generated before applying the redundancy elimination. Array accesses are guaranteed to be within their bounds and transformations applied on the instrumented code are always safe if alias restriction is enforced. Transformers and preconditions of the instrumented code are therefore dependable. However, the second approach, Insertion of unavoidable tests, is based directly on analyses computed for the input code. The preexisting analyses such as transformers, preconditions and array regions may not be correct, since the semantics of an incorrect code is undefined. So an array region is used only if the order in which arrays are written is verified.

As for the Elimination of redundant tests, the interprocedural array bound checking begins with the code instrumentation phase. Every bound violation is checked and the formal array sizes are guaranteed not to exceed the size of the actual arrays. Thus the elimination of redundant tests based upon preconditions is safe.

Neither intra- nor inter-procedural array bound checking are compatible with ugly and assumed-size arrays. The ugly assumed-size arrays cause premature aborts due to bound violations and the assumed-size arrays prevent array bound checking for these arrays. So array resizing must be performed before bound violation checking if we do not want to omit these kinds of arrays. Then we can apply instrumentation, preexisting analyses and simplification of bound checks on the instrumented code.

9.1.4 Alias Analysis

As explained in Chapter 7, there are at least three levels to verify the alias restriction violation on formal and common arrays. Figure 9.4 illustrates the three different interpretations.

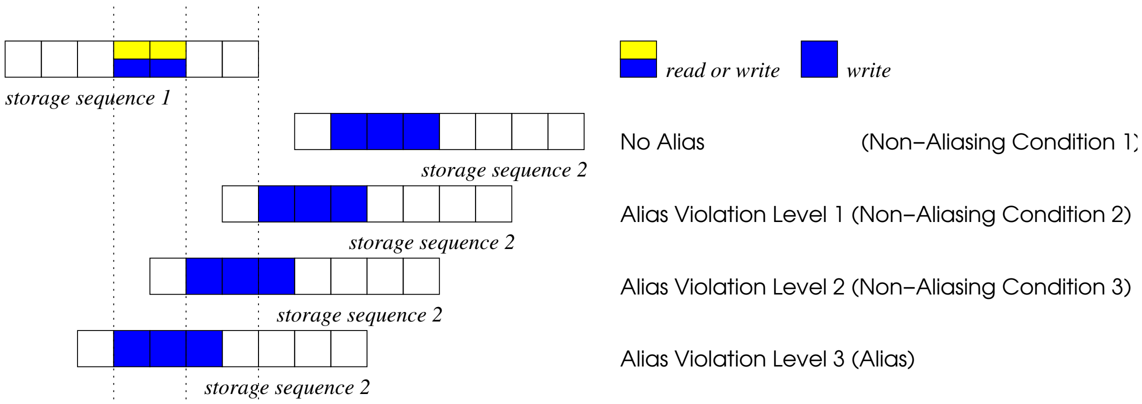

Figure 9.4: Alias violations on array variables

The simplest but strictest level is to treat arrays as scalar variables (Alias Violation Level 1 in Figure 9.4). Writing on any array element is not allowed if this array is aliased with another one. Expecting users to follow this rule is not realistic because memory management is often performed by passing sections of huge workspace used as a user managed heap.

The second and less strict level (Alias Violation Level 2 in Figure 9.4) is studied in Chapter 7. If the written array element is in the overlapping section of two storage sequences, there is an alias violation. The storage sequence of an array is computed from its offset in the corresponding area and the declared size in its procedure. This size is potentially unknown because of the assumed-size problem.

The third and finest level is the comparison among really referenced array elements (Alias Violation Level 3 in Figure 9.4). There is an alias violation if and only if an array element is defined and the aliased element of another array is referenced. That is data dependence arcs are generated between references to two different arrays. At this level, array resizing is not needed for ugly and assumed-size arrays because the computation of an array element address does not involve the upper bound of the last dimension. The third level is the most complicated to implement. Static analysis is usually not accurate enough and dynamic analysis is the only alternative.

The experimental results in Chapter 7, page 127, show that four out of ten benchmarks in SPEC95 CFP violate the alias restriction even if only the finest level is checked. Two more benchmarks violate the second level, but this could be avoided by using the non-aliasing condition to compute the new array declarations. In fact, alias violations are detected in these two benchmarks because we perform the alias checking on ugly and assumed-size arrays, unlike other compilers. So our choice of the second level is reasonable and compatible with FORTRAN programming practices. A more important issue is the impact of aliasing on the dependence graph and hence on code optimizations. New scheduling constraints cause real alarms for program transformations.

We now discuss how non-aliasing conditions could be used to resize arrays. We also have three non-aliasing conditions that correspond to the three levels of alias violation (Figure 9.4). The first condition is the simplest and strictest one: the intersection of two storage sequences is empty. The second condition is: the intersection of the defining sequence of one variable and the storage sequence of the other is empty. The third condition is: the intersection of the defining sequence of one variable and the referencing sequence of the other is empty. This means that we do not compare the variable declaration but the really referenced array elements.

We choose the Non-Aliasing Condition 1 (Figure 9.4) that seems to be the most appropriate according to our empirical study of SPEC95 CFP. Passing disjoint array sections to the called procedure is often used to allocate space dynamically and causes no problem neither for aliasing nor for the standard interpretation. Under this condition, we infer new upper bound value for the pointer-typed arrays. However, by forcing the non-aliasing condition, intraprocedural array bound violations may be created because the new size may be too small with respect to the actual array references in the called procedure. By doing this, we in fact transform an alias violation into an array bound violation. This problem is discussed in Section 9.2.

Alias verification uses the write effect analysis (Chapter 7, page 120). If the aliased variables are not involved in writing effects of a module, there is no violation. Memory effect analysis is also a preexisting analysis in PIPS whose correctness depends on the program standard conformance. If aliases between variables on entry of a module are not taken into account, we only have the write effects on variables that are syntactically modified by this module. However, these incomplete effects do not impact on our alias verification because of the aliasing relation. If none of the aliased variables is syntactically written, the other variables in the aliased set are not written neither, so the write effects give us correct information. Otherwise, if at least one variable is written, it is in the write effect list and the alias verification checks for this variable and the other variables in the aliased set.

Alias analysis is also related to array bound checking. Figure 9.5 shows the impact of the no out-of-bound assumption on alias verification. The intra- and inter-procedural bound violations on array `F1` in respectively the first and second program create aliasing between array elements `F1(11)` and `F2(1)`. So a write on `F1(11)` or `F2(1)` causes an alias violation, which is not detected by the alias analysis because it assumes that there is no bound violation. In the intraprocedural example, the intersection between the storage sequences of array `F1` and `F2` must be empty, and in the interprocedural one, the storage sequence of array `A` and `B` in the blank common cannot overlap. So to make alias analysis exact, we must guarantee that there is no array bound violation in the program.

```
PROGRAM INTRA_ABC_ALIAS
REAL A(20)
CALL INTRA_SUB(A(1),A(11))
END
SUBROUTINE INTRA_SUB(F1,F2)
REAL F1(10),F2(10)
F1(11) = 1
END
```

```
PROGRAM INTER_ABC_ALIAS
COMMON A(10),B(10)
CALL INTER_SUB(A,B)
END
SUBROUTINE INTER_SUB(F1,F2)
REAL F1(11),F2(10)
F2(1) = 1
END
```

Figure 9.5: Alias analysis with intra- and inter-procedural array bound violations

9.1.5 Uninitialized Variables

Since the uninitialized variable analysis (Chapter 8) is directly based on imported array regions, we must guarantee that the program respects the standard about alias restriction and array references. So if this uninitialized variable verification is done after array resizing, alias and array bound checking (both intra- and inter-procedural versions), the preexisting analyses performed on the output code are dependable, and so is our uninitialized variable analysis.

9.2 Synthesis of Verifications

Figure 9.6 summarizes the constraints existing between the above program analyses and verifications.

