Generating Program Analyzers

Dissertation

Zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften (Dr.-Ing.)
der Technischen Fakultät
der Universität des Saarlandes

von

Diplom-Informatiker
Florian Martin

Saarbrücken
Juni 1999

Tag des Kolloquiums:	22.10.1999
Dekan:	Prof. Dr. W.J. Paul
Gutachter:	Prof. Dr. R. Wilhelm Prof. H.R. Nielson, Ph.D.
Vorsitzender:	Prof. Dr.-Ing. G. Weikum

Abstract

In this work the automatic generation of program analyzers from concise specifications is presented. It focuses on provably correct and complex interprocedural analyses for real world sized imperative programs. Thus, a powerful and flexible specification mechanism is required, enabling both correctness proofs and efficient implementations. The generation process relies on the theory of data flow analysis and on abstract interpretation. The theory of data flow analysis provides methods to efficiently implement analyses. Abstract interpretation provides the relation to the semantics of the programming language. This allows the systematic derivation of efficient provably correct, and terminating analyses. The approach has been implemented in the program analyzer generator PAG. It addresses analyses ranging from "simple" intraprocedural bit vector frameworks to complex interprocedural alias analyses. A high level specialized functional language is used as specification mechanism enabling elegant and concise specifications even for complex analyses. Additionally, it allows the automatic selection of efficient implementations for the underlying abstract datatypes, such as balanced binary trees, binary decision diagrams, bit vectors, and arrays. For the interprocedural analysis the functional approach, the call string approach, and a novel approach especially targeting on the precise analysis of loops can be chosen. In this work the implementation of PAG as well as a large number of applications of PAG are presented.

Zusammenfassung

Diese Arbeit befaßt sich mit der automatischen Generierung von Programmanalysatoren aus prägnanten Spezifikationen. Dabei wird besonderer Wert auf die Generierung von beweisbar korrekten und komplexen interprozeduralen Analysen für imperative Programme realer Größe gelegt. Um dies zu erreichen, ist ein leistungsfähiger und flexibler Spezifikationsmechanismus erforderlich, der sowohl Korrektheitsbeweise, als auch effiziente Implementierungen ermöglicht. Die Generierung basiert auf den Theorien der Datenflußanalyse und der abstrakten Interpretation. Die Datenflußanalyse liefert Methoden zur effizienten Implementierung von Analysen. Die abstrakte Interpretation stellt den Bezug zur Semantik der Programmiersprache her und ermöglicht dadurch die systematische Ableitung beweisbar korrekter und terminierender Analysen.
Dieser Ansatz wurde im Programmanalysatorgenerator PAG implementiert, der sowohl für einfache intraprozedurale Bitvektor-Analysen, als auch für komplexe interprozedurale Alias-Analysen geeignet ist. Als Spezifikationsmechanismus wird dabei eine spezialisierte funktionale Sprache verwendet, die es ermöglicht, auch komplexe Analysen kurz und prägnant zu spezifizieren. Darüberhinaus ist es möglich, für die zugrunde liegenden abstrakten Bereiche automatisch effiziente Implementierungen auszuwählen, z.B. balancierte binäre Bäume, Binary Decision Diagrams, Bitvektoren oder Felder. Für die interprozedurale Analyse stehen folgende Möglichkeiten zur Auswahl: der funktionale Ansatz, der Call-String-Ansatz und ein neuer Ansatz, der besonders auf die präzise Analyse von Schleifen abzielt. Diese Arbeit beschreibt sowohl die Implementierung von PAG, als auch eine große Anzahl von Anwendungen.

Extended Abstract

Program analysis is a method to determine information about the runtime behavior of a program at compile time without executing it. It is used in optimizing compilers in order to produce high quality code. There, the task of program analysis is to verify the applicability of efficiency increasing program transformations. Another area of use are program validation tools which verify certain aspects of a program like the timing behavior or the absence of common programming errors e.g. in the context of heap allocated storage.

The implementation of program analyzers is usually difficult and expensive, and gets even more complex in the presence of procedures. Implementing several program analyzers reveals recurrent tasks, like the implementation of appropriate equation solvers. The tradition of automatic software generation in the area of compiler construction has led to the design of a tool which supports the implementation of program analyzers.

This work presents the program analyzer generator PAG which allows the automatic generation of program analyzers from clear and concise specifications. It focuses on the generation of efficient interprocedural analyzers for real world sized imperative programs, but can also be applied to logic languages as well as to object oriented languages to a certain extent. PAG is based on the theories of abstract interpretation and data flow analysis. Abstract interpretation provides the relation to the semantics of the programming language and allows the systematic derivation of provably correct and terminating analyses. The use of provably correct analyses is of special importance for tools validating safety critical software. Data flow analysis provides a common framework to describe program analyses and offers many efficient algorithms, such as fixed point iteration.

A specialized high level functional language is used to specify the transfer functions which model the effects of statements during the analysis process. The abstract domain which is a lattice of the values of interest is specified by inductive definitions. Using such a high level and powerful specification mechanism enables the writing of concise and clear specifications even for complex analysis problems by freeing the user from many implementation details like the manual allocation of storage. Additionally, it allows to exchange the underlying implementation of different parts of the analyzer without the need for reformulating the analysis. E.g. PAG automatically tries to detect an efficient implementation for the abstract domain, by choosing from AVL trees, red black trees, binary decision diagrams, bit vectors, and arrays. It is also possible to exchange the underlying fixed point computation algorithm without changing the specification or the interface to the compiler in which the analyzer is integrated.

For PAG generated analyzers the user can choose from several different interprocedural analysis techniques such as the functional approach or the call string approach. The analyzers are quite fast, since PAG uses a number of carefully implemented optimizations. PAG can generate an optional visualization interface to the vcg tool to simplify the understanding and debugging of the program analyzer. Since the analyzers are implemented in ANSI C they can be easily integrated into existing compilers by instantiating their procedural interfaces to the control flow graph and the abstract syntax tree.

Loops are of special importance in program analysis since programs usually spend most of their time in loops. It has been long recognized that different execution contexts for procedures may induce different execution properties. There are well established techniques for the analysis of programs with procedures, but loops have not received similar attention. All executions are treated in the same way, although typically the first and later executions often exhibit very different properties. Therefore, a new technique has been implemented in PAG which allows the application of the well known and established interprocedural theory to the analysis of loops. It turned out that the call string approach has limited flexibility in its possibility to group several calling contexts together for the analysis. An extension to overcome this problem is presented that relies on the same theory but gives more useful results in practice. Additionally, extensions to analyze the first N iterations of a loop separately and to limit the number of execution contexts for each loop are shown.

PAG is implemented in a modular way in ANSI C. It has been used successfully in the ESPRIT project COMPARE, and other research projects. Additionally, it has proven its stability and usability in several universities all over the world. The web frontend PAG/WWW has been implemented merely for educational purposes. It can be used to explore the specification and the working of an analyzer interactively. It has been used in several courses about program analysis, and its online version is visited from all over the world usually several times a day.

This work describes the motivation for implementing PAG, and discusses the major design decisions in detail. The concepts of the specification mechanism are explained and demonstrated giving a number of examples. An overview of the implementation is presented, and the highlights are discussed in detail. The flexibility of the specification mechanism is demonstrated by presenting a variety of analysis problems that have been implemented using PAG. The generated analyses range from "simple" intraprocedural bit vector frameworks –like reaching definitions– to complex intraprocedural alias analysis. For the generated analyzers a performance evaluation is shown by applying them to a set of real world programs. This evaluation also gives insight to the effect of different analysis methods and optimizations.

Ausführliche Zusammenfassung

Programmanalyse ist eine Methode zur Vorhersage des Laufzeitverhaltens von Programmen ohne sie tatsächlich auszuführen. Ihr Hauptanwendungsgebiet sind optimierende Compiler, die die Ergebnisse der Programmanalyse benutzen, um die Anwendbarkeit von effizienzsteigernden Transformationen zu überprüfen. Ein anderes wichtiges Einsatzgebiet der Programmanalyse sind Werkzeuge zur Programmüberprüfung, z.B. um das Timing Verhalten eines Programmes zu bestimmen, oder um Fehler bei der Benutzung von dynamischem Speicher aufzudecken.

Die Implementierung von Programmanalysatoren ist kompliziert und aufwendig, und wird sogar noch schwieriger, wenn Prozeduren in Betracht gezogen werden. Die Implementierung mehrerer Programmanalysatoren zeigt gewisse wiederkehrende Aufgaben auf, wie zum Beispiel die Implementierung passender Gleichungslöser. Dies und die lange Tradition von Werkzeugen zur automatischen Generierung von Software im Übersetzerbau haben zum Entwurf eines Werkzeugs geführt, das die Implementierung von Programmanalysatoren unterstützt.

Diese Arbeit präsentiert den Programmanalysatorgenerator PAG, der die automatische Generierung von Programmanalysatoren aus klaren und prägnanten Spezifikationen ermöglicht. Dabei wird besonderer Wert auf die Generierung von effizienten Analysatoren zur interprozeduralen Analyse von Programmen realer Größe gelegt. Der Generator ist für imperative, logische und bis zu einem gewissen Grade auch für objektorientierte Sprachen geeignet. PAG basiert auf den Theorien der Datenflußanalyse und der abstrakten Interpretation. Die abstrakte Interpretation stellt den Bezug zur Semantik der Programmiersprache her, und ermöglicht somit die systematische Ableitung von beweisbar korrekten und terminierenden Analysen. Die Verwendung beweisbar korrekter Analysen ist insbesondere bei der Überprüfung von sicherheitskritischer Software wichtig. Die Datenflußanalyse liefert eine Methode zur Beschreibung von Programmanalysen, sowie eine Reihe von effizienten Algorithmen, wie z.B. Fixpunktiterationen.

Um die Transferfunktionen, die die Effekte der Programmanweisungen für die Analyse modellieren, zu spezifizieren, wird eine speziell entwickelte funktionale Sprache verwendet. Die abstrakten Wertebereiche der Analyse werden durch rekursive Definitionen beschrieben. Die Verwendung eines solchen abstrakten Spezifikationsmechanismus ermöglicht es, auch für komplexe Analysen kurze und prägnante Spezifikationen zu schreiben. Außerdem wird der Anwender dadurch nicht mit Implementierungsdetails, wie z.B. der Speicherverwaltung, konfrontiert. Zusätzlich wird es möglich, Teile der Implementierung des Analysators auszutauschen, ohne die Spezifikation anpassen zu müssen. Dadurch kann PAG automatisch eine passende Implementierung für die abstrakten Bereiche auswählen. Dazu stehen unter anderem AVL-Bäume, Rot-Schwarz-Bäume, Binary Decision Diagrams, sowie Felder und Bitvektoren zur Verfügung. Auch den Fixpunkt Algorithmus kann man austauschen, ohne die Spezifikation ändern zu müssen.

Für die interprozedurale Technik, die von den PAG-generierten Analysatoren verwendet wird, kann der Benutzer entweder den funktionalen Ansatz oder den Call-String-Ansatz wählen. Dank einer Reihe von sorgfältig implementierten Optimierungen sind die erzielten Analysegeschwindigkeiten hoch. Zur Unterstützung der Fehlersuche kann PAG für ausgewählte Berechnungen Eingaben für das Visualisierungsprogramm vcg erzeugen. Da die Analysatoren in ANSI C implementiert sind, können sie meist einfach in bestehende Compiler integriert werden. Es muß nur eine Reihe von Funktionen zum Zugriff auf den Kontrollflußgraphen und den abstrakten Syntaxbaum zur Verfügung gestellt werden.

Da Programme üblicherweise den größten Teil ihrer Laufzeit in Schleifen verbringen, sollte diesen besondere Aufmerksamkeit in der Programmanalyse gewidmet werden. Für Prozeduren ist bereits vor längerer Zeit erkannt worden, daß die Ausführung in verschiedenen Kontexten zu verschiedenem Verhalten führen kann. Daher gibt es etablierte Techniken zur Analyse von Prozeduren. Aber Schleifen haben bisher nicht dieselbe Aufmerksamkeit erlangt. Dort werden alle Ausführungen gleich behandelt, obwohl die erste Ausführung der Schleife sich typischerweise stark von allen anderen Ausführungen unterscheidet. Aus diesem Grund wurde eine neue Technik entwickelt und in PAG implementiert, die es erlaubt, die bekannten Verfahren der interprozeduralen Analyse auf die Analyse von Schleifen zu übertragen. Dabei hat es sich herausgestellt, daß der Call-String-Ansatz nur bedingt zu guten Ergebnissen bei der Analyse von Schleifen führt. Daher wurde ein neuer Ansatz entwickelt, der in der Praxis bessere Ergebnisse liefert. Außerdem werden Erweiterungen vorgestellt, die es erlauben, die ersten N Schleifendurchläufe getrennt zu analysieren und die Anzahl der analysierten Kontexte einer Schleife zu begrenzen.

PAG ist in ANSI C implementiert. Es wurde erfolgreich in dem ESPRIT Projekt COMPARE und in einer Reihe von Forschungsprojekten eingesetzt. Außerdem hat es seine Anwendbarkeit und Zuverlässigkeit durch den Einsatz an verschiedenen Universitäten auf der ganzen Welt bewiesen. Das Web Frontend PAG/WWW wurde implementiert, um PAG in der Ausbildung besser nutzbar zu machen. Es kann verwendet werden, um Spezifikationen und die Arbeitsweise von Programmanalysatoren interaktiv zu erlernen. Es wurde für eine Reihe von Kursen über Programmanalyse benutzt und wird in seiner Online-Version mehrmals am Tag verwendet.

Diese Arbeit beschreibt die Grundlagen der Implementierung von PAG und erörtert die Entwurfsentscheidungen. Der Spezifikationsmechanismus wird anhand einer Anzahl von Beispielen erläutert und darüberhinaus wird die Implementierung ausführlich dargestellt. Um die Flexibilität von PAG zu demonstrieren, werden einige Beispielanalysen vorgestellt, die mit Hilfe von PAG implementiert wurden. Diese reichen von einfachen Bitvektor Analysen, wie z.B. verfügbare Definitionen, bis hin zu komplexen interprozeduralen Alias Analysen. Die Leistung der generierten Analysatoren wird ausführlich evaluiert, indem sie auf eine große Zahl realer Programme angewendet werden. Die Auswertung diskutiert auch die Einflüsse der verschiedenen Optimierungen, die in dieser Arbeit vorgestellt werden.

Acknowledgments

I would like to thank my supervisor Prof. Dr. Reinhard Wilhelm for the opportunity of working in his group, and for the freedom to select such an interesting topic. Furthermore I would like to thank him for his advice and support. Also Martin Alt deserves my thanks, as he had the original idea to develop a program analyzer generator and was deeply involved during the first years in the development of PAG.

Furthermore I would like to thank Hanne and Flemming Nielson as well as Helmut Seidl for special impulses for the future development of PAG.

Many features of PAG resulted from stimulations by the users in the compiler design group of the Universität des Saarlandes: Martin Alt, Daniel Bobbert, Beatrix Braune, Christian Ferdinand, Michael Haase, Daniel Kästner, Marc Langenbach, Oliver Lauer, Niklas Matthies, Christian Probst, Jörg Pütz, Thomas Ramrath, Michael Schmidt, Jörn Schneider, Martin Sicks, Henrik Theiling, Stephan Thesing, and Dirk Werth. They often had the doubtful pleasure of coming into contact with the most recent versions of PAG. They found many bugs and shortcomings and needed a lot of patience while I was busy removing them.

Several people provided parts of the implementation of PAG. I would like to thank Martin Alt who implemented parts of the core, Oliver Lauer who implemented a lot of different efficient data structures, Stephan Thesing who wrote the manual, Daniel Bobbert and Niklas Matthies who implemented the WWW frontend. The implementation of the Clax frontend is based on code of Martin Alt and Georg Sander, the C frontend is based on an implementation of Susan Horwitz, which was adapted by Martin Alt. The Sun executable frontend was implemented by Thomas Ramrath based on the EEL Library of the University of Wisconsin, the frontend for the While language was implemented by Daniel Bobbert, two frontends for Power PC executables have been written by Dirk Werth and Jörg Pütz and by Henrik Theiling.

Michael Schmidt implemented the frontend generator Gon that was used by himself to create an alternative Clax frontend. It was also used by Marc Langenbach to generate the CRL frontend and by Christian Probst to implement a JAVA frontend.

Numerous external users of PAG have importantly contributed to the success of the system by providing useful comments and by reporting installation problems and other shortcomings. They deserve my thanks, too.

For careful proof reading this work in different stages I thank Christian Ferdinand, Reinhold Heckmann, Reinhard Wilhelm and my wife Verena.

Finally, I would like to thank my family and my coworkers for their patience and their support.

Contents

Chapter 1

Introduction

1.1 The Generation of Program Analyzers

Program analysis is a widely used technique for automatically predicting the set of values or behaviors arising dynamically at runtime when executing a program on a computer. Since these sets are in most cases not computable, approximations are necessary. The calculation of the desired information is made statically at compile time without actually executing the program. Information computed like that can be used in optimizing compilers (Wilhelm and Maurer, 1997) to verify the applicability of code improving transformations. Classical examples are all kinds of avoidance of redundant computations, like reusing results which have already been computed, moving loop invariant code out of loops, evaluating expressions already at compile time if all arguments are known, or removing computations if the result is not needed.

A more recent use of the information obtained by program analysis is in software validation, either to provide the programmer with warnings about suspicious aspects of the program such as the possible use of uninitialized variables (Evans, 1996) or to verify certain aspects of the program such as the timing behavior (Ferdinand, 1997b). Another example of software validation tools using program analysis that is very famous nowadays are tools to detect year 2000 (Y2K) problems in software systems e.g. (Eidorff et al., 1999).

Program analysis has a long tradition. It started out with data flow analysis in (Kildall, 1973) and was improved by (Kam and Ullman, 1977). Other important publications about data flow analysis are (Hecht, 1977; Marlowe and Ryder, 1990; Nielson et al., 1998). One impact of this theory was the introduction of a common framework to describe program analyses: the data flow analysis framework. Many analyses in the literature have been described using this framework (Aho et al., 1986; Fischer and LeBlanc, 1988; Muchnick, 1997). The emphasis in these books is often on practical implementations of data flow analyses.

A more general framework for static program analyses was introduced in (Cousot and Cousot, 1977) called *abstract interpretation*. It is semantics based, thus it provides a relation of the semantics of the programming language to the program analysis. Abstract interpretation executes a program using *abstract values* in place of concrete values. It allows the systematic derivation of data flow analyses and provides methods to prove their correctness and termination. Other contributions of abstract interpretation are methods to approximate fixed points that allow a broader applicability of the methods or can speed up the fixed point computation.

In both main application areas of program analysis described above its use becomes more and more important. In the area of compiler construction there is a growing demand for fast adaptation of compilers to new hardware. This demand emerges especially from the fast growing market of embedded processors and their applications. As a result from the cost pressure these embedded processors have more and more irregular architectures, which makes a (good) adaptation of the compilers extremely difficult. For these architectures target specific optimizations are very important. Also the area of software validation tools is a growing market. As mentioned above these tools are heavily based on program analysis.

In the area of compiler construction there is a long tradition in the automatic generation of software. Generators like lex and yacc for frontend generation are used routinely to implement all kinds of language interfaces. Backend generators like beg (Emmelmann et al., 1989) or code selector generators like iburg (Fraser et al., 1992) are getting more and more popular. For program analysis the theoretical foundations are well explored. The next logical step is a tool supporting the complicated and error prone task of implementing program analyzers.

Until today, generators that support the optimization phase have not been very successful. There have been a number of efforts to establish these kinds of tools, but none of them were widely used. The tools can be divided into two classes: one class focused on the theory of program analysis and was designed as a platform for developing and testing new analyses. But their specification mechanism usually did not allow an efficient implementation. The generated analyzers were in most cases too slow to be used even for medium sized programs. Another class of tools had their emphasis on the support of the implementation of fast and efficient program analyzers for the use in larger compiler systems. Most of these tools use a variant of ANSI C to specify the transfer functions. But to generate efficient code they rely on a lot of prerequisites for these C specifications so that the user has to have a very deep knowledge of the system to write these specifications. As a consequence the tools are only used by a small group of people usually in the surrounding of their creators. Additionally, these tools focus merely on a restricted class of problems like bitvector frameworks, what has restricted their applicability.

1.2 The Program Analyzer Generator PAG

The lack of working generators has motivated the design and the implementation of PAG (Alt and Martin, 1995; Martin, 1995; Alt et al., 1995; Martin, 1998), a system to generate interprocedural analyzers that can be integrated into compilers by instantiating a well designed interface. The system is based on the theory of abstract interpretation. The philosophy of PAG is to support the designer of an analyzer by providing languages for specifying the data flow problem and the abstract domains on a quite abstract level. This simplifies the construction process of the analyzers, supports the correctness proof, and leads to a modular structure of the specification. The user of PAG is neither confronted with the implementation details of domain functionality nor with the traversal of the control flow graph or syntax tree nor with the implementation of suitable fixed point algorithms. The use of the high level specification language also enables a number of automated optimizations, so that the generated analyzers are reasonably fast.

1.3 Overview of the System

1.3.1 Historical Development

The first version of the PAG system was developed in the ESPRIT Project #5399 COMPARE (COMpiler Generation for PARallel MachinEs) in the year 1994. The Project COMPARE aimed for the development of a configurable, parallel compiler system for different source languages (ANSI C, Fortran 90, Modula-P, and others) and different target architectures (Sparc, Transputer, and others) (Alt et al., 1994). A system to support the development of several full program analysis phases –more complex than bit vector analyses– was needed. This first version of PAG was developed within the context of a master thesis (Martin, 1995) and was able to generate interprocedural call string zero analyzers (context insensitive) (see Chap. 3). At that time no support for different sophisticated implementations of the data structures of the generated analyzers existed. PAG was used to generate a constant propagation and a simple must alias analysis. The source code was nearly 45,000 lines (about 1.2 MB) large.

After COMPARE the development continued and PAG was ported to different Unix systems. It was used with a number of frontends for various source languages apart from the COMPARE intermediate representation. Support for different interprocedural analysis methods was added and the possibility of using widening and narrowing was implemented. A bunch of efficient implementations of the runtime data structures was added as well as the facility of automatically selecting an appropriated data structure in the generator. Also the visualization interface was improved, and the WWW frontend came into being. Today the core system consists of about 166,000 lines of documented source code (4.3 MB) written in C and to a minor extend in the flex and bison specification languages. The source code is distributed over 415 files which are divided into the following modules:

- the runtime system 12,000 lines (250 KB)
- the data structure templates 33,000 lines (650 KB)
- the code generator 19,000 lines (560 KB)
- tools 11,000 lines (270 KB)
- the data structure generator 15,000 lines (440 KB)

- the Clax frontend 9,000 lines (240 KB)
- the Executable frontend 7,000 lines (190 KB)
- the C frontend 30,000 lines (920 KB)
- the While frontend 6,000 lines (160 KB)
- generated prototype and inlining files 24,000 lines (620 KB)

Apart from the source code the PAG distribution (as it is available now, May 1999) consists of:

- 3 MB documentation as Postscript files and READMEs
- 130 KB makefiles and configure scripts
- 300 KB example specifications
- 3.4 MB input files for the example analyzers

Additionally the following packages are available:

- the frontend generator Gon 1 MB total, thereof 23,000 lines (650 KB) C Code and 3,300 lines (90 KB) example specifications with inputs and 280 KB documentations
- PAG/WWW the PAG web interface 590 KB, thereof 2,200 lines (70 KB) Perl source code and 12,000 lines (300 KB) C code, 70 KB icons, 900 lines (50 KB) of templates, analysis specifications and programs, as well as 3,700 lines (100 KB) documentations as static HTML code.

Since January 1999 PAG has been used in the ESPRIT project #28198 JOSES (Java and CoSy technologies for Embedded Systems) and the DFG Transfer Project #14. The latter aims to develop a system for the prediction of worst case execution times for programs to be used in hard real time environments. To better meet the very specific requirements, the further development of PAG is an additional part of the project.

1.3.2 Availability

Since late summer 1998 PAG is available under a research license. The license fees are waived for universities. Until now 13 sites have signed a license agreement and got PAG from the ftp server. These are sites from Austria, Denmark, France, Germany, Israel, Sweden, and the United States.

The system has been tested on various Unix platforms including SunOs, Solaris, NetBsd, Silicon Graphics Irix and Linux. A version for Windows NT is currently under development.

The PAG/WWW interface is running on a server in Saarbrücken since July 1998. For a course on Principles of Program Analysis several additional servers have been running. A mirror server in the USA is projected. The access log files from the primary server show increasing interest in the project. Since July 1998 there have been 45,000 accesses (loads of a file) from outside the University of Saarbrücken. It is difficult to say how many different persons have used the system so far. But there have been 8,500 post operations, where each post operation corresponds to a single interactive use of the server.

1.4 Overview of the Work

In the next chapter the underlying theories of abstract interpretation and data flow analysis are briefly described. The description focuses on the parts interesting for PAG. Exact fixed point calculation techniques and approximating fixed points via widenings are examined in detail. It is also discussed how to prove that an analyzer specification is correct and that the analyzer terminates.

Some interprocedural analysis techniques are described in Chap. 3.

Chapter 4 is devoted to a specialized procedure to analyze loops. In classical data flow analysis loops did not receive as much attention as procedures: all iterations are treated alike, although the first iterations usually behave very different from all other executions. Our approach extends the techniques for interprocedural analysis to the analysis of loops.

The main design criteria for the implementation of PAG are discussed in Chap. 5.

Chapter 6 explains the high level specification mechanism used in PAG. It is not meant as a reference manual for the functional language. Instead it will demonstrate the expressive power of the specification mechanism by explaining the main concepts. It shows a number of examples taken from real specifications.

The implementation of PAG and the working of the generated analyzers are explained in Chap. 7. The description focuses on those parts which contain uncommon solutions or are of special interest.

The influence of various implementations and optimizations is shown in Chap. 8. It contains a practical evaluation of applying a set of generated analyzers to sets of input programs, where –for some analyzers– the programs are taken from real applications and are of reasonable size.

Chapter 9 gives an overview of the related work. Chapter 10 and Chap. 11 discuss the results of this work, and give an outlook to the future of PAG.

Chapter 2

Theoretical Background

2.1 Program Analysis and Abstract Interpretation

Program analysis is a widely used technique to determine runtime properties of a given program automatically without actually executing it. A program analyzer takes a program as input and computes some interesting properties. Most of these properties are undecidable. Hence, correctness and completeness of the computed information are not achievable together. In program analysis there cannot be any compromise on the correctness side; the computed information has to be reliable. Thus it cannot guarantee completeness. The quality of the computed information, usually called its *precision*, should be as good as possible.

(Cousot and Cousot, 1977) describe a general framework for static program analyses called *abstract interpretation*. It is semantics based, therefore it supports correctness proofs of program analyses. Abstract interpretation amounts to perform a program's computation using *value descriptions* or *abstract values* in place of concrete values.

One reason for using abstract values instead of concrete ones is to transform uncomputable problems to computable ones. In other cases computable problems are transformed into more efficiently computable problems. Another reason is to deduce facts that hold on all possible paths and different inputs.[1]

Examples of abstract interpretations from mathematics and everyday life are "computation with residues", "casting out of nines", and the "sign rules" (Wilhelm and Maurer, 1997).

There is a natural analogy described in (Jones and Nielson, 1995): abstract interpretation is to formal semantics as numerical analysis is to mathematical analysis. Problems without any known analytic solution can be solved numerically, giving approximate solutions, for example a numerical result r and an error estimate ϵ. The solution is acceptable for practical usage if ϵ is small enough.

[1] Also existence properties are statements about the existence of a path in the set of all possible paths.

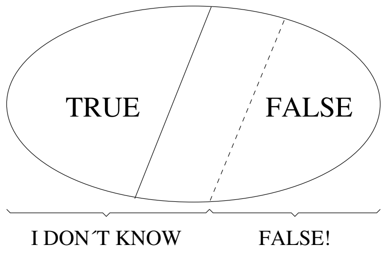


Figure 2.1: Approximation: erring on the safe side

An approximate program analysis is *safe*, if its results can always be depended on. So it depends on the usage of the analysis results on which side imprecisions are allowed. Let us regard the question: "Are two statements in a program dependent", with the goal to reorder them in an optimizing compiler. The program analysis may result in the answer "Yes" even if this is not true, because then the compiler would not reorder the statements. It would be unacceptable however, if the analysis result was "No" in case that they were dependent. Then the compiler would transform the program in a wrong way. Usually, the answers of the program analysis will be renamed in this case to "Yes?" (maybe yes) and "No!" (definitely no) to make clear on which side imprecisions may occur (see Fig. 2.1).

The number of "false" answers is a measure for the precision of the analysis. Often it can be improved at higher computational costs.

A detailed and complete overview of program analysis is given in (Nielson et al., 1998).

In PAG *data flow analysis* is used as a special case of abstract interpretation. Nevertheless some advantages are taken from the general theory of abstract interpretation e.g. the strong relation to the original semantics and fixed point approximation techniques like widening and narrowing.

2.2 Data Flow Analysis

In the following only those aspects of data flow analysis will be discussed that are important to PAG. A more complete discussion of data flow frameworks and their characterizing properties can be found in (Marlowe and Ryder, 1990).

Among the most typical applications of data flow analysis is the computation of properties for every program point in a given input program, where properties are described by abstract values in a certain domain D. The program points[2] of interest are the points directly before and directly after the execution of a statement.

The basis of data flow analysis is the control flow graph (CFG). It represents the program to be

[2]This work is focused on imperative programming languages. Nevertheless most things are applicable to logic and (imperative) object oriented programming languages as well, and to some extent also to functional languages.

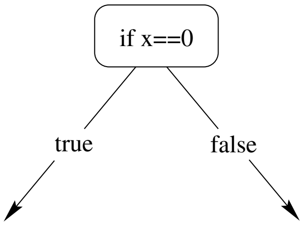


Figure 2.2: An example for using edge types

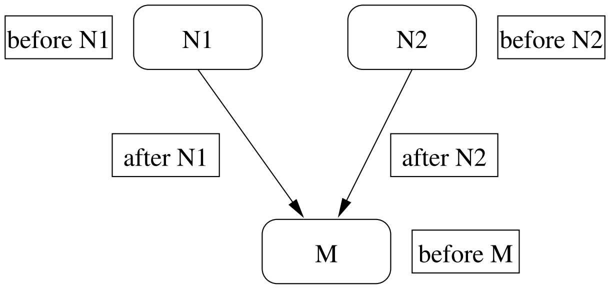


Figure 2.3: The values being valid directly before the execution of a node N are placed beside the node N (or omitted). The values being valid directly after the execution of N are placed beside the edge leaving N.

analyzed: for each basic operation such as the execution of a statement or the evaluation of a condition, there is a node in the CFG, and every edge represents a possible transfer of control during the execution. In building the CFG there is a choice in the way the statements are mapped to nodes. This can be done either on a fine or on a more coarse level. E.g. if the language to be analyzed contains short circuit evaluation of boolean expressions the evaluation of the expression "a and b" can either be modeled as one node or as two nodes: one for the evaluation of a and an optional one for the evaluation of b. Which way will be chosen depends on the analysis to be performed. The result of the analysis is a labeling of the CFG with abstract values from D that are valid directly before and directly after the operations corresponding to the nodes.

To compute these values, to every edge in the CFG a function $f : D \to D$ is assigned which describes the modification of the abstract values, whenever the control flows along this edge. These functions are called *transfer functions*.

Transfer functions are assigned to the edges of the CFG and not to nodes since for some analyses it can be useful to have different abstract functions for the different edges leaving a node. For example in the constant propagation (see Chap. 8) at a node `if x==0` (see Fig. 2.2) it can be concluded that `x` equals zero for the `true` edge, but not for the `false` edge.

The abstract value corresponding to the program point directly before the operation of a node n is assigned to the source of the edges leaving n. Since this value is the same for all edges leaving

n one can also think of assigning it to the node n itself. As there can be several transfer functions for a node n there can be also several values directly after the execution of n. These values are assigned to the end of the edges leaving n.

If the abstract values directly after the execution of the nodes are known, the values directly before the execution of the successor nodes can be calculated. Therefore, they will be omitted in the following presentation when they are not explicitly needed (see Fig. 2.3).

2.3 Basic Definitions

Definition 2.1 (Control Flow Graph)
A control flow graph *(CFG) is a graph* $G = (N, E, s, e)$*, with a finite set* N *of* nodes*, a set* $E \subseteq N \times N$ *of* edges*, a* start node s *and an* end node e*, where* $s, e \in N$*. If* $(n, m) \in E$*,* n *is called predecessor of* m *(*m *is successor of* n*).* s *is required to have no predecessor,* e *to have no successor.*

Additionally, it is required for the rest of this work that CFGs are *connected*, i.e. there are no nodes that cannot be reached from s, and that e is reachable from any node. An example for the CFG of a program is given in Fig. 2.4.

Definition 2.2 (Path)
A path π *from node* n_1 *to node* n_k *in a CFG* (N, E, s, e) *is a sequence of edges, beginning with a node* n_1 *and ending with some node* n_k *:* $\pi = (n_1, n_2), (n_2, n_3), \ldots, (n_{k-1}, n_k)$ *with* $(n_i, n_{i+1}) \in E$*.*

For PAG it is required that the set of abstract values forms a complete lattice. (In some cases it is possible to relax this prerequisite.)

Definition 2.3 (Complete Lattice)
A partially ordered set $D = (A, \sqsubseteq)$ *is called* complete lattice*, iff every subset of* A *has a greatest lower bound (*$\bigsqcap$*) and a least upper bound (*$\bigsqcup$*). The elements* $\bot = \bigsqcap A$*,* $\top = \bigsqcup A$ *are called* bottom *and* top *elements of* D*. The application of the greatest lower bound to two elements* $\bigsqcap\{a, b\}$ *is written as* $a \sqcap b$*. Similarly,* $\bigsqcup\{a, b\}$ *is written as* $a \sqcup b$*.*

Definition 2.4 (Chain)
A (ascending) chain $(x_i)_i$ *is a sequence* $x_0, x_1, \ldots,$ *such that* $\forall j : x_j \sqsubseteq x_{j+1}$*. A chain* $(x_i)_i$ *is called* strongly ascending*, iff* $\forall j : x_j \neq x_{j+1}$*. A chain* $(x_i)_i$ eventually stabilizes *iff* $\exists j : \forall n \geq j : x_j = x_n$*. Similar definitions apply to descending chains.*

The local (abstract) semantics of a CFG is expressed by *transfer functions* which assign a meaning to every edge of the CFG: *tf* : $E \to D \to D$. Depending on the properties of these functions the data flow problem can be characterized in several ways.

```
int fac(int n)
{
  int res;

  res=1;
  while (n>1) {
    res=res*n;
    n=n-1;
  }
  return res;
}
```

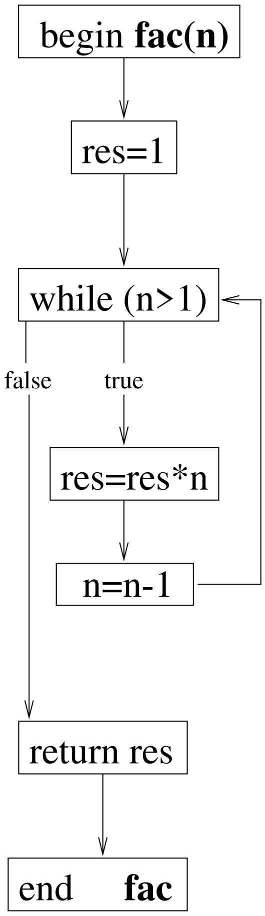


Figure 2.4: A program and its CFG

Definition 2.5 (monotone, distributive)
A function $f : D \to D$ *is called* monotone*, iff* $\forall d, d' \in D : d \sqsubseteq d' \Rightarrow f(d) \sqsubseteq f(d')$*.* f *is called* distributive*, iff* $\forall d, d' \in D : f(d \sqcup d') = f(d) \sqcup f(d')$*.*

To define the (abstract) semantics of a program first the semantics of a path has to be defined. The semantics of a path π represents the semantics of a program execution along that path without the effect of the statement in the last node.

