Lecture Notes in Computer Science 2628
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Thomas Fahringer Bernhard Scholz

Advanced Symbolic Analysis for Compilers

New Techniques and Algorithms
for Symbolic Program Analysis and Optimization

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Authors

Thomas Fahringer
University of Vienna
Institute for Software Science
Liechtensteinstr. 22, 1090 Vienna, Austria
E-mail: tf@par.univie.ac.at

Bernhard Scholz
Technische Universität Wien
Institut für Computersprachen E185/1
Argentinierstr. 8/4, 1040 Vienna, Austria
E-mail: scholz@complang.tuwien.ac.at

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): D.3.4, F.3.2, F.3, D.3, D.2, D.4

ISSN 0302-9743
ISBN 3-540-01185-4 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin GmbH
Printed on acid-free paper SPIN: 10873049 06/3142 5 4 3 2 1 0

1. Preface

The objective of program analysis is to automatically determine the properties of a program. Tools of software development, such as compilers, performance estimators, debuggers, reverse-engineering tools, program verification/testing/proving systems, program comprehension systems, and program specialization tools are largely dependent on program analysis. Advanced program analysis can: help to find program errors; detect and tune performance-critical code regions; ensure assumed constraints on data are not violated; tailor a generic program to suit a specific application; reverse-engineer software modules, etc. A prominent program analysis technique is symbolic analysis, which has attracted substantial attention for many years as it is not dependent on executing a program to examine the semantics of a program, and it can yield very elegant formulations of many analyses. Moreover, the complexity of symbolic analysis can be largely independent of the input data size of a program and of the size of the machine on which the program is being executed.

In this book we present novel symbolic control and data flow representation techniques as well as symbolic techniques and algorithms to analyze and optimize programs. Program contexts which define a new symbolic description of program semantics for control and data flow analysis are at the center of our approach. We have solved a number of problems encountered in program analysis by using program contexts. Our solution methods are efficient, versatile, unified, and more general (they cope with regular and irregular codes) than most existing methods.

Many symbolic analyses are based on abstract interpretation, which defines an abstraction of the semantics of program constructs in a given language, and an abstraction of program input data. The abstraction of data and programs leads to a less complex view which simplifies many program analyses. Some problems which were originally undecidable become decidable or can be solved only with a very high computational effort. On the other hand, information can get lost through abstraction; thus a solution of a problem based on abstract interpretation may no longer be a solution of the original problem. The art of symbolic program analysis is focused on the development of a suitable abstract interpretation for a given program and its input data which allows the problems of interest to be solved.

Previous approaches on symbolic program analysis frequently have several drawbacks associated with them:

- restricted program models (commonly exclude procedures, complex branching and arrays),
- analysis that covers only linear symbolic expressions,
- insufficient simplification techniques,
- memory- and execution-time-intensive algorithms,
- either control flow or data flow can be modeled but not both,
- unstructured, redundant and inaccurate analysis information, and complex extraction of analysis information,
- additional analysis is required to make the important relationship between problem size and analysis result explicit,
- recurrences are frequently not supported at all, or separate analysis and data structures are required to extract, represent, and resolve recurrence systems.

Furthermore, for program analyses there is commonly a need to express values of variables and expressions as well as the (path) condition under which control flow reaches a program statement. Some approaches do not consider path conditions for their analysis. Without path conditions the analysis accuracy may be substantially reduced. We are not aware of any approach that combines all important information about variable values, constraints between them, and path conditions in a unified and compact data representation. Moreover, it has been realized [E. Su, A. Lain, S. Ramaswamy, D. Palermo, E. Hodges, and P. Banerjee. Advanced Compilation Techniques in the Paradigm Compiler for Distributed-Memory Multicomputers. In *Proc. of the 9th ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995] that systems with interfaces for off-the-shelf software are critical for future research tool and compiler development. Approaches that rely on specialized representations for their analysis are often forced to reimplement standard symbolic manipulation techniques which otherwise could be taken from readily available software packages. Symbolic expressions are used to describe the computations as algebraic formulas over a program's problem size. However, symbolic expressions and recurrences require aggressive simplification techniques to keep the program contexts in a compact form. Hence, *computer algebra systems* (CASs) play an important role in manipulating symbolic expressions and finding solutions to certain problems encountered in program analysis. During the last decade CASs have become an important computational tool. General purpose CASs [D. Harper, C. Woo, and D. Hodgkinson. *A Guide to Computer Algebra Systems*. John Wiley & Sons, 1991], which are designed to solve a wide variety of mathematical problems, have gained special prominence. The major general purpose CASs such as Axiom, Derive, Macsyma, Maple, Mathematica, MuPAD, and Reduce, have significantly profited from the increased power of computers, and are mature enough to be deployed in the field of program analysis. One of the goals of this

book is to introduce techniques for closing the gap between program development frameworks and CASs. Symbolic algebraic techniques have emerged in the field of CASs to broaden the spectrum of program analyses for compilers and program development tools.

We describe a novel and unified program analysis framework for sequential, parallel, and distributed architectures which is based on symbolic evaluation, and combines both data and control flow analysis to overcome or at least to simplify many of the deficiencies of existing symbolic program analysis efforts, as mentioned above. At the center of our framework is a new representation of analysis information, the *program context*, which includes the following three components:

- variable values,
- assumptions about and constraints between variable values, and
- conditions under which control flow reaches a program statement.

A key advantage of our approach is that every component of program contexts can be separately accessed at well-defined program points without additional analysis. We describe an algorithm that can generate all program contexts by a single traversal of the input program. Program contexts are specified as n-order logic formulas, which is a general representation that enables us to use off-the-shelf software for standard symbolic manipulation techniques. Computations are represented as symbolic expressions defined over the program's problem size. Instead of renaming data objects, our symbolic analyses try to maintain the critical relationship between a program's problem size and the resulting analysis information. This relationship is important for performance-driven program optimization. Our symbolic analysis framework accurately models assignment and input/output statements, branches, loops, recurrences, arrays (including indirect accesses), and procedures. Recurrences are detected, closed forms are computed where possible, and the result can be directly retrieved from well-defined program points. Note that detecting recurrences and finding closed forms for recurrences are decoupled. The decoupling simplifies the extension of our recurrence solver with new recurrence classes. All of our techniques target both linear and nonlinear symbolic expressions and constraints.

We intensively manipulate and simplify symbolic expressions and constraints based on a system which we have built on top of existing software. We will describe a variety of new techniques for simplifying program contexts. Furthermore, we have developed several novel algorithms for comparing symbolic expressions, computing lower and upper bounds of symbolic expressions, counting the number of solutions to a system of constraints, and simplifying systems of constraints. While previous work mostly concentrated on symbolic analysis for shared memory architectures, this research also supports symbolic compiler analysis for distributed memory architectures covering symbolic dependence testing, array privatizing, message vectorization and coalescing, communication aggregation, and latency hiding.

We do not know of any other system that models a similar large class of program constructs based on a comprehensive and unified analysis representation (program context) that explicitly captures exact information about variable values, assumptions about and constraints between variable values, path conditions, and recurrence systems, and side-effect information about procedure calls.

Empirical results based on a variety of Fortran benchmark codes demonstrate the need for advanced analysis to handle nonlinear and indirect array references, procedure calls, and multiple-exit loops. Furthermore, we will show the effectiveness of our approach for a variety of examples including program verification, dependence analysis, array privatization, communication vectorization, and elimination of redundant communication.

Although symbolic program analysis has been applied to a variety of areas including sequential program analysis, parallel and distributed systems, real-time systems, etc., in this book we are mostly concerned with sequential, parallel, and distributed program analysis.

We have implemented a prototype of our symbolic analysis framework which is used as part of the Vienna High Performance Compiler (VFC, a parallelizing compiler) and P^3T (a performance estimator) to parallelize, optimize, and predict the performance of programs for parallel and distributed architectures. Although we examined our framework for Fortran programs on sequential, parallel, and distributed architectures, the underlying techniques are equally applicable to any similar imperative programming language.

Contents

1. Preface VII

2. Introduction 1
2.1 Motivation 1
2.2 Applications for Symbolic Analysis of Programs 6
2.3 Contributions 8
2.3.1 Symbolic Analysis Framework 8
2.3.2 Symbolic Analysis for Parallelizing Compilers 9
2.4 Organization 11

3. Symbolic Analysis of Programs 13
3.1 Introduction 13
3.2 Initializations, Assignments, and Input/Output Operations 14
3.3 Conditional Statements 17
3.4 Loops and Recurrences 21
3.5 Arrays 25
3.6 Procedures 31
3.7 Dynamic Data Structures 34
3.8 Summary 40

4. Generating Program Contexts 41
4.1 Introduction 41
4.2 Extended Control Flow Graph 41
4.3 Algorithm 43
4.4 Summary 47

5. Symbolic Analysis Algorithms and Transformations 49
5.1 Introduction 49
5.2 Preliminaries 51
5.3 Symbolic Expression Evaluation 52
5.3.1 Compute Lower and Upper Bounds of Symbolic Expressions 53
5.3.2 Rewrite φ-Expressions 56
5.3.3 Rewrite Policies 57

5.3.4 Simplify Expressions . . . 58
5.3.5 Determine Result . . . 59
5.4 Count Solutions to a System of Constraints . . . 59
5.4.1 Symbolic Sum Computation . . . 60
5.4.2 The Algorithm . . . 61
5.4.3 Algebraic Sum . . . 64
5.4.4 Miscellaneous . . . 64
5.5 Simplify Systems of Constraints . . . 66
5.6 Experiments . . . 67
5.6.1 Eliminate Redundant Constraints . . . 67
5.6.2 Counting Solutions to a System of Constraints . . . 70
5.6.3 Optimizing FTRVMT . . . 72
5.7 Summary . . . 73

6. Symbolic Analysis for Parallelizing Compilers . . . 75
6.1 Introduction . . . 75
6.2 Programming DMMPs . . . 76
6.3 Data Dependence Analysis . . . 79
6.4 Vienna High Performance Compiler . . . 81
6.5 Implementation . . . 88
6.6 Symbolic Analysis to Optimize OLDA . . . 92
6.7 Symbolic Analysis to Optimize HNS . . . 93
6.8 Summary . . . 96

7. Related Work . . . 99
7.1 Advanced Symbolic Analysis Algorithms . . . 106
7.2 Parallelizing Compilers . . . 107

8. Conclusion . . . 109

9. Appendix . . . 113
9.1 Control Flow Graphs . . . 114
9.2 Denotational Semantics . . . 115
9.3 Notation . . . 117
9.4 Denotational Semantic: Notation . . . 118
References . . . 119

Index . . . 127

2. Introduction

2.1 Motivation

Symbolic analysis of programs is a process of statically detecting program properties which covers data and control flow information at well defined program points. The information provided by symbolic analysis is useful in compiler optimization, code generation, program verification, testing, reverse-engineering, comprehension, specialization, and debugging, performance analysis, and parallelization.