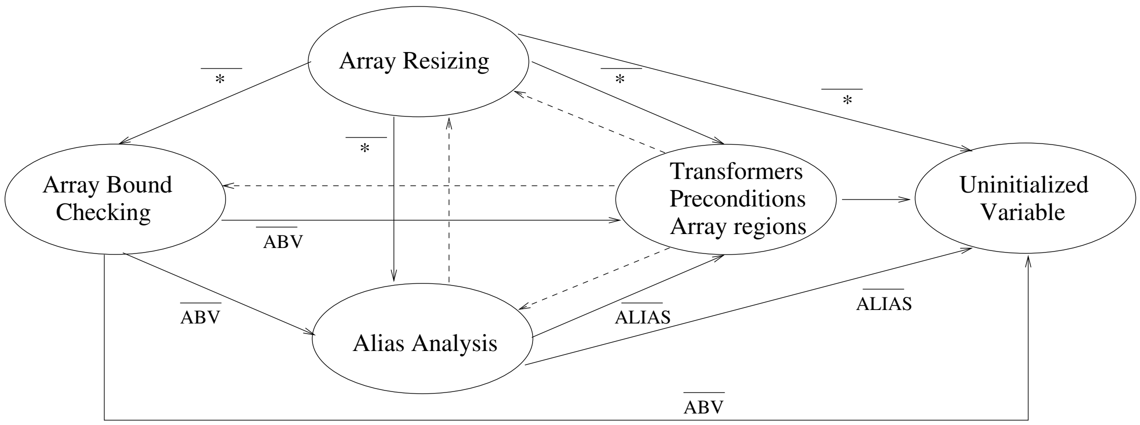

Figure 9.6: Constraints between program analyses and verifications

The solid arrow from one analysis to the other represents the fact that corresponding property must be guaranteed before performing the second analysis. More concretely, four arrows from array resizing to the other analyses mean that there must be no assumed-size arrays ($\overline{*}$) when generating array bound checks, performing alias analysis, preexisting analyses and uninitialized variable analysis, respectively. Three arrows from array bound checking to alias analysis, preexisting analyses and uninitialized variable analysis denote that there must be no bound violation ($\overline{\texttt{ABV}}$) before carrying out these analyses. The arrows from alias analysis to preexisting analyses and uninitialized variable analysis signify that there must be no alias violation ($\overline{\texttt{ALIAS}}$) when performing them. The arrow from preexisting analyses to uninitialized variable analysis specifies the fact that imported array regions are used to perform this checking.

The dashed arrows imply that the information from one analysis can be used to optimize the other. Preexisting analyses are exploited to support array resizing (preconditions for the argument association rules approach and array regions for the regions based approach), array bound checking (preconditions to remove intra- and inter-procedural redundant checks, array regions to insert unavoidable tests), alias analysis (preconditions to optimize generated checks). In addition, the non-aliasing condition from alias analysis could be used to assist array resizing.

So given all possible program verifications and the constraints among them, one important question is, *if I have a 'buggy' program, which verifications do I have to apply and in which order?* Figure 9.6 suggests the general order: array resizing, array bound checking, alias analysis, preexisting analyses and uninitialized variable analysis. However, this order cannot be applied straightforward because, as we have seen, array resizing may cause artificial alias violations. In order to simplify the problem, we distinguish two different cases.

9.2.1 Full Array Declarations

If the program has no ugly and assumed-size array declarations, the global verification is as follows:

1. Generate intra- and inter-procedural array bound checks, using the bound check generation phases in the Elimination of Redundant Tests and Interprocedural Array Bound Checking algorithms (page 78 and page 99, Chapter 6);

2. Verify alias restriction violations, using the Interprocedural Alias Propagation and Interprocedural Alias Checking algorithms (page 117 and page 121, Chapter 7);

3. Eliminate redundant checks, using transformers and preconditions computed for the instrumented code, as the elimination phase of the Elimination of Redundant Tests algorithm (page 78, Chapter 6);

4. Check uninitialized variables, using imported array regions as in the Used Before Set algorithm (page 140, Chapter 8).

Note that we distinguish between the bound check generation and the redundancy elimination phases for both intra- and inter-procedural bound checking. The array region based approach to array bound checking is not used. If no error is detected at compile-time and the program is not proved to be free of errors, the final code is then compiled and any standard violation generates a run-time error.

Theorem 9.2.1 The established verifying order is safe.

Proof: Since there are no ugly and assumed-size arrays in the program, we can generate intra- and inter-procedural bound checks for all arrays. After the first step, every array access is guaranteed to be within their bounds. Then the second step, alias analysis, is safe because the constraint between array bound checking and alias analysis is respected. Any formal or common variable that is aliased with another variable is not modified by the called procedure after alias checking.

Once all the array bound and alias violations are not going undetected by the instrumented code, the preexisting analyses, i.e transformers and preconditions are used to eliminate redundant checks. As for the imported array regions, these analyses are dependable. The last step is to perform the uninitialized variable analysis. The above order is safe since it satisfies all the constraints among different analyses. □

9.2.2 Incomplete Array Declarations

If pointer-typed (ugly and assumed-size) array declarations occur in the input program, the order of verifications becomes more complicated. We have to treat the problem of non-existing alias violations, caused by the top-down approaches to array resizing, i.e the argument association rule approaches. We have studied three solutions to this problem.

The first solution is to treat alias violations at the third level, based on the actually referenced array elements, not on the array declarations. By this approach, arrays are treated element by element. There is an alias violation if and only if an array element is defined and the aliased element of another array is referenced. Array resizing is only needed to perform array bound checking and preexisting analyses. The general order proposed by Figure 9.6 then can be applied. However, this alias violation level relies on such a complicated analysis that trade-offs must be made.

The second solution is, rather than avoiding the artificial alias violations caused by the parameter passing rules array resizing approaches, to study the impact of aliasing on the dependence graph. This solution is justified by the fact that actual alias violations do exist in many benchmarks, so a more important issue is the impact of aliasing on code optimizations. This impact analysis would point out that the alias violations caused by the over approximated approaches to array resizing do not create any scheduling constraints; so they are compatible with most compiler optimizations. However, the most precise data dependence analysis would compare all referenced array elements, which may cause the same problem as the first solution.

The third solution, chosen by us, is to use the non-aliasing condition to avoid artificial alias violations. However, this condition infers new array sizes that may cause array bound violations after adding new bound checks. In this case, we are not able to decide the nature of the error. With respect to the original program, is there an array bound violation or an alias violation? This is a classical problem with the order of program transformations in the compilation field. The results given by different combinations of transformations may be not equal, due to the mutual dependences among them. Since our ultimate goal is to make sure that no standard violation is going undetected, the exact origin of the error then can be sacrificed. In addition, explicit array sizes are not really needed if we are able to perform analyses and generate checks for the assumed-size arrays. Following this approach, the order of verifications would be established as follows:

1. Compute new upper bound for the last dimension of ugly and assumed-size arrays, using the Argument Association Rules and Code Instrumentation approaches (page 61 and page 66, Chapter 5);

2. Generate intra- and inter-procedural array bound checks, using the generation phases in the Elimination of Redundant Tests and Interprocedural Array Bound Checking algorithms (page 78 and page 99, Chapter 6). The last upper bound checks in the intraprocedural version and the comparison between formal and actual array sizes in the interprocedural version involving the ugly and assumed-size arrays are generated with respect to the value computed by step 1;

3. Verify alias restriction violations, using the Interprocedural Alias Propagation and Interprocedural Alias Checking phases (page 117 and page 121, Chapter 7). When an array resized by step 1 is involved in an alias violation, we use the non-aliasing condition to infer a new last upper bound for this array. In this case, alias checks are removed from the instrumented code for the array. Upper bound checks are added with respect to the new computed value;

4. Eliminate redundant checks, using transformers and preconditions computed for the instrumented code, as the elimination phase of the Elimination of Redundant Tests algorithm (page 78, Chapter 6). The size of ugly and assumed-size arrays is the size computed by step 1;

5. Check uninitialized variables, using imported array regions as in the Used Before Set algorithm (page 140, Chapter 8). The size of ugly and assumed-size arrays is also the size computed by step 1.

During this verification process, an error can be detected statically or the program can be proved to be free of out-of-bound, alias violation and used-before-set errors. Otherwise, the instrumented code is compiled and executed to catch errors at run-time. Checking on ugly and assumed-size arrays is not omitted.

Theorem 9.2.2 The established verifying order is safe.

Proof: At Step 1, new upper bounds of the last array dimensions are computed with respect to the declarations in the callers, using the argument association rules. Intra- and inter-procedural array bound checks are then generated for all arrays in the program at Step 2.

The application of alias analysis at Step 3 is safe because after Steps 1 and 2, there is no array bound violation in the code. If any array resized by Step 1 is involved in an alias violation, the non-aliasing condition is used to compute a new upper bound for this array. There are no more alias checks on pointer-typed arrays whose new bounds satisfy the non-aliasing condition. New bound checks are added to guarantee that no error whose origin is either an alias or an array bound violation is going undetected.