Definition 2.6 (Path Semantics)
The path semantics $[\![\ \pi\ ]\!]_{tf}$ *of a path* π *is a composition of the transfer functions* $tf : E \to D \to D$ *along the path:*

$$\begin{array}{lcl} [\![\ \epsilon\ ]\!]_{tf} & = & id_{D \to D} \\ [\![\ e_1, \ldots, e_n\ ]\!]_{tf} & = & [\![\ e_2, \ldots, e_n\ ]\!]_{tf} \circ tf(e_1) \end{array}$$

The solution of the data flow problem is the least upper bound of all path semantics, applied to an initialization element $\iota \in D$.

Definition 2.7 (Merge over all paths solution)

$$\mathit{MOP}(n) = \bigsqcup\{\, [\![\, \pi \,]\!]_{\mathit{tf}}(\iota) \mid \pi \textit{ is path from } s \textit{ to } n\}$$

I.e. all possible executions of a program reaching node n are combined with $\bigsqcup$. So *MOP*(n) is the abstract value valid directly before the execution of n. The combination of information ensures that the information is valid for all possible program executions, and is therefore independent of the input of the program.

Unfortunately, the *MOP* solution is not always computable. So one uses the 'minimal fixed point' solution *MFP*, which is computable if every ascending chain of the lattice D eventually stabilizes, and if *tf*(e) is monotone for all e in E.

Definition 2.8 (Minimal fixed point solution)
MFP(n) *is the least fixed point of the equation system*

$$\mathit{MFP}(n) = \begin{cases} \iota & \textit{if } n = s \\ \bigsqcup\{\mathit{tf}(e)(\mathit{MFP}(n')) \mid e = (n', n) \in E\} & \textit{otherwise} \end{cases}$$

For the existence of the least fixed point see below.

If *tf*(e) is monotone for all e in E, and all ascending chains in D eventually stabilize, then, according to the coincidence theorem (Kam and Ullman, 1977; Knoop and Steffen, 1992) it holds that $\forall n : \mathit{MOP}(n) \sqsubseteq \mathit{MFP}(n)$, i.e. *MFP*$(n)$ is an approximation to the above described *MOP* solution.

If the functions *tf*(e) are even distributive, then

$$\forall n : \mathit{MFP}(n) = \mathit{MOP}(n)^3$$

All problems considered here are formulated such that $a \sqsubseteq b$ means b approximates a. This can always be achieved by flipping the ordering around. So erring on the safe side means getting larger in terms of the lattice ordering. Thus, the *MFP* solution is a safe approximation of the *MOP* solution.

2.4 Computing the *MFP* Solution

To compute the solution for the system of recursive equations (one for each node) from Def. 2.8 a fixed point computation is necessary. First of all *Tarskis Fixed Point Theorem* ensures that the least fixed point for the equation system exists.

[3] The *MOP* solution is computable in this restricted case.

Theorem 2.1 (Tarskis Fixed Point Theorem)
Let $(L, \sqsubseteq)$ be a complete lattice. If $F : L \to L$ is a monotone function, then the set of all fixed points of F $Fix(F) = \{l \mid F(l) = l\}$ is nonempty and forms a complete lattice when ordered by $\sqsubseteq$.

This theorem ensures that the minimal fixed point of the equation system of Def. 2.8 exists. This can be seen by choosing $L = D^n$ where D is the lattice of abstract values and n is the number of nodes in the CFG, and choosing F as applying all equations from the definition at once.

Kleenes Fixed Point Theorem gives a way to compute the least fixed point of the equation system in an iterative fashion.

Theorem 2.2 (Kleenes Fixed Point Theorem)
Let $(L, \sqsubseteq)$ be a complete lattice, where all chains eventually stabilize. If $F : L \to L$ is a monotone function, then there exists a k such that $F^k(\bot) = F^{k+1}(\bot)$ and $F^k(\bot)$ is the least fixed point of F.

This ensures that an iterative computation computing $F^k(\bot)$ will eventually stabilize, and result in the minimal fixed point of F. An implementation of this fixed point iteration is usually more efficient if the transfer functions are not applied in parallel.

There are several classes of algorithms to compute fixed points. One class are the iterative computation techniques, as they are suggested by Kleenes theorem. These algorithms are used in PAG. But there are other techniques as well, such as the elimination techniques which are not so easily applicable to the general problem formulation. A number of iterative computation techniques are discussed in (Fecht, 1997).

Figure 2.5 presents an optimized *workset algorithm* to compute the *MFP* solution. To every node n, a value $v(n) \in D$ is assigned, which is supposed to contain *MFP*(n) after the computation. Initially, $v(n)$ is set to $\bot$ for all $n \neq s$, and $v(s) = \iota$. A node n will be contained in the workset if for one of its predecessors m the value $v(m)$ has changed since the last computation of $v(n)$. In each iteration a node from the workset is selected and a new value for n is computed. Then it is checked, whether this value has changed since the last computation. If it has changed, the value is propagated again to all successors of n, i.e. these are entered into the workset. The iteration ends when no more nodes are contained in the workset. In this stable state $v(n) =$ *MFP*(n) holds for all n.

The algorithm only works correctly if it is assumed that $\bot$ is never the result of any transfer function except if the argument is $\bot$ and that $\iota \neq \bot$. Only then it can be guaranteed that every node that has a non trivial effect is visited at least once, and that the fixed point is reached. This means that $\bot$ has only the interpretation "node not reached". If this is not the case like in the live variable problem (see Example 6.1), where $\bot = \emptyset$ stands for "no live variable", an additional artificial least element $\bot$ should be used i.e. the lattice has to be lifted (see Sec. 6.2). For the rest of the work it will be assumed that $\forall e \in E, d \in D : d \neq \bot \Rightarrow tf(e)(d) \neq \bot$ and $\iota \neq \bot$.

Input: a CFG $G = (N, E, s, e)$, a complete lattice D, transfer functions $tf : E \to D \to D$, and an initial value $\iota \in D$.

Output: the annotation $v : N \to D$.

```
(* Init *)
FORALL n ∈ N DO
    v(n) := ⊥;
OD
v(s) := ι;
workset := {n | (s,n) ∈ E};

(* Iteration *)
WHILE workset ≠ ∅ DO
  LET n ∈ workset IN (* select a node from the workset *)
    workset := workset \ {n};
    X := ⨆{tf(e)(v(m)) | e = (m,n) ∈ E};                    (1)
    IF X ≠ v(n) THEN
      v(n) := X;
      workset := workset ∪ {m | (n,m) ∈ E};
    FI
OD.
```

Figure 2.5: Workset iteration

The algorithm leaves open the selection of the actual node from the workset. A widely used technique is to replace the set by a priority queue, and to assign each node a static priority. Different node orderings are discussed in Sec. 2.6.

For some data flow problems it is required that the direction of the CFG (N, E, s, e) has to be reversed. These problems are called *backward problems*. In the resulting CFG (N, E^{-1}, e, s) all paths start at the exit node, and all algorithms can be applied without further changes.

Other classes of data flow problems are usually formulated as being interested in the greatest fixed point. There are two possible solutions to this. From the theoretical point of view all these problems can be reformulated in a lattice where the order of the elements is reversed. In this *dual lattice* all computations can be done as described above and the least fixed point is calculated. From a practical point of view (of the specifier of the analysis) it is not necessary to change the problem formulation, but to change the computation algorithm. This can be done by replacing $\perp$ with $\top$, $\sqcup$ with $\sqcap$ and so on. Then, of course all descending chains of the lattice need to stabilize eventually, but nothing changes on the principles. In PAG both ways can be chosen, but here, for the sake of simplicity, the theoretical discussion will be restricted to calculating least fixed points.

2.5 Speeding up the Computation

Several techniques can be used to speed up the computation of the *MFP* solution. Here the discussion will be restricted to the techniques used in PAG.

2.5.1 Edge Classes

Since in most cases the transfer functions for different edges leaving a node will be the same, classes of edges which are called *edge types* are introduced: every edge belongs to a certain class like `true` edges and `false` edges leaving an `if` node. Edges for which no special edge class is mentioned belong to the class `normal`. A transfer function for an edge (n, m) is determined by the source node n of the edge and by its edge type.

So it is possible to save space and computation time by applying the same transfer function for all outgoing edges of a node that belong to the same edge class. Another advantage is the reduction of specification size for the transfer functions.

2.5.2 Optimized Workset Iteration

This technique avoids recalculation of the information of all incoming edges when only one of them has changed. This algorithm only computes the value for the incoming edge that has changed. To obtain the new value for node n the lub between the old value and the newly calculated value is built. In order to see this assume (without loss of generality), that a node n has k incoming edges $e_1 = (m_1, n), \ldots, e_k = (m_k, n)$ and since the last computation of the i-th iterant $v^i(n)$ for n only the value $v^{i+1}(m_1)$ has changed. The computation of $v^{i+1}(n)$ from the workset algorithm in Fig. 2.5 can be rephrased as follows:

$$
\begin{array}{rl}
v^{i+1}(n) & \text{applying the definition of } v^{i+1}(n) \\
= & \bigsqcup \{ \mathit{tf}(e)\,(v^{i+1}\,(m)) \mid e = (m, n) \in E \} \\
 & \text{unfolding according to the assumption} \\
= & \mathit{tf}(e_1)\,(v^{i+1}\,(m_1)) \sqcup \bigsqcup_{2 \leq j \leq k} \mathit{tf}(e_j)\,(v^{i+1}\,(m_j)) \\
 & \text{replacing } v^{i+1}(m_j) \text{ by } v^i(m_j) \text{ for } 2 \leq j \leq k \\
= & \mathit{tf}(e_1)\,(v^{i+1}\,(m_1)) \sqcup \bigsqcup_{2 \leq j \leq k} \mathit{tf}(e_j)\,(v^{i}\,(m_j)) \\
 & \text{since } \mathit{tf}(e_1) \text{ is monotone and } v^i(m_1) \sqsubseteq v^{i+1}(m_1) \\
= & \mathit{tf}(e_1)\,(v^{i+1}\,(m_1)) \sqcup \mathit{tf}(e_1)\,(v^{i}\,(m_1)) \sqcup \bigsqcup_{2 \leq j \leq k} \mathit{tf}(e_j)\,(v^{i}\,(m_j)) \\
 & \text{folding according to the assumption} \\
= & \mathit{tf}(e_1)\,(v^{i+1}\,(m_1)) \sqcup \bigsqcup \{ \mathit{tf}(e)\,(v^{i}\,(m)) \mid e = (m, n) \in E \} \\
 & \text{applying the definition of } v^i(n) \\
= & \mathit{tf}(e_1)\,(v^{i+1}\,(m_1)) \sqcup v^i(n)
\end{array}
$$

Input: a CFG $G = (N, E, s, e)$, a complete lattice D, transfer functions $tf : E \to D \to D$, and an initial value $\iota \in D$.

Output: the annotation $v : N \to D$.

```
(* Init *)
FORALL n ∈ N DO
    v(n) := ⊥;
OD
v(s) := ι;
workset := {(m, n) ∈ E | m = s};

(* Iteration *)
WHILE workset ≠ ∅ DO
  LET e = (n, m) ∈ workset IN
    workset := workset \ {e};
    X := tf(e)(v(n)) ⊔ v(m);
    IF X ≠ v(m) THEN
      v(m) := X;
      workset := workset ∪ {(k, l) ∈ E | k = m};
    FI
OD.
```

Figure 2.6: Optimized workset iteration

In the algorithm (see Fig. 2.6) the workset no longer contains nodes but edges, for which a recalculation has to be done.

2.5.3 Widening and Narrowing

Another way to speed up the calculation was proposed by (Cousot and Cousot, 1992). In order to compute a minimal fixed point in the iteration process it is always safe to replace a value by a larger one (according to the lattice ordering). Doing so one can shorten the ascending chains encountered in the iteration and speed up the computation. The new iteration process may overshoot the least fixed point. But any solution above the least fixed point is a safe approximation of the least fixed point. By using this technique, called *widening*, it is even possible to deal with domains that have infinite ascending chains: it must be guaranteed that only finite chains can occur during the iteration. To improve the results obtained by widening, the Cousots suggest to use a second iteration process called *narrowing*. In this iteration it must be guaranteed that the sequence does not jump below the least fixed point.

Definition 2.9 (Widening)
A widening *is an operator* $\nabla : D \times D \to D$, *such that:*

$$\forall x, y \in D : x \sqsubseteq x \nabla y \text{ and } y \sqsubseteq x \nabla y$$

and for every ascending chain $(x_i)_i$ *the ascending chain* $(y_i)_i$, *defined by*

$$y_0 = x_0 \text{ and } y_{i+1} = y_i \nabla x_{i+1},$$

eventually stabilizes.

The upward iteration sequence with widening for a function $F : D \to D$ is defined as:

$$\begin{array}{lcl} X^0 & = & \bot \\ X^{i+1} & = & X^i \nabla F(X^i) \end{array}$$

Definition 2.10 (Narrowing)
A narrowing *is an operator* $\Delta : D \times D \to D$, *such that:*

$$\forall x, y \in D : (y \sqsubseteq x) \Rightarrow (y \sqsubseteq (x \,\Delta\, y) \sqsubseteq x)$$

and for every descending chain $(x_i)_i$ *the descending chain* $(y_i)_i$ *defined by*

$$y_0 = x_0 \text{ and } y_{i+1} = y_i \,\Delta\, x_{i+1}$$

eventually stabilizes.

The downward iteration sequence with narrowing for a function $F : D \to D$ is defined as:

$$\begin{array}{lcl} X^0 & = & \text{limit of the upward iteration} \\ X^{i+1} & = & X^i \,\Delta\, F(X^i) \end{array}$$

The widening operator ∇ on D can be lifted to a widening operator ∇^N on functions from a finite set of nodes N to D by defining $\nabla^N(f, g) = \lambda n.\nabla(f(n), g(n))$. The narrowing Δ can be extended similarly to Δ^N.

To obtain an algorithm using a widening ∇ after the computation of X in line (1) of the workset algorithm in Fig. 2.5 the statement $X := v(n) \nabla X$ has to be inserted. Widening can also be used for the optimized workset algorithm from Fig. 2.6, but may deliver more imprecise results as the widening operator is applied more often than necessary.

To improve the computed results with a narrowing Δ a second iteration process shown in Fig. 2.7 has to be appended to the workset algorithm of Fig. 2.5.

The optimization from Sec. 2.5.2 forces any sequence to become an increasing sequence by joining the new value to the one from the last iteration. But since a sequence with narrowing is a downward sequence the optimization is not applicable for the narrowing process.

```
(* Init *)
workset := {n | n ∈ N};

(* Iteration *)
WHILE workset ≠ ∅ DO
  LET n ∈ workset IN (* select a node from the workset *)
    workset := workset \ {n};
    X := ⨆{tf(e)(v(m)) | e = (m, n) ∈ E};
    X := v(n) Δ X;
    IF X ≠ v(n) THEN
      v(n) := X;
      workset := workset ∪ {m | (n, m) ∈ E};
    FI
OD.
```

Figure 2.7: The narrowing process to be used after the widening process

2.5.4 Basic Blocks

Another important way to speed up the computation and to save space is to use basic blocks instead of single instructions in the CFG.

Definition 2.11 (Basic block)
A basic block *is a sequence of nodes* $(n_1, \ldots, n_k)$ *of a CFG* $= (N, E, s, e)$ *such that* n_i *is the only predecessor of* n_{i+1} *and* n_{i+1} *is the only successor of* n_i*.*

Using a new CFG, where the nodes are basic blocks for the iteration process, one can speed up the computation, but it mainly saves space, because the abstract domain values are only stored for each basic block. This concept is extended to even larger blocks in PAG: here each node in a basic block has not necessarily got only one predecessor. It is sufficient to consider connected sets of nodes as extended basic blocks where each node has only one successor (and can have more than one predecessor). Because the lub with the value from the last iteration is calculated at the end of the extended basic block according to the optimization in Sec. 2.5.2 it is not necessary to consider edges leading to the middle of the extended basic block. For non distributive transfer functions this may deliver even better results than the values obtained from the original strategy, because the lossy calculation of the lub is deferred.

This extended basic block strategy can be described as duplicating the nodes in the basic block, such that each path through the basic block gets a unique copy of each node (see Fig. 2.8). By applying the basic block algorithm to this transformed control flow graph the same results as with the extended basic block strategy can be achieved. The advantage of the extended basic block strategy is that no transformation to the CFG has to be applied. The algorithm is shown in Fig. 2.9 and Fig. 2.10.

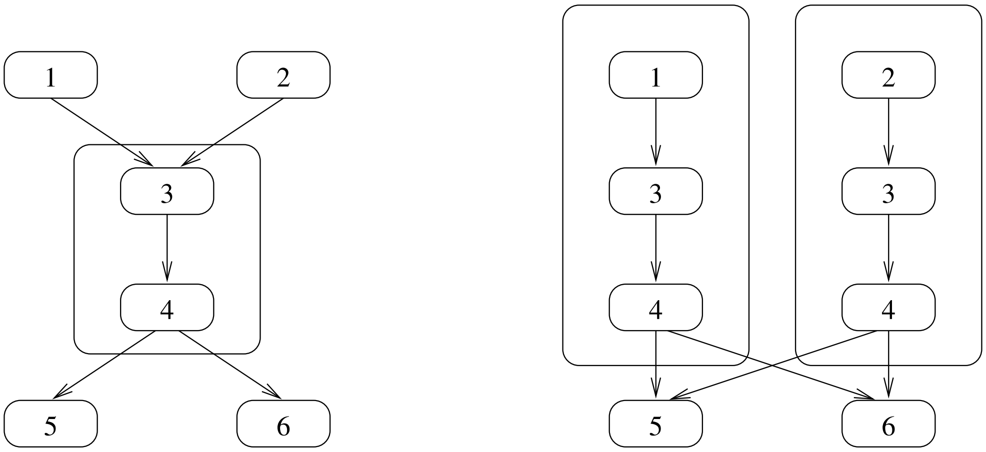


Figure 2.8: On the left side there is a CFG with one basic block and on the right there is the 'virtually' extended CFG with two extended basic blocks

2.6 Node Orderings

As already mentioned, the selection of a node from the workset can be performed according to priorities of the nodes. PAG supports the following static orderings:

dfs : depth first (pre)order. Nodes are ordered in a *depth first* visiting strategy of the CFG.

bfs : breadth first order. Nodes are ordered in a *breadth first* visiting strategy of the CFG.

scc$_d$: strongly connected components. This ordering first computes the sccs of the CFG. These are ordered topologically. The nodes within each scc are ordered depth first. This corresponds to the iteration method proposed in (Horwitz et al., 1987), except that PAG uses only one queue, so that minor differences can occur in the visiting order inside strongly connected components. It simplifies the runtime effort to be done during the iteration process.

scc$_b$: strongly connected components. This ordering is similar to **scc**$_d$, but it orders the nodes within the sccs breadth first.

wto$_d$: this ordering computes the sccs of the CFG, but orders them by a weak topological ordering of (Bourdoncle, 1993). The nodes of the sccs are ordered depth first.

wto$_b$: this ordering differs from **wto**$_d$ only by sorting the nodes of the sccs breadth first.

Input: a CFG $G = (N, E, s, e)$, a complete lattice D, transfer functions $tf : E \to D \to D$, and an initial value $\iota \in D$.

Output: the annotation $v : N \to D$.

Note: information is stored only at nodes with more than one successor in the main iteration process. The information for the other nodes is calculated in the post processing phase.

```
(* hmtos (has more than one successor):  determines if the argument node
   n has more than one successor *)
FUNCTION hmtos(n ∈ N):bool
  hmtos:= |{m | (n,m) ∈ E}| ≠ 1;
END

(* propagate the new contribution y through the extended basic block
   starting with n *)
PROCEDURE propagate(n ∈ N, y ∈ D)
  WHILE NOT hmtos(n) DO
    LET e = (n,m) ∈ E IN
        (* uniquely determined because NOT hmtos(n) *)
      y := tf(e)(y);
      n := m;
  OD
  y := y ⊔ v(n);
  IF y ≠ v(n) THEN
    v(n) := y;
    workset := workset ∪ {(k,l) ∈ E | k = n};
  FI
END

PROCEDURE init
  FORALL n ∈ N DO
    IF hmtos(n) THEN
      v(n) := ⊥;
    FI
  OD
  propagate(s, ι);
END
```

Figure 2.9: Workset iteration with extended basic block optimization - part 1

```
PROCEDURE calculate
  WHILE workset ≠ ∅ DO
    LET e = (n, m) ∈ workset IN
      workset := workset \ {e};
      propagate(m, tf(e)(v(n)));
  OD
END

PROCEDURE post
  FORALL n ∈ N DO
    if hmtos(n) THEN
      workset := workset ∪ {(k, l) ∈ E | k = n};
    ELSE
      v(n) := ⊥;
    FI
  OD
  WHILE workset ≠ ∅ DO
    LET e = (n, m) ∈ workset IN
      workset := workset \ {e};
      X := tf(e)(v(n)) ⊔ v(m);
      IF X ≠ v(m) THEN
        v(m) := X;
        workset := workset ∪ {(k, l) ∈ E | k = m AND NOT hmtos(l)};
      FI
  OD
END

(* main *)
  init();
  calculate();
  post();
END.
```

Figure 2.10: Workset iteration with extended basic block optimization - part 2

2.7 Correctness Proofs

Two important facts can be proven about program analyzers: termination and correctness. This section shows the framework for these proofs and concludes with the proof obligations for the designer of a program analysis. So a PAG user should prove these facts and can conclude that the program analysis terminates and is correct.

For the proof of the termination the following is required:

- the used abstract domain forms a complete lattice
- the lattice has either only finite chains, or a widening according to Def. 2.9 is used.
- if a narrowing is used it has to fulfill the conditions from Def. 2.10.

There are several ways to prove the correctness of a program analysis, but here no complete overview is given. Instead, the presentation will be restricted to the Galois connection approach of (Cousot and Cousot, 1977), which is restricted in the sense that it only applies to analyses where properties directly describe sets of values. In the literature, the term *first-order analyses* has been used for these analyses.

A correctness proof of a program analysis has to refer to the semantics of programs. Therefore, a form of the semantics is considered that represents a program as a CFG and the meaning of program statements by a function $f : E \to D_{conc} \to D_{conc}$ over a concrete domain D_{conc}[4].

Since program analysis calculates information for all possible input data, the *collecting semantics*[5] is considered. This associates with each program point the set of all states that can ever occur when program control reaches that point.

The collecting semantics *Coll*: $N \to D_{coll}$ maps program points to sets of states $D_{coll} = 2^{D_{conc}}$. Let *Init* $\subseteq D_{conc}$ be the set of all possible initial program states, i.e., all program states that may occur before the execution of s. Then

$$Coll(n) = \bigcup_{i \in Init} \bigcup \{ [\![\pi]\!]_f(i) \mid \pi \text{ is a path from } s \text{ to } n\}$$

D_{coll} forms a complete lattice with set inclusion $\subseteq$ as partial ordering.

The relation between the collecting semantics and the abstract semantics is described by a pair of functions *abs* and *conc*. The intuition is that the elements of the abstract domain D are descriptions of sets of values from the concrete domain.

$$conc : D \to D_{coll}$$

$$abs : D_{coll} \to D$$

[4]In this scenario $\perp \in D_{conc}$ has to be propagated to non reachable conditional branches.

[5]In (Cousot and Cousot, 1977), the term *static semantics* is used.

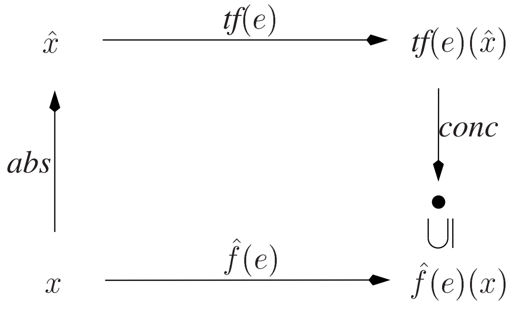


Figure 2.11: Concrete and abstract analysis domains.

Let $\hat{f} : E \to D_{coll} \to D_{coll}$ denote the extension of f to D_{coll}. The relation between $\hat{f}$ and *tf* should be described by local consistency (see Fig.2.11).

Definition 2.12 (Local consistency)
$\hat{f}$ and tf are locally consistent, *if*

$$\forall x \in D_{coll} : \forall e \in E : \hat{f}(e)(x) \subseteq conc(tf(e)(abs(x))).$$

For our purposes *abs* and *conc* should form a Galois connection.

Definition 2.13 (Galois Connection)
$(D_{coll}, abs, conc, D)$ *is a* Galois connection *for a collecting semantics D_{coll}, if*

1. *D forms a complete lattice*
2. $\forall x \in D_{coll} : x \subseteq conc(abs(x))$
3. $\forall x \in D : abs(conc(x)) \sqsubseteq x$
4. *abs and conc are monotone.*

A program analysis is correct, if any state that can be reached at a program point at runtime is predicted by the analysis. Due to undecidability problems, an abstract interpretation usually cannot be exact: the analysis will predict some states that do not occur at runtime, i.e., it is conservative.

From (Cousot and Cousot, 1977) follows for our framework that a program analysis is correct if it satisfies the following conditions:

1. the abstract domain D forms a complete lattice
2. $(D_{coll}, abs, conc, D)$ is a Galois connection; and

3. the transfer function *tf* is locally consistent with the concrete operation $\hat{f}$.
4. the widening and narrowing used in the analyzer fulfill the Def. 2.9 and Def. 2.10
5. the abstract initial value for the start node $\iota \in D$ describes all possible concrete init values *Init*. I.e. $abs(Init) \sqsubseteq \iota$.

Chapter 3

Interprocedural Solutions

Up to now only programs with one procedure (the main procedure) have been considered. A typical program however consists of several procedures. The usage of several procedures which may be called from different places in the program with different arguments and different values for global variables increases the complexity of program analysis. The main point of the following three sections is that in general a procedure called more than once will have different *call contexts* for the different calls, which means that the calls will have different elements of the abstract domain calculated for them. Best results can be obtained when the procedure is analyzed separately for each call context. But this may not only increase the complexity of the analysis drastically, but may even lead to non termination, if a procedure is analyzed for an infinite number of different contexts. There are several approaches which attack the problem of non termination and complexity explosion in different ways.

Inlining: every call to a procedure is replaced by the body of that procedure. This is only feasible for nonrecursive procedures, and the CFG may grow exponentially in terms of the nesting depth.

Invalidating: this approach invalidates *all* data at every call to a procedure. This strategy, used for instance by the GNU C compiler is easy to implement but very imprecise. It can be improved by invalidating only the information which may be changed by the procedure call. This requires an extra analysis prior to the data flow analysis.

Effect calculation: for every procedure a function (effect) is calculated that maps input values of the procedure to output values. Then these functions are used to perform the analysis (see Sec. 3.2).

Call string: the 'history' of a procedure call is incorporated into the underlying lattice D, allowing to analyze different call sequences separately, see Sec. 3.3.

Static call graph: this strategy looks at some statically defined call sequences and performs the data flow analysis for each sequence separately. Sequences exceeding a certain length are mixed together and loose precision. See Sec. 3.4 for further details.

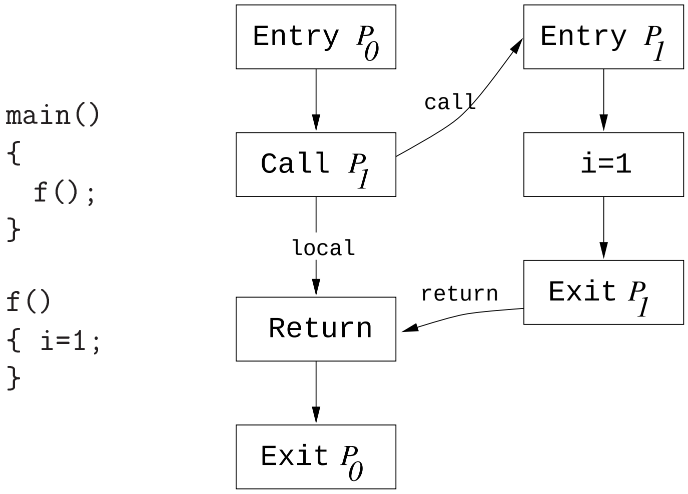


Figure 3.1: A program and its supergraph

PAG implements the effect calculation and the static call graph approach. With these two strategies all others can be simulated. First the call string approach and the effect calculation are described in detail. For a further discussion of these two strategies see (Sharir and Pnueli, 1981). Then the static call graph approach is described and it is shown how it can simulate the call string approach.

For an interprocedural analyzer it is usually necessary that the complete program source is available for the analysis. This can be seen in contrast to another goal in the area of compiler construction: separate compilation (modules). It is an open question in compiler research how to combine separate compilation and program analysis. PAG delivers best results by analyzing complete programs. But it offers the possibility to deal with references to unknown procedures such as calls of library routines. The solution is to invalidate all data flow information at the call of the unknown procedure. This behavior can be refined by the user of PAG by invalidating only those parts that are known to change, and by specifying this behavior for each library routine separately. E.g. the `printf` routine in C is known to change neither any global variable, nor any function parameter, nor any part of the heap of the program, and can be assumed to have the empty effect for most program analyses.

But nevertheless most optimizing transformations assume that there are no hidden calls to analyzed procedures which are not seen by the analyzer, since only then procedures can be optimized according to known call contexts.

3.1 Program Representation

In order to explain the interprocedural techniques, first the interprocedural control flow graph used in PAG has to be described.

The interprocedural CFG of a program is called *supergraph* and is constructed from the CFGs of the procedures:

Definition 3.1 (Supergraph)
Let $P_0, \ldots, P_n$ be the procedures of the program, where P_0 is the main procedure, and $G_0, \ldots, G_n$ the corresponding CFGs. The supergraph *of the program P, $G^* = (N^*, E^*, s^*, e^*)$ consists of a set of nodes N^* which is the union of all nodes of the G_i where each node representing a procedure call is replaced by 2 nodes:*

- *a* call *node* `Call` p_j,
- *a* return *node* `Return`.

The edge set E^* contains all the edges of the G_i. In addition E^* contains the following edges for each procedure call node 'call' *to a procedure P_j and the corresponding return node* 'return'*:*

- *a* call edge *from* 'call' *to the start node s_j of P_j*
- *a* return edge *from the exit node e_j of P_j to* 'return' *and*
- *a* local edge *from* 'call' *to* 'return'

The start node and the end node of G^* are the start and end nodes of G_0: $s^* = s_0, e^* = e_0$.

See Fig. 3.1 for an example.

The local edge is introduced to allow for the propagation of information which cannot be changed by the procedure call, e.g. values of local variables of the caller. This technique has been used many times in the literature – for instance in (Reps et al., 1995). It can also be used to simulate the return functions used in (Knoop et al., 1996) which correspond exactly to the concept of hyperedges in the analysis of logic programs (see e.g. (Fecht, 1997) for a detailed discussion). The desired behavior must be encoded into the transfer functions of the local and the return edges. The combination of the two data flow values is done by the least upper bound operator[1]. See Fig. 3.2 for an example situation. A constant propagation is shown, where the variable x is local to P_1, whereas y is global. Each of the two involved edges has to set those variables that should not be propagated along this edge to bottom, the neutral element of the least upper bound operator. So the local edge sets $y = \perp$, and the return edge from P_2 sets $x = \perp$.

3.2 Effect Calculation

Effect calculation analyzes each procedure once for each call context that arises during the computation. This can be implemented by storing the different call contexts for each procedure in a

[1]The function used to combine any two data flow values can be redefined in the specification of the analysis.

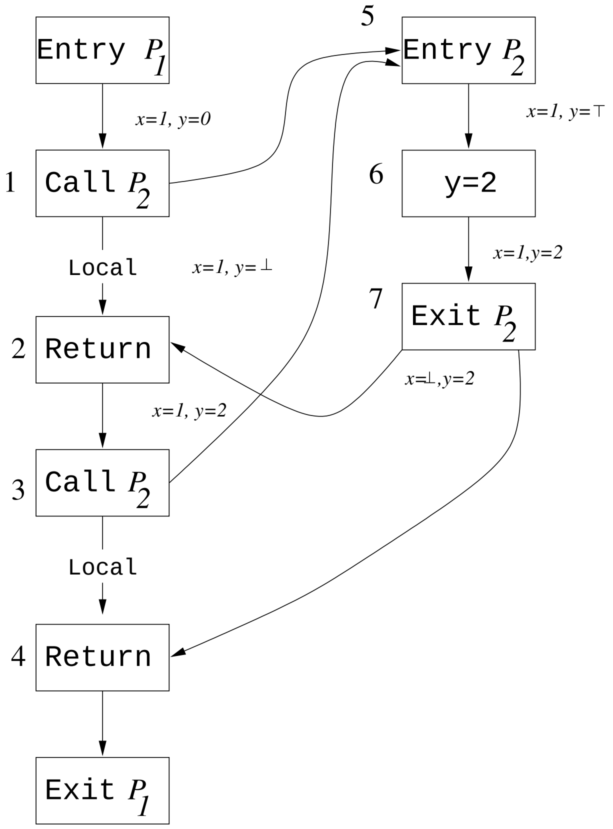


Figure 3.2: Local edge

table along with the corresponding abstract domain element for the exit of the procedure. The tables for the procedures can be seen as functions mapping incoming abstract values to outgoing abstract values. They represent abstract versions of the procedures[2].

The algorithm shown in Fig. 3.3 and Fig. 3.4 calculates the abstract function tables in a demand driven way. Each time the iteration algorithm reaches a call node it looks up the call context in the table of the called procedure. If an exit value is found, this is used as the result of the call. If no exit value is found its calculation is triggered. Since procedures can be (simultaneously) recursive this can trigger the calculation of other values. Because each call context is inserted only once in the workset infinite loops with the same call context for a procedure are avoided. E.g. if a procedure calls itself unconditionally with exactly the same call context the exit information for this procedure is $\perp$ which is interpreted as "not reachable". This is correct, as the procedure would loop forever.

Further conditions to guarantee termination must be imposed. For termination it is sufficient, if the abstract domain is finite since then the tables for every procedure cannot grow infinitely. No other good sufficient termination conditions are known to us, if the abstract functions are

[2] This is where the name of the approach comes from: calculating the effect of the procedures.

Input: a supergraph $G^* = (N^*, E^*, s^*, e^*)$, a complete lattice D, transfer functions *tf* : $E^* \to D \to D$, and an initial value $\iota \in D$.

Output: the function *MFP*: $N^* \to D$.

Note: the effect functions are represented by a two dimensional, partially defined array *PHI*: $N^* \times D \to D$, such that *PHI*(n, x) stores the value for the node n, where x is the calling context of the call of the enclosing procedure. The variable $workset$ contains pairs (n, x) of elements of $N^* \times D$, for which *PHI*(n, x) has changed and whose new values have not yet been propagated to the successors of n.

```
PROCEDURE init
  FORALL n ∈ N*, x ∈ D DO
   PHI(n, x) := ⊥;
  OD
  PHI(s*, ι) := ι;
  workset := {(s*, ι)};
END

(* propagates the new contribution y to the node n, where x is the
   calling context of the call of the enclosing procedure *)
PROCEDURE propagate(x ∈ D, y ∈ D, n ∈ N*)
 z := PHI(n, x) ⊔ y;
 IF z ≠ PHI(n, x) THEN
   PHI(n, x) := z;
   workset := workset ∪ {(n, x)};
 FI
END

PROCEDURE post
 FORALL n ∈ N* DO
   MFP(n) := ⨆{PHI(n, x) | x ∈ D};
 OD
END
```

Figure 3.3: Effect calculation algorithm - part 1

```
PROCEDURE effects
  WHILE workset ≠ ∅ DO
    LET (n,x) ∈ workset IN (* select a pair (n,x) from the workset *)
      workset := workset \ {(n,x)};
      y := PHI(n,x);
      IF n ≡ CALL P THEN
        z := PHI(EXIT P,y);
        IF z ≠ ⊥ THEN
          (* Pair (ENTRY P,y) already processed *)
          FORALL m successor of n DO
            propagate(x,z,m);
          OD
        ELSE
          propagate(y,y,ENTRY P);
        FI
      ELIF n ≡ Exit P THEN
        FORALL c = Call P DO
          FORALL u ∈ D WHERE PHI(c,u) = x DO
            FORALL m successor of c DO
              propagate(u,y,m);
            OD
          OD
        OD
      ELSE
        FORALL m successor of n DO
          propagate(x,tf(n,m)(y),m);
        OD
      FI
  OD
END

(* main *)
 init();
 effects();
 post();
END.
```

Figure 3.4: Effect calculation algorithm - part 2

tabulated. In other cases e.g. for bit vector problems one can do better.