In order to introduce some of the concepts and problems of symbolic analysis consider the code excerpt in Ex. 2.1.1. There are two integer variables `A` and `B`. In statement ℓ_1 a `read` statement assigns new values to both variables. From statement ℓ_2 to statement ℓ_4 we change values of variables `A` and `B` by a sequence of assignment statements. For the code in Ex. 2.1.1 we are interested in finding a different notation of program's computations. Although the input values of the program variables are not available at compile time, our objective is to determine the values of `A` and `B` at the end of the computation. For this purpose we require a technique to statically discover the relation between input values and the computations of a program. We want to establish a symbolic variable binding of program variables that describe a valid result of the computation for all possible input values of the program. In other words, we try to describe the variable values of a program as a function of the input data at compile-time which assumes that input data values are unknown. For this purpose we will employ symbolic analysis techniques that can deal with unknown variable values, manipulate symbolic expressions, propagate variable values through a program, and collect information about conditions under which control flow reaches a program statement.

We can proceed as follows: Instead of computing numbers for variables `A` and `B` we assign them symbolic values. Let us assume that variable `A` and `B` are undefined before statement ℓ_1. We denote this undefined variable binding of `A` and `B` as follows,

$$A = \bot$$
$$B = \bot$$

In ℓ_1 the read-statement assigns `A` and `B` values which are unknown at compile time. Therefore, we assign symbolic values $\bigtriangledown_1$ and $\bigtriangledown_2$ to the va-

T. Fahringer and B. Scholz: Advanced Symbolic Analysis, LNCS 2628, pp. 1–11, 2003.

Example 2.1.1 *Statement sequence*

```
     integer::A,B
ℓ1:  read(A,B)
ℓ2:  A := A+B
ℓ3:  B := A - B
ℓ4:  A := A - B
```

riables, respectively, where $\bigtriangledown_1$ and $\bigtriangledown_2$ denote the input values of the read statement.

$$A = \bigtriangledown_1$$
$$B = \bigtriangledown_2$$

Now, we continue our computation with the first assignment statement in ℓ_2. We propagate symbolic values of `A` and `B` and after statement ℓ_2 we obtain a symbolic variable binding as follows,

$$A = \bigtriangledown_1 + \bigtriangledown_2$$
$$B = \bigtriangledown_2$$

The symbolic description of variable `B` remains unchanged since statement ℓ_2 only affects variable `A`. We continue the symbolic computation for the remaining assignments and after symbolically evaluating the last statement ℓ_4 we get the following symbolic variable binding:

$$A = (\bigtriangledown_1 + \bigtriangledown_2) - \bigtriangledown_1$$
$$B = (\bigtriangledown_1 + \bigtriangledown_2) - ((\bigtriangledown_1 + \bigtriangledown_2) - \bigtriangledown_1)$$

The result exactly reflects the operations performed on the input values of the code fragment. Either we can compute the results by the program as given in Ex. 2.1.1 or we use our symbolic description of the computation which has strong resemblance to functional programming.

It is a compelling thought that further transformations can substantially simplify the symbolic description of the computation. Note that the integer numbers of program variables `A` and `B` have certain algebraic properties which must be obeyed. They form a ring $\mathbb{Z}$ of integers which is an integral domain that can be further classified as commutative ring with multiplication identity that satisfies the cancellation rule [67]. In Fig. 2.1 we have listed computational rules which hold for symbolic expressions E over $\mathbb{Z}$, which are used to describe the computations of a program as given in the example above. For instance, by applying the computational rules to the symbolic variable binding of our example we can significantly simplify the symbolic expression of the computations in Ex. 2.1.1:

$$A = \bigtriangledown_2$$
$$B = \bigtriangledown_1$$

Commutativity :	$\forall i, j \in E$
	$i + j = j + i$
	$i * j = j * i$
Associativity :	$\forall i, j, k \in E$
	$i + (j + k) = (i + j) + k$
	$i * (j * k) = (i * j) * k$
Distributivity :	$\forall i, j, k \in E$
	$i * (j + k) = (i * j) + (i * k)$
Identity :	$\forall i \in E$
	$i + 0 = i$
	$i * 1 = i$
Inverse :	$\forall i \in E$
	$i - i = 0$

Fig. 2.1. Properties of symbolic computations

Example 2.1.2 *Loop*

```
     integer::I
ℓ1:  read(A)
ℓ2:  do while A < 100
ℓ3:     A := 3*A + 1
ℓ4:  enddo
```

The symbolic description of the computations reveals that program variables `A` and `B` swap their input values. Just by applying symbolic analysis techniques, namely (1) *forward substitution* and (2) *symbolic simplification* techniques we could uncover the program semantics of the example. CASs offer an excellent interface for simplification techniques. Nevertheless, there is still the demand to develop techniques for obtaining a symbolic description of the program. In general this is a non-trivial task.

> *One of the goals of this book is to introduce a novel symbolic representation of computations, called* program contexts*, and algorithms to automatically generate program contexts.*

Consider the code excerpt in Ex. 2.1.2. We observe a loop that cannot be analyzed by simple forward substitution techniques as applied to the previous example in Ex. 2.1.1. A loop may imply an infinite number of iterations and in general we do not know the number of iterations at compile time. For instance, in the example code we can partially list the symbolic values of program variable `A` inside of the loop as listed in Fig. 2.2.

Iteration zero denotes the value of program variable `A` before entering the loop which stems from the read-statement in ℓ_1. The value of iteration 1 corresponds to the value of variable `A` at the end of the first iteration. Now,

Iteration	Symbolic value of `A`
0	$\bigtriangledown_1$
1	$3 \bigtriangledown_1 + 1$
2	$9 \bigtriangledown_1 + 4$
3	$27 \bigtriangledown_1 + 13$
⋮	⋮

Fig. 2.2. Sequence of symbolic values for program variable `A`

we can continue our symbolic analysis of the program ad infinitum. However, infinite symbolic lists are of little use for program analysis. Hence, we employ a mathematical vehicle called *recurrence* [82] for describing infinite lists of computations. A recurrence describes an infinite sequence of elements and it consists of a *recurrence relation* and a *boundary condition*. The recurrence relation expresses the nth element of a sequence as a function of previous elements and is also called *difference equation*. The boundary condition specifies few elements of the sequence to enable the recurrence to get started. Moreover, recurrences present a mathematical method to compute *closed forms* which express elements of the sequence just by their index number. Therefore, recurrences are very useful to describe symbolic computations inside of loops. We relate variable values from the previous iterations to variable values of the current iteration at the beginning of the loop, and we can classify several program paths of a loop as depicted for a program variable `V` in Fig. 2.3. The first class of paths enters the loop from outside. These paths constitute the boundary condition $v(0)$ for program variable `V` where $v(0)$ denotes the first element of the sequence. The second class of paths iterate the loop and determine the recurrence relation of a program variable whereas at the beginning of a loop the value of a variable is denoted by $v(k)$ and at the end of the loop by $v(k+1)$. The relation between $v(k+1)$ and $v(k)$ is used for constructing the recurrence relation. The third class of paths exits the loop.

For instance, the sequence of Fig. 2.2 represents the symbolic values of program variable `A` inside of the loop for the code in Ex. 2.1.2 and can be written as

$$\begin{aligned} a(0) &= \bigtriangledown_1 \\ a(k+1) &= 3a(k) + 1 \end{aligned}$$

where $a(0)$ denotes the value before entering the loop, $a(k+1)$ the value at the end of the k+1st iteration, and $a(k)$ the value at the beginning of the kth iteration.