Since the instrumented code respects the standard about alias restriction and array element references, the preexisting analyses such as transformers, preconditions and imported array regions are correct. Assumed-size arrays are analyzed using the upper bound computed by Step 1, which satisfies the parameter passing rules. We can safely apply redundancy elimination to remove redundant array bound and alias checks (Step 4) and the uninitialized variables analysis (Step 5) for the new code. The above order is safe since it satisfies all the constraints between the different analyses (see Figure 9.6). □

9.2.3 A Step-by-Step Example

The example in Figure 9.7 is used to illustrate our different verification steps.

```
PROGRAM MAIN
REAL WORK(100)
DO I = 1, 100
   WORK(I) = I
ENDDO
READ *,N
CALL RUN(N,WORK,WORK(N+1),WORK(2*N+1))
END

SUBROUTINE RUN(N,X,Y,Z)
REAL X(*),Y(*),Z(*)
DO I = 1, N
   Z(I) = Y(I)
   Y(I) = X(I)
ENDDO
END
```

Figure 9.7: Input code

Let X∗, Y∗ and Z∗ be respectively the symbolic upper bound of the assumed-size arrays X, Y and Z. The following steps are performed.

- Step 1: Computing new upper bounds using the argument association rules.

$$\begin{cases} X* = 100 \\ Y* = 100 - N \\ Z* = 100 - 2N \end{cases}$$

- Step 2: Generating intra- and inter-procedural array bound checks.

```
PROGRAM MAIN
REAL WORK(100)
DO I = 1, 100
   IF (I.LT.1) STOP 'Bound violation:  array MAIN:WORK, lower bound'
   IF (I.GT.100) STOP 'Bound violation:  array MAIN:WORK, upper bound'
   WORK(I) = I
ENDDO
READ *, N
IF (2*N+1.LT.1) STOP 'Bound violation:  array MAIN:WORK, lower bound'
IF (2*N+1.GT.100) STOP 'Bound violation:  array MAIN:WORK, upper bound'
IF (N+1.LT.1) STOP 'Bound violation:  array MAIN:WORK, lower bound'
IF (N+1.GT.100) STOP 'Bound violation:  array MAIN:WORK, upper bound'
CALL RUN(N, WORK, WORK(N+1), WORK(2*N+1))
END

SUBROUTINE RUN(N,X,Y,Z)
REAL X(*),Y(*),Z(*)
DO I = 1, N
   IF (I.LT.1) STOP 'Bound violation:  array RUN:Y, lower bound'
   IF (I.GT.100-N) STOP 'Bound violation:  array RUN:Y, upper bound'
   IF (I.GT.100-2*N) STOP 'Bound violation:  array RUN:Z, upper bound'
   Z(I) = Y(I)
   IF (I.LT.1) STOP 'Bound violation:  array RUN:X, lower bound'
   IF (I.GT.100) STOP 'Bound violation:  array RUN:X, upper bound'
   IF (I.GT.100-N) STOP 'Bound violation:  array RUN:Y, upper bound'
   Y(I) = X(I)
ENDDO
END
```

Figure 9.8: Generating all array bound checks

Each statement is preceded by the bound checks on its array references. Identical lower bound checks on arrays in statements Z(I) = Y(I) and Y(I) = X(I) are generated once. The intraprocedural array bound checking in program MAIN gives us the following constraints on N:

$$\begin{cases} \mathtt{N}+1 \geq 1 \\ \mathtt{N}+1 \leq 100 \\ 2\mathtt{N}+1 \geq 1 \\ 2\mathtt{N}+1 \leq 100 \end{cases} \Longleftrightarrow 0 \leq \mathtt{N} \leq 49$$

The intraprocedural array bound checking, with respect to array bounds computed by Step 1, when entering in the loop in module SUB gives us the following information:

$$\mathtt{N} \geq 1 \longrightarrow \begin{cases} \mathtt{N} \leq 100 \\ \mathtt{N} \leq 100-\mathtt{N} \\ \mathtt{N} \leq 100-2\mathtt{N} \end{cases} \Longleftrightarrow \mathtt{N} \geq 1 \longrightarrow \mathtt{N} \leq 33$$

There are no interprocedural checks for the call to RUN, since the resized bounds satisfy the argument association rules.

- Step 3: Verifying alias restriction violations.

  This verification is important since the parallelization of the DO loop in subroutine RUN is false if aliasing between arrays X, Y and Z is not taken into account. There is no alias violation when writing on Z(I) and Y(I) if disjoint array sections are passed to RUN. So by applying the non-aliasing condition, when $\mathtt{N} \geq 1$ we have new upper bounds:

$$\begin{cases} \mathtt{X}* = \mathtt{N} \\ \mathtt{Y}* = \mathtt{N} \quad (\text{when } \mathtt{N} \geq 1) \\ \mathtt{Z}* = \mathtt{N} \end{cases}$$

  New upper bound checks are added with respect to these values for array references X(I), Y(I) and Z(I) in module RUN.

```
SUBROUTINE RUN(N,X,Y,Z)
REAL X(*),Y(*),Z(*)
DO I = 1, N
   IF (I.GT.N) STOP 'Bound violation:  array RUN:Y by non-aliasing'
   IF (I.LT.1) STOP 'Bound violation:  array RUN:Y, lower bound'
   IF (I.GT.100-N) STOP 'Bound violation:  array RUN:Y, upper bound'
   IF (I.GT.100-2*N) STOP 'Bound violation:  array RUN:Z, upper bound'
   Z(I) = Y(I)
   IF (I.GT.N) STOP 'Bound violation:  array RUN:X by non-aliasing'
   IF (I.LT.1) STOP 'Bound violation:  array RUN:X, lower bound'
   IF (I.GT.100) STOP 'Bound violation:  array RUN:X, upper bound'
   IF (I.GT.100-N) STOP 'Bound violation:  array RUN:Y, upper bound'
   Y(I) = X(I)
ENDDO
END
```

Figure 9.9: Verifying alias violations with non-aliasing condition

These bound checks add no more constraints on N. We do not have to perform the actual/formal array size checking for the call to RUN because the bounds resized by the non-aliasing condition are always less than or equal to the bounds computed by the over-approximated array resizing approaches, with the argument association rules.

- Step 4: Eliminating redundant checks.

  The redundancy elimination is done by using transformers and preconditions computed for the new code. Useless tests, such as I.GT.100 and I.LT.1 in the main program are removed from the instrumented code. Constraints are also normalized, 2*N+1.LT.1 to N.LT.0 and 2*N+1.GT.100 to N.GT.49, etc.

  The simplified code is:

```
PROGRAM MAIN
REAL WORK(100)
DO I = 1, 100
   WORK(I) = I
ENDDO
READ *, N
IF (N.LT.0) STOP 'Bound violation:  array MAIN:WORK, lower bound'
IF (N.GT.49) STOP 'Bound violation:  array MAIN:WORK, upper bound'
CALL RUN(N, WORK, WORK(N+1), WORK(2*N+1))
END

SUBROUTINE RUN(N,X,Y,Z)
REAL X(*),Y(*),Z(*)
DO I = 1, N
   IF (I.GT.100-2*N) STOP 'Bound violation:  array RUN:Z, upper bound'
   Z(I) = Y(I)
   Y(I) = X(I)
ENDDO
END
```

Figure 9.10: Eliminating redundancy checks

  Since code hoisting has not been implemented in PIPS, a test still remains in the loop body in module RUN.

- Step 5: Checking uninitialized variables.