A bottleneck of the algorithm of Fig. 3.3, and Fig. 3.4 (which is mainly the original algorithm from (Sharir and Pnueli, 1981)) is the case for $n \equiv \texttt{Exit}\ P$ in the procedure `effects`. There are three nested loops over all calls, any element of D and all successors of all call nodes to find those entries in the tables of the callers for which the actual information was calculated. This can be avoided by keeping additional tables for each procedure to keep track for which call nodes and elements from D the information is calculated.

3.3 Call String Approach

In the call string approach the calls are distinguished by their path through the dynamic call tree. The idea can be understood as simulating the call stack of an abstract machine which contains frames for each procedure call that has not yet finished.

Such an abstract stack can be expressed as a sequence of call sites of the program. Despite the fact that there is only a finite number of call sites in each program, there is an infinite number of sequences of them. Even if one takes only those call sequences into account that can possibly occur during the runtime of the program, their number is still infinite for a simple recursive program.

To overcome this infinity the idea is to limit the possibly unbounded call sequences Γ (or *call strings*) to a fixed length K. Γ^K is the set of all call strings with length at most K. If a new procedure invocation occurs, the call c is appended to the call string and if the resulting string is longer than K its K-suffix is taken. After finishing a procedure with a call string γc the analysis can only continue at the return node corresponding to c. c is deleted from the call string, but due to the limiting of call strings to length K all possible calls which call the procedure containing the first element in γ have to be prefixed to γ.

The shortcoming of K-limiting the call strings is that information may be taken into account during the analysis resulting from an overestimation of the set of paths that can occur during an actual program execution, and so the result can be imprecise.

Formally, the updating of the call strings is expressed by assigning each edge $(m, n) \in E^*$ an updating relation $R_{(m,n)}$ in Γ^K, which is not necessarily one-to-one. (Two or more call strings α can be mapped by a $R_{(m,n)}$ to the same resulting call string β.)

First a K-limiting append $\circ^K$ for call strings from Γ^K and call sites is defined as

$$c_1 c_2 \ldots c_n \circ^K c = \begin{cases} c_1 c_2 \ldots c_n c & \text{if } n < K \\ c_2 \ldots c_n c & \text{if } n \geq K \end{cases}$$

Then

$$\alpha\ R_{(m,n)}\ \beta \text{ iff } \begin{cases} \alpha = \beta & \text{if } (m,n) \text{ is an intraprocedural edge} \\ \beta = \alpha \circ^K m & \text{if } m \text{ is a call node} \\ \beta\ R_{(m',n')}\ \alpha & \text{if } (m,n) \text{ is a return edge and } (m',n') \text{ the corresponding call} \\ & \text{edge} \end{cases}$$

The last case expresses that when returning from a procedure the inverse of the call is applied.

The tracing of the call strings is done by encoding them into the abstract domain: if the original domain is D then a domain $D^* : \Gamma^K \to D$ is constructed. The transfer function $tf : E^* \to D \to D$ is extended to a function $tf^* : E^* \to D^* \to D^*$ by defining:

$$tf^*\,(m,n)\,(\chi)\,(\gamma) = \bigsqcup \{tf\,(m,n)\,(\chi\,(\gamma')) \mid \gamma'\ R_{(m,n)}\ \gamma\}$$

This approach has two advantages compared to the effect calculation approach: first, it is possible to deal with abstract domains of infinite cardinality. Secondly, it is easily possible to cut down the complexity of the analysis by selecting small values for K.

The disadvantages are: analyses using the call string approach can be less precise than those using the effect calculation approach. By encoding the call strings into the analysis domain the updating of the call strings has to be done during the analysis. Since the analysis is an iterative process the same $R_{(m,n)}$ has to be calculated several times. One can do better by calculating as much as possible prior to the analysis.

3.4 Static Call Graph Approach

In this section the way PAG implements the call string approach will be described. This technique was introduced in (Alt and Martin, 1995) and can also be used to implement more general interprocedural techniques, where it is statically determined (with the help of the call graph) which call contexts should be distinguished (see Chap. 4) .

To keep different call paths apart, each node in the supergraph is annotated with an array of data elements from D. A pair (n,i) of a node and an index is called a *location* and describes the i-th element in the array of data flow values belonging to n. The intuition is that each location corresponds to a call string which reaches the procedure. For each procedure P_i, the number of data elements at nodes of this procedure is fixed and called its *multiplicity* $\mathsf{mult}(P_i)$. We set $\mathsf{mult}(P_0) = 1$, i.e. there is only one data element in the main procedure. mult can be extended to nodes $\mathsf{mult} : N^* \to \mathbf{N}$ by defining $\mathsf{mult}(n) := \mathsf{mult}(P_i)$, iff n is a node in P_i.

A *connector* is a set of functions $\{\mathsf{con_c}\}_\mathsf{c}$ which describe for each call c how the locations of the calling procedure are connected to the locations of the called procedure. I.e. $\forall$ `c = Call` P_j in P_i: $\mathsf{con_c} : \{1,\ldots,\mathsf{mult}(P_i)\} \to \{1,\ldots,\mathsf{mult}(P_j)\}$. This leads to the definition of a new kind of CFG, where the nodes are locations. The locations are connected along the edges

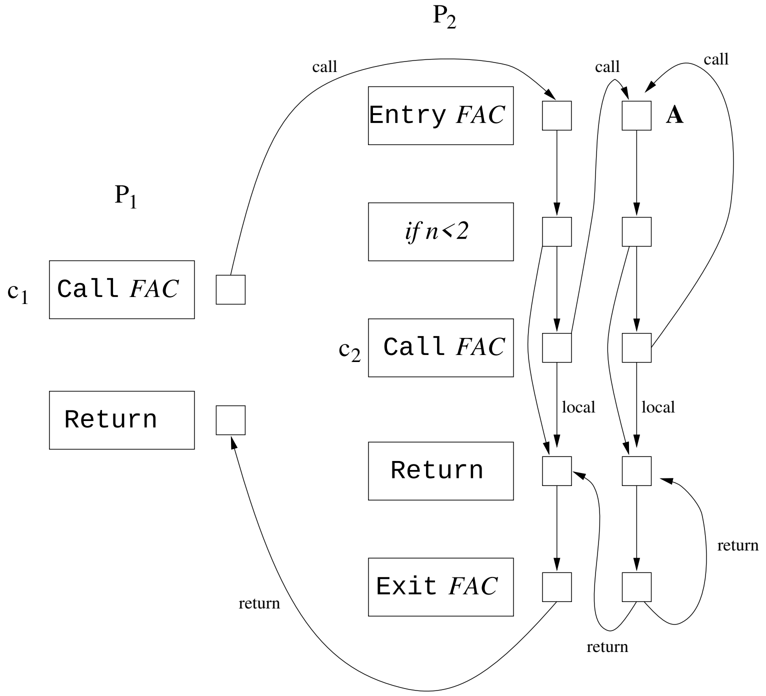


Figure 3.5: Connector example

of the supergraph. Within procedures the i-th location is connected with the i-th location of all successors (see Fig. 3.5). For all calls c the function con_c determines how the data flow elements of the calling procedure are connected to the elements of the called procedure. At the corresponding return edges the inverse relation of con_c is applied.

Definition 3.2 (Expanded Supergraph)
The expanded supergraph *for a supergraph $G^* = (N^*, E^*, s^*, e^*)$, a function* $\mathsf{mult} : N^* \to \mathbb{N}$ *and a connector $\{\mathsf{con}_\mathsf{c}\}_\mathsf{c}$ is defined as*

$G^*_E = (N^*_E, E^*_E, s^*_E, e^*_E)$*, with*

- $N^*_E = \{(n, i) \mid n \in N^*, i \in \{1, \ldots, \mathsf{mult}(n)\}\}$
- $s^*_E = (s^*, 1)$
- $e^*_E = (e^*, 1)$
- $((n_1, i_1), (n_2, i_2)) \in E^*_E$*, iff* $(n_1, n_2) \in E^*$ *and one of the following conditions holds:*

  i.) (n_1, n_2) *is a call edge and* $i_2 = \mathsf{con}_{n_1}(i_1)$

  ii.) (n_1, n_2) *is a return edge, c the corresponding call, and* $i_1 = \mathsf{con}_\mathsf{c}(i_2)$*,*

  iii.) $i_1 = i_2$ *otherwise*

Figure 3.5 is an example for an expanded supergraph with $\mathsf{mult}(P_1) = 1, \mathsf{mult}(P_2) = 2$, $\mathsf{con}_{c_1}(1) = 1$, $\mathsf{con}_{c_2}(1) = 2$, and $\mathsf{con}_{c_2}(2) = 2$.

3.4.1 Connectors

The connectors con_c and multiplicities $\mathsf{mult}(P_j)$ can be adjusted to perform different interprocedural analyses.

Simple Connector

$\forall n \in N^* : \mathsf{mult}(n) = 1, \forall$ Calls c: $\mathsf{con}_\mathsf{c} = id$. Only one location is used per supergraph node. No separation of different call paths is made: information is combined at the procedure entries. This is equivalent to call string zero.

Easy Connector

Set $\mathsf{mult}(P_i) = k_i$, where k_i is the number of incoming edges of `Entry` P_i, i.e. the number of static call sites to P_i. For all `c=Call` P_i in P_j and $\forall i \in \{1, \ldots, \mathsf{mult}(P_j)\}$ set $\mathsf{con}_\mathsf{c}(i) = x$, where $x \in \{1, \ldots, \mathsf{mult}(P_i)\}$ is fixed but unique for each call site of P_i.

This connector assigns a unique data element to every call of a procedure P. For call paths with more than one dynamic call from the same call site, information is mixed together for that procedure (e.g. at node **A** in Fig. 3.5). This connector is equivalent to call string one.

Full Call String Connector

The general call string K method can also be expressed by a connector. To do this the set of all K-limited call strings reaching a procedure p has to be calculated and is called $\mathit{ECS}_K(p)$. This is done under the assumption, that the *main* procedure is reached only by the empty call string ϵ. This calculation can be implemented as a recursive analysis over the supergraph shown in Fig. 3.6 and Fig. 3.7. The algorithm calculates the lists of valid call strings for each procedure, as well as the corresponding connectors. The multiplicity for a procedure is given by the length of the list of call strings. The algorithm leaves open the actual representation of the call strings. Usually the calls in the program will be represented as integers between 1 and M, where M is the number of calls in the program. Then a call string γ is a sequence of integers $c_n \ldots c_1 c_0$. It can be coded as an integer i if it is seen as an n-ary number to the base $M+1$: $i = \sum_{j=0}^{n} c_j * (M+1)^j$. But it turned out that for some programs (e.g. `bison` described in Chap. 8) these numbers i get too large for 64-bit integers even for $K = 3$. So in PAG call strings are implemented as lists of integers.

Input: a supergraph of a program.

Output: a list *ECS*(p) of valid call strings for each procedure p and the connector *CON* for each call c as set of argument/value pairs

Note: the iterative process is controlled by a workset contained in the variable *workset*

```
(* Returns the position of elem in the list [1..length(list)] or 0 if elem
   is not in the list *)
FUNCTION POS(elem, list):integer
  POS := 1;
  WHILE list ≠ [] DO
    IF (head(list) = elem) THEN
      RETURN pos;
    FI;
    list := tail(list);
    POS := POS + 1;
  OD;
  (* elem not found in list *)
  RETURN 0;
END;
```

Figure 3.6: The calculation of the call string connector – part 1

```
(* Init *)
FORALL procedures p DO
  ECS_K(p) = [];
OD;
ECS_K(main) := [ε];
workset := {(main,ε)};
FORALL calls c DO
  CON(c) := ∅;
OD;

WHILE workset ≠ ∅ DO
  LET (p,γ) ∈ workset IN
    workset := workset \ {(p,γ)};
    pos := POS(γ,ECS_K(p));
    FORALL c ≡ 'call q' in p DO
      γ' = γ ∘^K c;
      pos' := POS(γ',ECS_K(q)) ;
      IF pos' = 0 THEN
        (* γ' not in ECS_K(q) *)
        ECS_K(q) := append(ECS_K(q),[γ']);
        workset := workset ∪ {(q,γ')};
        pos' := POS(γ',ECS_K(q)) ;
      FI;
      CON(c) := CON(c) ∪ {(pos,pos')};
    OD;
OD.
```

Figure 3.7: The calculation of the call string connector – part 2

Chapter 4

Analysis of Loops

4.1 Introduction

Since loops are usually executed several times the program states will be different for the different executions. In many cases there will be an initialization effect, so that some aspects of the state encountered during the first iteration will be different from all other states occurring during the execution.

In data flow analysis, however, the data flow information for the first run through a loop will be combined with the value for all other executions since the meet over all paths solution is either computed or approximated. So it seems to be useful to keep them distinguished.

A solution is presented which relies on extensions of the interprocedural analysis techniques from the previous Chapter. It offers the advantage that similar problems –loops and procedures– can be treated in the same formal framework. Furthermore it allows to use the existing theory and implementations of the interprocedural analysis for the analysis of loops.

The necessity for better loop analyses has also been claimed by (Harrison III, 1995), who proposed to transform loops into procedures to use the techniques from interprocedural analysis. Also the community of instruction schedulers has invented several techniques for the specialized treatment of loops (software pipelining).

The main idea of applying interprocedural analysis to loops is to extend the procedure concept to a special block structure in the control flow graph for a program, so that the interprocedural techniques can be applied to all pieces of code that are executed several times and not only to procedures. Such blocks can be analyzed like procedures in interprocedural analysis, thus allowing for a separation of the information for different paths through the control flow graph.

The solution presented here allows not only the application of interprocedural techniques to loops but to arbitrary blocks of code. It will be shown that the call string approach is not always well suited for the analysis of nested loops. A new technique called VIVU is presented to overcome these deficiencies.

Category	Abb.	Meaning
always hit	ah	The memory reference will always result in a cache hit.
always miss	am	The memory reference will always result in a cache miss.
not classified	nc	The memory reference could neither be classified as ah nor am.

Figure 4.1: Categorizations of memory references

4.2 Motivation

Two examples for the analysis of loops will now be considered: the available expression analysis (Nielson et al., 1998) and the cache behavior prediction (Alt et al., 1996a; Ferdinand et al., 1997).

4.2.1 Cache Analysis

Caches are used to improve the access times of fast microprocessors to relatively slow main memories. They are an upper part of the storage system hierarchy and fit in between the register set and the main memory. They can reduce the number of cycles a processor is waiting for data by providing faster access to recently referenced regions of memory. (Hennessy and Patterson, 1996) claim that 'modern' workstations are equipped with microprocessors that have cycle times of about 2 to 40ns and DRAM (Dynamic Random Access Memory) that has a cycle time of 90ns and more, which was back in 1996. They also claim that the speed of microprocessors is growing about 60% a year, but the DRAM speed is increasing only by 7% a year. This means that the cache is of fast growing importance to bridge this gap.

Cache behavior prediction is a representative of a large class of analysis problems that are of high practical relevance, e.g., in the area of hard real time systems. Real time systems require a timing validation based on bounds of the execution time. Closely related to cache behavior prediction is pipeline behavior prediction for which similar analysis requirements exist.

The goal of cache analysis is to compute a categorization for each memory reference that describes its cache behavior. The categories are described in Fig. 4.1.

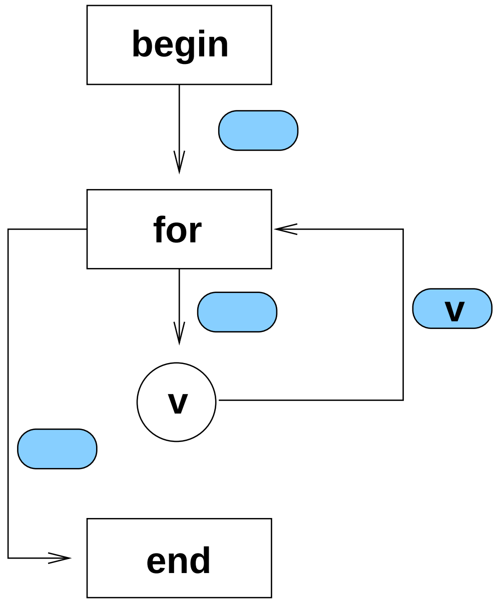
The cache analysis consists of two parts. On the one hand, a *must* analysis computes all memory locations that will surely be in the cache for all executions. On the other hand, all memory locations are calculated that *may* be in the cache. If a memory location m is referenced at a program statement p and m is contained in the abstract must cache for the entry of p this reference is guaranteed to be a cache hit. If m is not in the abstract may cache for the entry of p it is guaranteed to be a cache miss.

Example 4.1
Let us consider a sufficiently large data cache and the program fragment from Fig. 4.2 a) for the must analysis.

In the first execution of the loop, the reference to v *will result in a cache miss, since* v *is not in*

```
/* Variable v not in the data cache */
for i:=1 to ..  do
     ⋮
     y:=v
     ⋮
end
```

a)



b)

Figure 4.2: Motivating example 1

the cache. In all further iterations the reference to v will result in a cache hit, if the cache is sufficiently large to hold all variables referenced within the loop.

The control flow graph for this program is shown in Fig. 4.2 b). An empty oval means that v is not in the abstract must cache and an oval with v means that v is in the abstract cache. In the classical approach, the first iteration and all further iterations are not distinguished. A combination function combines the data flow information at the entry of the loop with the data flow information from the end of the loop body to obtain a new data flow value at the beginning of the loop. Since the combination function for the must cache analysis is based on set intersection, the combined data flow information will never include the variable v, because v is not in the abstract cache state before the loop is entered. The reference to the variable v cannot be classified as ah*. For a WCET (Worst Case Execution Time) computation this is a safe approximation, but nevertheless not a good one.*

4.2.2 Available Expression Analysis

An expression is said to be *available* at a program point p if it has been evaluated at some program point before p, and if it has not been modified (by changing the value of one of its variables) since.

The abstract domain for the analysis is the powerset of all expressions occurring in the program.

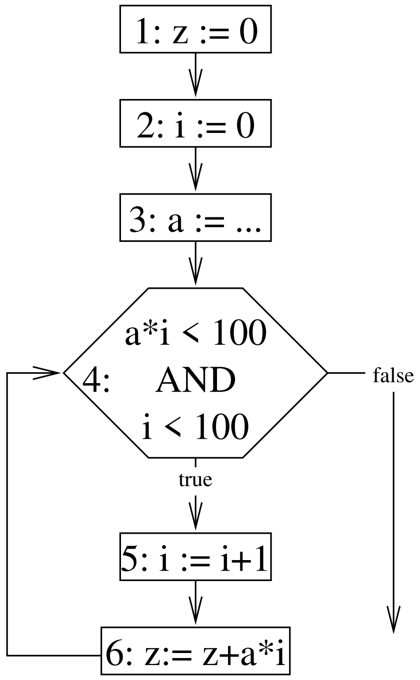


Figure 4.3: Example for available expression analysis

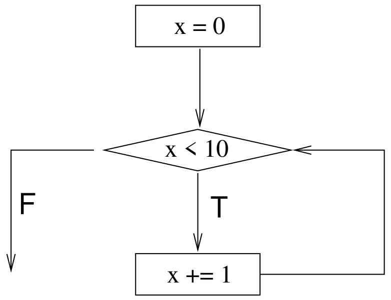
The combination function is set intersection, and the transfer functions for assignments of the form `x:=e` remove all expressions containing an occurrence of `x`, as they will change their value, and add all subexpressions of `e`.

For the control flow graph of Fig. 4.3 the first and all further iterations of the main loop are not distinguished. The set of available expressions directly before the execution of the nodes are:

node	available expressions
1	{}
2	{}
3	{}
4	{}
5	{a*i}
6	{}

It cannot be concluded that `a*i` is available at node 4 in the second and all further iterations of the loop, since the information that the expression is available after node 6 is intersected with the information that nothing is available after node 3. But based on the information that `a*i` is available at node 4 after the first iteration it could be decided to unroll the loop once and then to eliminate the recalculation of `a*i` at node 4.

```
x = 0;
while (x<10) {
   x += 1;
}
```



a) b)

Figure 4.4: A loop and its CFG

4.3 Extending Interprocedural Analysis

In interprocedural analysis, pieces of code (procedures) are analyzed separately for the different data flow values that reach the beginning of the code (calling contexts). This is done to get better analysis results for pieces of code that are executed more than once in different contexts. The motivating examples of Sec. 4.2 have shown that the bodies of loops are also executed several times in different contexts. Therefore, the techniques for interprocedural analysis are now extended such that they can be applied to loops as well.

To do so the concept of procedures is generalized to *blocks*. Blocks have entry and exit nodes which are the only ways to enter or leave the block. Additionally, there may be edges from inside the block to the entry of other blocks (which correspond to procedure calls in the interprocedural context and are therefore referred to as *call edges*). For each of these call edges there has to be a corresponding edge from the exit of the called block back to a node inside the calling block (which will be called *return edges*). Each node in the CFG can belong only to one block.

The transformation of a loop to a block can be seen in Fig. 4.4 and Fig. 4.5. The loop in Fig. 4.4 a) corresponds to the CFG in b) which is transformed to the CFG in Fig. 4.5. The description above requires the introduction of an edge r corresponding to the edge c in the transformed CFG. This edge r in Fig. 4.5 allows the continuation of the calling block after the called block returns. But as a loop is “tail recursive” there is nothing to do after the return. Therefore, the return edge degenerates to a self loop at the exit node. This becomes clearer if the loop of Fig. 4.4 a) is transformed into the recursive procedure in Fig. 4.6 a) which has the supergraph of Fig. 4.6 b). But although self loops can be omitted for an intraprocedural data flow analysis, in the interprocedural setting the inverse call connector is assigned to them.

Now Def. 3.1 of the interprocedural supergraph can be extended to arbitrary blocks.

Definition 4.1 (Supergraph (generalized))
A supergraph $G^* = (G, P)$ consists of a control flow graph $G = (N, E, s, e)$ with a set of nodes N and a set of edges E, a start node s, an end node e, and a partition $P \subset 2^N$ of nodes,

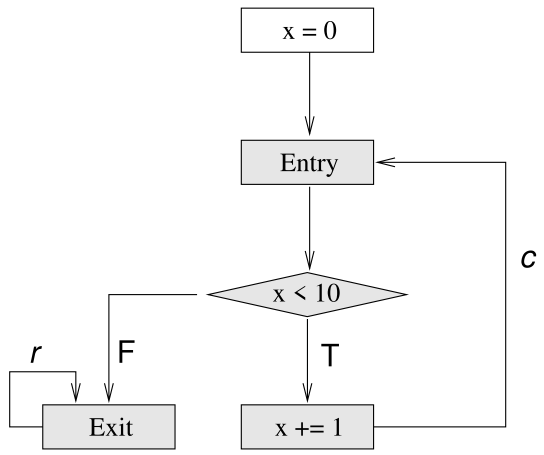


Figure 4.5: The transformed CFG

where each class P_i represents a block with unique entry and exit nodes ($\mathsf{entry}_i, \mathsf{exit}_i$). Class P_0 represents the main block and therefore $s = \mathsf{entry}_0$ has no predecessors and $e = \mathsf{exit}_0$ no successors. Each edge from a node $n_1 \in P_i$ to entry_j (call edge) has a corresponding edge from exit_j to a node $n_2 \in P_i$ (return edge). All other edges are intra partition edges.

A block in the supergraph is said to be *(directly) recursive* if it has an edge to its own entry. A set of blocks in the supergraph is *simultaneously recursive* if their subgraphs are strongly connected. A supergraph is called *recursive* if it contains (a set of simultaneously) recursive blocks. In this work only procedures and loops are used as blocks but the approach is not limited to them. It can be applied to each block of code, but is only useful for blocks which have more than one entry.

Definition 3.2 of an expanded supergraph with different connectors can be directly applied to the supergraph with blocks.

4.4 VIVU

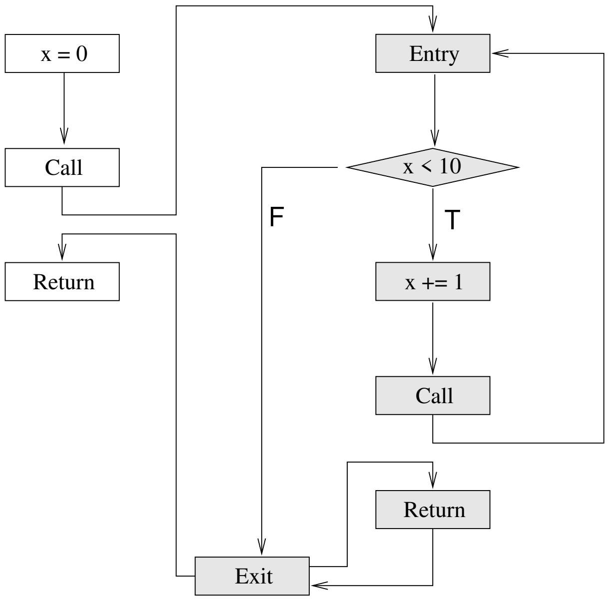
In practice it has turned out that the call string approach is not optimal for nested loops. Even if the length of the call string is chosen to be the nesting depth of the loops many paths are separated although their distinction is not interesting.

Figure 4.7 shows an example for two nested loops. The boxes beside the CFG nodes represent locations, and the locations are connected with dotted arrows according to the (call string) connector. The call edges are labeled with $L1_f, L2_f, L1_o, L2_o$ where f stands for the first calls and o for other (all further) calls of loop $L1$ or loop $L2$. Possible call strings which reach $L2$ are $L1_f(L2_f L2_o^* L1_o)^* L2_f L2_o^*$.

For two nested loops a call string length of two should be sufficient. The call string approach with $K = 2$ considers all suffixes of length two of the paths. These (and their interpretation) are:

```
x = 0;
loop();


void loop(void) {
   if (x<10) {
   x += 1;
   loop();
   }
}
```



a) b)

Figure 4.6: The transformation from a loop to a recursive procedure

- $L1_f L2_f$ (first iteration of the outer and inner loop)
- $L2_f L2_o$ (second iteration of the inner loop, any iteration of the outer loop)
- $L2_o L2_o$ (iteration ≥ 3 of the inner loop, any iteration of the outer loop)
- $L1_o L2_f$ (iteration ≥ 2 of the outer loop, first of the inner loop)

If an analysis problem is assumed for which initialization effects can be expected (like in cache analysis or for available expression analysis) then it is not important to separate the second from all other iterations, but the first from all other iterations. This leads to the following separation scheme, where the contexts are different from the ones obtained by the call string approach:

outer loop	inner loop	context
first	first	$L1_f L2_f$
first	other	$L1_f L2_o$
other	first	$L1_o L2_f$
other	other	$L1_o L2_o$

For the two nested loops this results in the expanded supergraph in Fig. 4.8.

One can see that a kind of call strings is formed, where the elements are again call edges. In the following these strings will be called *call descriptions* to distinguish them from the call strings

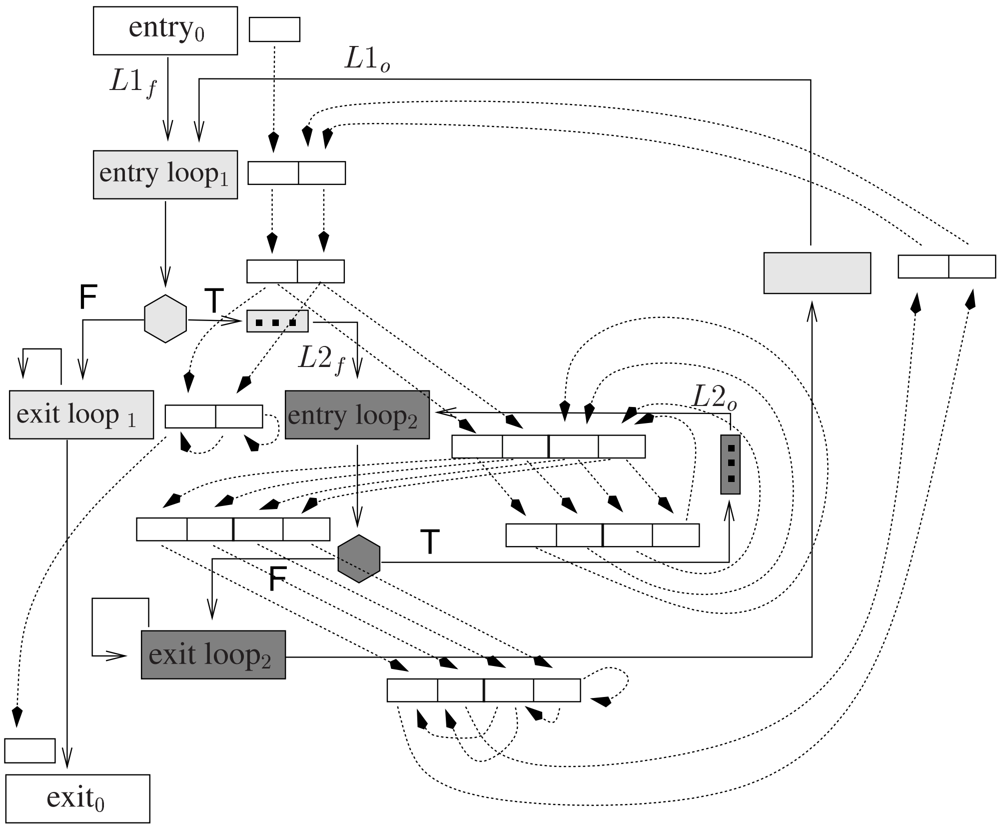


Figure 4.7: Call string approach for two nested loops

in the call string approach. The length of a description is the nesting depth of the actual block which is two in the example for the inner loop. In contrast to the call strings each position in the call description corresponds to a nesting depth, and when a block is entered repeatedly only the corresponding position in the description is updated, and the elements are no longer shifted to the left as in the original call string method. In Fig. 4.8 for example (see Fig. 4.9 for details) the inner loop is recursively called over the call edge $L2_o$ with the call description $L1_f L2_f$ (first location). The resulting call description is $L1_f L2_o$ instead of $L2_f L2_o$. In Fig. 4.10 it is shown that for the call edge $L2_f$ which leads from the body of $L1$ to $L2$, $L2_f$ is appended to the call description of $L1$ ($L1_f$ or $L1_o$), because $L2$ is entered for the first time (no call to $L2$ appears in the call description).

The approach of call descriptions can be applied to nonrecursive blocks as well. All paths through the call graph are kept apart which corresponds to call string ∞ for nonrecursive blocks. Here all call edges c are appended to the call description. Figure 4.11 shows an example situation, in which a block is entered by three calls. The calling blocks are assumed to be a loop block called from the main block, and the other two calls are assumed to be in the main block.

Simultaneously recursive blocks are not covered by the above description, since whenever a

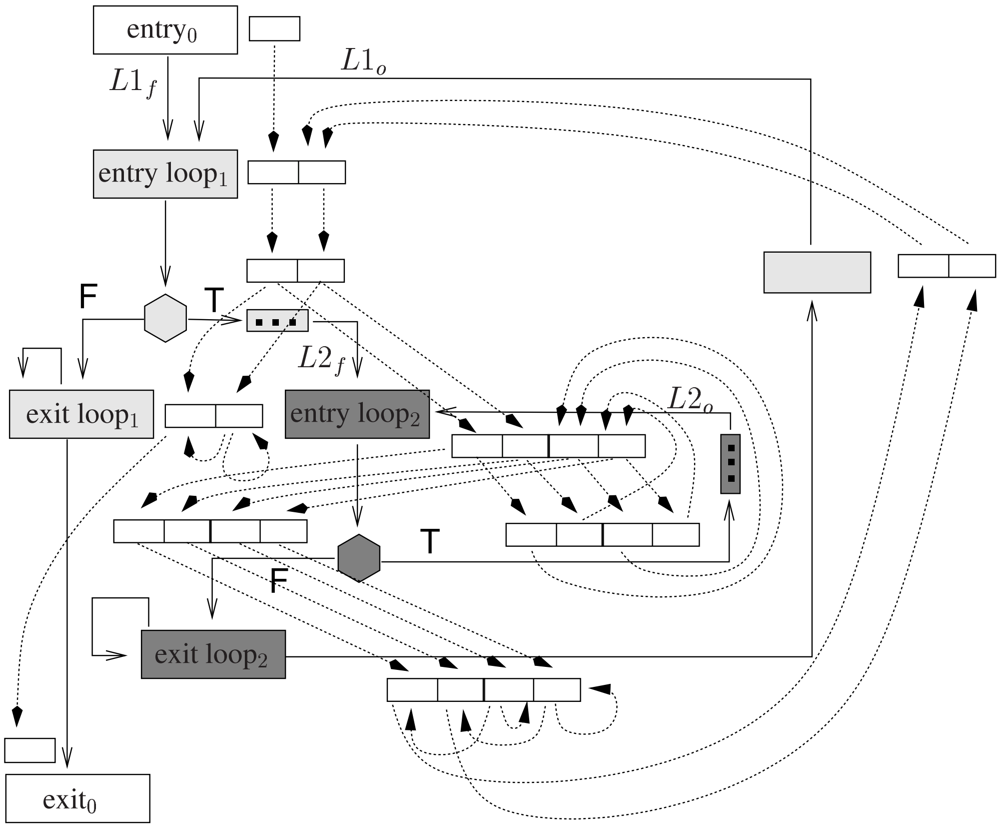


Figure 4.8: VIVU for two nested loops

block b is entered repeatedly through a call cycle (with more than one block) the first call to b is not the last element of the call cycle. But the method can be extended if the replacement of a call is not limited to the last position in the call cycle. I.e. if a block b is entered over a call edge c in a call cycle (a call c' to b already appears in the call description) then c' is replaced by c in the call description. As a result only one call to every procedure can appear in a call description. As an example a simultaneously recursive supergraph is shown in Fig. 4.12 where the blocks are collapsed to single nodes (which is essentially the call graph). In this example the call description c_1 in p stands for the first execution of p and c_1c_2 is the first execution of q. For both p and q the call description c_3c_2 represents all other executions.

4.4.1 Approximation

For nonrecursive procedures the method described so far corresponds to full inlining, and for recursive ones it separates the first pass through them from all other passes through them. This is where the name for this method comes from: Virtual Inlining (of procedures) and Virtual Unrolling (of loops). The number of call descriptions for inner loops in large programs can be

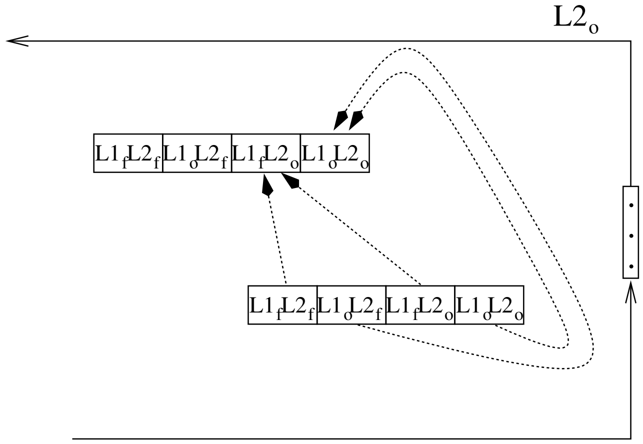


Figure 4.9: VIVU for $L2_o$ in detail

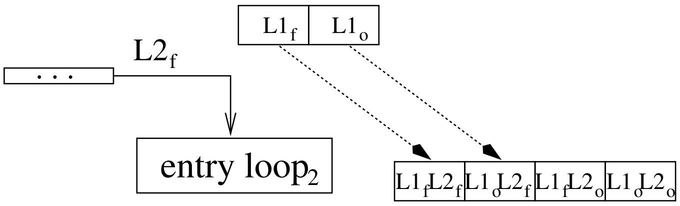


Figure 4.10: VIVU for $L2_f$ in detail

very high which leads to quite long analysis times. The `lloops` benchmark described in Chap. 8 is a kind of worst case input for the VIVU approach because it only consists of several deeply nested loops. It has 5,677 instructions, and the number of locations is 688,371 which are about 121 locations per node on average. Therefore, a method is now introduced which allows to approximate this exact VIVU strategy.