In mathematical literature [64, 82] we have various approaches for finding closed forms of recurrence relations. One of the approaches is based on *generating functions*. Generating functions are an ingenious method by which

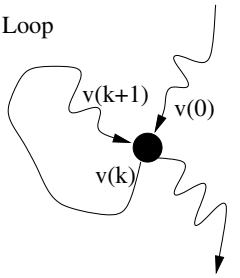

Fig. 2.3. Modeling of loops

recurrences can be solved very elegantly. Like Fourier transforms, generating functions transform a problem from one conceptual domain into another in order to solve the problem in the new domain. The *generating function* for the sequence $v(0), v(1), \ldots$ is

$$V(z) = \sum_{k=0}^{\infty} v(k) z^k.$$

Note that the sum in the definition might not always converge. We now show how generating functions can be used to solve a simple recurrence as given in our example.

$$v(k) \Rightarrow V(z) = \sum_{k} v(k) z^k$$

$$v(k+1) \Rightarrow \frac{V(z) - v(0)}{z} = \frac{\sum_k v(k) z^k - v(0)}{z}$$

$$c \Rightarrow \frac{c}{1-z}$$

Fig. 2.4. Replacement rules

First, we transform our recurrence relation by the rules given in Fig. 2.4. The first two rules replace the occurrence of v(k) and v(k+1) in the recurrence relation. The last rule replaces a constant c by its generating function. By applying these rules to our example we get the following relation,

$$\frac{A(z) - a(0)}{z} = 3A(z) + \frac{1}{1-z},$$

which has to be solved for $A(z)$. In the next step we substitute $a(0)$ by $\bigtriangledown_1$ and isolate $A(z)$ on the left-hand side:

$$
\begin{aligned}
A(z) &= \frac{\bigtriangledown_1}{1} 1 - 3z + \frac{z}{(1-3z)(1-z)} \\
&= \frac{\bigtriangledown_1}{1-3z} + \frac{1}{2(1-3z)} - \frac{1}{2(1-z)} \\
&= \left[\bigtriangledown_1 \sum_k 3^k z^k\right] + \left[\frac{1}{2}\sum_k 3^k z^k\right] - \left[\frac{1}{2}\sum_k z^k\right] \\
&= \sum_k \underbrace{\left[\bigtriangledown_1 3^k + \frac{1}{2}3^k - \frac{1}{2}\right]}_{a(k)} z^k
\end{aligned}
$$

Now we must return to the original problem domain. We know that for a sequence $1, 1, 1, \ldots$ the generating function is $1 + z + z^2 + \ldots = 1/(1-z)$ and for sequence $1, c, c^2, \ldots = 1/(1-cz)$. Finally, we see that the corresponding sequence is given by,

$$
a(k) = 3^k \bigtriangledown_1 + \frac{(3^k - 1)}{2}, \quad \text{if } k \geq 0 \tag{2.1}
$$

CASs are reasonable effective in finding closed forms of recurrence relations. Nevertheless, there is still a need to develop techniques for extracting recurrence relations of program variables.

> *One of the goals of this book is to introduce a unified symbolic analysis framework for the extraction and definition of recurrences for program variables in order to symbolically describe the computations of program variables inside of loops and under complex control flow.*

2.2 Applications for Symbolic Analysis of Programs

Symbolic analysis can be applied to a variety of problems. A main operational area of symbolic analysis is centered around compilers. By its nature, symbolic analysis is an expensive technique and it should be applied with care. Standard problems where less program analysis precision is required are better solved with conventional non-symbolic analysis. For instance, it is widely accepted [67, 102, 101, 19, 20, 98, 84] that current parallelizing compilers are not very effective in optimizing parallel programs that fully utilize the target multiprocessor system. The poor performance of many compiler analyses can be attributed to ineffective parallelization of programs (for instance, High Performance Fortran *HPF* [74] and Fortran 90 [85]) that have a strong potential for unknowns such as number of processors and sizes of allocateable arrays. Most compilers provide limited techniques to discover the relationship between a program's problem size (input data, array and machine sizes, etc.) and analysis information. Some others have poor capabilities to analyze dependences between references with linearized subscripts

and non-linear terms in subscript expressions. Non-linear terms can stem from recurrences or tiling with symbolic block sizes [10]. Commonly, compilers do not employ interprocedural analysis at the cost of analysis accuracy [71]. As noted in [20, 102, 101], ineffective approaches to gather and propagate sufficient information about variables through the program continues to have a detrimental impact on many compiler analyses and optimizations. As a consequence worst case assumptions are frequently made or program analysis is done at runtime which increases the execution overhead. Accurate data dependence analysis is necessary to exploit the inherent parallelism of sequential programs. Classical techniques usually fail to provide accurate data dependence information to support parallelization techniques such as array privatization[101] and communication optimization[53]. Therefore, sophisticated symbolic analysis that can cope with program unknowns is needed to overcome these compiler deficiencies.

Another application for symbolic analysis are program development tools which assist the programmer and system designer. Instead of testing and debugging, symbolic analysis techniques provide analysis information for statically detecting program anomalies. For example memory anomalies [97] such as memory leaks and nil-pointer dereferences cause severe problems in modern software design. Such errors are particularly hard to detect as they are usually very difficult to reproduce and to identify the source of the error in the program [3]. Clearly, it is of paramount importance to detect memory errors at compile time as compared to exhaustively testing programs at runtime which can cause a significant performance slowdown. A static analysis tool that targets the detection of memory anomalies, therefore, can greatly simplify program development and testing.

Tools that are able to detect deadlocks in concurrent programs [16] are of key importance in modern software. Concurrent language constructs increase the expressiveness of a language. Nevertheless, designing and programming a concurrent system is a tedious task and can result in erroneous program behavior. It is very hard to reproduce such anomalies and to identify the source of the error. Clearly, a tool that simplifies debugging and testing can significantly improve the quality of software.

Other applications can be found in safety-critical real-time computer systems which can be characterized by the presence of timing constraints. A tool for assessing and verifying the correctness of timing behavior is crucial. For example symbolic analysis has been employed to deduce time functions [96, 14] for real-time programs. Unfortunately modern hardware complicates time analysis, and caches are a major obstacle towards predicting execution time for modern computer architectures. To overcome or at least to simplify the problems of classical approaches, symbolic analysis [17] can be effectively

employed in order to deduce reasonably accurate analytical cache hit functions which compute the *precise* number of cache hits.

Symbolic analysis techniques can be equally important for program testing [80], program verification [27], software specialization [30], software reuse [28], pattern matching and concept comprehension [37], and other areas where complex programs with unknowns must be examined and accurate program analysis is required.

2.3 Contributions

In this section we outline the major contributions of this book and discuss the importance in compiler optimizations and other program development tools.

2.3.1 Symbolic Analysis Framework

The quality of many optimizations and analyses of compilers significantly depends on the ability to evaluate symbolic expressions and on the amount of information available about program variables at arbitrary program points. We will describe an effective and unified symbolic analysis framework for the programming language Fortran that statically determines the values of variables and symbolic expressions, assumptions about and constraints between variable values and the condition under which control flow reaches a program statement. We introduce the *program context*, a novel representation for comprehensive and compact control and data flow analysis information. Program contexts are described as logic formulas which allows us to use computer algebra systems for standard symbolic manipulation. Computations are represented as algebraic expressions defined over a program's problem size. Our symbolic evaluation techniques comprise accurate modeling of assignment and input/output statements, branches, loops, recurrences, arrays, dynamic records and procedures. All of our techniques target both linear as well as non-linear expressions and constraints. Efficiency of symbolic evaluation is highly improved by aggressive simplification techniques.

In contrast to abstract interpretation [31] our symbolic analysis can be seen as a kind of compiler which translates programs to symbolic expressions and recurrences. Therefore, we do not need an abstract domain which must honor certain algebraic properties (e.g., complete lattice). Other approaches such as SSA form [34] and G-SSA form [5] commonly require additional algorithms to determine variable values and path conditions. G-SSA form does not represent this information in a unified data structure. We are not aware of an approach that combines all important information about variable values, constraints between them, and path conditions in a unified and

compact data representation. Moreover, it has been realized [98] that systems with interfaces for off-the-shelf software such as computer algebra systems are critical for future research tool and compiler development. Approaches that rely on specialized representations for their analysis are commonly forced to re-implement standard symbolic manipulation techniques which otherwise could be taken from readily available software packages.

Contribution 1: *We present a comprehensive and compact control and data flow analysis information, called* program context *for solving program analysis problems. Program contexts include three components: variable values, assumptions about and constraints between variable values, and path condition.*

Contribution 2: *Program context components can be separately accessed at well-defined program points and are an amenable program analysis description for computer algebra systems.*

Contribution 3: *Our approach targets linear and non-linear symbolic expressions and the program analysis information is represented as symbolic expressions defined over the program's problem size.*

Contribution 4: *We introduce an algorithm for generating program contexts based on control flow graphs. The algorithm comprises accurate modeling of assignment and input/output statements, branches, loops, recurrences, arrays, dynamic records and procedures.*

2.3.2 Symbolic Analysis for Parallelizing Compilers

Symbolic analysis has emerged as an advanced technique to support the detection of parallelism and optimization of parallel programs. We have employed our symbolic analysis framework to optimize communication [50] by extracting single element messages from a loop and combining them to vector-form (*communication vectorization*), removing redundant communication (*communication coalescing*), and aggregating different communication statements (*communication aggregation*). A key technology for parallelizing compilers is to compute data dependences inside of loops as precise as possible. Therefore, symbolic analysis is of paramount importance in the field of parallelizing compilers. For instance, Parafrase-2 [88, 67] and Nascent [56] introduced symbolic analysis to recognize and compute closed forms of induction variables. Parafrase-2 also applies symbolic analysis for dependence testing, eliminating dead-code, and scheduling of parallel loops. The Parascope project [79] included interprocedural symbolic analysis. In the Polaris restructuring compiler [23] symbolic analysis supports range propagation (lower/upper bounds of symbolic expressions), array privatization and dependence testing. The Paradigm compiler [98] incorporates a commercial symbolic manipulation package to improve parallelization of programs for distributed memory architectures. The SUIF compiler [69] employs linear symbolic

analysis for scalar, array and interprocedural analysis. Previous approaches on symbolic compiler analysis frequently have several drawbacks associated with them:

- restricted program models (commonly exclude procedures, complex branching and arrays)
- analysis that covers only linear symbolic expressions
- insufficient simplification techniques
- memory and execution time intensive algorithms
- unstructured, redundant and inaccurate analysis information, and complex extracting of analysis information
- additional analysis is required to make the important relationship between problem size and analysis result explicit
- recurrences are frequently not supported at all or separate analysis and data structures are required to extract, represent, and resolve recurrence systems.