  Imported array regions are computed for the instrumented code. All array elements of WORK are well defined by the main program, before being possibly used in the subroutine RUN. The program is proved to be free of used before set errors. The code with imported array regions is given in Figure 9.11.

```
      PROGRAM MAIN
      REAL WORK(100)
      DO I = 1, 100
         WORK(I) = I
      ENDDO
C     <WORK(PHI1)-IN-MAY-{1<=PHI1, PHI1<=98}>
      READ *, N
C     <WORK(PHI1)-IN-MAY-{1<=PHI1, PHI1<=2N, N<=49, 0<=N}>
      IF (N.LT.0) STOP 'Bound violation:  array MAIN:WORK, upper bound'
C     <WORK(PHI1)-IN-MAY-{1<=PHI1, PHI1<=2N, N<=49}>
      IF (N.GT.49) STOP 'Bound violation:  array MAIN:WORK, upper bound'
C     <WORK(PHI1)-IN-EXACT-{1<=PHI1, PHI1<=2N}>
      CALL RUN(N, WORK, WORK(N+1), WORK(2*N+1))
      END

C     <X(PHI1)-IN-EXACT-{1<=PHI1, PHI1<=N}>
C     <Y(PHI1)-IN-EXACT-{1<=PHI1, PHI1<=N}>
      SUBROUTINE RUN(N,X,Y,Z)
      REAL X(N),Y(N),Z(N)
C     <X(PHI1)-IN-EXACT-{1<=PHI1, PHI1<=N}>
C     <Y(PHI1)-IN-EXACT-{1<=PHI1, PHI1<=N}>
      DO I = 1, N
         IF (I.GT.100-2*N) STOP 'Bound violation:  array RUN:Z, upper bound'
C     <Y(PHI1)-IN-EXACT-{PHI1==I}>
         Z(I) = Y(I)
C     <X(PHI1)-IN-EXACT-{PHI1==I}>
         Y(I) = X(I)
      ENDDO
      END
```

Figure 9.11: Checking uninitialized variables with IN regions

The final code is given in Figure 9.12. When executing the program, if $\texttt{N} < \texttt{0}$, there is a lower bound violation on array `WORK` in the main program. If $\texttt{N} > \texttt{49}$, there is an upper bound violation on `WORK` and if $\texttt{33} < \texttt{N} \leq \texttt{49}$, there is an upper bound violation on array `Z` in module `RUN`, which is detected with respect to the actual array declarations. The bound checks generated after using the non-aliasing condition do not create bound violations, since all array references in `RUN` are within the inferred bounds. In conclusion, there is no standard violation in this code if and only if $\texttt{0} \leq \texttt{N} \leq \texttt{33}$.

```
PROGRAM MAIN
REAL WORK(100)
DO I = 1, 100
   WORK(I) = I
ENDDO
READ *, N
IF (N.LT.0) STOP 'Bound violation:  array MAIN:WORK, lower bound'
IF (N.GT.49) STOP 'Bound violation:  array MAIN:WORK, upper bound'
CALL RUN(N, WORK, WORK(N+1), WORK(2*N+1))
END

SUBROUTINE RUN(N,X,Y,Z)
REAL X(*),Y(*),Z(*)
DO I = 1, N
   IF (I.GT.100-2*N) STOP 'Bound violation:  array RUN:Z, upper bound'
   Z(I) = Y(I)
   Y(I) = X(I)
ENDDO
END
```

Figure 9.12: Final code

9.3 Conclusion

It is important to study the dependences of one program analysis with another. The correctness of a program verification technique cannot be proved if these dependences have not been taken into account. We proposed a safe order to apply all the implemented analyses to guarantee the correctness of programs.

There is no perfect solution if implicit array declarations are used. For programs with full array declarations, the safe verifying order is array bound check generation, alias violation checking, preexisting analyses for code simplification and then uninitialized variable checking. The order with incomplete array declaration programs is array resizing by argument association rules, bound check generation, alias analysis with the non-aliasing condition, then preexisting analyses and uninitialized variable checking. We cannot always decide between an alias violation or an out-of-bound error message. Artificial alias violations may be created by array resizing only if the argument association rules are used. The non-aliasing condition then may create out-of-bound errors. We choose the solution with the non-aliasing condition to transform an alias violation into an out-of-bound error. In fact, our experimental results with the SPEC95 CFP benchmarks show that there is no bound violations caused by the non-aliasing condition and all the artificial alias violations are removed, as in the case of the step-by-step example. Moreover, conditions when a program respects the standard could be established by this approach.

Our study reveals also the problem of code quality. Good programming practices, i.e the standard conformance on array declarations and alias restriction, etc enable static analyses and therefore are important factors to reduce risks caused by software systems.

Chapter 10

CONCLUSION

The English speaking reader will find the translation of this chapter in Section 10.2.

10.1 Français

Nous avons étudié dans cette thèse quatre analyses et techniques de vérification de programme : le redimensionnement de tableaux, le test de débordement de tableaux, l'analyse des alias et l'analyse de variables non initialisées. Tous ces analyses sont implantées et ont été expérimentées avec des benchmarks et des applications industrielles. Les résultats expérimentaux obtenus avec le benchmark SPEC95 CFP sont donnés avec chaque vérification. Les résultats obtenus avec des applications industrielles sont confidentiels mais ils ont démontré la robustesse de l'implantation et montré l'intérêt d'une certaine souplesse vis-à-vis de la norme.

À notre connaissance, le problème de la déclaration incomplète de tableaux n'a pas été abordé dans la littérature. Cette déclaration incomplète empêche d'autres analyses, et nuit à la compréhension et à la mise au point de programmes. Elle est aussi l'origine du surcoût de temps d'exécution dans certains langages de programmation comme MATLAB et JAVA, à cause du redimensionnement dynamique. Les approches sur le redimensionnement de tableaux développées dans le Chapitre 5 ont résolu les problèmes liés aux tableaux de type *ugly* et *assumed-size* dans FORTRAN et elles peuvent être aussi appliquées à d'autres langages comme C pour déduire les informations sur les pointeurs.

Deux algorithmes de test intraprocédural de débordement de tableaux et un algorithme de test interprocédural sont étudiées dans le Chapitre 6. Des mesures sur le nombre de tests enlevés, le volume d'information nécessaire pour la mise au point, les temps de compilation et d'exécution ont été effectuées sur différents compilateurs et plates-formes pour montrer l'efficacité et le coût limité de deux optimisateurs intraprocéduraux (Élimination des tests redondants et Insertion des tests inévitables). Afin d'aboutir à une vérification complète des accès aux tableaux, nous avons présenté dans cette thèse une vérification de cohérence entre sites d'appel et déclarations de modules pour prouver l'absence ou détecter des erreurs liées au passage de tableaux dans les appels de procédure. Cette analyse n'est abordée ni dans les autres travaux ni dans la plupart des compilateurs commerciaux.

L'analyse des alias du Chapitre 7 nous a permis de lever l'hypothèse pessimiste sur les alias et d'analyser des programmes avec des hypothèses moins sévères sur leur qualité. Nous avons développé des algorithmes pour détecter la violation des restrictions sur les alias dans le langage FORTRAN, une option manquante dans la plupart des compilateurs. Cette vérification de la norme est utile pour débugger, pour aider le programmeur à corriger des erreurs et pour gagner des performances en appliquant des optimisations sans risque. De plus, l'impact des alias sur le graphe de dépendance de programme est aussi étudié dans cette thèse.

En utilisant des analyses interprocédurales avancées sur les régions de tableaux, un petit nombre de variables potentiellement non initialisées est révélé par notre analyse statique du Chapitre 8. Cette analyse permet de gagner beaucoup de temps pour la mise au point des systèmes de grande taille. Les erreurs concernant l'utilisation d'une variable avant sa définition sont aussi détectées au moment de l'exécution pour éviter d'obtenir des résultats incorrects, des comportements imprévisibles ou même des échecs du système.

Les interactions entre les différentes vérifications et analyses de programme ainsi que leur ordonnancement sont étudiés dans le Chapitre 9 pour montrer comment on peut vérifier correctement des programmes faux.

Les contributions principales présentées dans cette thèse sont les suivantes :

1. *Une combinaison efficace d'analyse statique et instrumentation de code.* Cette thèse montre que l'analyse de programme peut être considérée comme un outil puissant pour améliorer la sécurité des logiciels. L'analyse statique y est profitablement combinée avec des techniques dynamiques pour réduire l'instrumentation de code et donc le surcoût de l'exécution. Grâce à l'information disponible fournie par l'analyse statique, seuls quelques fragments du code sont instrumentés. Cette méthode facilite aussi la détection des erreurs au moment optimal, pendant le cycle de développement du logiciel. En comparaison avec les autres techniques de vérification de programmes, notre approche est complètement automatique et n'a pas besoin d'interactions humaines comme des annotations d'utilisateur et des assertions. L'analyse statique peut être vue comme un outil efficace pour détecter des erreurs dans les systèmes de taille importante à un coût raisonnable.