For the VIVU approach the idea is to limit the number of elements in the call description to K like in the call string approach. This means that whenever a block is entered and the call description has already reached length K then the leftmost call in the description is dropped. For a program which has only direct recursive blocks (loops) it is no longer distinguished whether the outermost $n - K$ surrounding loops are in the first or another iteration for loops on a nesting level n deeper than K. Intuitively, this works well because the initialization effects on the inner loop do not depend very much on the state of the outer loops. This is of course an approximation which does not always lead to optimal results.

For programs which have only nonrecursive blocks this approximation is exactly the call string approach. However, recursive blocks are handled differently.

Of course, only those simultaneously recursive procedures are detected that are on a call cycle of length at most K. Call cycles of a size larger than K are treated as in the call string approach.

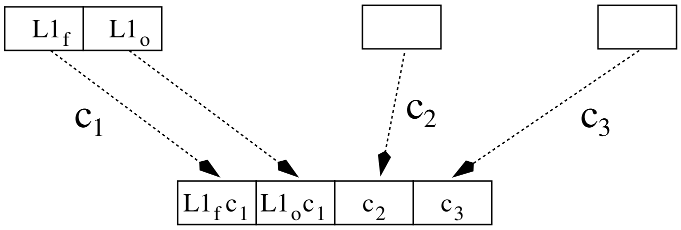


Figure 4.11: VIVU for nonrecursive calls

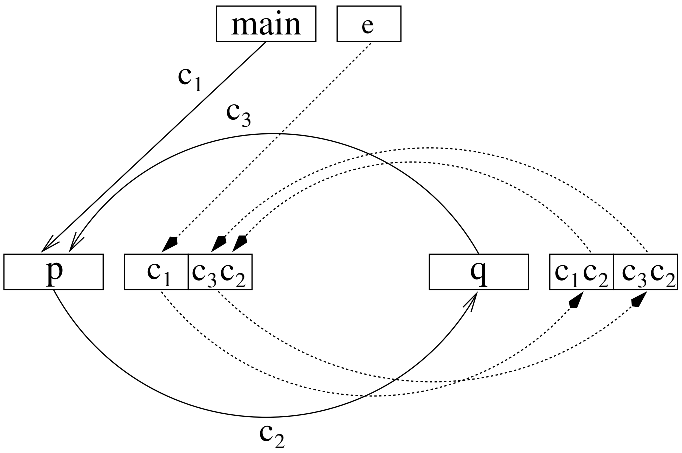


Figure 4.12: VIVU for simultaneously recursive blocks

4.4.2 Formal Description

The VIVU approach can be described in the same formal framework as the call string approach in Sec. 3.3.

The updating relation R_e which reflects how the call descriptions are updated along the edges is the same as for the call string approach except for the fact that now $\oplus^K$ is used instead of $\circ^K$, where $\oplus^K$ defines the append of call edges to K limited call descriptions:

$$c_1 c_2 \ldots c_n \oplus^K c = \begin{cases} c_1 \ldots c_{i-1} c c_{i+1} \ldots c_n & \text{if } c \text{ and } c_i \text{ are calls to the same block}^1 \\ c_1 c_2 \ldots c_n c & \text{else if } n < K \\ c_2 \ldots c_n c & \text{else if } n \geq K \end{cases}$$

$$\alpha \; R_e \; \beta \text{ iff } \begin{cases} \alpha = \beta & \text{if } e \text{ is an intra partition edge} \\ \beta = \alpha \oplus^K e & \text{if } e \text{ is a call edge} \\ \beta \; R_{e'} \; \alpha & \text{if } e \text{ is a return edge corresponding to the call edge } e' \end{cases}$$

[1] if $c_1 c_2 \ldots c_n$ is a valid call description c_i is uniquely determined!

The transfer function can be extended to act on $\Gamma^K \to D$ instead of on D in the same way as for the call string approach.

$$tf^* (m, n) (\chi) (\gamma) = \bigsqcup \{ tf (m, n) (\chi (\gamma')) \mid \gamma' \; R_{(m,n)} \; \gamma \}$$

To turn the approximative call description approach into a connector the same method as for call strings (see Sec.3.4.1) can be used. Only the updating operation $\circ^K$ has to be replaced by $\oplus^K$.

4.4.3 Extension

For some analyses it can be useful to separate not only the first iteration from all others, but also some other initial iterations like the second iteration and so on. In the pipeline analysis (Schneider, 1998; Schneider and Ferdinand, 1999) for example, there are initialization effects not only in the first iteration, but in the first n iterations. Another example would be an analysis in order to predict the behavior of a branch prediction unit of a microprocessor. These units work by recording the last n (nowadays usually 3) results of a conditional jump instruction. Then these results are used to predict the result of the next execution of the jump instruction to prefetch the right instructions from the memory in order to keep the pipeline filled.

In the operation $\oplus^K$ a call c to a block b replaces another call to the same block b in the call description. To separate the first L iterations in $\oplus^K_L$ now L calls to the same block are allowed in a call description γ. When γ already contains L calls to b and another call c to b should be added then the leftmost call to b is dropped and all other calls to b are shifted to the left in the call description, and c is inserted on the place of the rightmost call to b.

To define this formally a projection function Π_c is defined which gives the set of indices of all calls to the block b called by c in a call description. Furthermore, a transformation function Ω_c is defined mapping indices to calls when applied to a call description to express the described shifting of calls

$$\Pi_c(c_1 \ldots c_n) = \{ \; i \mid c_i \text{ and } c \text{ are calls to the same block } b \; \}$$

$$\Omega_c(c_1 \ldots c_n)(i) = \begin{cases} c_i & \text{if } c_i \text{ and } c \text{ are calls to different blocks} \\ c_{i+j} & \text{where } j = \mathit{min}\,(\Pi_c\,(c_{i+1} \ldots c_n)) \\ & \text{and the set is not empty} \\ c & \text{otherwise} \end{cases}$$

For the following, it is assumed that $n \leq K$ and $|\Pi_c(c_1 \ldots c_n)| \leq L$

$$c_1 c_2 \ldots c_n \oplus^K_L c = \begin{cases} c_1 \ldots c_n c & \text{if } n < K \text{ and } |\Pi_c(c_1 \ldots c_n)| < L \\ c_2 \ldots c_n c & \text{if } n = K \text{ and } |\Pi_c(c_1 \ldots c_n)| < L \\ (\Omega_c(c_1 \ldots c_n)(i))_i & \text{otherwise} \end{cases}$$

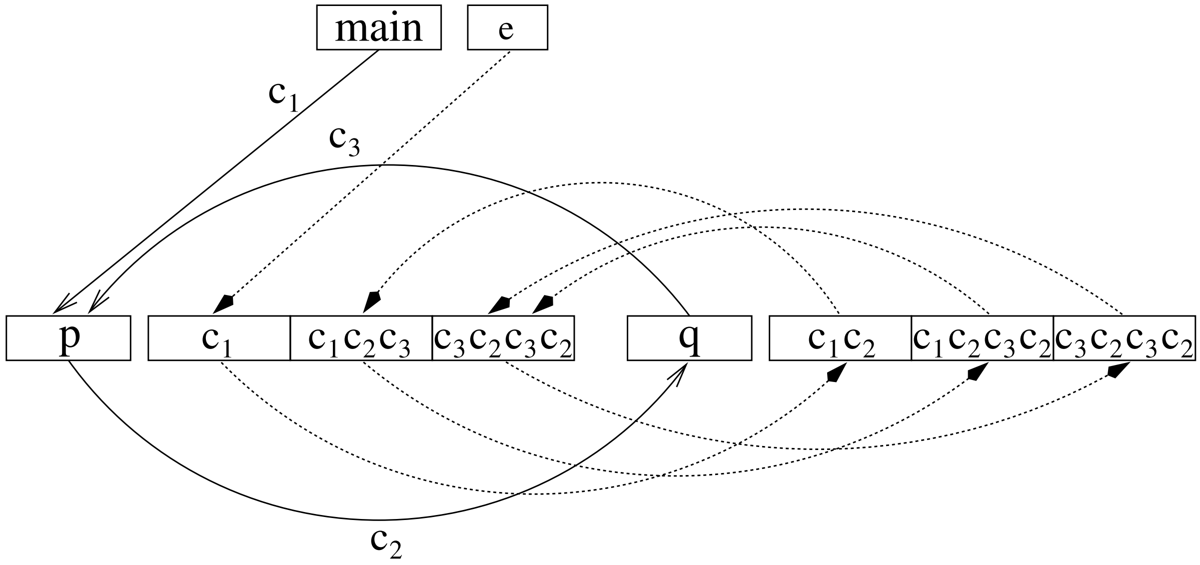


Figure 4.13: VIVU for simultaneously recursive blocks with $L = 2$

To clarify this, the example of Fig. 4.12 is revisited with $L = 2$ in Fig. 4.13. When p is called via c_3 with the call description $c_1c_2c_3c_2$ (second location) then the calls to p are shifted to the left in the call description: c_1 is dropped, c_3 changes its place from the third position to the first position, and the new c_3 is placed on the third position. So $c_1c_2c_3c_2 \oplus_2^K c_3 = c_3c_2c_3c_2$ for each K larger than 3. Only if L and K are chosen such that $L \leq K$ directly recursive blocks can be detected by this extended strategy.

4.5 Revisiting Motivation

By applying the VIVU connector to the example of Sec. 4.2.1 one can see in Fig. 4.14 that the reference to `v` can be determined as a cache miss for the first iteration and as a cache hit for all other iterations. This allows the prediction of a much tighter upper bound of the worst case execution time of the loop.

For the motivating example of the available expression analysis (see Fig. 4.3) the VIVU connector gives the result that `a*i` is available at node 4 in all iterations except the first. With this result an optimizing compiler can decide to unroll the loop once and then to optimize the loop body further.

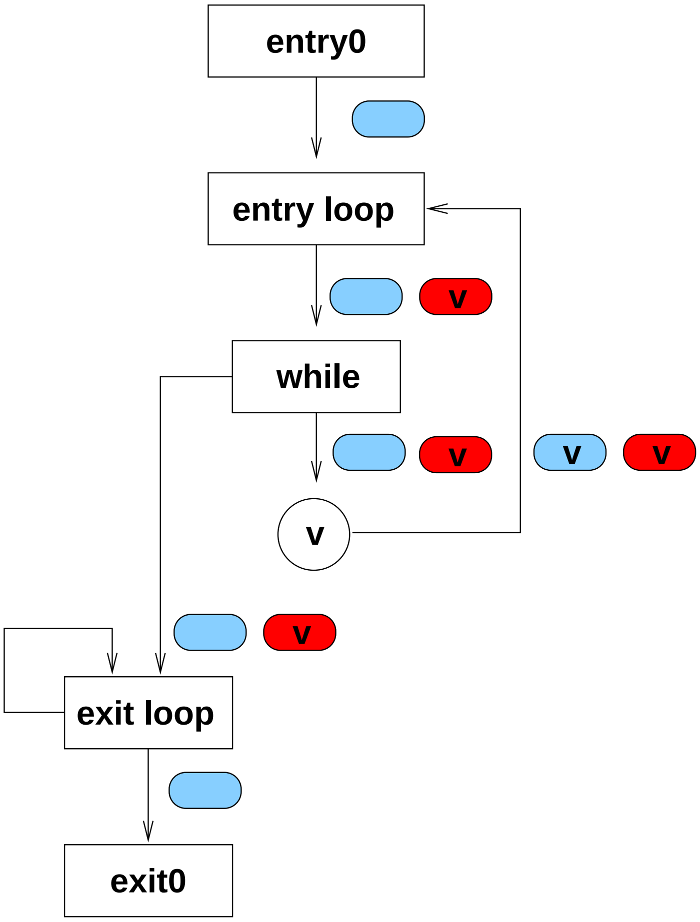


Figure 4.14: Example for cache prediction

Chapter 5

Generating Analyzers

From Chap. 2, one can see what is needed to implement a data flow analyzer:

1. the complete lattice D of abstract values with a lub operation and a bottom element, and an equality function to test for stabilization
2. a transfer function which can be applied to a CFG edge giving a monotone function from D to D. This function is used to construct an equation system for each control flow graph.
3. a fixed point solving algorithm with the appropriate data structures, e.g. an iteration algorithm and a workset
4. parameters such as the direction of the analysis and the initial value ι
5. optional widening and narrowing functions

Furthermore a control flow graph is needed as input for the analyzer.

Some of these parts, e.g. the interface to the CFG or the implementation of the fixed point iterator, do not change much from one analyzer to the other. To shorten the time of implementation and to simplify the maintenance of the resulting analyzer it seems to be desirable to have a generator which allows to specify the changing parameters and to generate the whole analyzer from that specification. For such a system a balance between the efficiency of the generated code and the flexibility of the specification system has to be found.

The advantages of generating analyzers instead of hand coding them are much shorter development periods and easier maintainability. The generation can also enable the development of more complex systems which were hardly feasible before. But it should be clear that in most cases carefully hand coded analyzers are faster than automatically generated analyzers. This is analogous to compilers: nowadays most programs are developed in a high level language, since it is much faster and easier to program in a high level language than to write assembler code.

Nevertheless, for some of the hot spots of a program it still can be worthwhile to hand code them in assembler, if the compiler output lacks the required quality.

In the following the criteria for developing PAG are described.

5.1 Design Decisions

In (Nielson, 1996) the author states: "To be useful, a tool should have an easy user interface (specification mechanism) compared with the complexity of the task to be performed. [...] Next the tool should have a wide area of applicability. [...] Also the tool should produce a result of sufficiently high quality that the programmer sees no reason to redo the effort manually. Finally, the tool should be seen as a consecutive way of exploiting existing theories for how to conduct the task at hand."

The goals that have been followed when developing PAG are very similar:

Generality: the generator should be applicable to all types of data flow problems. There should be no restriction to problems with certain properties such as distributive bitvector problems. The restrictions that exist in PAG are due to the way the programs to be analyzed are modeled: only those languages can be analyzed whose program execution can be described by a control flow graph. This is certainly true for all imperative languages including assembler languages and also for logic languages (Fecht, 1997). This model can also be applied to a restricted class of (imperative) object oriented languages. The problem for the latter is the potentially unknown control flow for the invocation of virtual methods. Therefore, conservative assumptions have to be made, e.g. assuming a possible control flow to all procedures that might be called. These pessimistic assumptions can potentially be relaxed using a control flow analysis prior to the control flow graph construction, or interleaving the control flow analysis with a data flow analysis (Nielson et al., 1998).

Efficiency: the generated analyzers should be applicable in real compilers. They have to be able to analyze large real world programs. This can be seen in contrast to other program analyzer generators that aim mostly to research or educational purposes. Of course, the goal of efficiency has to be seen in the context of the development of PAG: it is still a research project without major industrial involvement. So, man power is lacking to exploit all possibilities for improvements and optimizations.

Expressiveness: the specification mechanism should be powerful enough to express complex data flow analyses in a concise way. It must be clear enough to write well structured and understandable specifications.

Interprocedural: there should be the possibility to generate interprocedural analyzers with minimal specification overhead (e.g., only the specification of the parameter and return value handling should be needed). Also the specification of the intraprocedural analysis method should be possible. The implementation of the interprocedural analysis method should be exchangeable, without the need to change the specification.

Modularity: as far as possible the different parts of the analyzer should be implemented as modules with a well defined interface, making it easy to replace the implementations of the modules. This enables the insertion of newly developed algorithms into the existing analyzer generator.

Portability: the generated analyzers as well as the analyzer generator itself should be portable to many platforms.

As a result from the last point it was decided to use ANSI C as implementation language for the generator as well as for the generated analyzers. This also simplifies the integration into existing compilers which are mainly written in ANSI C or other imperative languages.

For the choice of the specification language first the question of the language paradigm to choose had to be considered: is an imperative or a declarative language the better choice? Using an imperative language offers the advantage that most programmers are familiar with those languages. Also, it could nicely coincide with the decision to use ANSI C as implementation language. And additionally in most cases, imperative languages allow to write the fastest programs. But if one considers automatic or semi automatic proof techniques to reason about some aspects of an analyzer specification like proving the monotonicity of the transfer function, or the correctness of a widening, a declarative language seems to be more suited than imperative languages. This is also true for imperative languages which have a clearer semantics than ANSI C, such as Pascal or Modula.

The use of a functional language allows the shortest and easiest specification of most data flow problems. This is the reason why many (formal) descriptions of data flow analyzers are written in declarative style (either functional or mathematical) (Nielson et al., 1998; Sagiv et al., 1996). Using a declarative language as specification language frees the user from many implementation details, e.g. memory allocation and garbage collection.

To achieve the goals of generality and expressiveness in PAG it has been decided to use a functional language and also a powerful mechanism to specify the abstract domains. Their implementation is then generated. This frees the user from considering implementation details. Additionally, this enables different implementations of the same functionality. The implementation to be used can either be chosen automatically or can be selected by the user.

The second design decision about the specification language was whether to use an existing functional language such as ML or Haskell or to design a new language especially tailored for that purpose.

The major drawback of a new language is that no one is used to the syntax of this language. This can be avoided by choosing a syntax close to existing functional languages.

Using ML (or other existing functional languages) as specification language for the analyzer with a translation directly to executable code would make it difficult to migrate the generated analyzer to different machine types. Also the integration into existing compilers would become very harmful: interfacing ML with ANSI C is very difficult. And the ML-C interface is not part of the ML standard, meaning that this interface varies between the different ML implementations if it is implemented at all. Additionally, the size of the ML runtime system is for the most implementations quite large.

Since ANSI C was chosen as implementation language there was a need for a functional specification language that can be translated directly to C. It has been decided to design a new functional language with syntactic elements from Haskell (Bird, 1998) and extensions which support the special need when designing data flow analyses. Some restrictions have been imposed on the functional language to simplify the translation to ANSI C. Differences to existing functional languages are induced by the special needs when specifying data flow analyses.

Another important decision was to base each PAG generated analyzer on a control flow graph. This avoids the need for an explicite representation of the equation system. The advantages of this approach are that the solver can be especially tuned for the application area and that no separate phase to generate the equation system from the CFG is needed. The drawbacks are that the equation solver from PAG cannot be used directly to solve other kinds of equations, and that it is difficult to use other equation solving methods than the iterative solution methods, e.g. elimination techniques.

5.2 Integration

Since a data flow analyzer is never a stand alone tool, it is necessary to integrate it into an existing compiler. The integration has to be done on both sides: the input side and the output side. For the input side PAG relies on the CFG construction by the compiler. It uses a simple interface with functions such as `fetch_predecessors`, `get_start`, `get_type_of_edge` and so on, to walk through the graph. Additionally, it is assumed that the nodes are labeled with subparts of the syntax tree, which may be inspected by the transfer functions. Hence, there is also a fixed interface to examine the syntax tree. The structure of the syntax tree is described in a special input file. In order to use a PAG generated analyzer in an existing compiler it is necessary to implement this interface and to describe the structure of the syntax tree. This has to be done once per compiler. Additionally, there are several such interfaces for existing compilers or frontends.

The interface on the output side is kept simple: there is a set of C-functions that allow an optimizer phase to examine the results produced by PAG. It is also possible to write (complex) evaluation functions in the functional language which can be called in the subsequent compiler phase.

PAG comes with a graphical tool that can visualize the CFG with the calculated data; this allows for an easy design and test of a data flow analysis.

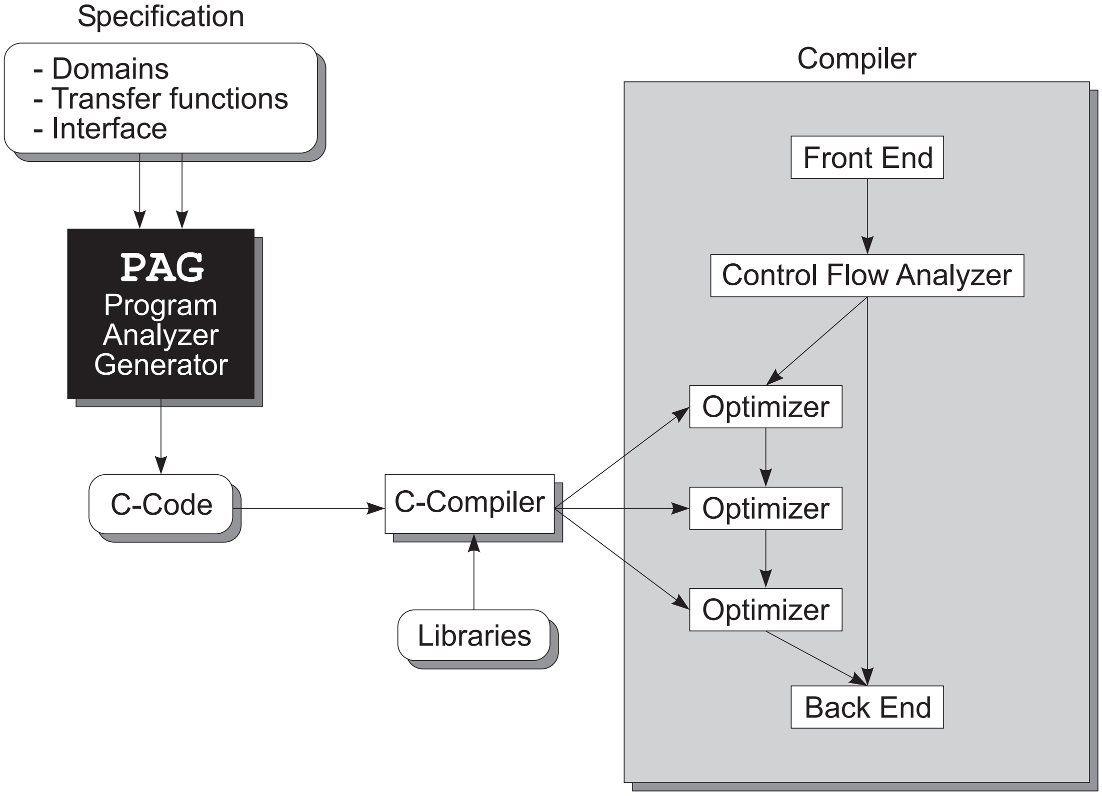


Figure 5.1: Integration of analyzers

5.3 The Generation Process

In order to generate an analyzer with PAG (assuming an already existing frontend interface) two specifications have to be written: one for the definition of the data structures, including the abstract domain, in DATLA, and one that contains the analysis parameters, the definition of the transfer functions and optionally some support functions. (A detailed description follows in Chap. 6.) From these files, PAG generates C-files and a compile script which creates a library that has to be linked to the compiler. In the compiler a call to the analyzer has to be inserted.

An overview of the interaction of the generated analyzer with the compiler is shown in Fig. 5.1.

Chapter 6

Specifying the Analyzer

In this chapter the specification mechanism of PAG is explained using a number of examples. Some technical details are omitted. For further references see the manual (Thesing et al., 1998).

There are several points in the specification process where specifications, parameters or options may be used to influence PAG or the generated analyzer: the specification of the abstract domains, the transfer functions and a number of parameters which are written into specification files. Other options such as the used iteration algorithm are passed as command line options to PAG. The generated analyzer can be influenced by a number of compile time flags which have to be passed to the C compiler when translating the generated code. Finally, there are a couple of runtime parameters which influence the behavior of the generated analyzer.

This chapter focuses on the main specification files. For a better understanding of them a part of the frontend description has to be explained. Only a few compile time and runtime parameters are pointed out. Their majority is explained in Chap. 7.

A complete example of an analyzer specification is given in Appendix A. It is the specification of an analyzer to detect assignments to unused variables. The analysis is called *strong live variables* and its purpose is described further down.

6.1 Declaration of Global Values

Analyzer runtime parameters from the frontend, the command line, or other parts of the compiler can be used to parameterize the generated analyzer. They can be used in all parts of the analyzer specification. These parameters have to be declared. They can be used to import external values which are constants for each analyzer run, e.g. the number of variables in the input program from the frontend, or analyzer command line options. C functions have to be implemented to retrieve the values of the global parameters.

6.2 The Datatype Specification

In this part of the specification the datatypes for the analyzer are defined. This is done in a language called DATLA (DAta Type definition LAnguage).

PAG distinguishes two kinds of datatypes: sets and lattices, where lattices means complete lattice according to Def. 2.3 and has therefore a partial ordering, top and bottom elements, and the lub and glb operations.

Sets and lattices are constructed bottom up. There are a number of predefined simple datatypes such as numbers, and there are operators that construct new datatypes from other datatypes, called *functors*. For instance one can build the set of all functions from numbers to Boolean values which then can be used as input for other functors.

There are two parts of the specification: the first part is for the definition of sets (without ordering) and the second part is for the definition of lattices. Each definition is an equation with a single name that is to be defined on the left side and a functor applied to a couple of names on the right side.

Basic sets are:

1. `snum`: the set of signed integers
2. `unum`: the set of unsigned integers
3. `real`: the set of all floating point numbers
4. `chr`: the character set (ASCII)
5. `string`: the set of all character sequences

Basic lattices are:

1. `lsnum`: the signed integers with additional $+\infty$ and $-\infty$ ($\top$ and $\bot$) ordered by $<$
2. `lunum`: the unsigned integers with additional $+\infty$ ($\bot$ is 0)
3. `bool`: the truth values `true` and `false` ordered by `true` $>$ `false`
4. `[a..b]`: interval lattices are subsets of the signed numbers ordered by $<$
5. enumeration of elements and of a complete partial order

These lists can be extended by user-defined types. They have to be implemented according to the rules given in Chap. 7. Note that a lattice can also be used whenever a set is required, as a lattice is a set with a partial ordering.

Complex sets can be formed from the basic ones through the following functors (also lattice functors can be used to create sets):

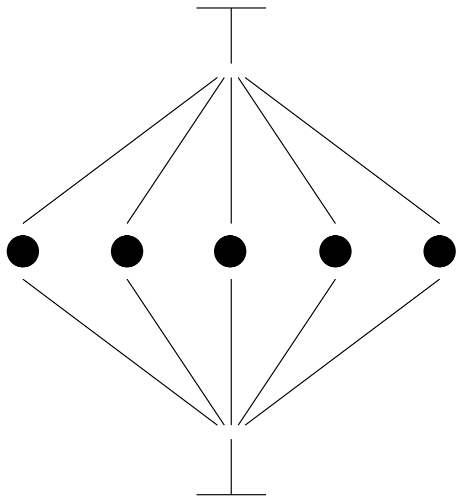


Figure 6.1: The flat functor

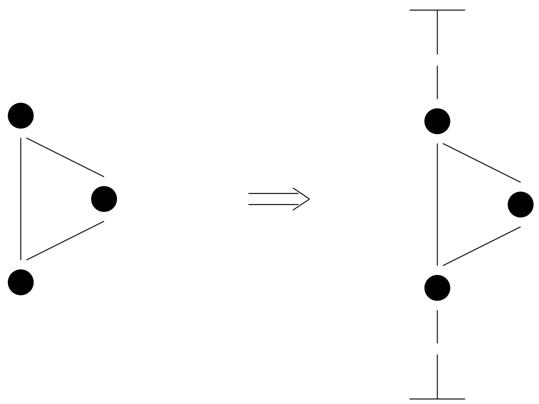


Figure 6.2: The lift functor

sum: `S1 ++ S2 ++ ...++ Sn` disjoint sum of a finite number of sets: this is the union of the sets, but if several sets contain the same element these are distinguished

list: `list(S)` lists of the elements of S

Some functors require their arguments to be lattices (L), others only require them to be sets (S). The following lattice constructing functors exist in PAG:

flat: `flat(S)` flattening of a set S: each element of S is only comparable to itself and to the two new elements $\perp$ and $\top$ (see Fig. 6.1)

lift: `lift(L)` lifting of a lattice L: a new top and bottom element are added that are greater respectively smaller than all other elements. The ordering of L is preserved (see Fig. 6.2).

power set: `power(S)` denotes the set of all subsets of S ordered by set inclusion

tuple: `L1 * L2 * ...* Ln` construction of the tuple space of a finite number of lattices: the ordering is component–wise

function: `S -> L` denotes the lattice of functions from a set S into a lattice L with a point–wise ordering

dual: `dual(L)` construction of the dual lattice of the lattice L (reverting the ordering)

reduced lattice: `reduce(L,f)` denotes the reduced lattice E over another lattice L with a reduction function $f : L \to L$. E is defined as the image of L under f. f is specified in FULA and must be monotonic and idempotent. Then the image is the set of all fixed points of f and forms a complete lattice (see Theorem 2.1). The reduced lattice can be used for example to implement the combination of odd/even analysis with the analysis of signs, see (Nielson et al., 1998; Codish et al., 1995).

For some functors there exist different implementations, e.g. unsorted tables, arrays or several tree implementations for the function functor. PAG tries to choose a good implementation. This includes packing sets to bit vectors whenever reasonable. For details see Chap. 7. Further informations about the data can be expressed in DATLA as follows:

- size restriction: the maximal value of a global value can be restricted e.g.:
  `a_way: unum < 24`
- implementation selection: the `use` clause allows to select a functor implementation directly e.g.:
  `assoc = store_line -> age use(arrayfunc)`

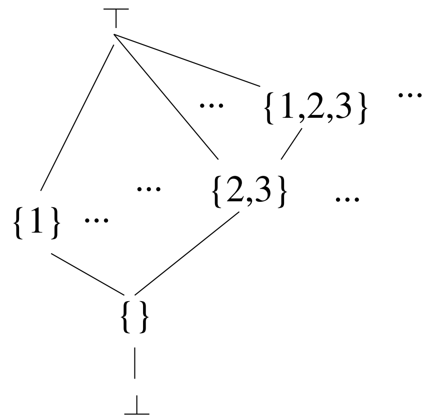


Figure 6.3: The lattice `var`

Example 6.1 (Strongly live variables)
The strongly live variable analysis (Giegerich et al., 1981) is an extended version of the live variable problem. It tries to detect for each node in the CFG the set of all variables which may be used on a path from the node to the exit, and is a backward problem. In the live variable analysis an assignment of the form `x := a + b` *makes* `a` *and* `b` *alive. In the strong variant of the analysis* `a` *and* `b` *are considered to be alive before the assignment only if* `x` *is alive after the assignment. This makes the analysis invariant under elimination of unnecessary assignment statements.*

```
GLOBAL
  maxvar : snum

SET
  vars   = [0 .. maxvar]

LATTICE
  varset = set(vars)
  var    = lift(varset)
```

The abstract domain of this analysis consists of sets of variables (`varset`*). An additional lift operation is applied to fulfill the condition* $\forall e \in E, d \in D : d \neq \bot \Rightarrow tf(e)(d) \neq \bot$ *from Sec. 2.4. This definition results in the lattice in Fig. 6.3.*

Example 6.2 (Interval analysis)
In this analysis from (Cousot and Cousot, 1992) the concrete domain of integers as values of variables is abstracted by intervals. An interval $[a, b]$ *is an abstraction for all numbers* x *between* a *and* b*. One can calculate with those intervals (addition, subtraction, and so on). Intervals are ordered by subset inclusion. Hence the* lub *of two intervals is the least interval which contains both. A condition such as* `x <= v` *in a program can be used to refine the interval for* `x`*, e.g., for the true branch this can be expressed by: '*`x` *has a value that lies between* $-\infty$ *and* `v` *intersected with the interval for* `x` *before the statement'. This may lead to cases where it is detected that a branch in the CFG cannot be reached.*

One should remark that the abstract domain has infinite ascending chains, therefore a widening is needed. Results can be improved by narrowing.

```
LATTICE
upper_bound = lsnum
lower_bound = dual(lsnum)
interval    = lower_bound * upper_bound
env         = snum -> interval
dom         = lift(env)
```

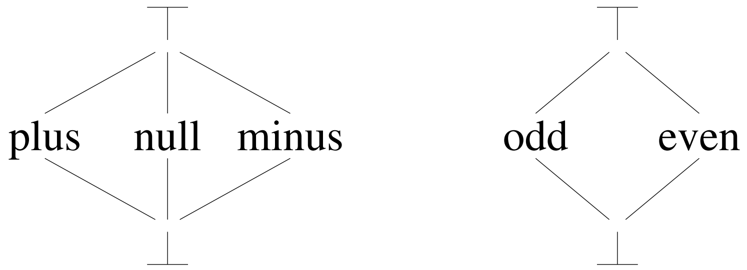


Figure 6.4: The lattices `sign` and `oddeven`

Here `env` *maps variables (encoded as signed numbers) to intervals which consist of a lower and an upper bound. A bound is either a number or* $+\infty$ *(*$\top$*) or* $-\infty$ *(*$\bot$*). Note that not all elements of* `interval` *denote valid intervals. The abstract domain* `dom` *has an additional bottom element to indicate non reachable paths.*

Example 6.3 (Combination of sign analysis with odd/even analysis)
This analysis demonstrates the use of the reduce functor. It implements a combination of a sign analysis and an odd/even analysis.

```
SET
sign_set    = (null, minus, plus)
oddeven_set = (odd, even)

LATTICE
sign        = flat(sign_set)
oddeven     = flat(oddeven_set)
pair        = sign * oddeven
dfi         = reduce(pair,reducef)
```

The definitions for `sign_set` *and* `oddeven_set` *enumerate the elements of the sets. The lattices* `sign` *and* `oddeven` *are constructed using the* `flat` *functor. These definitions result in the lattices from Fig. 6.4. The reason for using the reduce functor is that there are elements in the combined abstract domain (*`pair`*) which describe the same set of concrete values. These redundant elements are removed by the reduction function. This function* `reducef` *is defined in the continuation of the example down below.*

6.3 Description of the Frontend

Before it is explained how the transfer functions are defined, a short introduction to the description of the syntax tree and the edge types delivered by the frontend is given, since the transfer functions are defined in terms of the syntax tree and the edge types.

The form of the tree is described by a tree grammar with two additions. First, there is a notation (*) to introduce lists of nonterminals. Second the notation *nonterminal* == *simple_type* is used to identify a class of nonterminals with a built-in type. Such rules indicate that the elements of *nonterminal* can be converted to elements of *simple_type*.

Example 6.4

```
SYNTAX

START : Unlabstat

Unlabstat  : M_Assign(var:Var,exp:Exp)
           | M_While(exp:Exp,body:Stat*)
           ...

Exp        : M_Binop(op:Op,e1:Exp,e2:Exp)
           | M_Unop(op:Op,e:Exp)
           | M_Var_exp(var:Var)
           ...

Op         : M_op_leq()
           | M_op_plus()
           ...

Var        : M_simpl_var(id:Identifier)
           | M_select_var(var:Var,exp:Exp); // reference to an array

Identifier == snum; // variables are encoded as signed numbers from the
                    // frontend for efficiency reasons
           ...
```

The edge types that may occur in a control flow graph produced by a frontend are listed in the frontend description as well. Only the names introduced there may be used as edge types in the specification of the transfer functions.

Example 6.5

```
EDGES

true_edge
false_edge
call_edge
local_edge
return_edge
```

6.4 The Transfer Function Description

The main part of the description of an analyzer is the specification of the transfer functions. These are expressed in the functional language FULA.

6.4.1 Overview

FULA is a monomorphic first order functional language with eager evaluation. It borrows many features and syntax elements from the language Haskell (Bird, 1998). It has static scoping rules and the user defined types from DATLA.

In FULA each expression has a unique type that can be derived statically by a type inference algorithm. There are no implicit type casts in FULA. Any change of the type must be made explicitly.

For occurrences of a variable the corresponding definition is the syntactically innermost definition for that variable. Binding constructs are function definitions, `case` and `let` expressions as well as ZF[1] expressions (details will be shown later).

In the following, only those constructs from FULA are explained that are different from standard functional language terminology or cannot be derived from the context.

6.4.2 Datatypes

All types declared in DATLA can be used in FULA. Also types defined in the syntax tree description can be used in FULA.

There are two different types of functions in FULA: *static functions* defined in the language itself and *dynamic functions* declared in DATLA. The latter are seen as datatypes and can be arguments to the first sort. This distinction is made because of the first order character of the language. So it is possible to write a FULA function that takes a DATLA function as an argument. Also it is possible to have a static function returning a dynamic function.

For a cast rule of a syntactic type in the syntax tree description, e.g., `Identifier == snum`, PAG defines a function `val-Identifier` to turn a syntax tree leave (`Identifier`) into a built-in type (`snum`).