Furthermore, for parallelizing compilers there is a need to express values of variables and expressions as well as the (path) condition under which control flow reaches a program statement. Some approaches [71, 68, 27] do not consider path conditions for their analysis. Without path conditions the analysis accuracy may be substantially reduced.

To demonstrate the effectiveness of our approach that is based on *program contexts* we have implemented our symbolic analysis framework in the context of the Vienna High Performance compiler (VFC) and P^3T a performance estimator for parallel and distributed architectures. We have parallelized several Fortran programs that have been taken from the Perfect, RiCEPS (Rice Compiler Evaluation Program Suite) and Genesis benchmark suites [12, 1] as well as from the WIEN95 software package [13]. By applying our symbolic analysis techniques we could achieve significantly better results compared to previous work.

Contribution 5: *We apply our symbolic analysis framework in order to compute data dependences.*

Contribution 6: *We employ our symbolic analysis framework to optimize communication by extracting single element messages from a loop and combining them to vector-form (*communication vectorization*), removing redundant communication (*communication coalescing*), and aggregating different communication statements (*communication aggregation*).*

Contribution 7: *We have implemented our symbolic analysis approach in the context of the Vienna High Performance compiler (VFC) and have demonstrated the effectiveness of our approach for programs that have been taken from the Perfect, RiCEPS (Rice Compiler Evaluation Program Suite), Genesis benchmark suites, and WIEN95 software package.*

2.4 Organization

In Chapter 3 we introduce a novel symbolic analysis framework which is based on program contexts. In Chapter 4 we present an algorithm for computing program contexts based on control flow graphs. In Chapters 5 we present various symbolic algorithms to evaluate symbolic expressions, to simplify symbolic constraints, and to determine the number of solutions to a set of symbolic constraints. In Chapter 6 we describe the application of our symbolic analysis framework and discuss preliminary results of a prototype implementation in the context of a parallelizing compilers and a performance estimator. In Chapter 7 we compare existing approaches with our novel symbolic analysis approach and give a short summary of parallelizing compilers. We will conclude in Chapter 8 with an outlook on future work. In the appendix we present relevant background material and notations that are necessary to understand the work described in this book.

3. Symbolic Analysis of Programs

3.1 Introduction

Programming tools and compilers require sophisticated techniques to statically reason about program semantics. Existing program analysis techniques often fail to give precise answers about the meaning of a program due to imprecise approximations. In order to overcome the deficiencies of existing program analysis approaches we have continued the research in the field of symbolic analysis [53, 17, 97, 52, 15, 16]. Symbolic analysis is a static, global program analysis that examines each expression of the input program only once and attempts to derive a precise mathematical characterization of the computations. The result of this analysis is called "program context" [53], which comprises program semantics for an arbitrary program point. The program context is a symbolic representation of variable values or behaviors arising at run-time of the program. Therfore, symbolic analysis can be seen as a compiler that translates a program into a different language. As a target language we employ *symbolic expressions* and *symbolic recurrences*.

Despite sophisticated simplification techniques, symbolic analysis can not always compute useful answers due to theoretical limitations of semantic analysis where undecidable problems can occur [67]. However, in contrast to previous work our approach models program semantics as precise as possible. The price for this accuracy is reflected in an increased computational complexity.

For all statements of the program, our symbolic analysis deduces *program contexts* that describes the variable values, assumptions regarding and constraints between variable values and the condition under which the program point is reached. A program context c is defined by a triple $[s, t, p]$ that includes a state s, a state condition t, and a path condition p.

- State s:
  The *state* s is described by a set of variable/value pairs $\{v_1{=}e_1,\ldots,v_k{=}e_k\}$ where v_i is a program (scalar or array) variable and e_i a symbolic expression describing the value of v_i for $1 \leq i \leq k$. For each program variable v_i there exists exactly one pair $v_i = e_i$ in s.
- State condition t:
  In traditional program execution each time a branch statement is reached,

T. Fahringer and B. Scholz: Advanced Symbolic Analysis, LNCS 2628, pp. 13–40, 2003.

the values of program variables unambiguously determine the control flow. Based on a symbolic state s symbolic analysis cannot determine the control flow statically purely on the state. Hence, some assumptions about variable values must be taken in order to symbolically evaluate branch nodes. These assumptions about variable values are described by a *state condition* t. Additional constraints on variable values such as those implied by loops (recurrences), variable declarations and user assertions (specifying relationships between variable values) are also added to the state condition.

- Path condition p:
  The path condition p describes the condition under which control flow reaches a given program statement. The path condition p is specified by a logic formula that comprises the conditional expressions of branches that are taken to reach the program statement.

More formally, a context $c = [s, t, p]$ is a logical assertion $\overline{c} = s \wedge t \wedge p$, where $\overline{c}$ is a predicate over the set of program variables and the program input which are free variables. If for all input values, $\overline{c_{i-1}}$ holds before executing the statement ℓ_i, then $\overline{c_i}$ is the strongest post condition [36] and the program variables are in a state satisfying $\overline{c_i}$ after executing ℓ_i. Note that all components of a context — including state information — are described as symbolic expressions and recurrences. An unconditional sequence of statements ℓ_j $(1 \leq j \leq r)$ is symbolically evaluated by $[s_0, t_0, p_0]$ ℓ_1 $[s_1, t_1, p_1]$ $\ldots$ ℓ_r $[s_r, t_r, p_r]$. The initial context $[s_0, t_0, p_0]$ represents the context that holds before ℓ_1 and $[s_r, t_r, p_r]$ the context that holds after ℓ_r. If ℓ_j in the sequence $\ldots$ $[s_j, t_j, p_j]$ ℓ_j $[s_{j+1}, t_{j+1}, p_{j+1}]$ $\ldots$ does not contain any side effects (implying a change of a variable value) then $s_i = s_{i+1}$.

In the following we describe the symbolic analysis of various program constructs including assignments, input/output, branches, loops, recurrences, arrays, procedures, and dynamic records.

3.2 Initializations, Assignments, and Input/Output Operations

The code excerpt of Ex. 3.2.1 includes assignment and input/output statements. For every statement ℓ_j the corresponding context $[s_j, t_j, p_j]$ is displayed.

A statement in "[...]" denotes that this statement is not an executable statement but serves only for the purpose of illustrating the context of a given program statement.

In general, at the start of the program all variable values are undefined ($\perp$). More elaborate initializations are conducted for constants or explicit initializations of variables. For instance, Fortran 90 contains named constants, of the form **real,parameter ::** `A = 2.997` . These constants cannot be assigned values during the execution of the program whereas initialization

Example 3.2.1 *Statement sequence*

```
      integer::D1,D2,A,B
         [s0 = {D1 = ⊥, D2 = ⊥, A = ⊥, B = ⊥}, t0 = true, p0 = true]
ℓ1:   read(A,B)
         [s1 = δ(s0; A = ▽1, B = ▽2), t1 = t0, p1 = p0]
ℓ2:   D1 := (A-B)*A
         [s2 = δ(s1; D1 = (▽1 − ▽2) ∗ ▽1), t2 = t1, p2 = p1]
ℓ3:   D2 := 2*(A-B)
         [s3 = δ(s2; D2 = 2 ∗ (▽1 − ▽2)), t3 = t2, p3 = p2]
ℓ4:   read(B)
         [s4 = δ(s3; B = ▽3), t4 = t3, p4 = p3]
ℓ5:   assert(D1/D2 > A/2 - 1)
         [s5 = s4, t5 = t4, p5 = p4]
```

expressions such as **real ::** `A = 0.0` that assign an initial value to a variable in a type declaration statement and whose value might be changed during program execution.

For these cases the initial variable value is given by the value of the constant or the initialization expression, respectively. Additional constraints can be added to the path condition that are imposed by type information (`kind` function in Fortran 90) or provided by the user. However, by default the state condition t_0 and path conditions p_0 are set to *true*. In Section 3.6 the initialization of variables in procedures is discussed, which requires a parameter passing mechanism.

The statement ℓ_1 in Fig. 3.2.1 is a read-statement that assigns variables A, B unknown values during compile-time denoted by a special symbol ($\bigtriangledown$). For the analysis the symbolic input data values are of special interest as they are propagated through the program and thereby maintaining the important relationship of variable values and input data. Similar is done for other problem sizes such as machine and array sizes.

Most statements imply a change of only few variables. In order to avoid large lists of variable values in state descriptions only those variables whose value changes, are explicitly specified in the program context. For this reason we introduce function δ,

$$s_{j+1} = \delta(s_j; v_1 = e_1, \ldots, v_l = e_l) \tag{3.1}$$

The function δ changes the symbolic value of variables v_i $(1 \leq i \leq l)$ in state s_j. The value of the variables are specified by a list of assignements given as second parameter in function δ. For example in the Fig. 3.2.1 the read-statement transforms the state s_0 by changing the symbolic values of A, B. The new state s_1 is given by $\delta(s_0; A = \bigtriangledown_1, B = \bigtriangledown_2)$. The values of variables D_1 and D_2 remain unchanged and are still $\bot$.