2. *Une aide pour la mise au point de programme.* Les leçons tirées des résultats expérimentaux nous permettent de cibler notre détection des erreurs pour la mise au point, une des activités fondamentales du génie logiciel. Pour détecter un nombre maximal d'erreurs dans un passage, un message `PRINT` avec des informations utiles est employé au lieu d'un message `STOP`. À la différence de certains analyseurs syntaxiques qui s'arrêtent après avoir détecté une erreur, l'analyse statique et l'instrumentation nous permettent de tracer toutes

les erreurs possibles et d'éviter les compilations et exécutions répétitives. En plus, puisque la norme FORTRAN est souvent violée par des programmes réels, nous avons proposé des nouvelles notions de vérification, comme le test de débordement de section dans le Chapitre 6, des conditions plus flexibles sur la violation des alias et l'impact des alias sur le graphe de dépendence dans le Chapitre 7. La distinction entre bogues *insignifiants* et bogues *dangereux* permet de réduire le nombre de fausses alarmes et de prendre en compte le fait que les programmeurs en FORTRAN ne respectent pas toujours la norme. Toutes ces informations sont utiles à la maintenance, la réutilisation et l'adaptation des systèmes logiciels.

3. *Une intégration complète dans un compilateur source-à-source.* Des vérifications efficaces ont été implantées en réutilisant des techniques d'analyse interprocédurale qui sont déjà mises en oeuvre dans des compilateurs commerciaux. Notre implantation dans PIPS représente environ 1600 lignes de code C pour le redimensionnement de tableaux, avec les trois approches : les règles d'association d'argument, les régions de tableaux et l'instrumentation de code. Pour le test de débordement de tableaux, cela correspond à environ 1200 lignes de code C pour l'Élimination des tests redondants, 700 lignes pour l'Insertion des tests inévitables et 500 lignes pour le test de débordement interprocédural. Il est de moins de 2000 lignes pour l'analyse des alias, y compris les phases de propagation et de vérification. L'analyse de variables non initialisées prend environ 600 lignes de code C. Le total est donc de 6600 lignes pour les différentes méthodes de vérification. Des tests de non régression sont effectués fréquemment pour évaluer la robustesse de la mise en oeuvre. Une base de test est mise en place pour valider ces implantations.

4. *Des expériences qualitatives et quantitatives.* Des expériences ont été effectuées avec des extraits d'applications industrielles et trois benchmarks importants : Linpack, PerfectClub et SPEC95 CFP. Nous nous sommes concentrés en particulier sur le benchmark SPEC95 CFP, dix programmes publiquement disponibles. Les nombres de lignes et de sous-programmes varient, respectivement, de 190 lignes et 1 sous-programme (*tomcatv*) à 7764 lignes et 105 sous-programmes (*wave5*). Nous avons comparé les performances de nos analyses avec celles des différents compilateurs commerciaux et sur des plates-formes différentes (SUN, SGI et IBM), et aussi avec d'autres travaux dans la litérature. Les résultats de ces expériences montrent que les benchmarks dans SPEC95 CFP sont des programmes bien débuggés. Ils ne souffrent pas d'erreurs graves, sauf une violation interprocédurale de débordement de tableau et quelques utilisations de variables avant leur initialisation dans *wave5*. De plus, nos vérifications ont été appliquées sur deux applications industrielles de taille assez importante. La première contient plus de 83000 lignes et la deuxième plus de 200000 lignes. Des bogues ont été trouvés dans ces programmes et les résultats obtenus sont utiles pour la maintenance et la réutilisation.

Les diverses vérifications étudiées et des expérimentations effectuées dans le cours de cette thèse suggèrent quelques améliorations et ouvrent de nouvelles perspectives pour les travaux futurs:

1. *Analyse de graphe de flot de contrôle.* Les analyses préliminaires comme les transformeurs, les préconditions et les régions de tableau peuvent être améliorées en utilisant une analyse sémantique plus précise sur les graphes de flot de contrôle. Par exemple, le résultat de l'analyse de variables non initialisées du Chapitre 8 peut être considérablement amélioré si les différents chemins du graphe de flot de contrôle sont pris en compte en calculant les régions importées de ce graphe.

2. *Analyse d'expressions non-linéaires.* Cette analyse n'est pas effectuée par PIPS et c'est pourquoi nous n'avons pas de pourcentages élevés de tests enlevés sur certains benchmarks, comme l'illustrent les résultats expérimentaux du Chapitre 6. D'autre part, les tableaux d'indirection rendent aussi les analyses de programme inefficaces dans certains cas.

3. *Autres analyses de région de tableau.* Comme mentionné auparavant, d'autres mises en oeuvre de régions de tableau peuvent être adoptées pour améliorer le temps de compilation ou pour avoir des régions de tableau plus exactes. La deuxième approche du test de débordement intraprocédural de tableaux peut être plus efficace si les régions de tableau sont traitées dimension par dimension. L'information sur les éléments importés de tableau peut être plus précise si la liste des sections complémentaires de tableau est conservée en exécutant quelques opérations convexes sur des régions de tableau.

4. *Redimensionnement de tableaux par des conditions de non alias.* Nous envisageons une autre approche complémentaire pour le redimensionnement de tableaux en employant les conditions de non alias. Cette approche évite les fausses alarmes causées par la surapproximation des déclarations de tableau donnée par d'autres approches.

5. *Condition de correction totale.* Un travail futur pourrait être d'établir la condition de correction totale dans certains cas en utilisant l'analyse statique pour garantir la fiabilité, la robustesse et la qualité des programmes. Cette condition globale est obtenue au prix de la perte d'information sur l'origine des erreurs mais elle permet de vérifier la validation des lectures des paramètres.

6. *Autres langages de programmation.* Nous pouvons appliquer nos vérifications à d'autres langages de programmation dont les problèmes existants sont encore des problèmes ouverts comme la déclaration de tableau, le débordement de tampon et l'analyse des alias dans le langage C, le surcoût de test de débordement de tableaux dans JAVA, etc.

La combinaison d'analyse statique et instrumentation de code semble promise à un avenir certain dans le domaine de la vérification du logiciel.

10.2 English

We have studied in this dissertation four different program analyses and verification techniques: array resizing, array bound checking, alias analysis and uninitialized variable analysis. All these analyses have been implemented and experimented with different benchmarks and industrial applications. The experimental results with SPEC95 CFP benchmarks are given for each of the verifications. Results obtained with industrial applications cannot be published. However they show that our implementation is robust and scales up to 100 000 lines of Fortran code and hundreds of subroutines. They also showed that a strict conformance to the standard was not a practical goal. Fortran programmers use relaxed practical rules that must be taken into account instead or on top of the standard. Moreover debugging is more efficient if detected errors are traced and do not stop the executions. In one run, which may last for hours, several bugs can thus be identified.

To our knowledge, there has not been much related work done on the problem of incomplete array declarations that make program analyses, code understanding and debugging difficult. This is also the reason of large execution time overhead for dynamic resizing in some programming languages such as MATLAB and JAVA. The two main array resizing approaches developed in Chapter 5 help to solve the ugly and assumed-size array problems in FORTRAN and they can be applied to other languages such as C to have additional information on pointers.

Two intraprocedural and one interprocedural array bound checking algorithms are studied in Chapter 6. A complete experimental set about the number of removed checks, debugging information, compilation and execution times has been measured on different compilers and platforms to show the effectiveness and the limited cost of the two intraprocedural optimizers (Elimination of redundant tests and Insertion of unavoidable tests). To complete array range checking, we introduce in this thesis an interprocedural verification about the actual/formal array sizes to prove the absence of or to detect errors when passing array arguments in procedure calls. This analysis is neither addressed in other work nor present in commercial compilers.

The alias analysis in Chapter 7 lets us avoid the worst-case assumption about aliases and perform program analyses under less restrict assumptions about program quality. We developed algorithms to check for violation of aliasing rules in FORTRAN, an option missing in most compilers. This standard check is useful to debug code, to help programmers to correct errors in order to gain performance by applying optimizations safely. In addition, the impact of aliasing on a program dependence graph is also studied.

By using advanced interprocedural array region analyses, we show how detecting potentially uninitialized variables pointed out by our compile-time analysis in Chapter 8 can improve the testing and validation process by saving debugging time. Used before set errors are caught at run-time to avoid memory violation problems such as incorrect results, unpredictable behavior or system crashes.

The interactions between the above verifications and program analyses as well as their ordering to show how can we verify correctly a buggy program are proposed in Chapter 9.

The results presented in this dissertation summarize the following main contributions:

1. *An effective combination of static analysis and code instrumentation.* The thesis demonstrates that program analysis can be a powerful tool for improving software security. Static analysis is profitably combined with run-time techniques to reduce code instrumentation and therefore execution overhead. With available information provided by static analysis, only some code fragments requiring checking are instrumented. This method also makes it easy to perform error checking at the optimal time, during the development process. In comparison to other program verification techniques, our approach is completely automatic and does not need human interactions such as user annotations and assertions. Static analysis can be used to build an effective tool for detecting errors in large scale systems at reasonable cost.