6.4.3 Function Definitions

Static functions are defined by pattern matching. For each function there may be an optional type definition e.g. `f :: snum, snum -> unum`. Function definitions must not be nested.

[1] Also known as list comprehensions.

6.4.4 Control Constructs

The language FULA contains several control constructs, which are in fact expressions:

if-expression: `if b then e1 else e2 endif`

let-expression: `let p1 = e1; ... in exp` matches the expressions `e`*i* against the patterns `p`*i*. The variables occurring in the patterns can be used in the expression `exp`. If not all patterns match this results in a runtime error. The value of the whole let expression is the value of `exp`.

case expression: evaluates to a guarded expression when all its guards match. E.g. the expression

```
case e1, e2 of
  p1, p2 => exp1;
  p3, p4 => exp2;
endcase
```

evaluates to `exp1` if `p1` matches `e1` and `p2` matches `e2`. The variables introduced in `p1` and `p2` can be used in `exp1`. If no case matches a runtime error occurs.

6.4.5 Expressions

Expressions include

- constants (built-in and enumerated)
- function applications: DATLA and FULA functions can be applied to zero or more expressions. The number of arguments must exactly match the function definition since oversupply or undersupply are not allowed in first order languages. For the application of dynamic functions curly braces are used: `f{e}`, whereas for static function applications round braces have to be used: `f(e)`.
- print expressions: `print` (e_0) `++` e is equivalent to the expression e but with the side effect that e_0 is printed
- built-in function application in pre- and infix notation (see below)

6.4.6 Predefined Functions and Operators

For the datatypes of FULA various functions or operators can be applied. For all types, there is an equality test `=` and an inequality test `!=`. For every lattice the constants `top` and `bot` as well as the infix functions `lub` and `glb` are defined. It is also possible to use the predicates `<,>,<=,>=` which refer to the ordering of the lattice.

For lattices defined as `D = flat(E)` or `D = lift(E)` there exists a pair of functions `drop:: D -> E` and `lift:: E -> D`, which allow the conversion between the two types. If `drop` is applied to `top` or `bot` it results in a runtime error.

Especially useful is the strict binding in `let` expressions: using `p`*i* `<= e`*i* instead of `p`*i* `= e`*i* in a `let` expression. If `e`*i* is an element of a flat or lift type and evaluates to top or bottom the whole `let` expression evaluates to top respectively bottom. Otherwise the pattern `p`*i* is matched against `drop(e`*i*`)`.

For disjoint sums such as `foo = snum ++ real ++ chr`, insertion functions as `up-foo-2:: real -> foo` are defined. They insert an element into the sum from the corresponding constructing set. Also extraction functions as `down-foo-1:: foo -> snum` and test functions as `is-foo-3:: foo -> bool` are defined.

For tuples the selection operator to select the *i*-th element of an *n*-tuple `t` is `t!`*i*`!`*n*[2].

For power sets an element `e` can be inserted to a set `s`: `s`∼`e`, or deleted `s#e`. Also its membership can be tested `e?s`.

A constant dynamic function can be created by `[->e]`. The value of a dynamic function `f` for the argument `v` can be changed to `e` by `f\[v->e]`. To convert an element from a lattice in an element of the dual lattice (and vice versa) the function `dual` can be used.

Additionally, a number of conversion operations exists e.g. to convert a signed number to a character.

6.4.7 Patterns

Patterns can be used in function definitions as well as in `let`, `case` and ZF expressions. Only *linear* patterns are allowed, i.e., variables are not allowed to occur more than once per pattern. Pattern expressions may be nested.

The following patterns are defined:

1. constants: these can be elements of the predefined types, e.g., snums as well as user defined constants of enumerated sets or domains. Two special constants are `bot` and `top` the bottom and top elements of all lattices
2. variables: they match always
3. empty list and empty set: `[]` and {}
4. cons pattern: $p\!:\!p$

[2]The reason why the arity n of the tuple has to be given is explained in the next chapter.

5. tuple: $(p,\cdots)$. a tuple pattern matches if the sub-patterns match
6. as pattern: `p as i` binds the value matched by `p` to the name `i` for later references
7. wild card: it is denoted by an underbar `_` and matches every value

6.4.8 ZF Expressions

ZF expressions can be used to construct lists, sets and dynamic functions. They are created in an iterative process: a generator can be used to iterate over all elements in a list or set. E.g. the generator `x <-- s` successively generates all elements contained in the set `s`. There exists a generator for functions as well: `(x,y) <- f\z` generates all key/value pairs `(x,y)` from `f` for which `y` $\neq$ `z`. If several generators appear in a ZF expression all combinations of elements will be generated. Additionally, filter expressions can be used in ZF expressions to filter out some generated elements: `x <-- s, if x >= 0` will generate only those `x` from `s` which are not negative. Also local definitions as in `{ a !! x <-- s, a=x*3}` can be used. Note that for set and function generators the order of the generated elements is not determined.

Example 6.6 (ZF expressions)

- *remove all negative numbers from a set* `set`
  `{ x !! x <-- set, if x >= 0 }`
- *add one to every element of a set*
  `{ i+1 !! i <-- {1,2,3,4}}` *results in* *{2,3,4,5}*
- *creation of a list from a set (order of the result list is not determined):*
  `[ i+j !! i <-- {5,6}, j <-- {1,2}]` *results in [6,7,7,8]*

Example 6.7 (Strongly live variables cont.)

```
get_id(M_simpl_var(id))      = val-Identifier(id);
get_id(M_select_var(var,_)) = get_id(var);

use_var(M_select_var(var,exp)) = use(exp) lub use_var(var);
use_var(_)                     = bot;

use                   :: Exp -> varset;
use(M_Binop(_,e1,e2)) = use(e1) lub use(e2);
use(M_Unop(_,e))      = use(e);
use(M_Var_exp(var))   = {get_id(var)} lub use_var(var);
use(_)                = {};
```

The function `get_id` *extracts the name of a variable from the left side of an assignment. This can be either a variable or an array reference. For the latter case the function is defined recursively over the syntax tree.*

The functions `use` *and* `use_var` *compute the set of used variables for expressions or left sides of assignments. They are defined mutually recursive over the structure of expressions or left sides of assignments respectively. Note that* `lub` *corresponds to the union.*

Example 6.8 (Interval analysis cont.)

```
is_ok(l,u) = dual(l) <= u;

eval :: Exp, env -> interval;
eval(exp,en) =
  case exp of
    M_Var_exp(M_simpl_var(id)) => en{val-Identifier(id)};
    M_Int_exp(int) => let i = val-Integer(int) in (dual(i),i);
    M_Binop(M_op_plus(),e1,e2) =>
      let (l1,u1) = eval(e1,en);
          (l2,u2) = eval(e2,en);
      in (l1+l2,u1+u2);
    ...
  endcase;
```

Interval intersection (`glb`*) can deliver intervals where the lower bound is greater than the upper bound if applied to intervals with an empty intersection. Therefore, a test function* `is_ok` *is used to detect illegal intervals.*

Function `eval` *evaluates an expression in a given environment to an interval. This function has to contain a complete expression evaluator and is defined according to the structure of the expressions.*

Example 6.9 (Combination of sign analysis with odd/even analysis cont.)

```
reducef(null,top) = (null,even);
reducef(null,odd) = (bot ,bot );
reducef( _   ,bot) = (bot ,bot );
reducef(bot , _ ) = (bot ,bot );
reducef( a   , b ) = ( a   , b  );
```

The reduce function `reducef` *projects all elements in the lattice* `pair` *that represent the same set of concrete values to the least lattice element representing the same set of concrete values.* `(null,top)` *as well as* `(null,even)` *represent only the value 0.* `(null,odd)` *represents no value like all lattice elements containing a* `bot` *in any position. For all other elements* `reducef` *is the identity function.*

6.4.9 Transfer Functions

A FULA source consists of two parts: one for definitions of auxiliary functions and one for the transfer functions.

As explained in Chap. 2 the transfer function for an edge is defined in terms of the source node and its edge type. Transfer functions are written in a special notation: they do not need a name and are defined via patterns matching the label of the control flow graph and the edge type. They have an implicit parameter named `@` which is the incoming data flow value and they have to return a data flow value again.

Example 6.10 (Strongly live variables cont.)

```
TRANSFER

M_Assign(var,exp),_:
  let flow <= @;
        id  = get_id(var);
  in
    if id?flow then
      lift((flow # id) lub use_var(var) lub use(exp))
    else @
    endif;
```

Assignments of the form `id := exp` *only affect the set of live variables if the variable* `id` *is alive after the statement (*`id?flow`*). Then* `id` *is removed from the set of live variables (*`flow # id`*) and the variables used in* `exp` *and on the left side of the assignment (for array index expressions) are added.*

Example 6.11 (Interval analysis cont.)

```
TRANSFER

M_Assign(M_simpl_var(id),exp),_ :
  let f <= @; in
    lift(f\[val-Identifier(id)->eval(exp,f)]);
```

Here for an assignment of the form `id := exp` *and any kind of edge type the value of the identifier* `id` *is bound to the result of the* `eval` *function which evaluates* `exp` *in the environment* `f` *to an interval (see above). If the incoming flow value is top or bottom the outgoing one will be the same (*`f <= @`*).*

```
M_While(M_Binop(M_op_leq()
                ,M_Var_exp(M_simpl_var(id)),exp),_)
```

```
        ,true_edge:
let f <= @;
    id = val-Identifier(id);
in
  let erg = f{id} glb (bot, (eval(exp,f))!2!2);
  in if is_ok(erg)
     then lift(f\[id->erg])
     else bot
     endif;
```

Consider a statement `while(id <= exp)` *and the true case. First the expression* `exp` *will be evaluated in the environment* `f`*. From this interval the upper bound is taken (*`!2!2` *is the second element of a pair). Then a new interval is formed with this result and with $-\infty$ as lower bound, because the value of* `id` *has to be smaller than the highest value* `exp` *may assume. Now this new interval is intersected with the interval to which* `id` *is bound to. Finally, it has to be tested whether this (*`erg`*) forms a valid interval. If this interval is empty the true case of the conditional statement cannot be reached which is expressed by propagating bottom to it.*

6.5 Analyzer Description

In order to generate analyzers some additional parameters can be specified:

1. `direction`: values can be `forward` or `backward`. This specifies whether the data flow is along the edges of the control flow graph or in the opposite direction.
2. `carrier`: it has to be followed by a name which is the type of the abstract domain that was defined in DATLA, i.e. the type of the flow values.
3. `combine`: the value is the name of a FULA function that merges information which comes over different edges. The default is the lub.
4. `init`: is followed by an expression describing the initial value that is associated with every control flow node. Usually, this is the neutral element of the combine function. Default is bottom.
5. `init_start`: this is followed by an expression describing the initialization value for the start node or the end node for a backward problem.
6. `equal`: this optional parameter is followed by the name of a FULA function to test for stabilisation (usually the equality).
7. `widening`: followed by the name of the widening function (optional).
8. `narrowing`: followed by the name of the narrowing function (optional).

Example 6.12 (Strongly live variables cont.)

```
PROBLEM strongly_live
direction : backward
carrier   : var
init_start: lift(bot)
```

Strongly live variables is a backward problem. It is assumed that at the end node no variables are alive. Therefore, `init_start` *is the empy set of live variables, whereas all other nodes are (by default) initialized with* `bot` *which indicates non reachable paths.*

Example 6.13 (Interval analysis cont.)

```
PROBLEM interval
direction : forward
carrier   : dom
init_start: lift([->(dual(0),0)])
widening  : wide
narrowing : narrow
```

It is assumed that at the start node all variables are set to zero. Therefore, the initial value is an environment that maps all variables to the interval from zero to zero.

6.6 PAG Command Line Parameters

The iteration algorithm to be used is specified as a command line parameter to PAG. Up to now there are four different algorithms implemented:

1. the workset algorithm from Fig. 2.5
2. algorithm 1. with the optimization from Sec. 2.5.2
3. algorithm 2. based on extended basic blocks (see Sec. 2.5.4)
4. algorithm 2. based on effect calculation (see Sec. 3.2)

The interprocedural technique of the first three algorithms is based on the static call graph approach. All algorithms can be combined with widening and narrowing.

6.7 Compiletime and Runtime Options

The behavior of the analyzers can be influenced by a number of runtime parameters. This includes

- `global_node_ordering`: this integer variable selects the node ordering to be used for the workset (see Sec. 2.6)
- `global_select_connector`: this runtime variable can be used to select a connector. Up to now the tree connectors from Sec. 3.4.1 and the VIVU connector are implemented.
- `global_call_string_length`: this variable selects the length of the call string, if the full call string connector has been selected, and the limiting (K) if the VIVU connector is applied.
- `global_vivu_unrolling`: this variable selects the number of loop unrollings (L) in the VIVU approach.

Other options and compile time flags to influence the quality of the code, the memory management, and the graphical debugger are described in Chap.7.

Chapter 7

Implementation

This chapter describes the implementation of PAG, the runtime system, and additionally available packages. It focuses on the overall structure and the modularization of the system. Of special importance for the designing of PAG were the aspect of modularizing the system together with the design of an efficient architecture. The underlying algorithms will be sketched and they will be described more detailed if they are either of special importance or are solved in a non standard way. The algorithms to solve a data flow problem on a control flow graph have already been described in Chap. 2 and Chap. 3.

7.1 Portability

The PAG distribution features a configure script created through the autoconf tool (Mackenzie, 1996) which tries to figure out important system characteristics such as the size of pointers, the byte ordering, compiler characteristics and the location of include files and libraries. Usually, no manual configuration is required to install PAG on a UNIX system. PAG and its generated optimizers have successfully been tested on various systems including SunOS, Solaris, NetBSD, Silicon Graphics and Linux. A version for Windows NT is currently under development.

7.2 The Core Modules

7.2.1 Overview

PAG is implemented in ANSI C in a modular way. The goal of this modularization was to enable the replacement of each part of the PAG system by other implementations without interfering too much with other parts of the system. Of course, the range of different implementations for the modules is bounded by the interfaces between the modules and the design decisions described in Sec. 5.1.

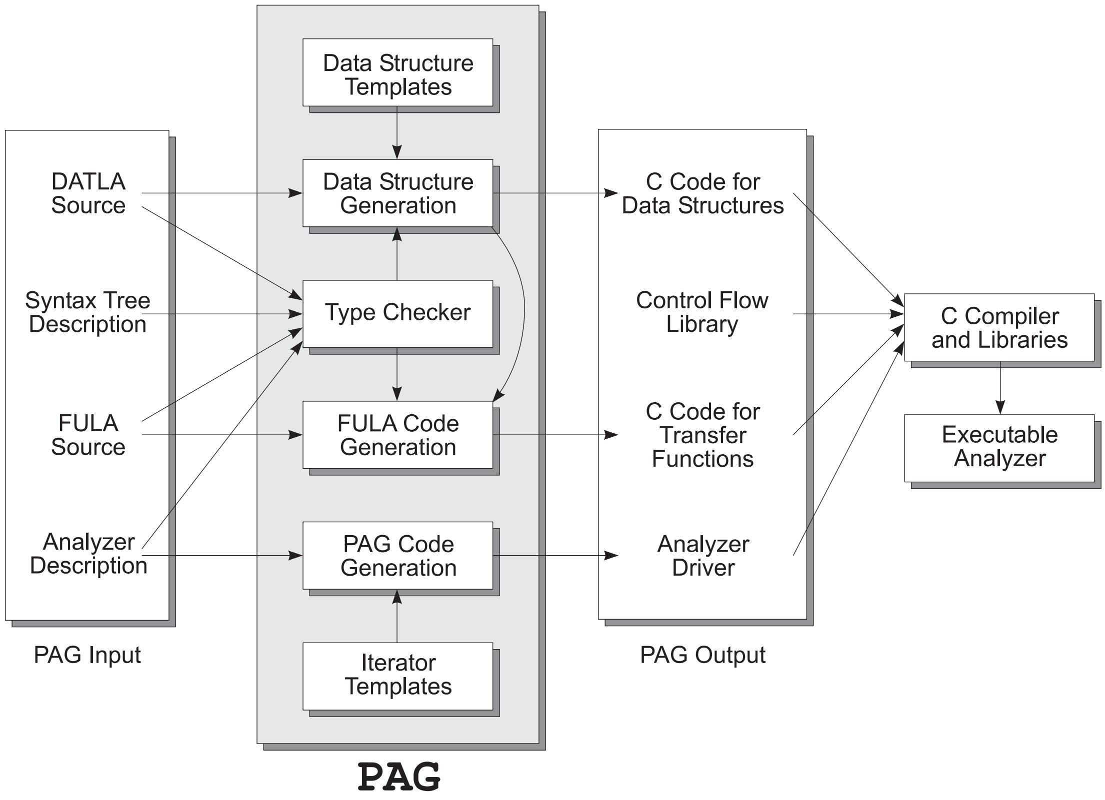


Figure 7.1: The structure of PAG

Figure 7.1 shows an overview of the module structure of PAG. There are four different kinds of inputs: the datatype definitions, the grammar underlying the abstract syntax tree, the definition of the transfer functions and the analyzer parameters. PAG works as follows: first a type checking of all four parts of the input is performed to ensure the type correctness and to calculate the type information needed in the subsequent phases. Then the generation of the data structures, the compilation of the transfer functions and the generation of the equation solver follow.

7.2.2 The Type Checker Module

The implementation of the type checker is based on the polymorphic type inference algorithm of (Cardelli, 1987). A polymorphic type inferencer was chosen since it allows to infer a type automatically without the need for user annotations.

The algorithm derives a type expression for every expression in the functional language. These type expressions are either basic type names, type variables, or type constructors applied to a fixed number of arguments. A type constructor constructs either an n-tuple, a static function (FULA defined), a dynamic function (DATLA defined), a set, a list, a sum, a flat/lift type or a syntactic list. Types introduced from the abstract syntax tree (except for lists) are treated as basic types. Other basic types are snum, unum, bool, real, chr, string, interval types and user defined enumerations.

Whenever an application of a function or a built-in operator occurs the algorithm tries to unify the type expression for the function with the types of the arguments and the return value. In order to keep track of the types for names bound in an outer scope a type environment is used. This type environment is initialized by the types of the built-in functions. The representation of type expressions has been chosen such that type variables are represented uniquely, i.e. there is only one occurrence of the same type variable in the whole type structure. To instantiate a type variable v during the unification process with a type term t, an instance field in the representation of the type variable is used: whenever this instance field is not empty the type variable is equivalent to the type where its instance field points to. This indirection frees from the necessity to detect all references to the same type variable in the type structure built so far: they all are instantiated at once.

To create polymorphic type variables, as they are needed for built-in operations (e.g. length of a list) a list of generic type variables is calculated by the type checking algorithm for each point of the functional program. This list is used when retrieving types from the type environment to generate new (fresh) copies of the generic type variables, so that they can be instantiated in different ways at different places in the functional program.

The basic polymorphic type inference algorithm has a number of restrictions:

- To construct a type expression for a program expression it must be evident from the syntax which type constructor has to be used. E.g. ZF expressions to construct lists or sets have to be syntactically different in order to construct either a list type or a set type.
- The arity of a type constructor must be known, e.g. the arity of a tuple must be derivable from the syntax when applying the tuple selection function: `t!n!2` selects the n-th element from the pair t.
- The algorithm does not allow the automatic insertion of type casts in the program, so that each conversion between two different types has to be made explicitly.
- There is no overload resolution mechanism. The few functions that are overloaded in PAG like *plus* have a universal type like `*,* -> *` in the initial type environment. An additional test is performed to ensure that the type variables for each occurrence of the function are bound to types allowed for this operation (e.g. snum or interval types).
- The place in a program where the unification fails is not necessarily the place of the type error. Therefore it is sometimes difficult to give the user informative feedback.

These restrictions can be circumvented by using more sophisticated type concepts and a different unification strategy, or extensions of the existing type checking algorithm. Parts of this work have already been experimentally implemented in the web interface to PAG (see Sec. 7.4), and are currently integrated into PAG itself (see Chap. 10).

7.2.3 The Data Structure Generation

The Functionality to be Generated

For each type defined in DATLA an *abstract type* in C is created. Abstract types consist of a type definition and a set of functions called its *signature* to access the abstract type. The members of abstract types are called *objects*. Each object represents a FULA *value*, but a value can be represented by different (even structurally different) objects. As a convention in PAG, a function f of the signature of an abstract datatype T is implemented by a C function called T_f. The signature can be divided into several parts. One is the general part which imposes some basic object handling functionality.

Some of these functions are:

- `void T_init(void)`: initialize the abstract type `T`, especially the global variables of `T`
- `void T_duplicate(T x)`: create a new shallow copy of an object `x` of type `T` (the subobjects of `x` are not copied)
- `BOOL T_eq(T x,T y)`: tests whether the two objects `x` and `y` of type `T` represent the same value (not just pointer equality)

Another part is used for the printing of an object. To enable the user of PAG to define his own versions of these functions their definitions are protected by compile time flags via `#ifndef`. E.g. for a numeric type which represents variables of the program to be analyzed its print function can be overridden by a function printing the variable name.

- `char *T_print(T x)` and `void T_print_fp(FILE *f,T x)`: print the textual representation to either a temporarily allocated string or a file. These functions are protected by the compile time flag `T_WRITE`.
- `char *T_vcgprint(T x)`: create a vcg representation of `x`. This function is protected by a separate compile time flag `T_VCGWRITE`.

There is a separate part of the object signature which is required whenever the abstract type represents a lattice (is defined in the LATTICE section of DATLA). This part contains functions to compare two objects and to build the least upper (greatest lower) bound of two objects.

Another part of the signature depends on the DATLA functor to be implemented: for each functor there is a set of functions to be contained in the signature of the abstract type created through the functor. E.g. for a type `T` defined as `T = set(T1)` functions like `T T_add_elem(T,T1)` to add an element of type `T1` to a set of type `T` have to be in the interface of the abstract type `T`. For details see (Thesing et al., 1998).

How the Generator Works

All DATLA functors with fixed arity are implemented using templates. Since in ANSI C there is no built-in possibility to use templates, PAG uses its own template mechanism: the template instantiator replaces all occurrences of the meta variables `$T`n in a template file by some given strings representing the argument types for the functor.

Functors with variable arity (tuples and disjoint sums) are implemented by functions written in C which emit the implementation of the abstract datatype directly.

The Object Representation

Objects of abstract types are placed in the PAG runtime heap. The functions do not deal with the objects themselves but only with references to the objects. It is also possible to replace references to objects by the objects themselves if they are of the same size (or smaller) than an object reference (*unboxed representation*). Since the implementation of a type is fully encapsulated the decision of representing it either boxed or unboxed does not touch any other abstract type.

In PAG the heap representation of an object can be shared, i.e., several pointers to the same object can exist. Therefore it is not allowed to change an object[1]. So each function updating an object has to make a copy of the object in advance. Since this is expensive (time and space consuming) a destructive update can be performed whenever it is ensured that only a single reference to the object exists. This information can be either obtained from an analysis over the FULA program[2] or arises from the structure of the generated code for some parts of the translation process. The functions performing a destructive update are marked by the suffix `_destr` of the function name.

The General Representation of Sets

PAG offers infinite data structures like snum or reals as well as sets over these types. Therefore a way of representing sets with an infinite number of elements has to be found. In PAG not all sets with an infinite number of elements can be constructed. All possible constructions are starting from the sets which can be expressed directly in FULA. These are the empty set (bottom), finitely enumerated sets, and the set containing all elements (top). All other sets occurring during the execution of a PAG generated analyzer are constructed from these sets by a finite series of basic operations (insertions, deletions, intersections, and unions).

An example for a set which cannot be constructed in this way is the set of all odd numbers.[3]

In PAG sets are represented as a (finite) enumeration of elements and a flag. The flag indicates

[1] It is allowed to replace an object by another object representing the same value.

[2] This is not implemented yet.

[3] The set of the odd numbers can be constructed in FULA either by a ZF expression or by recursive functions. In both cases an infinite number of basic operations has to be performed and so the construction does not terminate in either case.

Union:

Argument representation	$\oplus$	$\ominus$
$\oplus$	$e_1 \cup e_2,\ \oplus$	$e_2 \setminus e_1,\ \ominus$
$\ominus$	$e_1 \setminus e_2,\ \ominus$	$e_1 \cap e_2,\ \ominus$

Intersection:

Argument representation	$\oplus$	$\ominus$
$\oplus$	$e_1 \cap e_2,\ \oplus$	$e_1 \setminus e_2,\ \oplus$
$\ominus$	$e_2 \setminus e_1,\ \oplus$	$e_1 \cup e_2,\ \ominus$

Figure 7.2: Operations on two sets $s_1 = (e_1, r_1)$ and $s_2 = (e_2, r_2)$. Each row stands for a representation flag r_1 of s_1 and each column for a representation flag r_2 of s_2. The entries show the operation to be performed on the enumerations of the sets and the representation of the resulting set.

whether the set contains exactly the enumerated elements (*positive representation* $\oplus$) or whether it contains all elements of the element type except the enumerated ones (*negative representation* $\ominus$).

Insertions and deletions are expressible by adding and removing elements from the enumeration without changing the representation flag.

To implement union and intersection four cases are considered depending on the representation of the two arguments (see Fig. 7.2). Three operations on the underlying enumerations are needed: union $\cup$, intersection $\cap$, and subtraction $\setminus$ ($e_1 \setminus e_2$ denotes all elements from e_1 without those from e_2).

For sets over finite types it is important to know whether the argument type of the set functor has finite size, since then the same set can be represented in two different ways (positive or negative). If the size is finite it is also important to know the size.

For example if `nset` is defined as:

```
num   = [0..1]
nset  = set(num)
```

then the set $\{0\}$ can be represented as $(0, \oplus)$ or $(1, \ominus)$.

The General Representation of Functions

Total functions from A to B are implemented by an enumeration of pairs from $A \times B$ and an additional default element from B. The function represented by $(a_1, b_1), \cdots, (a_n, b_n), b_d$ is defined as

$$\begin{array}{rcl} f & : & A \to B \text{ with} \\ f(x) & = & \begin{cases} b_i \text{ if } x = a_i \\ b_d \text{ otherwise} \end{cases} \end{array}$$

The representation satisfies the invariant $\forall i : b_i \neq b_d$ and all a_i are pairwise disjoint. For this type of function representation, the class of representable functions cannot be left by any FULA operation.

For functions over finite types the size of this type has to be known.

The Various Functor Implementations

There are several implementations for most of the DATLA functors. Each implementation can be parameterized to restrict the signature to be generated: e.g. it is possible to leave out the lattice part of the signature. Some functor implementations require additional prerequisites such as the existence of a total order on some arguments. For each functor there is at least one generally applicable implementation.

As explained above objects may be represented in an unboxed way. This does not only save space in the heap, but it also allows faster access to the objects. Therefore, it is desirable to represent as many objects as possible as unboxed values. The basic types snum, unum, bool, chr, and interval types are represented in an unboxed way. Even some functor types admit unboxed representations if their argument types are of restricted size, i.e. have only a restricted number of elements. This applies to the functors lift, flat, tuple, and for some implementations even to set[4].

For the functors set, function, and list there are several different implementations which are described below. For the other functors there are only a straight forward implementation and an unboxed implementation.

The various set and function implementations are based on four data structures: AVL trees, red black trees, dynamic hash tables and binary decision diagrams (BDD). For all of them a persistent implementation was chosen which allows updating operations without copying the whole data structure, but only a small part e.g. making only a copy of that path of a tree to the leave which has been changed and sharing the rest of the tree.

The following implementations for sets are possible:

standard: the set is represented as a flag indicating negative or positive representation and an unsorted list of elements as the enumeration. This implementation imposes no prerequisites on the argument type.

AVL tree: consists of a flag indicating negative or positive representation and an AVL tree of elements to implement the enumerations efficiently. It requires a total ordering on the argument type.

red black tree: similar to AVL tree.

hash tables: uses a dynamic hash table to represent the enumeration of elements. It requires that the hash function of the argument type yields the same hash value for all equivalent elements. In the actual implementation this is not the case if two (or more) different representations for the same value exist, e.g. for sets over finite types.

[4]Using a bit vector of the maximal length of a machine pointer.

Operation	Std.	AVL	red black	hashing	Array	Bit vector
search	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$
insert	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$
delete	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$
insert pers.	$O(n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
delete pers.	$O(n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
union pers.	$O(n_1 * n_2)$	$O(n_1 \log n)$	$O(n_1 \log n)$	$O(n^2)$	/	$O(n)$
meet pers.	$O(n_1 * n_2)$	$O(n_1 \log n_2)$	$O(n_1 \log n_2)$	$O(n^2)$	/	$O(n)$
minus pers.	$O(n_1 * n_2)$	$O(n_1 \log n_2)$	$O(n_1 \log n_2)$	$O(n^2)$	/	$O(n)$
generator	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	$O(N)$	$O(N)$

Figure 7.3: Runtimes for operations in the various functor implementations: for operations with two arguments n is the sum of the sizes of the arguments n_1 and n_2. N is the maximal size of n.

bit vector: a bit vector can be used to represent a set over a finite argument type T if there exists a bijective function from $[1..\text{size\_of(T)}]$ to T, which can be calculated efficiently.

BDD: a BDD may be used to represent a set via its characteristic function. Therefore, it is required to have a finite argument type which can be represented as a vector of Boolean values. This is true for bit vectors and all types that are represented unboxed.

Functions have also been implemented by AVL trees, red black trees and hashing to efficiently store the enumeration of argument/value pairs. The same restrictions as for sets apply to the first argument type of the function functor. To implement function types, BDDs can be used whenever the argument type is either a bit vector or has an unboxed representation and the result type is either bool or has two elements. Finally functions can be implemented using arrays when the first argument has an unboxed representation.

In PAG lists can be implemented either in a standard way or by using a blocked representation. In this blocked representation N consecutive elements are allocated together in one list node, i.e. the list consists of linked blocks where each block can contain at most N elements. For this representation insertion and deletion are more expensive than for the standard implementation. But in most cases list traversal is faster since this representation will usually show better data locality and therefore make better use of the cache (Rubin et al., 1999).

Figure 7.3 summarizes the complexity of the most important operations for the different implementations. BDDs are not listed, since their runtimes cannot be easily compared with the other implementations. For PAG the persistent operations, where the original object is not destroyed, are of special importance. The operation ‘minus’ is used to realize union/intersection for certain combinations of negative/positive representations of sets (see Fig. 7.2). The operation ‘generator’ is used for the generator expressions in ZF expressions.

Choosing a Functor Implementation

The task to choose a concrete implementation is performed by the DATLA generator. It constructs a tree for each type to be generated, where the nodes are labeled with functors and the children correspond to the arguments of the functors. In a bottom up pass over the trees information is being collected about the types which is used to select the implementations (or using the user supplied choice).

The collected information includes the existence of a total order and size information. For the size information three cases can be distinguished: the size is finite and can be computed at generation time, the size is finite but depends on (analyzer) runtime values such as the number of variables in the input program, or the size is known to be infinite.

Depending on this information an unboxed representation is chosen whenever possible. After that choice the following preference list applies for sets and functions:

1. if the size of the (first) argument type is finite, use a bit vector implementation for sets or an array implementation for functions
2. if the (first) argument type has a total order, an AVL tree is used
3. if the size of the (first) argument type is finite, a hash table is used
4. the standard implementation is used otherwise.

Neither the BDD implementation nor the red black trees are used automatically since it turned out that they are useful only in special cases which are not detected automatically. For a further discussion see (Lauer, 1999) and Chap. 8.

Collecting information about the application frequencies for the operations on different abstract types could be helpful in making better choices (Schonberg et al., 1981). One could imagine to use either a static analysis of the FULA code or to provide a profiling mechanism to obtain the information, but this is currently not implemented.

7.2.4 The Garbage Collector

Heap memory is allocated from the C heap in blocks of a fixed size (usually 8K). There are two different kinds of garbage collection which may be used for the abstract types: a mark and compact garbage collector and a reference counting garbage collector. Both collectors work on distinct memory partitions, and work together so that the number of references from the mark and compact area to the reference counting area is updated after each mark and compact garbage collection.

The mark and compact garbage collector is used by default. The collector works in three phases: the mark phase marks all objects in the heap which are still reachable from the stack. This is

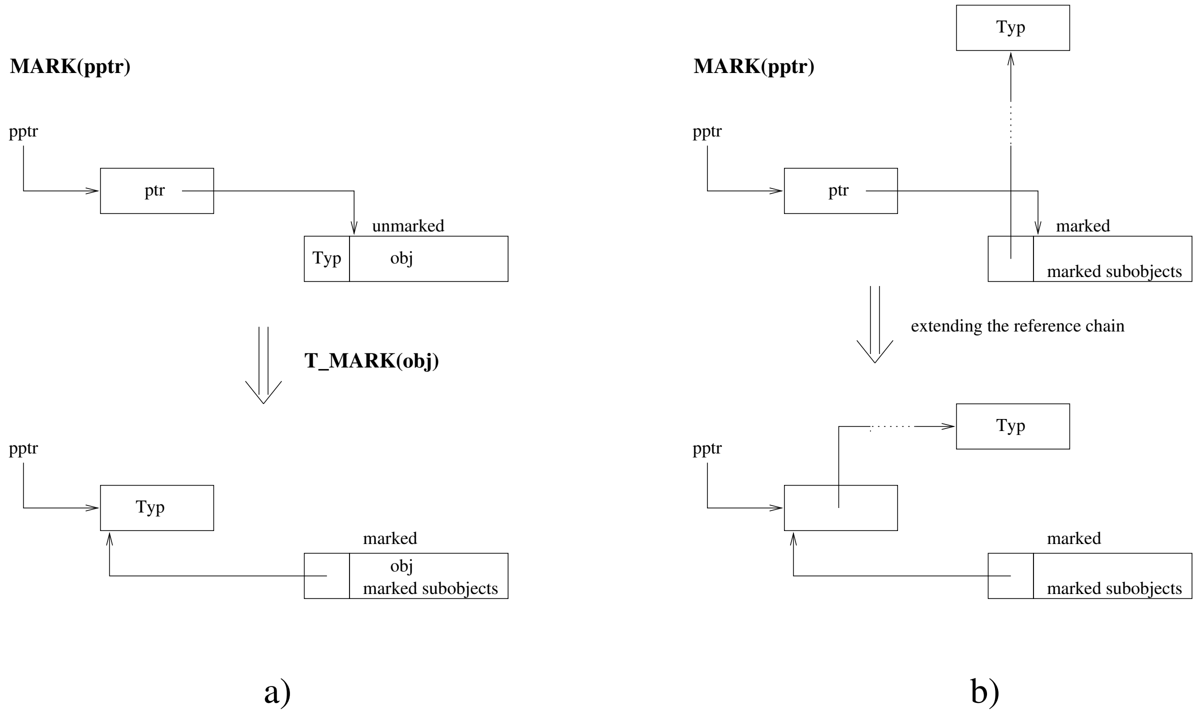


Figure 7.4: The mark phase during garbage collection. a) marks an unmarked object starting a new reference chain, b) adds a new pointer to the reference chain of an already marked object

implemented by recursive procedures. In this first phase reference chains are built up which allow to find all pointers that point to the same object by the classical pointer reversal technique (Schorr and Waite, 1967). One additional pointer per heap object is needed that contains the type information for the object and is the start of the reference chains during the garbage collection (see Fig. 7.4). In the second phase the new place for each object is calculated and the pointers to the object are updated. During this update the pointer reversal is undone. In the last phase the live objects are moved to their new places.

For the mark and compact collector in PAG the portable way of determining all pointers pointing from outside to the heap is to restrict the garbage collection to points in the equation solving process where the FULA stack is empty, so that the root set is the set of all variables in the equation system together with the static references from the C stack. For each abstract datatype, information about the kind of allocated heap space is registered in the garbage collector: the size of the allocated area and a function pointer to the mark function for objects of type T.

In case that heap space is very short, the memory management has a sharing phase which allows to redirect all references to identical copies of the same object in the heap to a single instance of this object. The runtime of this sharing phase is quadratic in the size of the actual heap and should be used only if necessary.

There are several runtime parameters for the generated analyzers to control the garbage collection and the sharing phase which allow to influence the frequencies of the different phases:

- `low_perc`: whenever less than `low_perc` percent of the actual heap are free a garbage collection is triggered
- `high_perc`: if after a garbage collection less than `high_perc` percent of the actual heap are free the heap is enlarged
- `share_min`: the minimum number of memory banks (usually 8K) which have to be in use to trigger a sharing phase
- `share_num`: the minimal number of garbage collections that have to be performed between two sharing phases

It seems to be useful to do further investigation in the area of garbage collection such as using generational techniques (Jones, 1996).