Statements ℓ_2 and ℓ_3 only alter the values of variables D_1 and D_2. Statement ℓ_4 is again a read statement that assigns a new input data value $\bigtriangledown_3$ to

B. For verifying whether the assert statement ℓ_5 holds at this statement we evaluate the assert expression with respect to the program context immediately preceding ℓ_5 which is $[s_4, t_4, p_4]$. For evaluating an expression we define function *eval*

$$eval(e, [s, t, p])$$

Function *eval* symbolically evaluates the value of expression e for a specific program context [s,t,p]. In our example function *eval* is applied to the predicate of an assert statement in statement ℓ_5 as follows:

$$\begin{aligned} eval(\frac{D1}{D2} > \frac{A}{2} - 1, [s_4, t_4, p_4]) &= \frac{(\bigtriangledown_1 - \bigtriangledown_2) * \bigtriangledown_1}{2 * (\bigtriangledown_1 - \bigtriangledown_2)} > \frac{\bigtriangledown_1}{2} - 1 \\ &= \frac{\bigtriangledown_1}{2} > \frac{\bigtriangledown_1}{2} - 1 \\ &= 0 > -1 \\ &= true \end{aligned}$$

The assert statement is true at statement ℓ_5 under the assumption that $\bigtriangledown_1 - \bigtriangledown_2$ never evaluates to zero. If the difference was zero, the division operation would cause a program error in statement ℓ_5.

The path condition in our example of Fig. 3.2.1 does not vary from statement to statement. At the start it is initialized by true and in absence of control flow branches the path condition remains true for all program statements of the code.

Other symbolic analyses such as Static Single Assignment(SSA) Form [34] combined with forward substitution techniques can also determine that the assert statement is true at ℓ_5. However, conventional scalar forward substitution [103] of $D1$ and $D2$ fails to verify the assert statement. The definitions in ℓ_2 and ℓ_3 are killed by the read statement ℓ_4 and not enough information is available at statement ℓ_5.

Figure 3.1 lists the symbolic evaluation rules of simple assignments (AS), statement sequences (SQ), `read`-statements (RE), and the evaluation function *eval* expressed as denotational semantic rules. A brief introduction to denotational semantics [62, 86] can be found in Appendix 9.4.

The rule (AS) is applied for an assignment `<var> := <expr>`. It is a function that maps the assignment with an input context to a new context. The new context keeps track of the changed variable value of variable `var` whereby the new context uses function δ to change the symbolic variable value. The new variable value is obtained by symbolically evaluating the expression with $eval(< \texttt{expr} >, [s, t, p])$. Note that the state condition and path condition remain unchanged since there is no control flow involved.

Rules (E1)-(E3) describe the symbolic evaluation of expression. Rule (E1) represents the evaluation of constants in an expressions. Rule (E2) is used to extract the symbolic value of a variable from state s and rule (E3) is used for translating an operation to its symbolic domain. In general, function *eval*

$$
\begin{array}{ll}
 & \mathcal{F} : \texttt{<stmt>} \to C \to C \\
\text{(AS)} & \mathcal{F}[\![\texttt{<var> := <expr>}]\!] = \lambda[s,t,p] \in C. \\
 & \quad [\delta\,(s; \texttt{<var>} = \mathit{eval}(\texttt{<expr>}, [s,t,p]))\,, t, p] \\
\text{(SQ)} & \mathcal{F}[\![\texttt{<stmt>}_1\ \texttt{<stmt>}_2]\!] = \lambda[s,t,p] \in C. \\
 & \quad (\mathcal{F}[\![\texttt{<stmt>}_2]\!](\mathcal{F}[\![\texttt{<stmt>}_1]\!][s,t,p])) \\
\text{(RE)} & \mathcal{F}[\![\texttt{read <v>}]\!] = \lambda[s,t,p] \in C. \\
 & \quad [((\lambda\alpha.\delta\,(s; v = \bigtriangledown_\alpha, \iota = \alpha + 1))\mathit{eval}(\iota, [s,t,p])), t, p] \\
 & \\
 & \mathit{eval} : \texttt{<expr>} \times C \to E \\
\text{(E1)} & \mathit{eval}(\texttt{<const>}, [s,t,p]) = \texttt{<const>} \\
\text{(E2)} & \mathit{eval}(\texttt{<var>}, [s,t,p]) = e_i, \\
 & \quad \text{where } s = \{v_1 = e_1, \ldots, \texttt{<var>} = e_i, \ldots\} \\
\text{(E3)} & \mathit{eval}(\texttt{<expr>}_1\ \mathit{op}\ \texttt{<expr>}_2, [s,t,p]) = \\
 & \quad \mathit{eval}(\texttt{<expr>}_1, [s,t,p])\ \mathit{op}\ \mathit{eval}(\texttt{<expr>}_2, [s,t,p])
\end{array}
$$

Fig. 3.1. Symbolic evaluation of assignments

transforms expressions of the input program to symbolic expressions based on a given context $[s,t,p]$. It is important to stress that not only straight forward mapping from the input program to the symbolic domain is used. Sophisticated simplification [48, 46, 45] such as shown in the previous example are of key importance to keep the result of symbolic analysis as short as possible. For statement sequences according to Rule (SQ) we symbolically analyze the first statement with the program context before the statement. Then, the resulting program context is taken as the program context for the remaining statements.

The rule (RE) describes the symbolic effect of a read-statement. To record input values we employ a symbolic counter ι, that is incremented every time when a read statement is analyzed. The variable of the read statement is assigned a numbered generic symbolic value $\bigtriangledown_\iota$ where ι is the index in the input stream. With the numbered generic symbolic value we can identify specific input values which is crucical for maintaining the relationshop between variable values and input data.

3.3 Conditional Statements

In case of branches, symbolic analysis has to follow multiple control flow branches because symbolic values of variables might not suffice to select which branch to go. The impact of the control flow on the variable values is described by the state condition, whereas the condition under which the control flow reaches a program statement is given by the path condition. Both state and path conditions are specified as logic formulae. If a conditional statement is encountered, the conditional expression is evaluated by using the variable

Example 3.3.1 *Conditional statements*

ℓ_1: ***read***`(Y)`
$[s_1 = \{X = x_1, Y = \bigtriangledown_1\}, t_1 = true, p_1 = true]$
ℓ_2: ***if*** `(Y < 0)` ***then***
$[s_2 = s_1, t_2 = t_1, p_2 = (p_1 \wedge \bigtriangledown_1 < 0) = \bigtriangledown_1 < 0]$
ℓ_3: `X:=-2*Y`
$[s_3 = \delta(s_2; X = -2 * \bigtriangledown_1), t_3 = t_2, p_3 = p_2]$
ℓ_4: `Y:=-X`
$[s_4 = \delta(s_3; Y = 2 * \bigtriangledown_1), t_4 = t_3, p_4 = p_3]$
ℓ_5: ***else***
$[s_5 = s_1, t_5 = t_1, p_5 = (p_1 \wedge \bigtriangledown_1 \geq 0) = \bigtriangledown_1 \geq 0]$
ℓ_6: `X:=2*Y`
$[s_6 = \delta(s_5; X = 2 * \bigtriangledown_1), t_6 = t_5, p_6 = p_5]$
ℓ_7: ***end if***
$[s_7 = \delta(s_1; X = x_2, Y = y_1),$
$t_7 = \gamma(\bigtriangledown_1 < 0;\ x_2 = -2 * \bigtriangledown_1,\ y_1 = 2 * \bigtriangledown_1;\ x_2 = 2 * \bigtriangledown_1, y_1 = \bigtriangledown_1),$
$p_7 = (p_4 \vee p_6) = true\ ]$
ℓ_8: ***if*** `(Y ≥ 0)` ***then***
$[s_8 = s_7, t_8 = t_7, p_8 = p_7 \wedge y_1 \geq 0]$
$[s'_8 = \delta(s_7; Y = \bigtriangledown_1),\ t'_8 = \gamma(\bigtriangledown_1 < 0; x_2 = -2 * \bigtriangledown_1;\ x_2 = 2 * \bigtriangledown_1),\ p'_8 = \bigtriangledown_1 \geq 0]$
ℓ_9: `Y:=2*Y`
$[S_9 = \delta(s'_8; Y = 2 * \bigtriangledown_1), t_9 = t'_8, p_9 = p'_8]$
ℓ_{10}: ***end if***
$[s_{10} = \delta(s_7; Y = 2 * \bigtriangledown_1),\ t_{10} = \gamma(\bigtriangledown_1 < 0;\ x_2 = -2 * \bigtriangledown_1;\ x_2 = 2 * \bigtriangledown_1),$
$p_{10} = true\ ]$

bindings of the current context. If it cannot be statically determined whether for all input data sets either the true or the false branch has to be followed, then symbolic analysis has to consider both branches. For instance, in the program of Ex. 3.3.1 we assume that at the beginning of this code segment, X has the value x_1 and the state and path conditions t_1 and p_1 are *true*. In ℓ_2 a conditional expression without side-effects is evaluated, which implies no change in the program state. However, the corresponding path condition of all statements within the then-part of this statement includes $\bigtriangledown_1 < 0$ where $\bigtriangledown_1$ is the current value of Y. The corresponding state condition remains unchanged as the variable values are unequivocal. The same is valid for all statements in the else-part whose path condition includes $\bigtriangledown_1 \geq 0$. A confluence of two branches occurs after statement ℓ_7.