2. *A support for program debugging.* The lessons learned from experimental results permit to target our error detection techniques for debugging purpose, one of the core software engineering activities. To detect a maximum number of errors in one pass, a `PRINT` message with useful debugging information is used instead of a `STOP` statement. Unlike some code parsers that report only one error in one pass, static analysis and instrumentation enable us to trace all these errors and to avoid repeated compilation and execution. Moreover, since the standard is often violated by real-world programs, new notions of checking are proposed, such as the area bound checking in Chapter 6, the more flexible alias violation conditions and the aliasing impact analysis in Chapter 7. The distinction between *trivial* and *dangerous* bugs permits to reduce the number of false alarms and to take into account the ways FORTRAN programmers have purposely violated the standard. All this information is really needed to maintain, reuse, re-engineer and enhance software systems.

3. *A complete integration in a source-to-source compiler.* Efficient program verifications are performed by reusing interprocedural analysis techniques already implemented in commercial compilers. Our implementation in PIPS represents about 1600 lines of C code for array resizing with the three approaches: argument association rules, array region based and code instrumentation. With array bound checking, it is about 1200 lines of C code for the Elimination of Redundant Tests approach, 700 for the Insertion of Unavoidable Tests approach and 500 for Interprocedural Array Bound Checking. It is less than 2000

lines for the alias analysis, both propagation and verification phases. The uninitialized variables analysis takes about 600 lines of C code. The total number is 6600 lines for different methods of different verifications. Non-regression tests are run frequently to test the robustness of the implementation. We have developed a test database to validate these implementations.

4. *Qualitative and quantitative experiments.* Experiments have been performed with excerpts from industrial applications and three important benchmarks: Linpack, Perfect Club and SPEC95 CFP. We focused more particularly on the SPEC95 CFP benchmarks with ten publicly available programs. The numbers of lines and subprograms vary, respectively from 190 lines and 1 subprogram (*tomcatv*) to 7764 lines and 105 subprograms (*wave5*). We measured the performance of our analyses with different commercial compilers (SUN, SGI and IBM) on different platforms, and also with other related work. Experiment results show that SPEC95 CFP benchmarks are rather well debugged programs. They do not suffer from significant errors, except the interprocedural array bound violation and used before set errors in *wave5*. In addition, our verifications have been applied to two industrial applications that are under development. The first one contains more than 83000 lines and the second one 200000 lines. Bugs have been found in these programs and the obtained results are useful for the maintenance and reverse engineering processes.

The different verifications studied in the thesis suggest some improvements and open up some new perspectives for the future work:

1. *Control flow graph analysis.* The preexisting analyses such as transformers, preconditions and array regions can be improved by using more precise semantic analysis on control flow graphs. For example, the result of the uninitialized variables analysis in Chapter 8 can be improved if different paths in a control flow graph are taken into account when computing its imported array regions.

2. *Nonlinear expression analysis.* This analysis is not handled by PIPS so, as shown by experimental results in Chapter 6, we do not have high percentages of removed checks for several benchmarks. Moreover, subscripted array subscriptions or indirection arrays also make program analyses inefficient in some cases.

3. *Other array region analyses.* As mentioned, other implementations of array regions can be used to improve the compilation time or to keep array regions more exact. We can treat array regions dimension by dimension for array bound checking, or store the list of complementary array sections when performing some convex operations on array regions, in order to have more precise information on the imported array elements for the used

before set analysis.

4. *Array resizing by non-aliasing condition.* We are planning to implement another complementary approach for array resizing by using the non-aliasing condition. This approach avoids the false alarms caused by the over-approximation of array declarations given by other approaches.

5. *Total correctness condition.* One of our future works is to establish the total correctness condition in some cases by using static analysis in order to guarantee the program reliability, robustness and quality. Information about the origin of error may be lost but this condition permits us, for example, to verify as soon as possible the validation of input parameters.

6. *Other programming languages.* We can apply our verifications to other programming languages whose existing problems are still open issues such as the array declarations, buffer overflows and aliasing problems in C, the overhead of array bound checks in JAVA, etc.

The combination of static analysis and code instrumentation has indeed a promising future in the software verification domain.

Bibliography

[ABC+88] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeane Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5:617–640, 1988.

[ABS94] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, June 1994.

[AI91] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

[AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, San Francisco, 2002.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, Denmark, May 1994.

[ANS83] ANSI. *Programming Language FORTRAN, ANSI X3.9-1978, ISO 1539-1980.* American National Standard Institute, New York, 1983.

[App01] John Appleyard. Comparing Fortran compilers. *ACM SIGPLAN - Fortran Forum*, 20(1):6–10, 2001.

[AR01a] Aneesh Aggarwal and Keith H. Randall. Related field analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 214–220, June 2001.

[AR01b] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, June 2001.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, Massachusetts, 1986.

[Asu92] Jonathan M. Asuru. Optimization of array subscript range checks. *ACM Letters on Programming Languages and Systems*, 1(2):109–118, June 1992.

[Ban79] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *ACM Symposium on Principles of Programming Languages*, pages 29–41, January 1979.

[Bar77] Jeffrey M. Barth. An interprocedural data flow analysis algorithm. In *ACM Symposium on Principles of Programming Languages*, pages 119–131, January 1977.

[BCCH95] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *International Workshop on Languages and Compilers for Parallel Computing*, volume 892 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[BCF97] Denis Barthou, Jean François Collard, and Paul Feautrier. Fuzzy data flow analysis. *Journal of Parallel and Distributed Computing*, 40(2):210–226, 1997.

[Bei90] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 1990.

[BGS97] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Refining data flow information using infeasible paths. In *European Software Engineering Conference*, pages 361–377, September 1997.

[BGS00] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bounds checks on demand. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–333, June 2000.

[Bir67] Garrett Birkhoff. *Lattice Theory*. American Mathematical Society, 1967.

[BJ01] William Brew and Maggie Johnson. Value lattice static analysis. *Dr. Dobb's Journal*, pages 30–38, March 2001.

[BR01] Thomas Ball and Sriram K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 97–103, June 2001.

[Bud88] Timothy Budd. *An APL Compiler*. Springer-Verlag, New York, 1988.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer induced aliases and side effects. In *ACM Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Second International Symposium on Programming*, pages 106–130, April 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[CC00] Patrick Cousot and Radhia Cousot. Temporal abstract interpretation. In *ACM Symposium on Principles of Programming Languages*, pages 12–25, January 2000.

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *ACM Symposium on Principles of Programming Languages*, pages 55–66, January 1991.

[CGP99] Edmun M. Clarke, Jr. Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, 1999.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM Symposium on Principles of Programming Languages*, pages 84–96, January 1978.

[CI95] Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. In *International Workshop on Languages and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 1995.

[CI96] Béatrice Creusillet and François Irigoin. Exact vs. approximate array region analyses. In *International Workshop on Languages and Compilers for Parallel Computing*, volume 1239 of *Lecture Notes in Computer Science*, pages 86–100. Springer-Verlag, 1996.

[CK88] David Callahan and Ken Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

[CK89] Keith D. Cooper and Ken Kennedy. Fast interprocedural alias analysis. In *ACM Symposium on Principles of Programming Languages*, pages 49–59, January 1989.

[Coo84] Keith D. Cooper. Analyzing aliases of reference formal parameter. In *ACM Symposium on Principles of Programming Languages*, pages 281–290, January 1984.

[Cou86] Deborah S. Coutant. Retargetable high level alias analysis. In *ACM Symposium on Principles of Programming Languages*, pages 110–118, January 1986.

[Cou97] Patrick Cousot. Program analysis: The abstract interpretation perspective. *ACM SIGPLAN Notices*, 32(1):73–76, 1997.

[Cre95] Béatrice Creusillet. IN and OUT array region analyses. In *Workshop on Compilers for Parallel Computers*, pages 233–246, June 1995.

[Cre96] Béatrice Creusillet. *Analyses de régions de tableaux et applications*. PhD thesis, Centre de Recherche en Informatique, Ecole des Mines de Paris, December 1996.

[CRL01] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Complexity of points-to analysis of Java in the presence of exceptions. *IEEE Transactions on Software Engineering*, 27(6):481–512, June 2001.

[DD98] Jozo J. Dujmovic and Ivo Dujmovic. Evolution and evaluation of SPEC benchmarks. *ACM SIGMETRICS*, 26(3):2–9, 1998.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *ACM SIGPLAN Notices*, 29(6):230–241, 1994.