7.2.5 The Language FULA

As stated in Sec. 5.1 FULA was designed to be translated directly to C. So each function in FULA corresponds directly to a C function and the FULA runtime stack is represented by the C stack.

Using C as implementation language has two major advantages: the produced code is very portable and one can rely on the optimizations of the underlying C compiler. For instance it is not necessary to avoid dead assignments when producing the code for pattern matching, since they are removed automatically by most optimizing C compilers.

The translation of FULA into C can be described with recursive code functions in the style of (Wilhelm and Maurer, 1997). The code functions are a set of simultaneously recursive functions which take (parts of) the abstract syntax tree of a FULA program and produce code (C code in this case) through mutually recursive calls. Additionally, an environment is passed around which maps FULA variables to C variables bound in an outer scope. There are several code functions called in different contexts: the F_Code function to translate FULA function definitions to C function definitions, the V_Code function to generate C statements for FULA expressions which evaluate the expression and store the result in a C variable, the E_Code function to translate FULA expressions into C expressions, the P_Code function to translate pattern matching, and the Z_Code function to translate ZF expressions.

The definition of these functions is mostly straightforward. Therefore only the interesting parts are highlighted here. The complete definitions of these functions may be found in (Martin, 1995).

After parsing the FULA source a number of transformations is applied: first, the different cases for a function definition are grouped together, then a number of high level FULA constructs such as the strict binding in let expressions are translated on a FULA to FULA level. After this step, only the core of the FULA language has to be translated through the code functions.

Before the translation itself can be started, it has to be determined for which expressions the E_Code function can be applied: only those expressions can be translated to C expressions which do not introduce new variable bindings. So it is determined in a bottom up run over the FULA syntax tree, which expressions have ZF-, let-, or case expressions as subexpressions. These expressions have to be translated with the V_Code function.

In the F_Code function it is possible to build hash tables or caches to memoize the result of a function application. The use of these features has to be indicated by the user in the FULA code[5], and may speed up the calculation dramatically. To cope with the case-wise definition of functions the F_Code function calls the P_Code function. Also the tail recursion optimization is handled by the F_Code function: directly recursive functions are translated to the creation of a new variable binding and a goto to the begin of the function.

The P_Code function translates a pattern matching through recursive calls into a series of if-then-else statements and assignments to create the new variable bindings. The implementation follows the description in (Jones, 1987).

The Z_Code function translates ZF expressions directly to possibly nested while loops and uses destructive updating of the constructed data structure to be more efficient. Each nested while loop results from a generator in the ZF expression. It is required that the implementations of the functors set, function and list contain a generator signature which has to define a type that is called `T_cur` to store information about the iteration process. Objects of this type are allocated directly on the stack. Additionally, four functions have to be implemented:

- `void T_cur_reset(T_cur *c, T x)`[6]: initializes the `T_cur` object pointed to by `c` to generate all elements contained in `x`.
- `BOOL T_cur_is_empty(T_cur *c)`: tests whether the last element has been reached.
- `T T_cur_get(T_cur *c)`[7]: returns the actual element.
- `void T_cur_next(T_cur *c)`: advances the cursor to the next element.

For all these functions it is assumed that the set (or list or function) from which the elements are generated has not changed during the iteration process. For sets and functions the order in which the elements are generated does not matter.

Since sets may be represented negatively there has to be a possibility to enumerate all elements of a type T in order to implement a cursor interface. This additional part of the signature of all abstract types T is called the all-cursor interface and is the same as the cursor interface, except that all elements of a type are generated[8].

[5]For the simple case where the function has no arguments memoization is used automatically.

[6]When `T` is defined as `T = A -> B` the function requires an additional argument b_d of type `B`: all pairs `(a,b)` will be generated where $b \neq b_d$.

[7]When `T` is defined as `T = A -> B` the return type is pairs of `A` and `B`.

[8]This generation may be infinite.

7.2.6 Equation Solving

The template mechanism is also used to generate the equation solver. The templates for the iterative equation solvers are instantiated with the type of the abstract values and the other data flow parameters (init, init_start, direction, equality, narrowing, widening).

As described in Sec. 5.1 the equation system is not stored explicitly to save space. Instead the CFG is used as a graph describing the static dependencies. All solvers currently implemented are iterative solvers controlled by a workset which is organized as a priority queue and contains either nodes, edges or pairs of nodes and abstract values.

The priorities are assigned to each node by a separate module before the equation solving. If the workset contains edges the priority of the source node of an edge is used as the priority for the edge.

The modules currently implemented are:

1. the naive approach from Fig. 2.5
2. the optimized version storing edges in the workset from Fig. 2.6
3. the basic block version of the optimized algorithm from Fig. 2.9, 2.10
4. the functional approach from Fig. 3.3, 3.4

The solvers 1) - 3) are based on the static call graph approach.

7.2.7 Integration

PAG also creates a driver for the analysis which calls the different stages of the analysis and takes care of initializing the different modules and does some cleanup afterwards.

The calculated information can be accessed in two ways: either it is retrieved from the equation solver and then accessed by the functions from the signature of the abstract datatype, or FULA functions can be used to do a required post processing.

7.2.8 Inlining

The Gnu C compiler (Stallman, 1998) allows function inlining, which is a very effective optimization for small non-recursive functions. This optimization is limited to intra module inlining. The extensive modularization of PAG leads to a large number of very simple functions, spread over a variety of source files e.g. snum_eq(a,b) which performs only an equality test on ints. In order to enable inlining for these functions over module boundaries the PAG distribution contains an inliner which creates header files enabling the inlining of all functions by the preceding keyword INLINE. As translating a program with extensive inlining can be very costly with gcc the inlining can be switched on and off by a compile time flag.

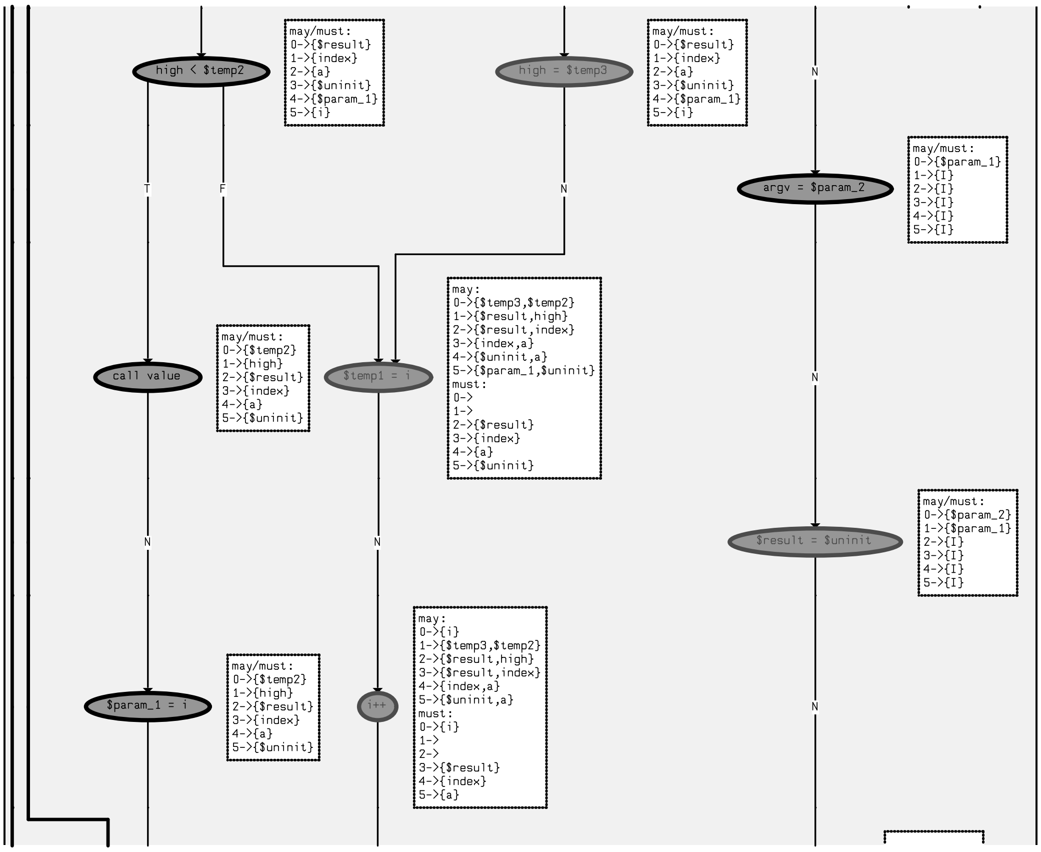


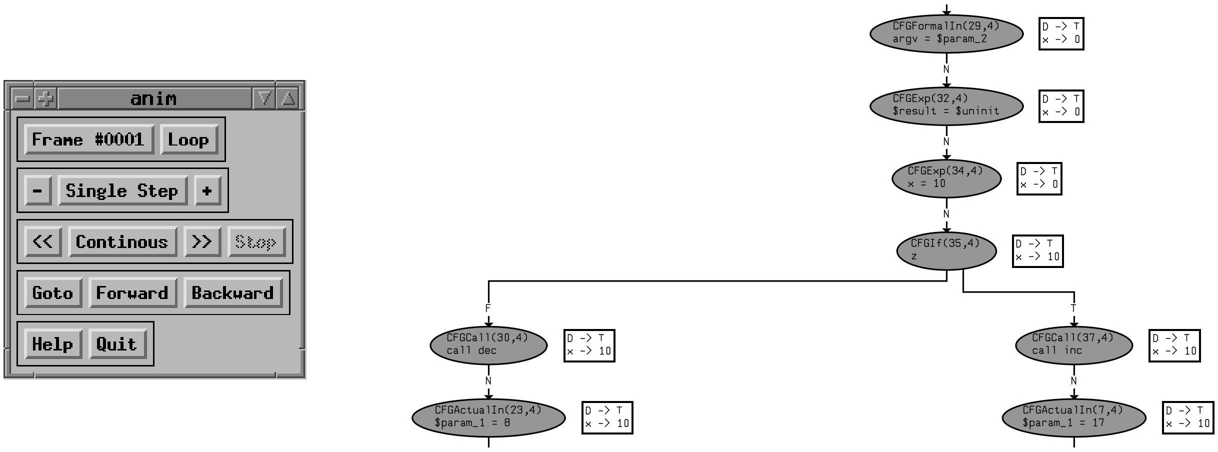
Figure 7.5: Cache behavior prediction

7.3 Debugging and Visualization

The PAG distribution offers an interface to the graphical visualization system vcg (Sander, 1994; Sander, 1996; Sander, 1999).

By the debug facility of PAG, input files can be created for the vcg tool that show the CFG and the calculated data on every node. These files can be created for a single step as well as for a sequence of steps. Those sequences can be animated using anim that is a driver for the vcg tool. (see Fig. 7.6 a) and Fig. 7.7).

For all data structures generated from DATLA a default visualization function is created. For complex structures like heap graphs the user can override these functions to achieve a visualization that represents the intuition behind the data structures more nicely. In Fig. 7.5 a snapshot from debugging the cache analysis is shown. Figure 7.6 b) shows constant propagation and 7.8 shows the debugging of a heap analysis.



a) b)

Figure 7.6: a) The animation tool and b) Debugging constant propagation

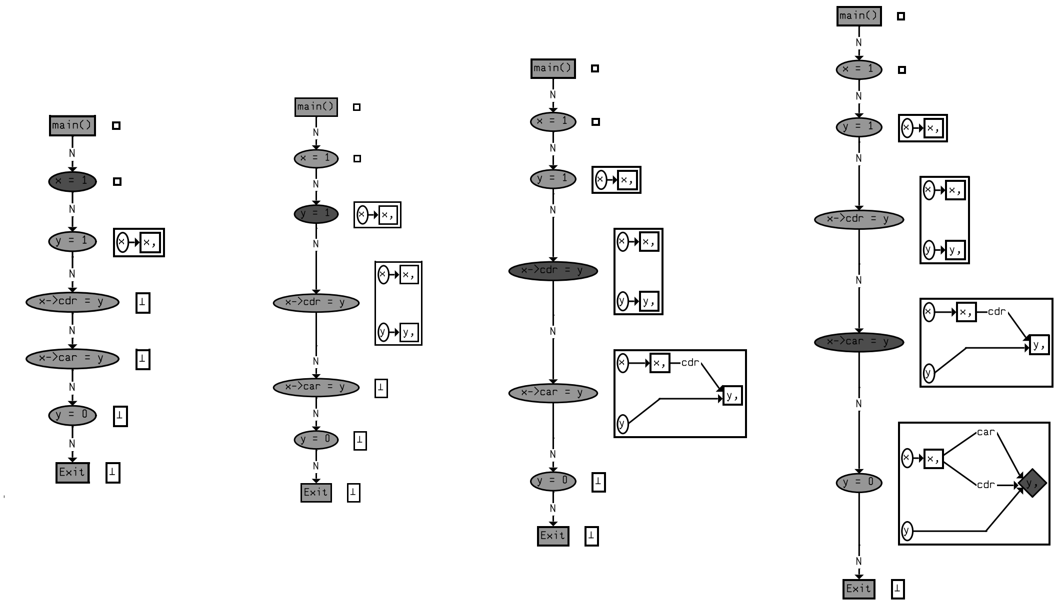


Figure 7.7: Animation sequence for heap analysis

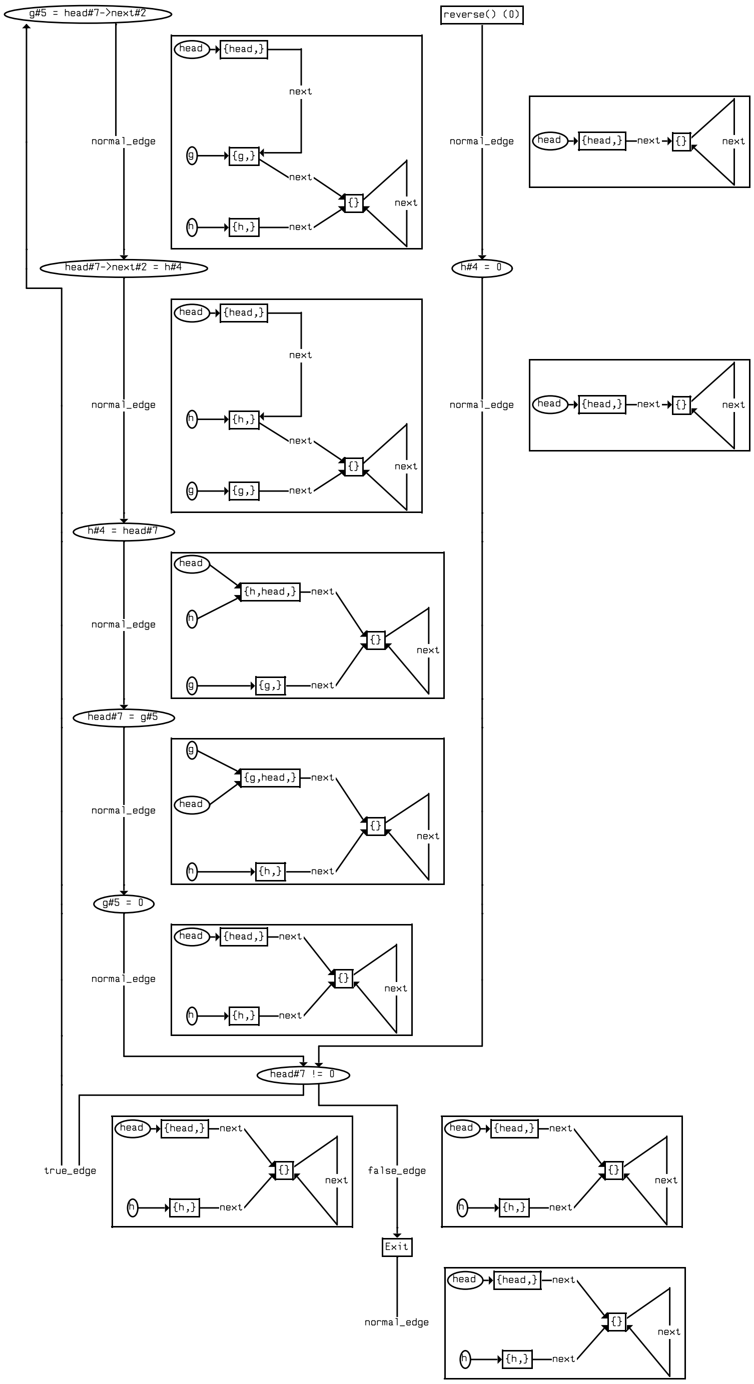


Figure 7.8: Debugging heap analysis

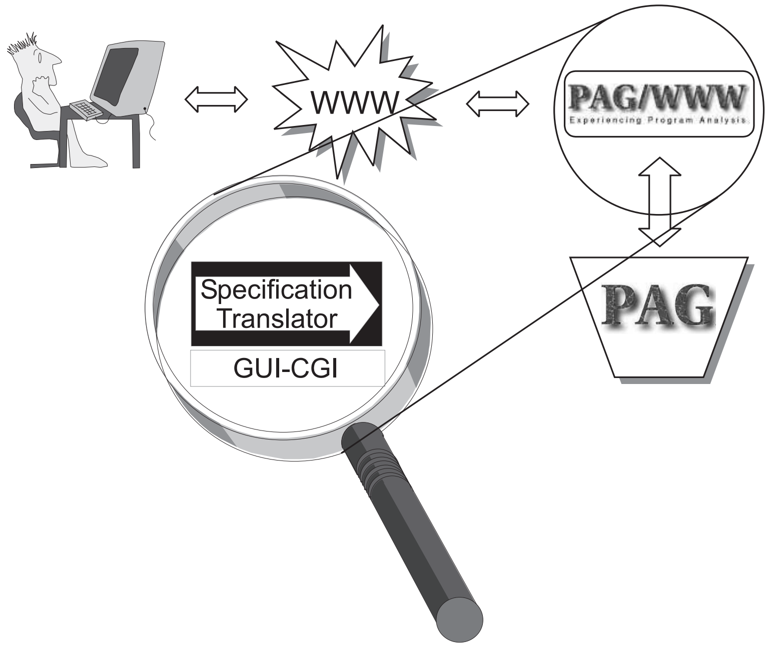


Figure 7.9: The PAG/WWW system

7.4 The Web Interface

The web interface to PAG called PAG/WWW was built to support tutorials on data flow analysis and abstract interpretation and to provide easy access to PAG. PAG/WWW (Bobbert, 1998) was designed on top of the full PAG system with restricted functionality and a fixed frontend (see Fig. 7.4). Additionally, it consists of the WWW gateway using the common gateway interface (CGI). It is implemented in Perl (Wall et al., 1996) and passes the user's input to PAG, processes the output or the error messages, initiates the compiling of the analyzer and finally starts the analyzer creating the visualization of the PAG run. The analysis results and the analysis steps can easily be browsed either in a graphical representation (Fig. 7.10) or in a textual representation (Fig. 7.11).

PAG/WWW has also been used as a test implementation for a new and simpler version of FULA (Fig. 7.12). It has a translator which translates the new FULA syntax into the old one of the PAG system. Most of this translation work is possible since a different type checking algorithm is used, and some features of the full system have been left out. E.g. it is not possible to define disjoint sums in PAG/WWW which simplifies the syntax of FULA. In contrast to the full system it is required to write type declarations for each FULA function, so that the specification of the function can be checked against this declaration, which simplifies the feedback to the user in case of an error.

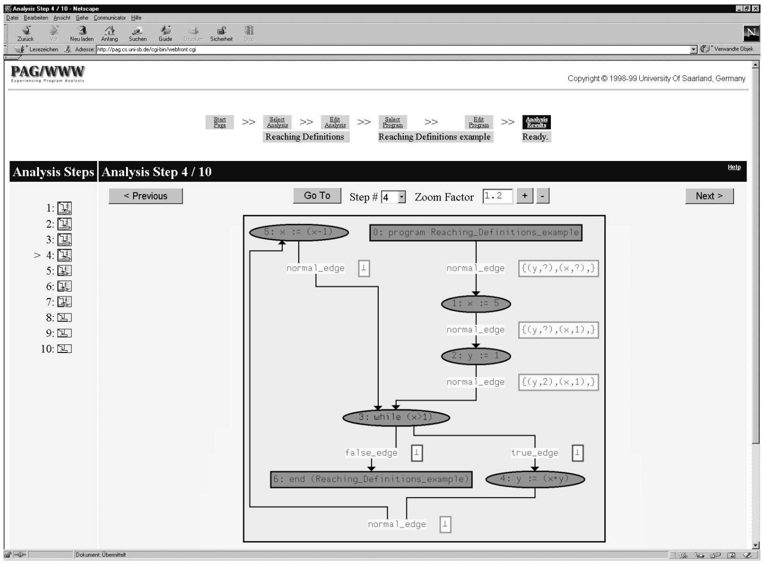


Figure 7.10: A screenshot from PAG/WWW

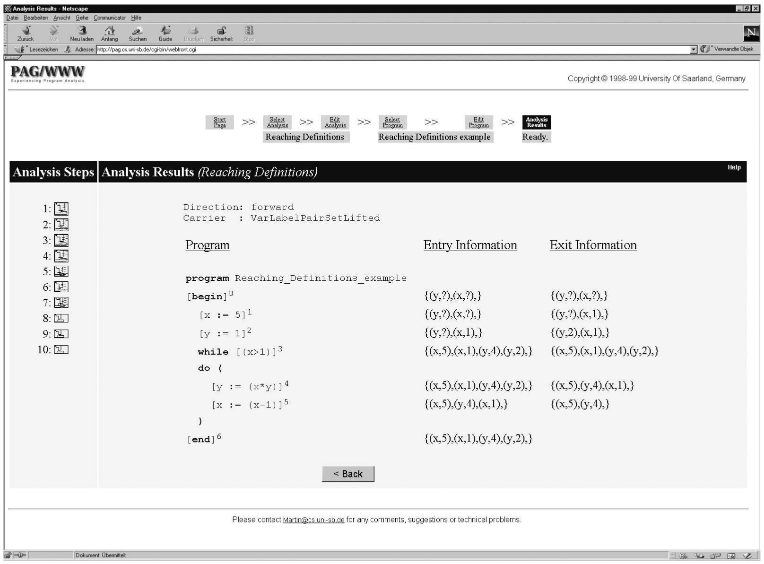


Figure 7.11: A screenshot from PAG/WWW

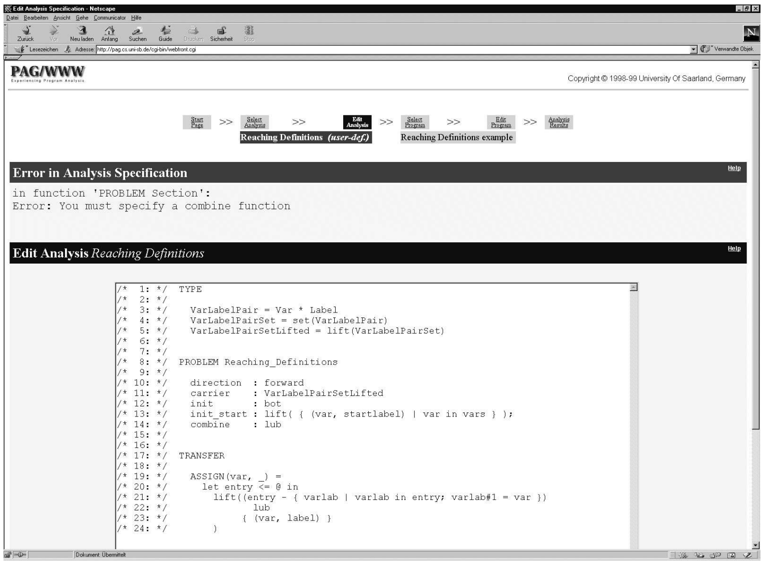


Figure 7.12: A screenshot from PAG/WWW

7.5 Generating Frontends

It turned out that PAG is used in several research projects and for teaching program analysis, where the goal is mostly not to integrate program analyzers into existing compilers, but to test the possibilities of program analysis for different languages. There is a great demand for implementing frontends which are (at first) used only as frontends to PAG. So it is no longer true for these cases that only an interface for an existing control flow graph and an abstract syntax tree has to be written.

Although generators such as flex and bison (Paxson, 1995; Donnelly and Stallman, 1995) are of great help for implementing frontends they still lack support for building abstract syntax trees and constructing control flow graphs.

This led to the development of a tool Gon ('Generator ohne Name' or 'Generator withOut Name') (Schmidt, 1999a) that integrates the generators for the lexical and syntactical analyses and offers specification mechanisms for creating the abstract syntax trees as well as the control flow graph (see Fig. 7.13) and provides the interface to PAG.

The specification formalisms (Schmidt, 1999b) for the lexical and syntactic analysis widely coincide with flex and bison. The specification for building the abstract syntax tree is given as a tree grammar annotation. If it is omitted then the parse tree from the syntactic analysis is built. To construct the CFG, to every production rule in the context free grammar a rule for constructing a CFG is assigned. It describes how the CFG for the non terminal of the left side of the rule is constructed from the CFG's of the non terminals on the right side. With these rules a CFG can be built in a bottom up pass over the abstract syntax tree. Additional mechanisms are needed to

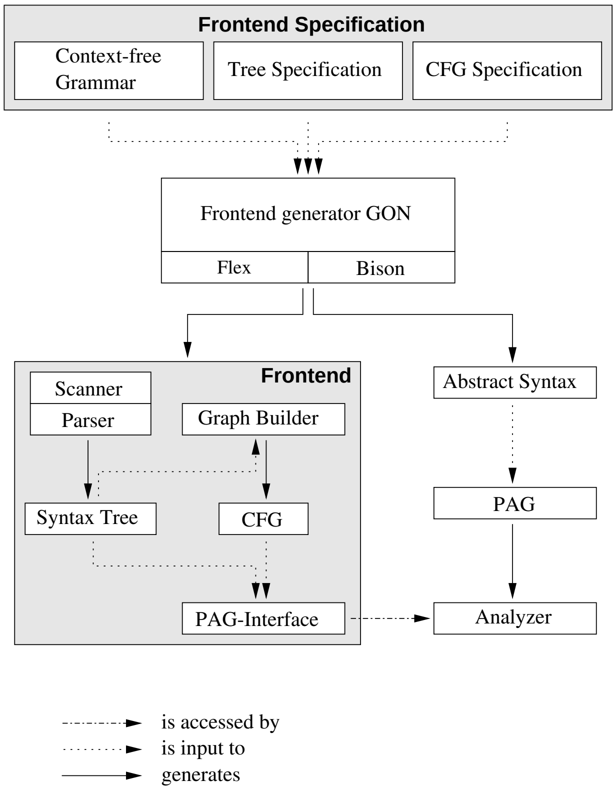


Figure 7.13: An overview of Gon

cover gotos and calls. Therefore symbol tables can be managed by Gon. The user of Gon has to annotate the syntax rules corresponding to the declaration of a label and a procedure as well as the beginning of a new scope. Also the rules corresponding to goto and call statements have to be annotated. After a successful creation of the control flow graph for each goto/call node the matching name in the inner most enclosing scope is searched, and edges to the corresponding nodes are inserted.

Gon aims mainly for imperative programming languages with static scoping. It also offers the possibility to extend the specification with C-Code as it is known from flex and bison.

Gon has been used so far for generating frontends for several languages including Clax, CRL[9] (Ferdinand et al., 1999), Java, and Java byte code.

[9]CRL is an intermediate representation for various kinds of assembler languages.

Chapter 8

Practical Results

In this chapter practical examples are presented. It tries to give an impression of the flexibility of PAG, and its applicability. It also tries to show that the generated analyzers are reasonably fast and applicable to large real world programs. Various analyses that have been implemented using PAG are explained and the sizes of the specifications are listed. Some figures are presented to show the time and space requirements of the generated analyzers. The quality of the analyses themselves is not discussed, because it depends mainly on the analyzer designer. The influence of the various optimizations and analysis concepts discussed throughout this work are exemplified for some analyses and programs.

8.1 Various Analyses

Various analyses have been specified with PAG with several frontends. In the following first the frontends are briefly described, and then a number of the analyses is listed, but this listing is not meant to be complete.

8.1.1 Analyses for C

The PAG C frontend is able to work with programs consisting of several files, but all of them have to be preprocessed by the C preprocessor. Nearly all ANSI C features except for a few not very common ones, such as the definition of a variable and a type with the same name, are supported. In contrast to the Gnu C compiler it is more pedantic with respect to the ANSI standard and rejects programs, e.g., if there are additional semicolons on top level. Thus, it often requires some work to prepare existing programs for the analysis.

Since the frontend has to keep the syntax trees and control flow graphs for all input files (and additional information) it is quite space consuming. Therefore, it can be difficult to analyze very large programs.

The frontend does a number of program transformations which simplify the analysis: the parameter transfer and the handling of the return value for procedures are made explicit by using assignments to temporary variables. Additionally, function calls are moved outside of expressions, so that a function or procedure call only occurs as a separate control flow node.

As an additional advantage the frontend allows the unparsing of a C program. This makes it possible to write C to C optimizers.

Constant Propagation

This analysis (Callahan et al., 1986; Wegman and Zadeck, 1985) tries to find variables in C programs that have a constant value at certain program points for each program execution. A number of variations have been specified: conditional constant propagation as well as unconditional constant propagation, both combined with full constant propagation or copy constant propagation. All variants are interprocedural. The conditional version of the analysis tries to take advantage of conditions, e.g., in the true case of `if (x==0) ... else ...;` the variable `x` is surely `0`. The full version (in contrast to the copy constant problem) needs an expression evaluator to consider each form of right sides of assignments. In the copy constant analysis only those right sides of assignments are taken into account which are either a literal constant or consist of a single variable. The abstract domain for the copy constant propagation is finite (for each program), which makes the functional approach applicable. Figure 7.6 shows a debugger screenshot from this analysis.

All variants of constant propagation are combined with a hand written transformation phase that replaces constant variables by their values and performs simplifications of the expressions enabled by the replacements: e.g. the expression `x+3` is replaced by `6` if the analysis calculates a constant value `3` for `x` at this program point. The simplification also includes the removal of unreachable branches of conditionals whenever the predicate of the condition can be simplified to a constant.

Shape Analysis

This analysis was presented in (Sagiv et al., 1996; Sagiv et al., 1998; Sagiv et al., 1999). It computes information on the layout of data in the heap for imperative programming languages with destructive updating. The abstract domain elements are graphs that represent the union of all possible heap structures. The graph maps variables to abstract heap nodes. These nodes are linked using annotated edges, where the annotations stand for selectors. The heap nodes are labeled with a set of variables that point to this node. Screenshots from the debugger are shown in Fig. 7.7 and Fig. 7.8. The version considered here uses a single shape graph for each node of the CFG (Sagiv et al., 1996). The analysis was specified for a subset of C.

Memory Error Detection

This analysis is a refined version of the shape analysis discussed above (Dor et al., 1998). It uses sets of shape graphs instead of single shape graphs for each node in the CFG. Additionally, the refinements from (Sagiv et al., 1998) are used and conditions with respect to pointers are interpreted.

Among others the analysis tries to detect the deallocation of still reachable storage, the dereferencing of NULL pointers, and memory leaks, i.e., missing deallocation of storage that is no longer used. Like the shape analysis it is implemented for a subset of C.

8.1.2 Analyses for Executables

Several frontends to analyze executables have been implemented. To support the analysis of executables, CRL has been developed. CRL stands for Control flow Representation Language and is a generic intermediate format to represent control flow graphs of different assembler languages. This language was designed as part of the Transfer Project #14. To analyze executables of a certain machine only a translator from the executable format to CRL has to be written. The CRL frontend itself is generated using Gon. The figures presented further down are for Sun executables in `a.out` format. All analyses for executables presented here are used in the worst case execution time calculation of Transfer project #14 of the USES (Universität des Saarlandes Embedded Systems Group) (Ferdinand et al., 1999).

Cache Analysis

This analysis was presented in (Alt et al., 1996a; Ferdinand et al., 1997; Ferdinand et al., 1998) and is discussed in Example 4.2.1. It tries to predict the contents of the cache for each program point. A number of variants have been specified: with or without persistence analysis (Ferdinand, 1997a), multilevel caches, and a variant which takes the different interleavings of multiple processes into account (Kästner and Thesing, 1998). The variant considered here includes the persistence analysis for a one level instruction cache.

Reference Determination

This analysis (Sicks, 1997) is used in the context of data cache behavior prediction and tries to determine address ranges of data references. To do so, the intervals of index variables for arrays are calculated. It has turned out that an interval analysis alone is not very helpful for analyzing executables. Therefore, a combination of an interval analysis and a must alias analysis is used. The analysis can also be used to detect upper bounds on the number of executions of loops. Such bounds are needed for the prediction of the worst case execution path for real time programs.

Pipeline Analysis

This analysis (Schneider, 1998; Schneider and Ferdinand, 1999) predicts the behavior of the pipeline of a microprocessor. It is designed in such a way that the results of a preceding cache analysis can be used, so no explicit consideration of the cache state is needed in the concrete pipeline semantics.

Compared to concrete cache states the concrete pipeline states are usually small. This simply allows to consider sets of concrete pipeline states as abstract domain.

The abstract pipeline update function reflects what happens when a new instruction enters the pipeline. It takes the current set of pipeline states into account, in particular the resource occupations, the contents of the prefetch queue, the grouping of instructions, and the classification of memory references as cache hits or misses.

Pipeline analysis has been implemented for a SuperSparc 1 processor.

8.1.3 Analyses for Clax

Clax (Sander et al., 1995; Alt et al., 1996b) is a small Pascal like language with arrays. It was defined to study the aspects of the implementation of compilers and has been used in a number of compiler construction courses. The language is well suited to study program analyses, since it is not overloaded with features complicating the analysis without giving new insights, but there are only a small number of example programs and no real applications.

Strongly Live Variables

This extended version of the live variable problem was presented by (Giegerich et al., 1981). In assignments of the form `x := a + b` the variables of the right side are only considered to be live above the statement, if `x` is alive after the statement. This makes the analysis invariant under the elimination of unnecessary assignment statements. Parts of the specification are discussed in the Examples 6.1, 6.7, 6.10. The full specification is shown in Appendix A. In contrast to the live variable problem this analysis is not a bitvector framework since it is not distributive.

Interval Analysis

This analysis approximates the runtime values of variables by intervals. It is explained in detail in the Examples 6.2, 6.8, and 6.11. It also has been specified for the While frontend. Furthermore, it has been combined with a number of widening strategies.

Dominator Analysis

This simple analysis calculates the set of dominators for each node in the CFG: A node n is said to be *dominated* by a node m if all paths from the start node of the CFG to n contain m. This analysis does not require any examination of the syntax tree, so it can be used for all frontends.

8.1.4 Analyses for the While Language

The While language is an imperative interprocedural language taken from (Nielson et al., 1998). It was designed to study program analyses. So it concentrates on the important concepts of imperative languages and leaves out unnecessary details. The frontend is the basis for PAG/WWW. The following analyses have been implemented using PAG/WWW.

Classical Bitvector Analyses These are the specifications of the classical four bitvector analyses (Nielson et al., 1998): available expressions, reaching definitions, live variables, and very busy expressions. Here for all problems an interprocedural version has been specified.

Sign Analysis This analysis (Nielson, 1999; Martin, 1999b) tries to detect the sign of a variable. The abstract domain of the analysis is the power set of {-, 0, +}.

Odd/Even Analysis This analysis (Nielson, 1999; Martin, 1999b) tries to detect, whether all possible values for a variable are odd or even. The abstract domain of the analysis is the power set of {even, odd}.

Upward Exposed Uses The upward exposed uses analysis (Nielson, 1999; Martin, 1999b) is the dual of the reaching definitions analysis. It determines for each definition of a variable which uses it might have.

8.2 The Test Environment

All measurements have been made on a Pentium II system with 400Mhz and 256MB of main memory, running Linux 2.0.36 (Red Hat 5.2) and gcc 2.7.2.3. For the measurements the PAG version 0.9.5.20 has been used.

In the following the standard setting for the experiments is described. If an experiment uses different settings this is mentioned explicitly.