Two different contexts $[s_4, t_4, p_4]$ and $[s_6, t_6, p_6]$ reach this point. At the confluence node we have two symbolic values for variable X and Y. Therefore, we create an artificial variable x_2 and y_1 which are bound to two different expressions of variables X and Y depending on the path condition of the branches. For X, $x_2 = -2 * \bigtriangledown_1$ if the conditional expression of ℓ_2 holds, otherwise $x_2 = 2 * \bigtriangledown_1$. In the state condition this is represented by $(\bigtriangledown_1 < 0 \wedge x_2 = -2 * \bigtriangledown_1) \vee (\bigtriangledown_1 \geq 0 \wedge\ x_2 = 2 * \bigtriangledown_1)$. Similar is true for y_1, the value of Y, which can be bound to two different expressions depending on which

$$(1)\,[s, t = t' \wedge \gamma(cnd; x = e; x = f), p] =$$
$$\begin{cases} [s_{x\to e}, t'_{x\to e}, p_{x\to e}] & \text{if } (p_{x\to e} \Rightarrow cnd) \wedge (p_{x\to f} \Rightarrow cnd) \\ [s_{x\to f}, t'_{x\to f}, p_{x\to f}] & \text{if } (p_{x\to e} \Rightarrow \neg cnd) \wedge (p_{x\to f} \Rightarrow \neg cnd) \\ [s, t, p] & \text{otherwise} \end{cases}$$

$$(2)\,[s, t = t' \wedge \gamma(cnd; x = e; x = f), p] =$$
$$\begin{cases} [s_{x\to e}, t'_{x\to e}, p_{x\to e}] & \text{if } p \Rightarrow cnd \\ [s_{x\to f}, t'_{x\to f}, p_{x\to f}] & \text{if } p \Rightarrow \neg cnd \\ [s, t, p] & \text{otherwise} \end{cases}$$

$$(3)\,[s, t = t' \wedge \gamma(cnd; x = e; x = f) \wedge \gamma(cnd'; y = e'; y = f'), p] =$$
$$\begin{cases} [s, t' \wedge \gamma(cnd; x = e; x = f) \wedge \gamma(cnd'; y = e'_{x\to e}; y = f'), p] & \text{if } cnd' \Rightarrow cnd \\ [s, t' \wedge \gamma(cnd; x = e; x = f) \wedge \gamma(cnd'; y = e'; y = f'_{x\to e}), p] & \text{if } \neg cnd' \Rightarrow cnd \\ [s, t' \wedge \gamma(cnd; x = e; x = f) \wedge \gamma(cnd'; y = e'_{x\to f}; y = f'), p] & \text{if } cnd' \Rightarrow \neg cnd \\ [s, t' \wedge \gamma(cnd; x = e; x = f) \wedge \gamma(cnd'; y = e'; y = f'_{x\to f}), p] & \text{if } \neg cnd' \Rightarrow \neg cnd \\ [s, t, p] & \text{otherwise} \end{cases}$$

Fig. 3.2. Simplification rules of γ

branch is taken at ℓ_2. In ℓ_7, ℓ_8, and ℓ_{10} of the example we use a function γ to avoid long logic expressions. Function γ is defined as,

$$\begin{aligned} &\gamma(cnd;\ x_1 = e_1, \ldots, x_k = e_k;\ x_1 = f_1, \ldots, x_k = f_k) = \\ &\quad (cnd \wedge x_1 = e_1 \wedge \cdots \wedge x_k = e_k) \vee \\ &(\neg\ cnd \wedge x_1 = f_1 \wedge \cdots \wedge x_k = f_k), \end{aligned} \tag{3.2}$$

where cnd is a conditional expression, $\neg cnd$ the negation of cnd, x_i $(1 \leq i \leq k)$ represent variable values, and e_i, f_i are symbolic expressions. In Fig. 3.2 we show some simplification rules of contexts with γ functions where cnd, cnd' are conditional expressions and e, f, e', f' are symbolic expressions.

Based on the definition of the γ function we can write the state condition t_7 for the previous code as $\gamma(\bigtriangledown_1{<}0;\ x_2{=}{-}2{*}\bigtriangledown_1, y_1{=}2{*}\bigtriangledown_1;\ x_2{=}2{*}\bigtriangledown_1, y_1{=}\bigtriangledown_1)$.

The path condition p_8 is equal to $p_7 \wedge (y_1 \geq 0)$ where $y_1 \geq 0$ is the constraint induced by the conditional expression of ℓ_8. Context $[S_8, t_8, p_8]$ can be rewritten to $[S_8,\ \gamma(\bigtriangledown_1{<}0; x_2{=}{-}2 * \bigtriangledown_1;\ x_2{=}2 * \bigtriangledown_1) \wedge \gamma(\bigtriangledown_1{<}0; y_1{=}2 * \bigtriangledown_1; y_1{=}\bigtriangledown_1), p_8]$ according to the following rewrite rule:

$$\begin{aligned} &\gamma(cnd;\ x_1 = e_1, \ldots, x_k = e_k;\ x_1 = f_1, \ldots, x_k = f_k) = \\ &\quad \gamma(cnd;\ x_1 = e_1;\ x_1 = f_1) \wedge \\ &\quad \gamma(cnd;\ x_2 = e_2, \ldots, x_k = e_k;\ x_2 = f_2, \ldots, x_k = f_k) \end{aligned} \tag{3.3}$$

We can simplify $[s_8, t_8, p_8]$ to $[s'_8, t'_8, p'_8]$ based on rule (1) of Fig. 3.2. Note that $s_{x \to e}$ means that all occurrences of x in s are simultaneously substituted by e. The first two rules of Fig. 3.2 are used to simplify a context immediately after a branch whose conditional expression impacts the associated state condition. By using rules (1) or (2) we can eliminate a γ function $\gamma(cnd; ...; ...)$ in the state condition of a context c and simplify c if the path condition of C implies cnd or $\neg cnd$. The third rule may simplify a context at a confluence node that implies a new γ function. This rule can eliminate additional unknowns (variable values) in γ functions if the condition of one γ function implies the condition of another γ function. If we replace a γ function by its equivalent logic expression, then we can also use the simplification techniques as described in [90].

Another confluence of two branches occurs after statement ℓ_{10}. Two different contexts $[s_7, t_7, p_7 \wedge y_1 < 0]$ and $[s_9, t_9, p_9]$ reach this point. Context $[s_7, t_7, p_7 \wedge y_1 < 0]$ is a dummy context of the missing else branch of statement n_8. The dummy context can be easily made explicit by adding an artificial else branch with an *empty* statement to every "*if then statement-sequence end if*" construct. $[s_7, t_7, p_7 \wedge y_1 < 0]$ can be rewritten to $[s'_7{=}\delta(s_7;\mathrm{Y}{=}2{*}\bigtriangledown_1)$, $t'_7{=}\gamma(\bigtriangledown_1{<}0; x_2{=}{-}2 * \bigtriangledown_1;\ x_2{=}2 * \bigtriangledown_1)$, $p'_7{=}2 * \bigtriangledown_1{<}0]$ based on the rewrite rule for the γ function and rule (1) of Fig. 3.2. At the confluence node after ℓ_{10} the value of X can have two different values depending on whether or not the true-branch of ℓ_8 has been taken. The value of Y is identical in both contexts that reach this point.

More formally, we introduce a folding operator $\odot$ for collapsing analysis information at confluence nodes [30, 80]. The folding operator joins two contexts $[s', t', p']$ and $[s'', t'', p'']$ and performs a symbolic state composition $[s''', t''', p''']$. Let

$$[s', t', p'] \odot [s'', t'', p''] = [s''', t''', p'''], \tag{3.4}$$

where

$$s' = \{v_1 = e_1, \ldots, v_n = e_n\}, \tag{3.5}$$
$$s'' = \{v_1 = f_1, \ldots, v_n = f_n\}, \tag{3.6}$$

then the result $[s''', t''', p''']$ is defined such that

$$g_i = \begin{cases} e_i & : \ \text{if } e_i = f_i, \\ \text{new symbol} & : \ \textit{otherwise}, \end{cases}$$
$$s''' = \{v_1 = g_1, \ldots, v_i = g_i, \ldots, v_n = g_n\},$$
$$\Omega' = \bigwedge_{1 \le i \le n, e_i \neq f_i} (g_i = e_i),$$

$$
\begin{array}{ll}
& \mathcal{F} : \texttt{<stmt>} \to C \to C \\
\text{(I1)} & \mathcal{F}[\![\texttt{if <cond> then <stmt> end if}]\!] = \\
& \quad \lambda[s,t,p] \in C.\ (\mathcal{F}[\![\texttt{<stmt>}]\!]\,[s,t,p \wedge \mathit{eval}(\texttt{<cond>},[s,t,p])]) \odot \\
& \qquad\qquad\qquad [s,t,p \wedge \neg \mathit{eval}(\texttt{<cond>},[s,t,p])] \\
\text{(I2)} & \mathcal{F}[\![\texttt{if <cond> then <stmt>}_1 \texttt{ else <stmt>}_2 \texttt{ end if}]\!] = \\
& \quad \lambda[s,t,p] \in C.\ (\mathcal{F}[\![\texttt{<stmt>}_1]\!]\,[s,t,p \wedge \mathit{eval}(\texttt{<cond>},[s,t,p])]) \odot \\
& \qquad\qquad\qquad (\mathcal{F}[\![\texttt{<stmt>}_2]\!]\,[s,t,p \wedge \neg \mathit{eval}(\texttt{<cond>},[s,t,p])])
\end{array}
$$

Fig. 3.3. Symbolic evaluation of if-statements

$$
\begin{aligned}
\Omega'' &= \bigwedge_{1 \le i \le n, e_i \not\equiv f_i} (g_i = f_i), \\
t''' &= (t' \vee t'') \wedge ((p' \wedge \Omega') \vee (p'' \wedge \Omega'')), \\
p''' &= p' \vee p''.
\end{aligned}
$$

The definition of the folding operator $\odot$ considers only variables with different symbolic values in state s' and state s''.

For those variables with different symbolic values in s and s' we introduce new symbols whose values are bound either to the symbolic value of s' or s'' depending on the path condition as given in state condition t'''. The path condition of the folded contexts is the disjunction of p' and p''.

The denotational semantic rules for symbolically evaluating if-statements are given in Fig. 3.3. Rule (I1) is used in order to evaluate if-statements without else-branches. Similarily, Rule (I2) is applied to if-statements with else-branches. Before evaluating the branches, the condition `<cond>` of the if-statement is evaluated. The path condition for evaluating the branches is given as the conjunction of the original path condition s and either the evaluated condition or the negated one depending on the branch. The result of both branches are joined by applying the folding operator $\odot$.