[DGS93] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical data flow framework for array reference analysis and its application in optimization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–77, June 1993.

[DJP00] Giorgio Delzanno, Georg Jung, and Andreas Podelski. Static analysis of array bounds as infinite-state model checking. Unpublished, 2000.

[DLFR01] Manuvir Das, Ben Liblit, Manuel Fahndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Static Analysis*, volume 2126 of *Lecture Notes in Computer Science*, pages 260–278. Springer-Verlag, 2001.

[DRS01] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cleanness checking of string manipulation in C programs via integer analysis. In *Static Analysis*, volume 2126 of *Lecture Notes in Computer Science*, pages 194–212. Springer-Verlag, 2001.

[EGHT94] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 87–96, December 1994.

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, January/February 2002.

[Fea88] Paul Feautrier. Parametric integer programming. *Recherche Operationnelle/Operations Research*, 22(3):243–268, 1988.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[Fea01] Paul Feautrier. *Compiler Optimizations for Scalable Parallel Systems*, volume 1808 of *Lecture Notes in Computer Science*, chapter 6: Array Dataflow Analysis, pages 173–220. Springer-Verlag, 2001.

[Fou27] J.B.J Fourier. Analyse de travaux de l'académie royale des sciences, pendant l'année 1824, partie mathématique. *Histoire de l'Académie Royale des Sciences de l'Institut de France*, 1827.

[FW00] Phyllis G. Frankl and Elaines J. Weyuker. Testing software to detect and reduce risk. *The Journal of Systems and Software*, 53:275–286, 2000.

[GK94] John Gough and Herbert Klaeren. Eliminating range checks using static single assignment form. Technical Report WSI-9413, Wilhelm Schickard Institut, Universitat Tubingen, Germany, 1994.

[GL00] Junjie Gu and Zhiyuan Li. Efficient interprocedural array data-flow analysis for automatic program parallelization. *IEEE Transactions on Software Engineering*, 26(3):244–261, March 2000.

[GLL95] Junjie Gu, Zhiyuan Li, and Gyungho Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Supercomputing*, December 1995.

[GLS01] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 47–58, June 2001.

[GMS00] Manish Gupta, Sayak Mukhopadhyay, and Navin Sinha. Automatic parallelization of recursive procedures. *International Journal of Parallel Programming*, 28(6):537–562, 2000.

[Gor79] Michael J.C. Gordon. *The Denotational Description of Programming Languages, An introduction.* Springer-Verlag, New York, 1979.

[Gor88] Michael J.C. Gordon. *Programming Language Theory and its Implementation.* Prentice Hall, Englewood Cliffs, 1988.

[GP01] Wolfgang Gellerich and Erhard Plodereder. Parameter-induced aliasing in Ada. In *Ada-Europe*, volume 2043 of *Lecture Notes in Computer Science*, pages 88–99. Springer-Verlag, 2001.

[Gup90] Rajiv Gupta. A fresh look at optimizing array bound checking. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 272–282, June 1990.

[Gup93] Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, March-December 1993.

[Hal79] Nicolas Halbwachs. *Détermination automatique de relations lineaires vérifiées par les variables d'un programme.* PhD thesis, Institut National Polytechnique de Grenoble, March 1979.

[Har77] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.

[HBCC99] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[HBCM92] Michael Hind, Michael Burke, Paul Carini, and Sam Midkiff. Interprocedural array analysis: How much precision do we need? In *Workshop on Compilers for Parallel Computers*, pages 48–64, June 1992.

[HBCM94] Michael Hind, Michael Burke, Paul Carini, and Sam Midkiff. An empirical study of precise interprocedural array analysis. *Scientific Programming*, 3(3):255–271, 1994.

[Hin01] Michael Hind. Pointer analysis: haven't we solved this problem yet ? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, June 2001.

[HJ92] Reed Hasting and Bob Joyce. Purify: fast detection of memory leaks and access errors. In *Winter USENIX Conference*, pages 125–136, January 1992.

[HMA+95a] Mary Hall, Brian Murphy, Saman Amarasinghe, Shih-Wei Liao, and Monica Lam. Interprocedural analysis for parallelization. In *International Workshop on Languages and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, pages 61–80. Springer-Verlag, 1995.

[HMA+95b] Mary W. Hall, B. Murphy, S. Amarasinghe, S. Liao, and M. Lam. Overview of an interprocedural automatic parallelization system. In *Workshop on Compilers for Parallel Computers*, pages 570–579, June 1995.

[Hoc96] Roger W. Hockney. *The Science of Computer Benchmarking.* SIAM, Philadelphia, 1996.

[Hol00] Gerard J. Holzmann. Software verification at Bell Labs: One line of development. *Bell Labs Technical Journal*, pages 35–45, January-March 2000.

[HP01] Michael Hind and Anthony Pioli. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming*, 39:31–55, 2001.

[HPY01] Jay P. Hoeflinger, Yunheung Paek, and Kwang Yi. Unified interprocedural parallelism detection. *International Journal of Parallel Programming*, 29(2):185–215, 2001.

[HR99] Nicolas Halbwachs and Pascal Raymond. Validation of synchronous reactive systems: from formal verification to automatic testing. In *Asian Computing Science Conference*, volume 1742 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[HT01a] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–34, June 2001.

[HT01b] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, June 2001.

[IJT91] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: an overview of the PIPS project. In *International Conference on Supercomputing*, pages 144–151, June 1991.

[Iri93] François Irigoin. Interprocedural analyses for programming environments. In *Environments and Tools for Parallel Scientific Computing*, pages 333–350. Elsevier, 1993.

[JB01] Pramod G. Joisha and Prithviraj Banerjee. Correctly detecting intrinsic type errors in typeless languages such as MATLAB. In *International Conference on APL*, pages 7–21, June 2001.

[JK97] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third International Workshop on Automated Debugging*, May 1997.

[KAC+96] Ronan Keryell, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, François Irigoin, and Pierre Jouvelot. PIPS: A workbench for building interprocedural parallelizers, compilers and optimizers. In *European Parallel Tool Meeting*, October 1996.

[Kar76] Michael Karr. Affine relationship among variables of a program. *Acta Informatica*, pages 133–151, 1976.

[KDA02] Sumant Kowshik, Dinakar Dhurjati, and Vikram Adve. Ensuring code safety without runtime checks for real-time control systems. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2002.

[KMM+95] A. Kubota, I. Miyoshi, K. Maeyama, S. Goto, S. Mori, H. Nakashima, and S. Tomita. TINPAR: A parallelizing compiler for message passing multiprocessors. In *International Symposium on Parallel and Distributed Supercomputing*, pages 214–223, September 1995.

[KNT00] Mickael Kerboeuf, David Nowak, and Jean-Pierre Talpin. Specification and verification of a steam-boiler with Signal-Coq. In *13th International Conference on Theorem Proving in Higher Order Logics*, volume 1869 of *Lecture Notes in Computer Science*, pages 356–371. Springer-Verlag, 2000.

[KW95] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–278, June 1995.

[Les96] Arnauld Leservot. *Analyses interprocédurales du flot des données*. PhD thesis, Université Paris VI, March 1996.

[LGL01] Zhiyuan Li, Junjie Gu, and Gyungho Lee. *Compiler Optimizations for Scalable Parallel Systems*, volume 1808 of *Lecture Notes in Computer Science*, chapter 7: Interprocedural Analysis Based on Guarged Array Regions, pages 221–246. Springer-Verlag, 2001.

[LH01] Donglin Liang and Mary Jean Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Static Analysis*, volume 2126 of *Lecture Notes in Computer Science*, pages 279–298. Springer-Verlag, 2001.

[LMW01] O. Laurent, P. Michel, and V. Wiels. Using formal verification techniques to reduce simulation and test effort. In *Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 465–477. Springer-Verlag, 2001.

[LP99] Yuan Lin and David Padua. Demand-driven interprocedural array property analysis. In *International Workshop on Languages and Compilers for Parallel Computing*, volume 1863 of *Lecture Notes in Computer Science*, pages 303–317. Springer-Verlag, 1999.

[LPH01] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 73–79, June 2001.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *ACM SIGPLAN Notices*, 27(7):235–248, 1992.

[LYHR01] Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas W. Reps. Debugging via run-time type checking. In *Fundamental Approaches to Software Engineering*, pages 217–232, April 2001.