The times measured for the analyzers are the user and system times of the whole analyzer run including the time spent in the CFG interface. They depend on the compiler which is used for the integration. The times also include the time for sorting the nodes in the control flow graph for the workset. Not included are the times for parsing and CFG construction.

The space measured is the memory that the generated analyzer allocates from the C heap via malloc. It does not include the space allocated by the frontend or the stack space.

Analysis	Lines	Standard Time [m:s]	Standard Size [KB]	Optimized Time [m:s]	Optimized Size [KB]
copy constant propagation	274	0:08	291.0	0:47	483.0
conditional copy constant p.	651	0:13	307.6	1:01	502.7
constant propagation	621	0:13	307.8	1:00	500.5
conditional constant p.	641	0:13	309.7	1:00	658.2
shape analysis	561	0:16	373.9	9:16	2954.1
memory error detection	1202	0:50	460.8	-	-
cache analysis	144	0:05	138.6	0:12	175.2
reference determination	1361	0:23	234.0	4:36	1023.5
pipeline analysis	805	0:31	212.7	-	-
strongly live variables	101	0:06	156.6	0:37	319.8
interval analysis	155	0:07	164.1	0:40	342.5
dominator analysis	30	0:05	150.7	0:20	246.0
available expressions	240	0:10	165.9	1:03	402.7
reaching definitions	125	0:12	177.4	0:57	426.3
live variables	259	0:10	180.4	0:44	344.7
very busy expressions	253	0:11	168.4	1:01	406.4
detection of signs	266	0:14	197.9	2:10	785.3
odd/even analysis	206	0:14	184.4	1:10	495.6
upward exposed uses	209	0:16	179.2	0:59	442.2

Figure 8.1: The specification size and the creation time

All measurements of program sizes include only non empty non comment lines.

The functor implementations are automatically selected by the generator, i.e., no `use` clause has been specified. The iteration algorithm is the basic block workset algorithm and the ordering is **bfs**. The call string approach with length one is used as interprocedural method.

8.3 The Specification Sizes

In Fig. 8.1 the sizes of the specifications listed above are shown. These are the specifications of the abstract domains and of the transfer functions. The size of the frontend description is not included, since it has to be given only once per frontend. The figure also tabulates the time needed to generate and compile the analyzer with the standard optimization (-O2) and the sizes of the generated analyzers. The time is the user and the system time for the call to `make` measured with the Unix `time` command. The sizes include the whole runtime library as well as the code for the frontend. Furthermore, the figure shows the time to generate and compile the analyzers and the size of the generated analyzers using the PAG inlining mechanism and full optimization of the compiler (-O4). For some analyses the C compiler runs out of memory (denoted by a '-').

Name	Description	Inst.
`avl2`	inserts and deletes 1000 elements in an AVL tree	614
`dhry`	Dhrystone integer benchmark	447
`djpeg`	JPEG decompression (128x96 color image)	1760
`fdct`	JPEG forward discrete cosine transform	370
`fft`	fast Fourier transformation	1810
`lloops`	Livermore loops in C	5677
`matmult`	50x50 matrix multiplication	154
`matsum`	100x100 matrix summation	135
`ndes`	data encryption	471
`stats`	two arrays sum, mean, variance, standard deviation, and linear correlation	456

Figure 8.2: The test programs used for the cache analysis and their number of instructions

8.4 Analysis Times

8.4.1 Cache Analysis

In Fig. 8.3 the time and space requirements of the cache analysis with various parameters and programs are shown. The test programs and their number of machine instructions are shown in Fig. 8.2 with a program description. The cache parameters for the analysis have been chosen according to existing microprocessors: a SuperSparc (20KB, 5 way associative, line size 16B), a Pentium (8KB, 4 way associative, line size 32B), an Intel i960KB (512B, direct mapped, line size 16B), a Hewlett Packard PA7100 (128KB, direct mapped, line size 32B), and a MIPS R4000 (8KB, direct mapped, line size 64B). For the implementation of the cache an array implementation of the function functor has been chosen.

To judge the quality of the generated analyzers Fig. 8.4 shows the time and space consumption for the same analysis with an analyzer, where the abstract domain functionality and the transfer functions are handcoded in C. In this case the implementation is very short since there are only two abstract operations to be performed: *update* to model a reference to a memory location, and *combine* to combine two abstract domain elements. All other analysis parameters did not change compared to the previous experiment.

One can see that the performance gain is about factor four for large programs. Most small programs are analyzable with both –the generated and the hand written– analyzers in less than two seconds which is usually fast enough.

8.4.2 Constant Propagation

This evaluation uses interprocedural conditional constant propagation for C and a set of real world test programs. The programs are described in Fig. 8.5. They are listed with a short de-

Program	SuperSparc		Pentium		i960KB		HP-PA7100		MIPS R4000	
	s	MB	s	MB	s	MB	s	MB	s	MB
avl2	0.1	2.0	0.1	1.5	0.1	2.5	0.1	1.5	0.1	1.0
dhry	0.1	1.5	0.1	1.5	0.1	2.0	0.1	1.5	0.1	1.0
djpeg	2.0	6.0	1.0	4.5	2.0	7.0	0.1	3.0	0.1	2.0
fdct	0.1	1.0	0.1	1.0	0.1	1.0	0.1	1.0	0.1	1.0
fft	1.0	6.0	1.0	4.5	2.0	7.5	0.1	3.5	0.1	2.0
lloops	23.2	19.0	9.1	12.5	23.1	17.0	4.2	10.5	3.2	8.0
matmult	0.1	1.0	0.1	1.0	0.1	1.0	0.1	1.0	0.1	1.0
matsum	0.1	1.0	0.1	1.0	0.1	1.0	0.1	1.0	0.1	1.0
ndes	0.1	1.5	0.1	1.0	0.1	2.0	0.1	1.0	0.1	1.0
stats	0.1	2.0	0.1	1.5	0.1	2.0	0.1	1.5	0.1	1.0

Figure 8.3: Execution times and space consumption for the cache analysis

Program	SuperSparc		Pentium		i960KB		HP-PA7100		MIPS R4000	
	s	MB	s	MB	s	MB	s	MB	s	MB
avl2	0.1	2.0	0.1	1.5	0.1	2.5	0.1	1.5	0.1	1.0
dhry	0.1	1.5	0.1	1.0	0.1	2.0	0.1	1.0	0.1	1.0
djpeg	0.1	6.0	1.0	4.0	0.1	6.0	0.1	2.5	0.1	2.0
fdct	0.1	1.0	0.1	1.0	0.1	1.0	0.1	1.0	0.1	1.0
fft	0.1	6.0	0.1	4.0	0.1	6.0	0.1	3.0	0.1	2.0
lloops	8.1	10.5	3.1	7.5	8.0	11.0	1.0	7.0	1.1	6.5
matmult	0.1	1.0	0.1	1.0	0.1	1.0	0.1	1.0	0.1	1.0
matsum	0.1	1.0	0.1	1.0	0.1	1.0	0.1	1.0	0.1	1.0
ndes	0.1	1.5	0.1	1.0	0.1	1.5	0.1	1.0	0.1	1.0
stats	0.1	1.5	0.1	1.0	0.1	2.0	0.1	1.0	0.1	1.0

Figure 8.4: Execution times and space consumption for the partially hand coded cache analyzer

Program	Description	Lines	Flow Nodes	Procedures	Variables
`bison`	parser generator	6438	11722	155	3575
`cdecl`	C++ compiler part	2831	2401	33	841
`cloop`	benchmark	1488	2176	26	622
`dhry`	dhrystone	446	319	14	240
`ed`	editor	1506	2745	47	796
`find`*	Unix find program	5985	11773	212	4163
`flex`	scanner generator	5985	8688	129	2314
`flops`	benchmark	723	353	3	139
`gzip`*	compress	4056	6043	47	796
`linpack`	benchmark	821	796	13	278
`twig`	code generator	2092	3198	81	1085
`xmodem`	communication	2060	2965	31	1159
`xvcg`*	the vcg tool	298231	212134	1752	303374
`zpag`*	pag generated analyzer	20559	32525	777	25064

Figure 8.5: The test programs used for constant propagation

Program	Call String 0		Call String 1		Call String 2	
	s	MB	s	MB	s	MB
`bison`	2.0	4.5	4.3	8.0	17.1	10.7
`cdecl`	1.0	3.0	3.3	6.5	6.0	8.1
`cloops`	1.0	3.0	5.0	7.5	9.0	9.1
`dhry`	0.1	2.0	0.1	2.0	0.1	2.0
`ed`	0.1	2.5	1.0	4.0	2.0	4.5
`find`	6.0	8.0	157.1	14.5	9754.5	152.3
`flex`	3.0	5.5	7.0	8.5	16.0	11.1
`flops`	0.1	1.5	0.1	1.5	0.1	1.5
`gzip`	1.0	4.5	5.0	7.5	23.0	10.1
`linpack`	0.1	1.5	0.1	2.5	0.1	3.0
`xmodem`	1.0	4.5	3.0	6.0	4.0	6.6
`xvcg`	275.9	103.5	-	-	-	-
`zpag`	29.2	33.5	1635.3	124.0	-	-

Figure 8.6: Execution times and space consumption for constant propagation

Program	Description	Lines	s	MB
`avl`	the implementation of an avl tree	70	304.5	359.5
`list_search`	search an element in a linked list	39	0.1	1.0
`reverse`	reverse a linked list	34	0.1	1.0
`tree_copy`	copy a binary tree	49	2.0	7.5
`xtest`	inserts elements in a list	24	0.1	1.0

Figure 8.7: Execution times and space consumption for shape analysis

scription, their number of source code lines, the number of nodes in the control flow graph, the number of procedures and the number of variables. The programs marked with a star contain calls to procedure variables. In the CFG these calls are connected with all procedures in the program as possible callees. The analysis times for different call string lengths are shown in Fig. 8.6. For the larger programs not all call string lengths are possible. The ordering was $\mathbf{scc}_d$.

The long analysis times for the `find` program with call string length one and two result from the excessive use of procedure variables: there are 20 calls with unknown callees. This makes the call graph quite dense compared to other programs. This is also the reason why `xvcg` can be analyzed only for call string zero: it has 67 calls to unknown procedures. The other two test programs using procedural variables have just two calls to unknown procedures. Altogether the analysis times for the other programs are acceptable even for call string length two.

8.4.3 Shape Analysis

Shape analysis is a complex task. As stated above it is implemented for a subset of C. Therefore, the set of test programs is small. The results and the programs are listed in Fig. 8.7. The ordering was $\mathbf{scc}_d$. The analysis was done with call string length zero.

One can see that standard list and tree operations can be analyzed quickly. But for the `avl` program that consists of a series of mutual recursive functions for rebalancing the tree the analysis gets slow. Another reason for the longer analysis time is that the analysis is not able to figure out that a tree is constructed, and so the computed shape graphs get large and dense. (The worst case shape graph has 2^N nodes and $M * 2^{2^N}$ edges, where N is the number of variables in the program and M is the number of selector names.)

8.5 Influence of Optimizations

In this section the influence of the various optimizations discussed throughout this work is examined. The influence of the node orderings, the functor implementations, and the iteration algorithms is discussed. The experiments are executed with a smaller number of test programs which are carefully selected to underline the possible influence of the optimizations.

8.5.1 Node Ordering

The influence of the node ordering is exemplified on two analyses: cache analysis and conditional constant propagation. For the cache analysis the `lloops` program with the cache configuration for the HP-PA7100, `lloops` with the configuration of the i960KB, and `fft` with the configuration of the i960KB have been chosen. Except for the ordering all parameters have been chosen according to the experiment from Sec. 8.4.1. Figure 8.8 shows the results: sort is the time for computing the static priorities of the nodes, runtime is the complete running time of the analyzer including the time for sorting, and steps are the number of steps performed during the analysis. These steps are selections of basic blocks from the workset.

One can see that all these test cases suggest the use of **bfs**.

To examine the influence of the node ordering on the conditional constant propagation the experiment from Sec. 8.4.2 has been repeated for the test programs `bison` (Fig. 8.9), `flex` (Fig. 8.10) and `gzip` (Fig. 8.11) with different orderings.

Here one can see that more complicated sortings do not pay off for fast analyses and small programs: the times for `flex` and call string zero for $\mathbf{scc}_b$ and $\mathbf{wto}_d$ are about 50% sorting time. Overall one can see that either **bfs** or $\mathbf{scc}_d$ is to prefer.

These results are only valid for the two example analyses. For other analyses with more complex (or simpler) abstract operations the results may differ: if the abstract operations are complex then a better sorting may pay off. One can see that the orderings have to be chosen individually for each analysis, but there is a clear tendency for each of them which ordering to use. This decision is mostly independent of the programs.

8.5.2 Functor Implementations

To examine the influence of the different functor implementations, constant propagation and cache analysis have been used. In the constant propagation analysis the main datatype is an environment mapping variables to values. For this function functor four implementations can be used. For each implementation an analyzer has been generated and has been applied to the `gzip` program with call string length two. Figure 8.12 shows the results. It lists the time and space consumption and the percentage of the analysis time which has been spent for garbage collection. This last figure gives an impression of the amount of garbage that is produced during the analysis, i.e. heap space that was allocated and is no longer used.

For the constant propagation analysis the array implementation performs not very good since it produces a large amount of garbage. For the cache analysis the array implementation performs best compared to all other implementations. One can see that the amount of garbage created from an implementation plays an important role, and appropriate functor implementations have to be selected depending on how the datatypes are used.

Ordering	`lloops` & HP-PA7100 Runt.[s]	Sort[s]	Steps	`fft` & i960KB Runt.[s]	Sort[s]	Steps	`lloops` & i960KB Runt.[s]	Sort[s]	Steps
dfs	4.1	0.1	5453	4.0	0.1	7258	25.1	0.1	13170
bfs	4.2	0.1	5544	2.0	0.1	4904	23.1	0.1	12148
scc$_d$	4.1	0.1	5544	3.0	0.1	5240	23.1	0.1	12148
scc$_b$	23.1	0.1	28916	9.1	0.1	17075	165.2	0.1	79718
wto$_d$	22.0	0.1	28169	9.1	0.1	17219	165.1	0.1	78844
wto$_b$	8.0	0.1	10115	4.1	0.1	8906	56.1	0.1	27649

Figure 8.8: Influence of the ordering on the execution time of the cache analysis exemplified on different programs and architectures

Ordering	Call String 0 Runt.[s]	Sort[s]	Steps	Call String 1 Runt.[s]	Sort[s]	Steps	Call String 2 Runt.[s]	Sort[s]	Steps
dfs	2.0	0.1	10729	6.0	0.1	24802	15.0	0.1	59041
bfs	2.0	0.1	9572	6.0	0.1	23835	16.1	0.1	60061
scc$_d$	2.0	0.1	8095	4.3	0.1	23657	17.1	0.1	64633
scc$_b$	6.0	3.0	9772	10.1	3.0	28256	22.1	3.0	72392
wto$_d$	6.0	3.0	9792	10.0	3.0	28109	22.1	3.0	72245
wto$_b$	2.0	0.1	8061	6.0	0.1	24935	17.1	0.1	66789

Figure 8.9: Influence of the ordering on the execution time of the constant propagation for `bison`

Ordering	Call String 0 Runt.[s]	Sort[s]	Steps	Call String 1 Runt.[s]	Sort[s]	Steps	Call String 2 Runt.[s]	Sort[s]	Steps
dfs	2.0	0.1	7089	7.0	0.1	19023	16.0	0.1	43180
bfs	2.0	0.1	6476	7.1	0.1	18462	16.0	0.1	41173
scc$_d$	3.0	0.1	6618	7.0	0.1	18013	16.0	0.1	40316
scc$_b$	5.0	3.0	6871	9.1	3.0	19126	23.1	3.0	56815
wto$_d$	5.0	3.0	6909	9.0	3.0	19069	23.0	3.0	56685
wto$_b$	3.1	0.1	7333	7.1	0.1	19030	16.1	0.1	43188

Figure 8.10: Influence of the ordering on the execution time of the constant propagation for `flex`

Ordering	Call String 0 Runt.[s]	Sort[s]	Steps	Call String 1 Runt.[s]	Sort[s]	Steps	Call String 2 Runt.[s]	Sort[s]	Steps
dfs	1.0	0.1	5094	5.1	0.1	23010	25.1	0.1	90832
bfs	1.0	0.1	5777	5.0	0.1	21943	23.0	0.1	85375
scc$_d$	1.0	0.1	5777	5.0	0.1	21943	23.0	0.1	85375
scc$_b$	2.0	1.0	5355	8.1	1.0	25551	27.1	1.0	94253
wto$_d$	2.1	1.0	5355	8.0	1.0	25551	27.0	1.0	94253
wto$_b$	1.0	0.1	5094	6.1	0.1	23010	25.0	0.1	90832

Figure 8.11: Influence of the ordering on the execution time of the constant propagation for `gzip`

Functor Implementation	Constant Prop. & `gzip` Runtime [s]	gc [%]	MB	Cache & HP-PA7100 & `lloops` runtime [s]	gc [%]	MB
arrays	1864.1	92.9	70.6	4.0	0.1	10.5
avl trees	23.0	18.2	10.1	154.3	82.3	26.0
unsorted lists	20.0	36.8	10.1	260.2	3.1	15.5
hash tables	23.1	31.1	10.1	51.3	52.7	20.0
red black trees	26.1	47.9	10.1	228.5	79.7	35.0

Figure 8.12: Influence of the functor implementations for constant propagation and cache analysis

Algorithm	Constant prop. & `gzip` Runtime [s]	gc [%]	MB	Cache & i960KB & `lloops` Runtime [s]	gc [%]	MB
worklist algorithm	40.1	36.4	10.6	110.3	56.0	36.5
optimized algorithm	33.1	31.3	17.1	32.1	12.5	27.0
basic block algorithm	23.1	31.8	10.1	23.1	13.0	17.0

Figure 8.13: Influence of the iteration algorithm for constant propagation and cache analysis

8.5.3 Workset Algorithms

From the three workset algorithms based on the call string approach, the basic block algorithm performs best since it uses the least amount of space and thereby saves garbage collection time. To verify this, constant propagation on the program `flex` was used with call string length two and the cache analysis on the `lloops` program. The results are shown in Fig. 8.13.

As described in Chap. 2 the basic block iteration algorithm may deliver better results than the other two algorithms for non distributive problems, while all three algorithms deliver the same results for distributive problems. Together with its good performance it is clearly preferable. But for understanding and visualizing a program analysis the other two algorithms are simpler.

8.6 Influence of Analysis Concepts

8.6.1 Analysis of Loops

To examine the influence of the VIVU connector, cache analysis has been applied in three variants to the test programs from Fig. 8.2. Once with the classical control flow graph and the call string approach of length one (called *traditional*), once with the transformation of loops to procedures and the call string approach. The length of the call strings has been chosen as the length of the longest acyclic path in the call graph of the transformed program (containing loops and procedures as blocks). Finally, the cache analysis with the VIVU connector has been applied. Figure 8.15 shows the results of all three variants: it lists the percentage of locations for each

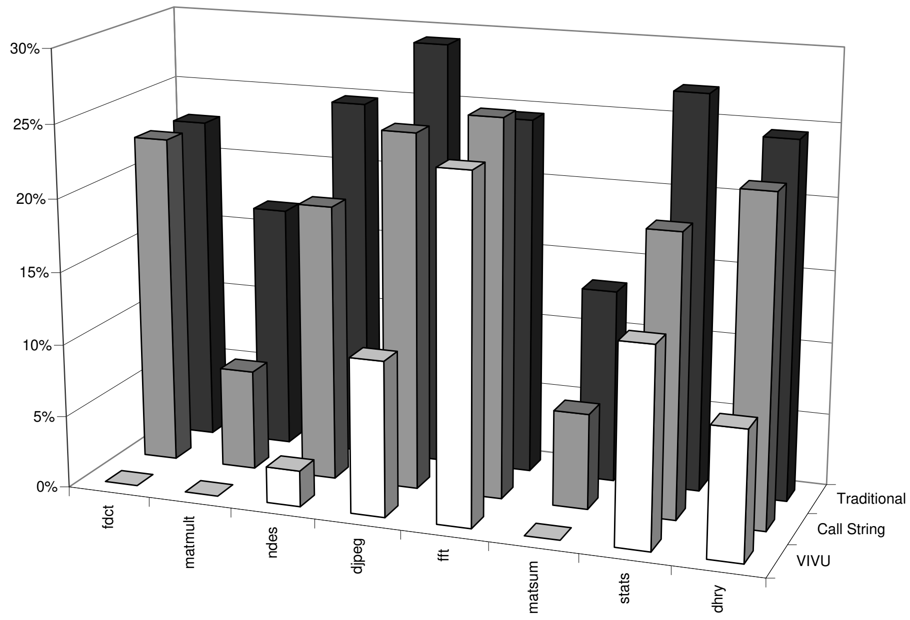


Figure 8.14: Evaluation of VIVU for cache analysis for a SuperSparc instruction cache.

program that result in a classification of the instruction as ah (always hits the cache) or am (always misses the cache) or nc (not classified), and the analysis times. Additionally, it shows the maximal length of the analyzed call strings for the call string method. Figure 8.14 shows the percentage of the non-classified instructions as bar charts.

For most programs there is a definite precision gain of VIVU over the traditional method, and for some programs VIVU allows to predict the cache behavior precisely. Of course VIVU can increase the number of analyzed contexts but the observation from our experiments was that this increase is moderate, as is the increase of the analysis times. One can also see that the call string approach is not well suited for the analysis of nested loops in the transformed control flow graph: for all test programs the VIVU method delivers better results, and the analysis times are similar.

8.6.2 Interprocedural Analysis

This section examines the two interprocedural techniques: call string approach and functional approach. This is done using conditional constant propagation and unconditional copy constant propagation. Note that the abstract domain of conditional constant propagation is not necessarily finite, and therefore the functional approach is not guaranteed to terminate. To examine the quality of the analysis results an optimization phase is used that replaces references to variables by their values if the value is known to be constant. Two measurements to compare the quality of the results are used: the theoretical precision which is the amount of all constant variables at

Name	Traditional				Call String					VIVU			
	ah	am	nc	Runt.[s]	N	ah	am	nc	Runt.[s]	ah	am	nc	Runt.[s]
`fdct`	74.86	2.43	22.70	0.1	2	74.86	2.43	22.70	0.1	86.92	13.08	0.00	0.1
`matmult`	73.38	9.74	16.88	0.1	5	80.20	12.87	6.93	0.1	91.03	8.97	0.00	0.1
`ndes`	72.19	2.97	24.84	0.1	5	77.72	3.25	19.02	0.1	89.87	7.65	2.49	0.1
`djpeg`	70.19	0.51	29.30	2.0	5	73.48	2.04	24.48	121.1	88.71	0.60	10.68	148.0
`fft`	67.89	7.52	24.59	1.0	6	70.74	3.34	25.92	42.1	74.73	1.53	23.74	18.1
`matsum`	73.33	13.33	13.33	0.1	5	76.97	16.45	6.58	0.1	85.83	14.17	0.00	0.1
`stats`	69.30	3.51	27.19	0.1	3	73.07	7.53	19.40	0.1	78.22	8.11	13.66	0.1
`dhry`	68.62	6.77	24.60	0.1	5	71.37	6.13	22.49	0.1	80.35	10.80	8.85	0.1

Figure 8.15: Evaluation of VIVU for cache analysis for a SuperSparc instruction cache.

each node over all nodes in the control flow graph, i.e. the lub of all locations of a node is built:

$$\sum_{n \in N} |\{var \mid \mathit{MFP}(n)(var) \neq \bot, \top\}|$$

As a measurement of the usability of the calculated information the number of source code transformations is given that can be done with the information obtained by the constant propagation. This is the number of replacements of variables by their values.

For each of the programs the functional analyzer was applied. It finds the best solution for any program in case of copy constant propagation. This is also true for full constant propagation if the functional analyzer terminates. The call string analyzers have been applied with call string length zero, one and two. As functional analyzers generated with PAG do not use the basic block optimizations, for the call string approach also the workset algorithm without the basic block optimization was used.

For each program four major columns are printed in Fig. 8.22 for copy constant propagation and in Fig. 8.23 for full constant propagation: three for call string approach of length zero, one, and two ($\mathcal{C}_0$, $\mathcal{C}_1$, $\mathcal{C}_2$), and one for the functional approach ($\mathcal{F}$). For each analysis method three numbers are given: the number of available constants, the number of foldings and the execution time of the analyzer.

To compare the different analysis approaches, for each analysis three bar charts are used: Fig. 8.16, 8.17, and 8.18 for the copy constant propagation and Fig. 8.19, 8.20, and 8.21 for the full constant propagation. The first bar charts (Fig. 8.16, 8.19) show the relative number of available constants of the call string approaches compared to the functional approach, the second bar charts (Fig. 8.17, 8.20) present the same comparison for the number of foldings, and the third bar charts (Fig. 8.18, 8.21) compare the runtimes of the different approaches.

One can see that for the copy constant propagation the analysis using call string with length one already enables all possible foldings. This is also true for the conditional constant propagation except for the `gzip` program. The functional analyzer for the full constant propagation gets trapped in an infinite loop for `bison`. For both analyses the call string analyzer finds the same available constants with length two as with length one except for the `flops` program. One can also see that the analysis times for the functional approach and the call string approach with length one are in the same range. So the functional approach seems to be preferable if termination can be guaranteed, but this conclusion is valid only for constant propagation. For a longer discussion of this topic see (Martin, 1999a; Alt and Martin, 1997).

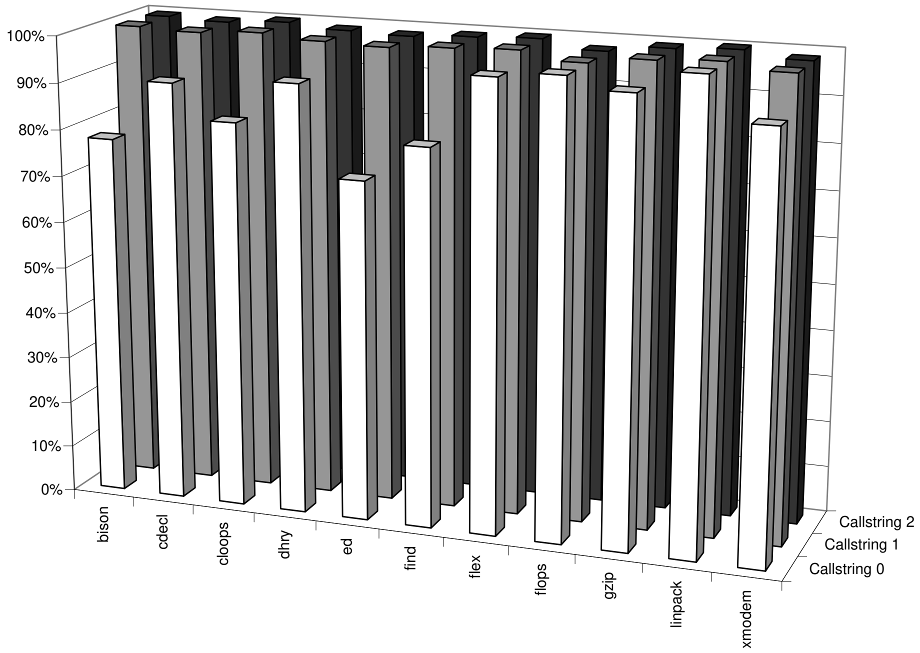


Figure 8.16: Copy constant propagation: percentage of available constants compared to the functional approach

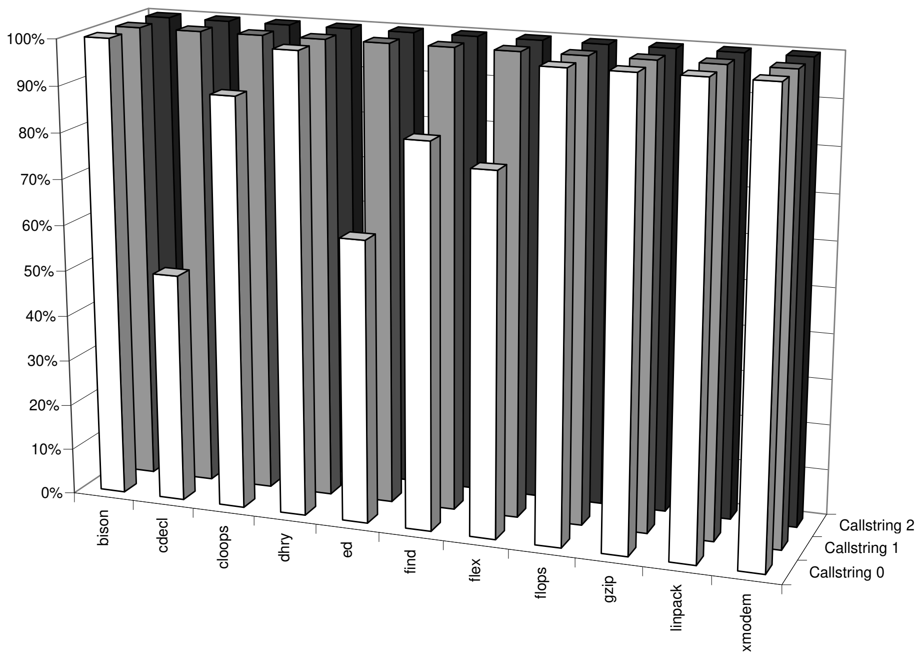


Figure 8.17: Copy constant propagation: percentage of constant foldings compared to the functional approach

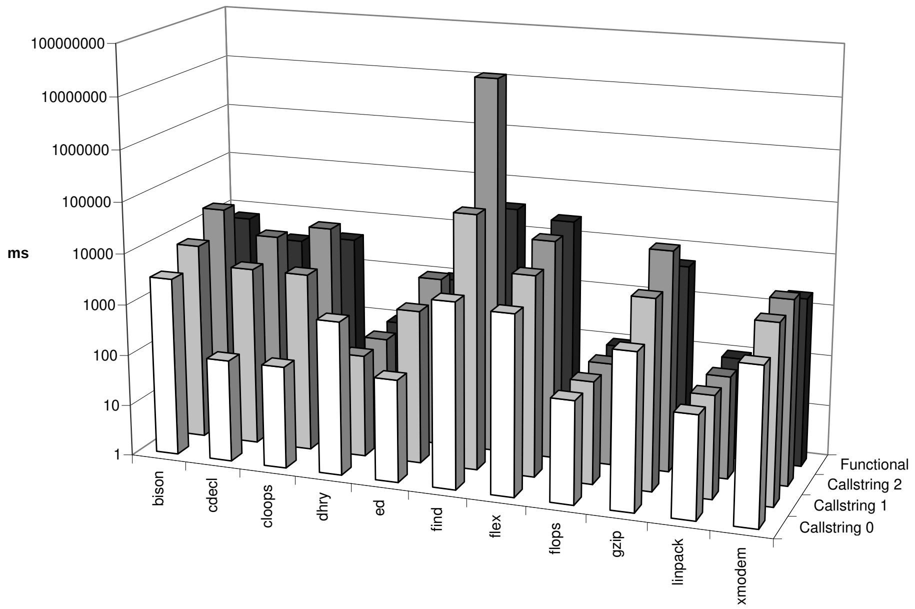


Figure 8.18: Copy constant propagation: runtimes

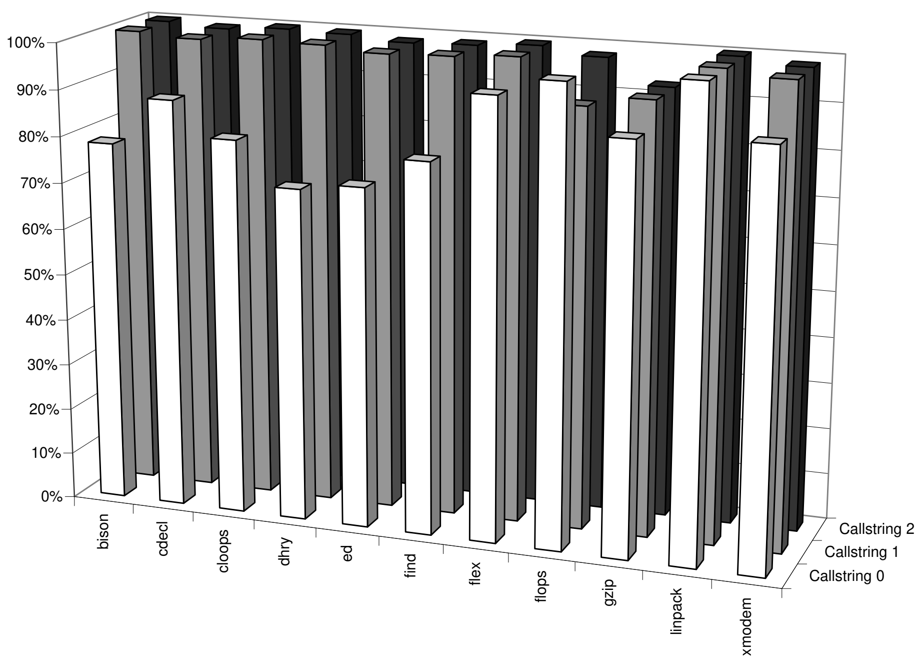


Figure 8.19: Full constant propagation: percentage of available constants compared to the functional approach

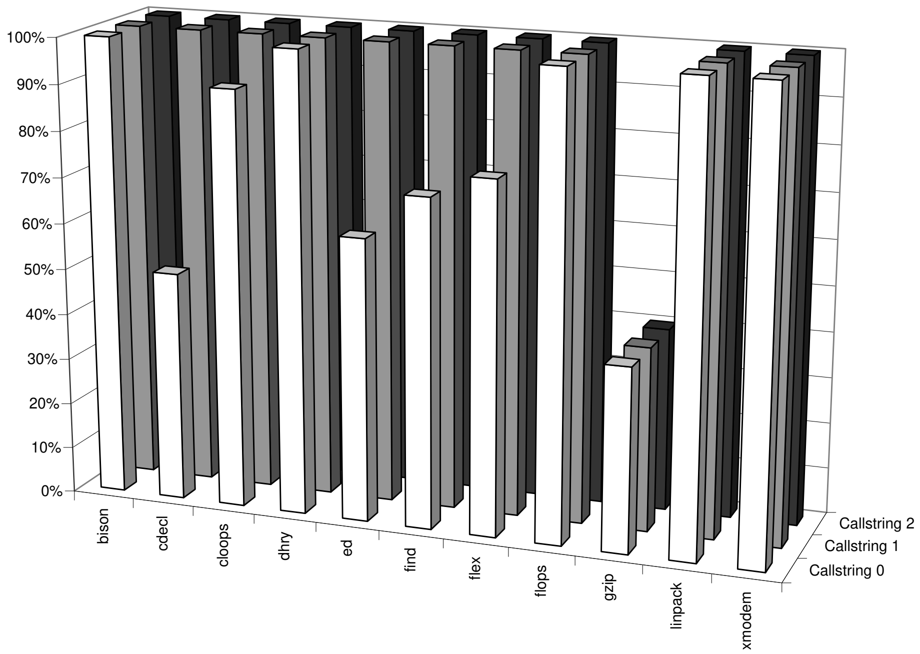


Figure 8.20: Full constant propagation: percentage of constant foldings compared to the functional approach

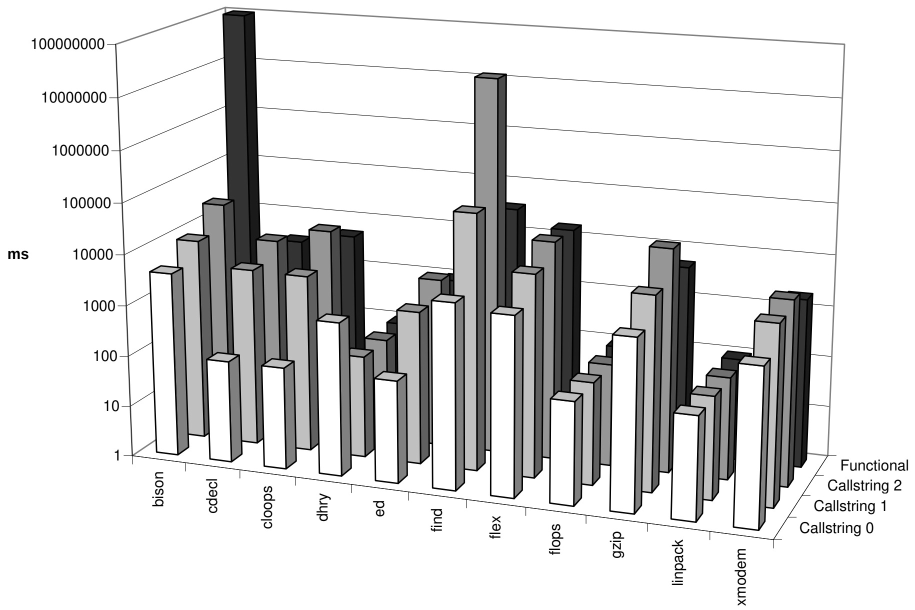


Figure 8.21: Full constant propagation: runtimes

Prog	$\mathcal{C}_0$			$\mathcal{C}_1$			$\mathcal{C}_2$			$\mathcal{F}$		
	Prec	Fold	Time	Prec	Fold	Time	Prec	Fold	Time	Prec	Fold	Time
`bison`	2456	7	3.1	3144	7	7.1	3144	7	20.1	3156	7	7.1
`cdecl`	1275	1	0.1	1392	2	3.1	1392	2	7.1	1404	2	3.1
`cloops`	1785	52	0.1	2138	58	3.1	2138	58	13.1	2140	58	4.1
`dhry`	174	0	0.1	186	0	0.1	186	0	0.1	188	0	0.1
`ed`	949	8	0.1	1275	13	1.0	1275	13	2.1	1294	13	1.0
`find`	1484	10	4.1	1809	12	93.2	1809	12	19873.2	1823	12	34.1
`flex`	4604	29	3.2	4749	37	8.2	4749	37	19.2	4763	37	24.1
`flops`	849	8	0.1	849	8	0.1	849	8	0.1	867	8	0.1
`gzip`	2285	3	1.1	2381	3	5.1	2381	3	20.1	2396	3	5.1
`linpack`	249	9	0.1	249	9	0.1	249	9	0.1	249	9	0.1
`xmodem`	1701	2	1.1	1847	2	3.1	1847	2	4.0	1873	2	2.0

Figure 8.22: Unconditional copy constant propagation

Prog	$\mathcal{C}_0$			$\mathcal{C}_1$			$\mathcal{C}_2$			$\mathcal{F}$		
	Prec	Fold	Time	Prec	Fold	Time	Prec	Fold	Time	Prec	Fold	Time
`bison`	2458	7	4.1	3137	7	9.2	3137	7	26.2	-	-	∞
`cdecl`	1284	1	0.1	1396	2	3.1	1396	2	6.1	1408	2	3.1
`cloops`	1914	59	0.1	2355	65	3.0	2355	65	12.1	2357	65	5.0
`dhry`	174	0	0.1	241	6	0.1	241	6	0.1	242	6	0.1
`ed`	976	8	0.1	1311	13	1.0	1311	13	2.1	1330	13	1.0
`find`	1618	10	4.1	2004	14	102.2	2004	14	20581.8	2026	14	35.1
`flex`	4715	29	3.1	4980	38	9.1	4980	38	19.2	4995	38	17.1
`flops`	920	8	0.1	849	8	0.1	920	8	0.1	938	8	0.1
`gzip`	2542	8	2.1	2693	8	6.1	2693	8	23.1	2904	20	5.1
`linpack`	249	9	0.1	249	9	0.1	249	9	0.1	249	9	0.1
`xmodem`	1752	4	1.1	1951	4	3.1	1951	4	4.1	1978	4	2.0

Figure 8.23: Conditional constant propagation

Chapter 9

Related Work

In this chapter the related work is described. It is divided into two main sections: one for the generation of analyzers and one for the theory of analysis of loops.