3.4 Loops and Recurrences

The power of symbolic analysis depends critically on its ability to analyze loops. However, loops impose severe problems for symbolic analysis. Loops have no static behavior. It is a non-trivial task to derive a mathematical function describing the semantics of loops since loops may introduce an infinite number of execution paths in a given program. By introducing *symbolic recurrences* we are able to capture the semantic behavior of loops without loosing analysis precision. However, it is not always possible to find static patterns in symbolic recurrences, i.e. determining closed forms, and manipulating symbolic recurrences. Nevertheless, symbolic recurrences is a vehicle in order to (1) describe loops in a symbolic domain and (2) to provide a solid interface for algebraic systems that solve recurrences [82].

Modeling loops implies a problem with variables and memory locations which are changed inside of loops. We denote such variables as *recurrence (induction) variables*. For example in a simple loop of the form `DO I,1,N .... END DO`, we have the problem that recurrence variable `I` is continousely changed with a value range from 1 to `N`. Everytime when the loop is executed, the value of `I` is incremented by one. We express this change in form of recurrences. For this example the reccurrence relation is given as $i(k+1) = i(k)+1$ where $i(k+1), k \geq 0$ is the value of variable `I` at the end of iteration $k+1$. Before entering the loop, the variable value of `I` is 1 and, therefore, we initialize the recurrence by $i(0) = 1$. In the remainder of this book, we refer to the initial value of a recurrence as *boundary condition*. Based on the recurrence relation and the boundary condition we symbolically represent the computations of the loop. To determine the value of `I` after the loop, we need the *recurrence condition* which symbolically determines the number of iterations for the recurrence. In our example the recurrence condition is given by $i(k) \leq N$. We have described the boundary condition, the recurrence relation and the recurrence condition in a pure mathematical denotation. To collect the loop information as part of the program context we introduce function

$$\mu(v, s, c)$$

which comprises a symbolic recurrence system. The recurrence relations of the recurrence variables are given in program context c as part of the state and state condition. The recurrence condition is represented by the path condition of c. The boundary condition of the reccurrence system is determined by state s of function μ, and variable v determines which recurrence variable is selected for the result of the μ function.

Consider the code in Ex. 3.4.1. There are two recurrence variables – `F` and `I` – whose values change during execution of the loop. Inside the loop we assign both recurrence variables the values $\underline{I}$ and $\underline{F}$ respectively after statement ℓ_1 which denote the symbolic values of either the previous iteration or the values before entering the loop. After the `while`-statement we collect the program context $[s_3, t_3, p_3]$ of statement ℓ_3 and the state condition of ℓ_0 for generating the μ functions of recurrence variables `I` and `F`. After the loop context $[s_4, t_4, p_4]$ assigns both recurrence variables `F` and `I` new symbolic values. Variable `F` is assigned the value of $\mu(F, s_0, [s_3, t_3, p_3])$ and `I` the value of $\mu(I, s_0, [s_3, t_3, p_3])$. Both μ-functions are a complete symbolic representation of the recurrence system of `F` and `I`. State s_0 contains the values of `F` and `I` before entering the loop ($\{..., F = y, I = b, ...\}$). Context $[s_3, t_3, p_3]$ determines (1) the recurrence relations as part of the state

$$F = 2 * \underline{F}$$
$$I = \underline{I} + b$$

Example 3.4.1 *Loops*

$[s_0 = \{M = m, F = y, I = b, B = b\}, t_0 = b > 0 \wedge m > 0, p_0 = \text{true}]$
ℓ_1: ***do while*** `I ≤ M`
$[s_1 = \{F = \underline{F}, I = \underline{I}\}, t_1 = true, p_1 = \underline{I} \leq m]$
ℓ_2: `F := 2*F`
$[s_2 = \delta(s_1; F = 2 * \underline{F}), t_2 = t_1, p_2 = p_1]$
ℓ_3: `I := I + B`
$[s_3 = \delta(s_2; I = \underline{I} + b, t_3 = t_2, p_3 = p_2]$
ℓ_4: ***end do***
$[s_4 = \delta\,(s_0; I = \mu(I, s_0, [s_3, t_3, p_3]), F = \mu(F, s_0, [s_3, t_3, p_3]))\,,$
$t_4 = t_0, p_4 = p_0] =$
$[s'_4 = \delta(s_0; F = y * 2^{\lceil \frac{m+1}{b} \rceil - 1}, I = b * \lceil \frac{m+1}{b} \rceil), t'_4 = t_4, p'_4 = p_4]$

and (2) the recurrence condition as part of the path condition, which is $\underline{I} \leq m$. Now we can rewrite our μ-function as a recurrence system in mathematical terms. First, the variables `F` and I on the left-hand side of the boundary condition s_0 are rewritten as $f(0)$ and $i(0)$, respectively. Second, in the program context $[s_3, t_3, p_3]$ we rewrite variables `F` and `I` on the right-hand side as $f(k)$ and $i(k)$, respectively where k is the recurrence index and determines the iteration. Third, variables `F` and `I` on the left-hand side are rewritten as $f(k+1)$ and $i(k+1)$.

Now we have a recurrence system in mathematical notation and we can try to simplify and to automatically compute closed forms for recurrence variables `F` and `I` at the loop exit (after the `end do`).

$$\begin{aligned} f(k+1) &= 2 * f(k), \quad k \geq 0 \\ i(k+1) &= i(k) + b, \quad k \geq 0 \\ f(0) &= y \\ i(0) &= b \end{aligned}$$

We first need to find the recurrence relations and the boundary conditions for $n \geq 0$ as given in the context $[s_4, t_4, p_4]$: The solution (loop invariant) to the previous recurrences is given as follows:

$$f(k) = y * 2^k \tag{3.7}$$
$$i(k) = b * (k+1) \tag{3.8}$$

Based on (3.8) and $i(n) > m$ (recurrence condition) we can determine the last iteration of the loop which is equivalent to finding a z for which the following holds

$$z = min\{k \mid i(k) > m\} \tag{3.9}$$
$$= min\{k \mid b * (k+1) > m\} \tag{3.10}$$
$$= \left\lceil \frac{m+1}{b} \right\rceil - 1 \tag{3.11}$$

The ceiling operation in (3.11) is necessary as the loop condition is evaluated based on integer arithmetic. In order to compute the values of F and I at the loop exit, we have to substitute z for k in (3.7) and (3.8):

$$F = y * 2^{\lceil \frac{m+1}{b} \rceil - 1}$$
$$I = b * \left\lceil \frac{m+1}{b} \right\rceil$$

which specifies the variable binding of `F` and `I` at the loop exit $[s'_4, t'_4, p'_4]$. As we can determine closed forms for the recurrence variables (`I` and `F`) of the loop, t'_4 and p'_4 are, respectively, identical with t_0 and p_0.

More formally, the denotational semantics of the while-loop is given in Fig. 3.4. A `while`-statement only effects recurrence variables. This is taken into account by assigning new symbolic values to recurrence variables in the state of the program context as shown in Rule (WI). The function *loopeval* analyses the loop body and initializes the recurrence systems of the recurrence variables $\{v_1, \ldots, v_l\}$. The function performs three steps: First, the boundary condition is stripped off from the program context $[s, t, p]$ before entering the loop.

$$s_0 = \{v_1 = eval(v_1, [s, t, p]), \ldots, v_l = eval(v_l, [s, t, p])\} \tag{3.12}$$

Second, in order to symbolically analyze the loop body, we compute an initial program context. This program context comprises the generic symbolic values for the recurrence variables and the loop condition.

$$s' = \delta(s; v_1 = \underline{v_1}, \ldots, v_l = \underline{v_l}) \tag{3.13}$$
$$c' = [s', \text{true}, eval(\texttt{<cond>}, [s', \text{true}, \text{true}])] \tag{3.14}$$

With the initial program context we can symbolically derive the recurrence relation as follows,

$$c'' = (F[\![\texttt{<stmt>}]\!]c') \tag{3.15}$$

Function *loopeval* expresses recurrence variables as symbolic recurrences by μ-functions.

$$loopeval(\texttt{<cond>}, \texttt{<stmt>}, [s, t, p]) = \delta(s; v_1 = \mu(v_1, s_0, c''), \ldots, v_1 = \mu(v_1, s_0, c'')) \tag{3.16}$$

The goal of our symbolic analysis with respect to loops is to detect the recurrence variables, determine the recurrence system, and finally find closed forms for recurrence variables at the loop exit by solving the recurrence system. The *recurrence system* is given by the *boundary conditions* (initial values for recurrence variables in the loop preheader), the *recurrence relations* (implied by the assignments to the recurrence variables in the loop body), and the *recurrence condition* (loop or exit condition). Our approach can detect all recurrences implied by recurrence variables and represent them by a recurrence system.

$$
\begin{array}{ll}
 & F : \texttt{<stmt>} \to C \to C \\
\text{(WI)} & F[\![\texttt{while <cond> do <stmt> end do}]\!] = \\
 & \quad \lambda[s,t,p] \in C.[\mathit{loopeval}(\texttt{<cond>}, \texttt{<stmt>}, [s,t,p]), t, p]
\end{array}
$$

Fig. 3.4. Symbolic evaluation of while-statement

We have implemented a recurrence solver written on top of Mathematica [105]. The recurrence solver tries to determine closed forms for recurrence variables based on their recurrence system. The implementation of our recurrence solver [95] is largely based on methods described in [56, 82]. In general it is not possible to find closed forms for every recurrence [56]. If we cannot find a closed form for a recurrence we introduce new value symbols for every unresolved variable. An unresolved variable is a recurrence variable for which no closed form can be determined. Value symbols of unresolved variables are linked with their associated recurrence system. Evaluation of unresolved variables is then done based on their value symbols instead of their recurrence system. In [47] algorithms are introduced to determine the monotonic behavior of unresolved variables. This can be important for applied symbolic analysis (for instance, dependence analysis) to reason about expressions containing unresolved variables.