[MAL93] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array dataflow analysis and its use in array privatization. In *ACM Symposium on Principles of Programming Languages*, pages 2–15, January 1993.

[May92] Dror Eliezer Maydan. *Accurate Analysis of Array References.* PhD thesis, Computer Science Department, Stanford University, September 1992.

[MCM82] Victoria Markstein, John Cocke, and Peter Markstein. Optimization of range checking. *ACM SIGPLAN Notices*, 17(6):114–119, June 1982.

[MDCE01] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers. Dynamic points-to sets: a comparison with static analyses and potential applications in program understanding and optimization. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 66–72, June 2001.

[Mey90] Bertran Meyer. *Introduction to the Theory of Programming Languages.* Prentice Hall, Englewood Cliffs, 1990.

[MMG00] Jose E. Moreira, Samuel P. Midkiff, and Manish Gupta. From flop to megaflops: JAVA for technical computing. *ACM Transactions on Programming Languages and Systems*, 22(2):265–295, March 2000.

[MMG+01] Jose E. Moreira, Samuel P. Midkiff, Manish Gupta, Pedro V. Artigas, Peng Wu, and George Almasi. The NINJA project. *Communications of the ACM*, 44(10):102–109, October 2001.

[MMS98] Samuel P. Midkiff, Jose E. Moreira, and M. Snir. Optimizing array reference checking in JAVA programs. *IBM Systems Journal*, 37(3):409–453, 1998.

[MN97] M. Manjunathaiah and Denis A. Nicole. Precise analysis of array usage in scientific programs. *Scientific Programming*, 6:229–242, 1997.

[MP89] Zohar Manna and Amir Pnueli. The anchored version of the temporal framework. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 201–284. Springer-Verlag, 1989.

[MP99] Vijay Menon and Keshav Pingali. A case for source-level transformations in MATLAB. In *ACM SIGPLAN 2nd Conference on Domain-Specific Languages*, pages 53–65, October 1999.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, San Francisco, 1997.

[MW93] Herbert G. Mayer and Michael Wolfe. Interprocedural alias analysis: implementation and empirical results. *Software - Practice and Experience*, 23(11):1201–1233, November 1993.

[Neu01] Peter G. Neumann. Risks to the public in computers and related systems. *ACM SIGSOFT Software Engineering Notes*, 26(6):6–15, November 2001.

[Neu02] Peter G. Neumann. Risks to the public in computers and related systems. *ACM SIGSOFT Software Engineering Notes*, 27(2):5–19, March 2002.

[NGL95] Trung N. Nguyen, Junjie Gu, and Zhiyuan Li. An interprocedural parallelizing compiler and its support for memory hierarchy research. In *International Workshop on Languages and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, pages 96–110. Springer-Verlag, 1995.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–344, June 1998.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag, Berlin, 1999.

[PF97] Harish Patil and Charles N. Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software - Practice and Experience*, 27(1):87–110, 1997.

[PHP98] Yunheung Paek, Jay Hoeflinger, and David Padua. Simplification of array access patterns for compiler optimizations. In *ACM SIGPLAN Conference* *on Programming Language Design and Implementation*, pages 60–71, June 1998.

[PHP02] Yunheung Paek, Jay Hoeflinger, and David Padua. Efficient and precise array access analysis. *ACM Transactions on Programming Languages and Systems*, 24(1):65–109, 2002.

[Pug92] William Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[QHV02] Feng Qian, Laurie Handren, and Clark Verbrugge. A comprehensive approach to array bounds check elimination for Java. In *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 325–341. Springer-Verlag, 2002.

[RG89] Stephen Richardson and Mahadevan Ganapathi. Interprocedural optimization: Experimental results. *Software - Practice and Experience*, 19(2):149–169, February 1989.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM Symposium on Principles of Programming Languages*, pages 49–61, January 1995.

[RLS+01] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems*, 23(2):105–186, 2001.

[RR00] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–195, June 2000.

[Ruf95] Erik Ruf. Context-insensitive alias analysis reconsidered. *ACM SIGPLAN Notices*, 30(6):13–22, June 1995.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Chichester, 1986.

[SG95] Madalene Spezialetti and Rajiv Gupta. Loop monotonic statements. *IEEE Transactions on Software Engineering*, 21(6):497–505, June 1995.

[SH97] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *ACM Symposium on Principles of Programming Languages*, pages 1–14, January 1997.

[Sha00] Kent Shaw. Run-time error checking. *Embedded Developers Journal*, pages 8–14, May 2000.

[SI77] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *ACM Symposium on Principles of Programming Languages*, pages 132–143, January 1977.

[SKL88] Birgit Schwarz, Walter Kirchgassner, and Rudolf Landwehr. An optimizer for Ada - design, experiences and results. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 175–184, June 1988.

[SP01] Natasha Sharygina and Doron Peled. A combined testing and verification approach for software reliability. In *Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 611–628. Springer-Verlag, 2001.

[Ste92] Joseph L. Steffen. Adding run-time checking to the portable C compiler. *Software - Practice and Experience*, 22(4):305–316, April 1992.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[SWSP01] Amie L. Souter, Tiffany M. Wong, Stacey A. Shindo, and Lori. L. Pollock. TATOO: Testing and analysis tool for object-oriented software. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 389–403. Springer-Verlag, 2001.

[TFI86] Remi Triolet, Paul Feautrier, and François Irigoin. Automatic parallelization of Fortran programs in the presence of procedure calls. In *European Symposium on Programming*, March 1986.

[TP95] Peng Tu and David A. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *International Conference on Supercomputing*, pages 414–423, July 1995.

[Wag00] David A. Wagner. *Static analysis and computer security: new techniques for software assurance.* PhD thesis, Computer Science, University of California at Berkeley, Fall 2000.

[Wal91] David W. Wall. Limits of instruction-level parallelism. *ACM SIGPLAN Notices*, 26(4):176–188, 1991.

[Wel78] J. Welsh. Economic range checks in Pascal. *Software - Practice and Experience*, 8:85–97, 1978.

[WFPS02] Peng Wu, Paul Feautrier, David Padua, and Zehra Sura. Instance-wise points-to analysis for loop-based dependence testing. In *International Conference on Supercomputing*, pages 262–273, 2002.

[Whi00] James A. Whittaker. What is software testing? and why is it so hard? *IEEE Software*, 17(1):70–79, January/February 2000.

[Wil97] Doran K. Wilde. A library for doing polyhedral operations. Technical Report RR-2157, Brigham Young University, August 1997.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. *ACM SIGPLAN Notices*, 33(5):249–257, 1998.

[ZC90] Hans P. Zima and Barbara M. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley, Reading, 1990.

[ZRL96] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 81–92, October 1996.

Personal Publications

1. Thi Viet Nga Nguyen, François Irigoin, Corinne Ancourt and Ronan Keryell. Efficient intraprocedural array bound checking. In *Second International Workshop on Automated Program Analysis, Testing and Verification* (WAPATV), Toronto, Canada, May 2001.

2. Thi Viet Nga Nguyen and François Irigoin. Efficient and effective array bound checking. Submitted to *ACM Transactions on Software Engineering and Methodology journal*, October 2001.

3. Corinne Ancourt and Thi Viet Nga Nguyen. Array resizing for code debugging, maintenance and reuse. In *ACM SIGLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (PASTE), pages 32-37, Utah, USA, June 2001.

4. Thi Viet Nga Nguyen and François Irigoin. Alias verification for Fortran code optimization. In *International Workshop on Compiler Optimization Meets Compiler Verification* (COCV), pages 56-70, Grenoble, France, April 2002.

5. Thi Viet Nga Nguyen and François Irigoin. Alias verification for Fortran code optimization. *Electronic Notes in Theoretical Computer Science, Elsevier Science*, 65(2), 2002.

6. Thi Viet Nga Nguyen and François Irigoin. Alias verification for Fortran code optimization (extended version of 4,5). Submitted to *Journal of Universal Computer Science, Springer-Verlag, Heidelberg, Germany*, September 2002.

7. Thi Viet Nga Nguyen and François Irigoin. Advanced program analyses and verifications. *First International Conference of French Speaking Vietnamese Computer Scientists* (RIVF), Hanoi, Vietnam, February 2003.

8. Thi Viet Nga Nguyen, François Irigoin, Corinne Ancourt and Fabien Coelho. Automatic detection of uninitialized variables. *12th International Conference on Compiler Construction*, Warsaw, Poland, April 2003.

9. Technical reports in PIPS (*http://www.cri.ensmp.fr/pips*).