9.1 Generation of Analyzers

A number of programming environments have been created to support the implementation of program analyses. They can generally be divided into two classes: generators which generate an implementation of the analysis automatically from a specification and frameworks which provide a number of classes or templates to support the user in implementing the analysis. For the generators the specification language can be different from the implementation language. For the frameworks large parts of the implementation have to be hand coded by the user. Sometimes they use a small specification language describing the instantiation of the general parts, but most of the specification has to be written in the implementation language.

In the following some program analysis environments are described. The descriptions refer to publications (see bibliography) and other publicly available material (e.g. web pages).

9.1.1 Generators

Spare

In (Venkatesch and Fischer, 1992) a tool –Spare– is presented which is based on abstract interpretation. It was developed mainly for testing purposes of program analyses. The system uses a high level specification mechanism. The user may specify the abstract domain either by enumeration or some higher constructs, as in PAG. In fact the specification mechanism of Spare was a starting point for our work. The abstract functions are specified in a functional language by structural induction over the abstract syntax tree, like a denotational semantics specification and not on the control flow graph as in PAG. The specifications are translated into specifications of the Synthesizer Generator (Reps and Teitelbaum, 1988). The analyses are incorporated into a generated language editor. The high level specification was also designed to make correctness proofs easier. Spare has no built-in facilities to handle interprocedural analyses.

In the paper it is reported that Spare was used to specify conditional constant propagation, available expressions, definition-use chains and others. There are no execution times given in the paper, but the authors state that they expect the tool to be useful for programs in a range from 10 to 50 lines.

System Z

In (Yi and Harrison III, 1993) an abstract interpretation based analyzer generator is presented. The implementation language for the generated analyzers is C. System Z features *projection expressions* to transform a complex abstract domain into a simpler one. These projection expressions are similar to Galois connections. In contrast to PAG System Z allows only domains of finite cardinality. Like Spare it is not based on control flow graphs but on abstract syntax trees.

An eager functional language is used as specification language with a couple of datatype constructors including enumerations, intervals, tuples, powersets, lifting, and function building.

The system has been used to analyze programs in a medium level intermediate language (MIL) for which ANSI C, Fortran and Scheme frontends exist. In the paper the implementations of two analyses are reported: constant propagation and alias analysis. The analyzer sizes are reported to be about 450KB which is a bit larger than PAG generated analyzers. The runtimes reported for context insensitive (call string zero) constant propagation are between 240s and 1,000s for four programs. The sizes of the programs are given only as the number of MIL expressions which ranges from 3,500 to 8,800.

DFA&OPT-MetaFrame

In (Klein et al., 1996) the authors propose a quite different approach which generates data flow analyzers from a modal specification. This is done by partial evaluation of a model checker and the modal formula. This approach was implemented for interprocedural analyzers. The framework is limited to distributive bitvector analyses. Functions with local variables or parameters are not supported. A practical evaluation is not given in the paper.

In (Knoop et al., 1996) some of the same authors describe a framework for specifying intra- and interprocedural analyses. As far as we know no tool has been implemented using the specification framework.

The idea of combining abstract interpretation and model checking is also followed by others. In (Schmidt, 1998) a model checker is used to evaluate data flow analyses and in (Cousot and Cousot, 1999) it is demonstrated that model checking is abstract interpretation.

Optimix

In (Aßmann, 1996) a data flow analyzer generator is described. This generator is based on edge addition graph rewrite systems which can be used to implement distributive bitvector frameworks. Like PAG, Optimix is based on a control flow graph. The specification mechanism is based on graph grammars. The use of graph rewriting systems enables an integration of restricted transformations of the analyzed program. There is no special support for interprocedural analyses.

As for PAG, the development of Optimix started in the ESPRIT project COMPARE. It works on the intermediate language of CoSy (Compiler System) which was described within the project, for which C and Fortran frontends are commercially available. In the project the generator has been used to generate a lazy code motion optimization which is based on two bitvector analyses. Furthermore a restricted variant of constant propagation was implemented. It calculates the definition-use chains for a program, and whenever all definitions for a use of `x` are of the form `x:=c` where `c` is the same constant for all definitions then the variable is considered to be constant at this program point. This variant of constant propagation is based on bitvector frameworks and is even more restrictive than the copy constant problem.

The author states in the paper that 60-80% of each example analyzer are generated. The remaining code has to be provided by hand crafting. In the paper the runtimes of the analyzers are not given directly. It is only said that for lazy code motion a compiler slow down of the factor 7.2 was experienced and that for constant propagation the factor 2.9 was measured.

9.1.2 Frameworks

Sharlit

As part of the SUIF project (Wilson et al., 1994) the tool Sharlit (Tjiang and Hennessy, 1992) was built. It supports the implementation of intraprocedural data flow analyzers by providing fixed point algorithms. The user has to implement the abstract domain functionality in C++. The transfer functions are described in a specification language that is based on C++. It lacks support for interprocedural analysis. For bitvector frameworks the *path compression* can be used which is a mixed fixed point computation algorithm using elimination techniques as well as iterative techniques.

The paper reports that Sharlit was used to build several optimizations for the optimizing SUIF-MIPS compiler: constant propagation, partial redundancy elimination, and strength reduction. The practical evaluation given in the paper is very short. It reports only the times for the whole compiler run (some hundred seconds) for several programs.

FIAT

FIAT (Hall et al., 1993) is –like Sharlit– used in the SUIF project. It is a framework for implementing context insensitive interprocedural analyses.

It supports the user in implementing data flow analyses in C by providing a template for an iterative fixed point solver and a common driver for all analyzers. The functionality for the abstract domains and the transfer functions has to be hand coded by the user.

FIAT has been used to support the implementation of a number of tools using data flow analyses including the automatic parallelization system in the SUIF compiler. The paper does not present any performance evaluation.

A Flexible Architecture for Building Data Flow Analyzers

In (Dwyer and Clarke, 1996) another framework for building data flow analyzers is presented. It defines interfaces for the various parts of a data flow analyzer: an interface for the control flow graph, a lattice interface, and a solver interface. Additionally, it offers a set of analyzer components to choose from. These components do not only include a solver, but also lattice templates for bitvectors, boolean variables and sets. For distributive bitvector frameworks the toolset allows to generate the flow functions from the kill and gen sets automatically. The implementation language of the components is Ada.

In the paper it is said that a number of analyses have been created using the tool for distributed systems as well as for petri nets. The implemented analyses include live variables, constant propagation and reachability problems. No performance figures are shown in the paper.

Vortex Compiler

In (Chambers et al., 1996) a framework for data flow analyses is presented which is used in the Vortex compiler (Dean et al., 1996). It supports the user in writing data flow analyses by providing a fixed point iteration algorithm. This algorithm supports context sensitive interprocedural analyses for which different analysis methods can be chosen. The flow functions and the lattice functionality have to be implemented by the user in C++.

Since the Vortex compiler is able to compile several object oriented languages such as C++ and Java, the framework contains support for constructing the call graph and the interprocedural control flow graph in form of a library. It is reported that several analyses in the compiler have been implemented using the framework including a ‘closure escape analysis’, a ‘may raise exception analysis’, and a constant propagation. There are no analysis times given.

Others

There is also a program analysis framework called PAF (Programming Languages Research Group, 1999), which constructs various program representations for C programs, but contains no additional support for program analyses.

Also some compilers contain their own framework for implementing data flow analyses, e.g., the McCat compiler (Hendren et al., 1993), the `cmcc` compiler (Adl-Tabatabai et al., 1996), and others. Most of these frameworks are only intended to be used in the compiler for which they were designed, and not as general tools.

9.2 Analysis of Loops

Code motion techniques to move loop invariant code out of loops are special data flow analyses which fit in the classical data flow framework. In contrast to that, the technique proposed in Chap. 4 is a general framework which can be applied to all data flow analyses in order to obtain more precise analysis results for programs containing loops.

Structure based analyses such as interval analysis (Ryder and Paull, 1986) are orthogonal to the technique presented here. They are used to solve data flow problems efficiently in the presence of loops.

In (Steffen, 1996; Knoop et al., 1999) a property oriented expansion of a program model was developed. The aims of this technique are similar to the goals of the VIVU analysis: separation of the program states that have different properties and are generated through different program paths for each program point. In the paper this is reached by unfolding all paths that result in different properties. In this method the set of all properties must be finite to guarantee termination. But even then the worst case complexity is worse than the one of the functional approach, since the expansion is not limited to certain call edges but is applied to each node with more than one predecessor. A practical evaluation is not given.

Chapter 10

Outlook

This chapter describes a possible future development of PAG. For this development three major goals can be formulated:

- increase of the efficiency of the generated analyzers
- modifying the structure of the system to support new application areas
- making PAG more widely known and used.

10.1 Improving Efficiency

Throughout this work several opportunities for improvements have been pointed out. Among these the most promising one seems to be in the area of storage management. One can see from Chap. 8 that a large part of the runtime of an analyzer is spent on garbage collection. Here an improved garbage collection algorithm can help. It is known from other functional languages that generational garbage collection algorithms can improve the performance. Another possibility to reduce the garbage collection time is to reduce the production of garbage, e.g. by using destructive operations on objects to which only one reference exists, and thereby saving the overhead of copying the object and collecting the original object as garbage. To achieve this it has to be found out which objects have only a single reference by a static analysis of the heap constructed by a FULA program. But also a dynamic reference count can provide this information. It seems to be worthwhile to use a combination of both techniques.

10.2 Structure Modification

Data flow analysis is based on the control flow graph, but for functional languages or to a certain extent for object oriented languages the control flow graph itself is not directly available, but has to be constructed by a program analysis (Nielson and Nielson, 1999). This leads to the need for a tool which supports also this type of program analysis. Therefore, the idea is to turn PAG into a more open system consisting of independent modules. The basic idea for this module structure is to have one or several modules to generate a constraint system. In case of data flow analysis this module will generate an equation system from the control flow graph of the program describing the desired *MFP* solution. For object oriented programming languages this constraint system can describe the control flow analysis as well as a data flow analysis.

If the interfaces for the modules are disclosed, this opens up the possibility to replace one or several modules of the system with other modules to perform different types of analyses. This can be done by any experienced user for his own analysis problem.

The constraint system will be formulated in a functional language and is solved by a possibly user provided constraint solving module. The separate representation of the constraint system also allows the implementation of non-iterative solvers such as elimination algorithms.

10.3 Making PAG Widely Used

In order to become widely used the system has to become even more user friendly and simple to use. To achieve this in Transfer Project #14 several improvements are planned. This includes the redesign of the FULA language. As explained in Chap. 7 the current type checking algorithm imposes some restrictions on the language which can be relaxed by using a new typechecking algorithm. Additionally, the syntax will be chosen more closely to existing functional languages. Using also the module concept of the languages ML or Haskell will allow to consider the DATLA functors as abstract datatypes for which several implementations exist. This module concept allows the user to write his own domain implementations and to define new functors. Then DATLA will turn to a little metalanguage to describe the composition of the modules.

Furthermore, a debugger for the language is planned. Until now the visualization of the generated analyzers can be done only after an analysis step, which is the application of a transfer function and possibly several combine operations. For complex analyses an interactive debugger for the functional language will be helpful.

Chapter 11

Conclusion

In the context of static program analysis, most work focuses on theoretical aspects. Only a few tools for generating program analyzers in a more practical setting have been implemented. Most of them show poor evaluations or are applicable to toy settings only. To overcome this situation we have developed the PAG tool. It generates program analyzers from specifications fully automatically without the need for user interaction. The tool is based on the theories of abstract interpretation and data flow analysis. The specification mechanism is based on a functional language allowing efficient implementations and correctness proofs over the specification. It is possible to specify even complex analyses in a concise way with a high degree of abstraction what helps in implementing a correct program analysis.

In contrast to intraprocedural analysis, interprocedural analysis can widely improve the quality of the results, but its implementation is inherently complex. In PAG interprocedural analysis methods are fully integrated and can be specified with minimal additional effort.

Motivated by poor results of several analyses in the presence of loops we have presented a generalization of the interprocedural analysis for loops and arbitrary blocks. By extending the existing methods through the static call graph technique it is possible to focus the analysis effort on the points of main interest.

The applicability of these methods has been shown by practical experiments. The newly developed VIVU approach makes it possible to predict for example the cache behavior of programs within much tighter bounds than the conventional analysis methods.

A variety of analyses have been implemented using PAG. They range from the classical bitvector frameworks to complex memory error detection analyses, and include also the field of predicting the dynamic behavior of modern processors in the context of a worst case execution time prediction for real time programming. The measurements show that it is possible to generate efficient interprocedural analyzers which are applicable to large real world programs.

PAG has been selected as the analyzer generator tool of choice by a variety of international projects for a large number of different applications due to the efficiency of the generated analyzers and the usability and adaptability of PAG.

The web frontend PAG/WWW was designed for learning and understanding data flow analysis. It has been used in a series of tutorials on program analysis. Due to a well designed functional interface it is possible to use PAG generated analyzers in a variety of environments, as various applications prove. If no such environment exists the frontend generator Gon supports the development of PAG-frontends.

By combining the advantages of the theory of abstract interpretation with data flow analysis it is possible to develop optimization phases for compilers which can be proven to be correct and are efficiently executable. The powerful specification mechanism enables the implementation of very complex analyses in a relatively short time. This simplifies the maintaining and the porting to different platforms.

Bibliography

Adl-Tabatabai, A.-R., Gross, T., and Lueh, G.-Y. (1996). Code Reuse in an Optimizing Compiler. In *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 51–96. ACM Press.

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison Wesley.

Alt, M., Aßmann, U., and van Someren, H. (1994). Cosy Compiler Phase Embedding with the Cosy Compiler Model. In Fritzson, P., editor, *Proceedings of the 5th International Conference on Compiler Construction*, volume 786 of *Lecture Notes in Computer Science*. Springer.

Alt, M., Ferdinand, C., Martin, F., and Wilhelm, R. (1996a). Cache Behavior Prediction by Abstract Interpretation. In Cousot, R. and Schmidt, D. A., editors, *Third Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 51–66. Springer.

Alt, M., Ferdinand, C., and Sander, G. (1996b). Graph Visualisation in a Compiler Project. In Eades, P. and Khang, Z., editors, *Software Visualisation*. World Scientific.

Alt, M. and Martin, F. (1995). Generation of Efficient Interprocedural Analyzers with PAG. In Mycroft, A., editor, *Second Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 33–50. Springer.

Alt, M. and Martin, F. (1997). Practical Comparison of Call String and Functional Approach in Data Flow Analysis. In Kuchen, H., editor, *Arbeitstagung Programmiersprachen*, volume 58 of *Arbeitsberichte des Institutes für Wirtschaftsinformatik*. Westfälische Wilhelms-Universität Münster.

Alt, M., Martin, F., and Wilhelm, R. (1995). Generating Dataflow Analyzers with PAG. Technical Report A 10/95, Universität des Saarlandes, Fachbereich 14.

Aßmann, U. (1996). How To Uniformly Specify Program Analysis and Transformation. In Gyimothy, T., editor, *Proceedings of the 6th International Conference on Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 121–135. Springer.

Bird, R. J. (1998). *Introduction to Functional Programming using Haskell*. Prentice-Hall, second edition.

Bobbert, D. (1998). PAG/WWW Experiencing Program Analysis – The Backend. Dokumentation des Fortgeschrittenenpraktikums, Universität des Saarlandes, Fachbereich 14.

Bourdoncle, F. (1993). Efficient Chaotic Iteration Strategies with Widenings. In Björner, D., Broy, M., and Pottosin, I. V., editors, *Proceedings of Formal Methods in Programming and their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer.

Callahan, D., Cooper, K. D., Kennedy, K., and Torczon, L. (1986). Interprocedural Constant Propagation. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, volume 21, pages 152–161.

Cardelli, L. (1987). Basic Polymorphic Typechecking. *Science of Computer Programming*, 8(2):147–172.

Chambers, C., Dean, J., and Grove, D. (1996). Frameworks for Intra- and Interprocedural Dataflow Analyses. Technical Report TR-96-11-02, University of Washington, Department of Computer Science and Engineering.

Codish, M., Mulkers, A., Bruynooghe, M., de la Banda, M. G., and Hermenegildo, M. (1995). Improving Abstract Interpretations by Combining Domains. *ACM Transactions on Programming Languages and Systems*, 17(1):28–44.

Cousot, P. and Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press.

Cousot, P. and Cousot, R. (1992). Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2-3):103–180.

Cousot, P. and Cousot, R. (1999). *Abstract Interpretation, Modal Logic and Data Flow Analysis*. Talk at Dagstuhl Seminar on Program Analysis, April 1999.

Dean, J., DeFouw, G., Grove, D., Litvinov, V., and Chambers, C. (1996). Vortex: An Optimizing Compiler for Object-Oriented Languages. In *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 83–100. ACM Press.

Donnelly, C. and Stallman, R. (1995). *Bison – The yacc-Compatible Parser Generator*. Version 1.25.

Dor, N., Rodeh, M., and Sagiv, M. (1998). Detecting Memory Errors via Static Pointer Analysis. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 27–34. ACM Press.

Dwyer, M. B. and Clarke, L. A. (1996). A Flexible Architecture for Building Data Flow Analyzers. In *18th International Conference on Software Engineering*, pages 554–564. ACM Press.

Eidorff, P. H., Henglein, F., Mossin, C., Niss, H., Sørensen, M. H., and Tofte, M. (1999). ANNODOMINI: From Type Theory to Year 2000 Conversion Tool. In *Conference Record of the 26th ACM Symposium on Principles of Programming Languages*, pages 1–14. ACM Press.

Emmelmann, H., Schroeer, F.-W., and Landwehr, R. (1989). BEG: A Generator for Efficient Back Ends. *ACM SIGPLAN Notices*, 24(7):227–237.

Evans, D. (1996). *LCLINT User's Guide*. Version 2.2.

Fecht, C. (1997). Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung. Dissertation, Universität des Saarlandes, Fachbereich 14.

Ferdinand, C. (1997a). A Fast and Efficient Cache Persistence Analysis. Technical Report 10/97, Universität des Saarlandes, Sonderforschungsbereich 124.

Ferdinand, C. (1997b). Cache Behavior Prediction for Real-Time Systems. Dissertation, Universität des Saarlandes, Fachbereich 14.

Ferdinand, C., Kästner, D., Langenbach, M., Martin, F., Schmidt, M., Schneider, J., Theiling, H., Thesing, S., and Wilhelm, R. (1999). Run-Time Guarantees for Real-Time Systems—The USES Approach. Submitted for publication.

Ferdinand, C., Martin, F., and Wilhelm, R. (1997). Applying Compiler Techniques to Cache Behavior Prediction. In *Workshop on Languages, Compilers and Tools for Real-Time Systems*, pages 37–46. ACM Press.

Ferdinand, C., Martin, F., and Wilhelm, R. (1998). Cache Behavior Prediction by Abstract Interpretation. *Science of Computer Programming*. To appear.

Fischer, C. N. and LeBlanc, J. R. J. (1988). *Crafting a Compiler*. Benjamin/Cummings.

Fraser, C. W., Hanson, D. R., and Proebsting, T. A. (1992). Engineering a Simple, Efficient Code Generator Generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226.

Giegerich, R., Möncke, U., and Wilhelm, R. (1981). Invariance of Approximative Semantics with Respect to Program Transformation. In *Proceedings GI – 11. Jahrestagung*, volume 50 of *Informatik Fachberichte*, pages 1–10. Springer.

Hall, M. W., Mellor-Crummey, J. M., Carle, A., and Rodríguez, R. G. (1993). FIAT: A Framework for Interprocedural Analysis and Transformations. In Banerjee, U., Gelernter, D., Nicolau, A., and Padua, D., editors, *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 522–545. Springer.

Harrison III, W. L. (1995). *Personal communication, Dagstuhl Seminar on Abstract Interpretation*. August 1995.

Hecht, M. (1977). *Flow Analysis of Computer Programs*. North Holland.

Hendren, L. J., Donawa, C., Emami, M., Gao, G. R., Justiani, and Sridharan, B. (1993). Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In Banerjee, U., Gelernter, D., Nicolau, A., and Padua, D., editors, *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, volume 757 of *Lecture Notes in Computer Science*, pages 406–420. Springer.

Hennessy, J. and Patterson, D. (1996). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.

Horwitz, S., Demers, A., and Teitelbaum, T. (1987). An Efficient General Iterative Algorithm for Data Flow Analysis. *Acta Informatica*, 24:679–694.

Jones, N. D. and Nielson, F. (1995). Abstract Interpretation: A Semantics-Based Tool for Program Analysis. In *Handbook of Logic in Computer Science*. Oxford University Press.

Jones, R. E. (1996). *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons.

Jones, S. L. P. (1987). *The Implementation of Functional Programming Languages*. Prentice-Hall.

Kam, J. and Ullman, J. D. (1977). Monotone Data Flow Analysis Frameworks. *Acta Informatica*, 7:305–317.

Kästner, D. and Thesing, S. (1998). Cache-Sensitive Pre-Runtime Scheduling. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, pages 127–141. ACM Press.

Kildall, G. A. (1973). A Unified Approach to Global Program Optimisation. In *Conference Record of the first ACM Symposium on Principles of Programming Languages*, pages 194–206. ACM Press.

Klein, M., Knoop, J., Koschützki, D., and Steffen, B. (1996). DFA & OPT–METAFrame: A Toolkit for Program Analysis and Optimization. In Margaria, T. and Steffen, B., editors, *Proceedings of the second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *LNCS*, pages 422–426. Springer.

Knoop, J., Rüthing, O., and Steffen, B. (1996). Towards a Tool Kit for the Automatic Generation of Interprocedural Data Flow Analyses. *Journal of Programming Languages*, 4(4):211–246.

Knoop, J., Rüthing, O., and Steffen, B. (1999). Expansion-Based Removal of Semantic Partial Redundancies. In Jähnichen, S., editor, *Proceedings of the 8th International Conference on Compiler Construction*, volume 1575 of *Lecture Notes in Computer Science*, pages 91–106. Springer.

Knoop, J. and Steffen, B. (1992). The Interprocedural Coincidence Theorem. In Kastens, U. and Pfahler, P., editors, *Proceedings of the 4th International Conference on Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, pages 125–140. Springer.

Lauer, O. (1999). Generierung effizienter Datentypen für PAG. Diplomarbeit, Universität des Saarlandes, Fachbereich 14.

Mackenzie, D. (1996). *Autoconf – Creating Automatic Configuration Scripts*. Edition 2.12, for autoconf Version 2.12.

Marlowe, T. J. and Ryder, B. G. (1990). Properties of Dataflow Frameworks: A Unified Model. *Acta Informatica*, 28(2):121–163.

Martin, F. (1995). Entwurf und Implementierung eines Generators für Datenflußanalysatoren. Diplomarbeit, Universität des Saarlandes, Fachbereich 14.

Martin, F. (1998). PAG – An Efficient Program Analyzer Generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67.

Martin, F. (1999a). Experimental Comparison of *call string* and *functional* Approaches to Interprocedural Analysis. In Jähnichen, S., editor, *Proceedings of the 8th International Conference on Compiler Construction*, volume 1575 of *Lecture Notes in Computer Science*, pages 63–75. Springer.

Martin, F. (1999b). Suggested Exercises for PAG/WWW. http://www.cs.uni-sb.de/∼martin/pag, Universität des Saarlandes, Fachbereich 14.

Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

Nielson, F. (1996). Semantics-Directed Program Analysis: A Tool-Maker's Perspective. In Cousot, R. and Schmidt, D. A., editors, *Third Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 2–21. Springer.

Nielson, F. and Nielson, H. R. (1999). Interprocedural Control Flow Analysis. In Swierstra, S. D., editor, *Proceedings of the 8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 20–39.

Nielson, F., Nielson, H. R., and Hankin, C. L. (1998). *Principles of Program Analysis - Flows and Effects*. Springer. Preliminary version of March 4, 1998; complete version to appear 1999.

Nielson, H. R. (1999). Compiler Optimization Course Description. http://www.daimi.au.dk/-∼hrn/COURSES/Courses99/pagwww.html, University of Aarhus.

Nielson, H. R. and Nielson, F. (1992). Bounded fixed-point iteration. *Journal of Logic and Computation*, 2(4):441–464.

Paxson, V. (1995). *Flex – a Fast Scanner Generator*. Edition 2.5.

Programming Languages Research Group (1999). Prolangs Analysis Framework. http://prolangs.rutgers.edu/public.html, Rutgers Unversity, New Jersey.

Reps, T., Horwitz, S., and Sagiv, M. (1995). Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Conference Record of the 22nd ACM Symposium on Principles of Programming Languages*, pages 49–61. ACM Press.

Reps, T. and Teitelbaum, T. (1988). *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer.

Rubin, S., Bernstein, D., and Rodeh, M. (1999). Virtual Cache Line: A New Technique to Improve Cache Exploitation for Recursive Data Structures. In Jähnichen, S., editor, *Proceedings of the 8th International Conference on Compiler Construction*, volume 1575 of *Lecture Notes in Computer Science*, pages 259–273. Springer.

Ryder, B. G. and Paull, M. C. (1986). Elimination Algorithms for Data Flow Analysis. *ACM Computing Surveys*, 18(3):277–315.

Sagiv, M., Reps, T., and Wilhelm, R. (1996). Solving Shape-Analysis Problems in Languages with Destructive Updating. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages*, pages 16–31. ACM Press.

Sagiv, M., Reps, T., and Wilhelm, R. (1998). Solving Shape-Analysis Problems in Languages with Destructive Updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50.

Sagiv, M., Reps, T., and Wilhelm, R. (1999). Parametric Shape Analysis via 3-Valued Logic. In *Conference Record of the 26th ACM Symposium on Principles of Programming Languages*, pages 105–118. ACM Press.

Sander, G. (1994). Graph Layout through the VCG Tool. In Tamassia, R. and Tollis, I. G., editors, *Proceedings of the DIMACS International Workshop on Graph Drawing*, volume 894 of *Lecture Notes in Computer Science*, pages 194–205. Springer.

Sander, G. (1996). Visualisierungstechniken für den Compilerbau. Dissertation, Universität des Saarlandes, Fachbereich 14.

Sander, G. (1999). Graph Layout for Applications in Compiler Construction. *Theoretical Computer Science*, 217(2):175–214.

Sander, G., Alt, M., Ferdinand, C., and Wilhelm, R. (1995). CLAX — A Visualized Compiler. In Brandenburg, F. J., editor, *Graph Drawing Workshop*, volume 1027 of *Lecture Notes in Computer Science*, pages 459–462. Springer.

Schmidt, D. A. (1998). Data-Flow Analysis is Model Checking of Abstract Interpretations. In *Conference Record of the 25th ACM Symposium on Principles of Programming Languages*, pages 38–48. ACM Press.

Schmidt, M. (1999a). Entwurf und Implementierung eines Kontrollflußgenerators. Diplomarbeit, Universität des Saarlandes, Fachbereich 14.

Schmidt, M. (1999b). *GON User's Manual*.

Schneider, J. (1998). Statische Pipline-Analyse für Echtzeitsysteme. Diplomarbeit, Universität des Saarlandes, Fachbereich 14.

Schneider, J. and Ferdinand, C. (1999). Pipeline Behaviour Prediction for Superscalar Processors by Abstract Interpretation. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, pages 35–44.

Schonberg, E., Schwartz, J. T., and Sharir, M. (1981). An automatic technique for selection of data structures in SETL programs. *ACM Transactions on Programming Languages and Systems*, 3(2):126–143.

Schorr, H. and Waite, W. M. (1967). An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures. *Communications of the ACM*, 10(8):501–506.

Sharir, M. and Pnueli, A. (1981). Two Approaches to Interprocedural Data Flow Analysis. In Muchnick, S. S. and Jones, N. D., editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall.

Sicks, M. (1997). Adreßbestimmung zur Vorhersage des Verhaltens von Daten-Caches. Diplomarbeit, Universität des Saarlandes, Fachbereich 14.

Stallman, R. M. (1998). *Using and Porting GNU CC*. Version 2.8.1.

Steffen, B. (1996). Property-Oriented Expansion. In Cousot, R. and Schmidt, D. A., editors, *Third Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 22–41. Springer.

Thesing, S., Martin, F., Lauer, O., and Alt, M. (1998). *PAG User's Manual*.

Tjiang, S. W. K. and Hennessy, J. L. (1992). Sharlit – A Tool for Building Optimizers. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 82–93. ACM Press.

Venkatesch, G. and Fischer, C. N. (1992). SPARE: A Development Environment for Program Analysis Algorithms. In *IEEE Transactions on Software Engineering*, volume 18.

Wall, L., Christiansen, T., and Schwartz, R. L. (1996). *Programming Perl*. O'Reilly.

Wegman, M. N. and Zadeck, F. K. (1985). Constant Propagation with Conditional Branches. In *Conference Record of the 12th ACM Symposium on Principles of Programming Languages*, pages 291–299. ACM Press.

Wilhelm, R. and Maurer, D. (1997). *Compiler Design*. International Computer Science Series. Addison–Wesley. second printing.

Wilson, R. P., French, R. S., Wilson, C. S., Amarasinghe, S. P., Anderson, J. M., Tjiang, S. W. K., Liao, S.-W., Tseng, C.-W., Hall, M. W., Lam, M. S., and Hennessy, J. L. (1994). SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12):31–37.

Yi, K. and Harrison III, W. L. (1993). Automatic Generation and Management of Interprocedural Program Analyses. In *Conference Record of the 20th ACM Symposium on Principles of Programming Languages*, pages 246–259. ACM Press.

Appendix A

Strongly Live Variable Analysis

```
SYNTAX

START : Unlabstat

Unlabstat : M_Assign(var:Var,exp:Exp)
          | M_If1(exp:Exp,then:StatSeq)
          | M_If2(exp:Exp,then:StatSeq,else:StatSeq)
          | M_Goto(id:Identifier)
          | M_While(exp:Exp,stats:StatSeq)
          | M_Call(id:Identifier,params:Params)
          | M_Return()
          | M_Read(var:Var)
          | M_Write(exp:Exp)
          | M_WriteStr(str:String);

Var       : M_simpl_var(id:Identifier)
          | M_select_var(var:Var,exp:Exp);

Exp       : M_Binop(op:Op,e1:Exp,e2:Exp)
          | M_Unop(op:Unop,e:Exp)
          | M_Var_exp(var:Var)
          | M_Int_exp(int:Integer)
          | M_Real_exp(real:Real)
          | M_True_exp()
          | M_False_exp();

Op        : M_op_le()
          | M_op_leq()
          | M_op_eq()
          | M_op_neq()
          | M_op_ge()
```

```
          | M_op_geq()
          | M_op_plus()
          | M_op_minus()
          | M_op_or()
          | M_op_times()
          | M_op_div()
          | M_op_modulo()
          | M_op_and();

Unop      : M_op_unminus()
          | M_op_not();

StatSeq   : M_NoStats();
          | M_Stats(head:Stat,tail:StatSeq);

Stat      : M_Stat(stat:Unlabstat)
          | M_LabStat(lab:Label,stat:Unlabstat);

Label     : M_Lab(id:Identifier);

Params    : M_NoExps()
          | M_Exps(tail:Params,head:Exp);

Integer    == snum;
Identifier == snum;
Real       == real;
String     == snum;


GLOBAL
  maxvar  : unum

SET
  vars      = [0..maxvar]

LATTICE
  varset    = set(vars)
  var       = lift(varset)

PROBLEM strongly_live
  direction : backward
  carrier   : var
  init_start: lift(bot)

TRANSFER
  M_Assign(var,exp),_:
```

```
    let flow <= @;
          id  = get_id(var);
    in
      if id?flow
        then lift((flow # id) lub use_var(var) lub use(exp))
        else @
      endif;

  M_If1(exp,_),_: let flow <= @; in
     lift(flow lub use(exp));

  M_If2(exp,_,_),_: let flow <= @; in
     lift(flow lub use(exp));

  M_While(exp,_),_: let flow <= @; in
     lift(flow lub use(exp));

  M_Read(var),_:
     let flow <= @;
         id    = get_id(var);
     in
       if id?flow
         then lift((flow # id) lub use_var(var))
         else @
       endif;

  M_Write(exp),_: let flow <= @; in
     lift(flow lub use(exp));

  _,_: @;


SUPPORT
  get_id(M_simpl_var(id))     = val-Identifier(id);
  get_id(M_select_var(var,_)) = get_id(var);

  use_var(M_select_var(var,exp)) = use(exp) lub use_var(var);
  use_var(_)                     = bot;

  use(M_Binop(_,e1,e2)) = use(e1) lub use(e2);
  use(M_Unop(_,e))      = use(e);
  use(M_Var_exp(var))   = {get_id(var)} lub use_var(var);
  use(_)                = bot;
```