3.5 Arrays

Let `A` be a one-dimensional array with n ($n \geq 1$) array elements. Consider the simple array assignment `A(I)=V`. The element with index `I` is substituted by the value of `V`. Intuitively, we may think of an array assignment as an array operation that is defined for an array. The operation is applied to the array and changes its internal state. The arguments of such an array operation are a value and an index of the newly assigned array element. A sequence of array assignments implies a chain of operations. For symbolic analyses we represent array operations upon an array as an element of an *array algebra* $\mathbb{A}$. The array algebra $\mathbb{A}$ is inductively defined as follows.

1. If n is a symbolic expression then $\perp_n \in \mathbb{A}$.
2. If $a \in \mathbb{A}$ and α, β are symbolic expressions then $a \oplus (\alpha, \beta) \in \mathbb{A}$.

In the state of a context, an array variable is associated with an element of the array algebra $\mathbb{A}$. Undefined array states are denoted by $\perp_n$ where n is a symbolic constant and represents the size of the array. An array assignment previously referred to as array operation is modelled by $\oplus$-*functions*. The semantics of the $\oplus$-function is given by

$$a \oplus (\alpha, \beta) = (v_1, \ldots, v_{\beta-1}, \alpha, v_{\beta+1}, \ldots, v_n)$$

where $(v_1, \ldots, v_n)$ represents the elements of array a and β denotes the index of the element with a new value α. An element a in $\mathbb{A}$ with at least one $\oplus$-function is a $\oplus$-*chain*[1]. Every $\oplus$-chain can be written as $\bot_n \bigoplus_{k=1}^{m}(\alpha_k, \beta_k)$. The *length of a chain* $|a|$ is the number of $\oplus$-functions in chain a.

For the following array assignment

$$
\begin{array}{ll}
 & [s_{i-1} = \{\ldots, A = a, \ldots\}, t_{i-1}, p_{i-1}] \\
\ell_i: & \texttt{A(I) = V} \\
 & [s_i = \delta(s_{i-1}; A = a \oplus (\mathit{eval}(V, [s_{i-1}, t_{i-1}, p_{i-1}]), \\
 & \quad \mathit{eval}(I, [s_{i-1}, t_{i-1}, p_{i-1}]))), t_i = t_{i-1}, p_i = p_{i-1}]
\end{array}
$$

we take the symbolic value $a \in \mathbb{A}$ of variable `A` and add a new $\oplus$-function to it. The $\oplus$-function has two parameters. The first one contains the symbolic value of the right-hand side of the array. The second one is the symbolic value of the index. Both symbolic values are computed by evaluating variables `I` and `V` with program context $[s_{i-1}, t_{i-1}, p_{i-1}]$.

Example 3.5.1 *Array assignments*

$$
\begin{array}{ll}
 & \textbf{\textit{integer, dimension}}\texttt{(100)::A} \\
 & \quad [s_0 = \{A = \bot_{100}, X = x\}, t_0 = \text{true}, p_0 = \text{true}] \\
\ell_1: & \texttt{A(x)=1} \\
 & \quad [s_1 = \delta\,(s_0; A = \bot_{100} \oplus (1, x))\,, t_1 = t_0, p_1 = p_0] \\
\ell_2: & \texttt{A(x+1)=1-x} \\
 & \quad [s_2 = \delta\,(s_1; A = \bot_{100} \oplus (1, x) \oplus (1 - x, x + 1))\,, t_2 = t_1, p_2 = p_1] \\
\ell_3: & \texttt{A(x)=x} \\
 & \quad [s_3 = \delta\,(s_2; A = \bot_{100} \oplus (1, x) \oplus (1 - x, x + 1) \oplus (x, x))\,, t_3 = t_2, p_3 = p_2] \\
\ell_4: & \texttt{A(x+1)=1+x} \\
 & \quad [s_4 = \delta\,(s_3; A = \bot_{100} \oplus (1, x) \oplus (1 - x, x + 1) \oplus (x, x) \oplus (1 + x, x + 1))\,, \\
 & \qquad t_4 = t_3, p_4 = p_3]
\end{array}
$$

Consider the program in Ex. 3.5.1 that consists of simple array assignments. At the beginning of the program fragment the value of variable `X` is a symbolic expression denoted by x. Array `A` is undefined ($\bot_{100}$). After the last statement, array `A` has symbolic value $A = \bot_{100} \oplus (1, x) \oplus (1 - x, x + 1) \oplus (x, x) \oplus (1 + x, x + 1)$. The left-most $\oplus$-function relates to the first assignment of program in Ex. 3.5.1 — the right-most one to the last statement. We can give a simplified representation of `A` since the last two statements overwrite the values of the first two statements. After simplification, variable `A` has the symbolic value $\bot_{100} \oplus (x, x) \oplus (1 + x, x + 1)$.

For simplifying $\oplus$-chains we need to find out if two symbolic expressions are equal or not. In general this question is undecidable [47, 67], although

[1] In the following we distinguish between an element a of the array algebra written in small capitals and a the program variable written in big capitals.

a wide class of equivalence relations can be solved in practice. The set of conditions among the used variables in the context significantly improves the evaluation of equivalence relation. We apply a partial *simplification operator* θ to $\oplus$-chains in order to simplify the $\oplus$-function. The operator θ is defined as follows,

$$\theta\left(\perp_n \bigoplus_{l=1}^{m}(\alpha_l,\beta_l)\right)=\begin{cases}\perp_n \bigoplus_{l=1,l\neq i}^{m}(\alpha_l,\beta_l), & \text{if } \exists 1\leq i<j\leq m:\beta_i=\beta_j\\ \perp_n \bigoplus_{l=1}^{m}(\alpha_l,\beta_l), & \text{otherwise}\end{cases} \tag{3.17}$$

The partial simplification operator θ seeks for two equal β expressions in a $\oplus$-chain. If a pair exists, the result of θ will be the initial $\oplus$-chain without the $\oplus$-function, which refers to the β expression with the smaller index i. If no pair exists, the operator returns the initial $\oplus$-chain; the chain could not be simplified. Semantically, the right-most β expression relates to the latest assignment and overwrites the value of the previous assignment with the same *symbolic index*.

The partial simplification operator θ reduces only one redundant $\oplus$-function. In the previous example θ must be applied twice in order to simplify the $\oplus$-chain. Moreover, each $\oplus$-function in the chain is a potentially redundant one. Therefore, the chain is potentially simplified in less than $|a|$ applications of θ. A partially complete simplification is an iterative application of the partial simplification operator and it is written as $\theta^*(a)$. If $\theta^*(a)$ is applied to a, further applying of θ will not simplify a anymore:

$$\theta(\theta^*(a))=\theta^*(a).$$

To access elements of an array we need to model array accesses symbolically. *Operator* ρ reads an element with index i of an array `A`. If index i can be found in the $\oplus$-chain, ρ yields the corresponding symbolic expression, otherwise ρ is the undefined value $\perp$. In the latter case it is not possible to determine whether the array element with index i was written. Let a be an element of $\mathbb{A}$ and $a=\perp_n \bigoplus_{l=1}^{m}(\alpha_l,\beta_l)$. The operator ρ is defined as

$$\rho\left(\perp_n \bigoplus_{l=1}^{m}(\alpha_l,\beta_l),i\right)=\begin{cases}\alpha_l, & \text{if } \exists l=\max\{\,l\,|\,1\leq l\leq m\wedge\beta_l=i\}\\ \perp, & \text{otherwise}\end{cases} \tag{3.18}$$

where i is the *symbolic index* of the array element to be found. If our symbolic analysis framework cannot derive that the result of ρ is β_l or $\perp$ then ρ is not resolvable.

We present four examples which are based on the value of `A` at the end of the program fragment in Ex. 3.5.1. Assume that array `A` has the symbolic value $\perp_{100}\oplus(x,x)\oplus(1+x,x+1)$ and, the values of variables `X` and `Y` are given by x and y, respectively.

$$
\begin{array}{lll}
(1) \quad \texttt{V=A(X)} & \Longrightarrow & V = \rho(A, x) = x \\
(2) \quad \texttt{V=A(X+1)} & \Longrightarrow & V = \rho(A, x+1) = x+1 \\
(3) \quad \texttt{V=A(X-1)} & \Longrightarrow & V = \rho(A, x-1) = \bot \\
(4) \quad \texttt{V=A(Y)} & \Longrightarrow & V = \rho(A, y) = \begin{cases} x, & \text{if } y = x \\ x+1, & \text{if } y = x+1 \\ \bot, & \text{otherwise} \end{cases}
\end{array}
$$

In the first array access (1) the element with index x is uniquely determined. The second one (2) is resolved as well. In the third access (3) the index $x-1$ does not exist in the $\oplus$-chain, therefore, the symbolic evaluation computes the undefined symbol $\bot$. In the last array access (4) we do not have enough information to determine a unique symbolic value for index y. Here, we distinguish between multiple cases depending on the value of y.

Array manipulations inside of loops are described by recurrences. A recurrence system over $\mathbb{A}$ consists of a boundary condition and a recurrence relation

$$
\begin{aligned}
a(0) &= b, b \in \mathbb{A}, \\
a(k+1) &= a(k) \bigoplus_{l=1}^{m} (\alpha_l(k), \beta_l(k)),
\end{aligned}
$$

where $\alpha_l(k)$ and $\beta_l(k)$ are symbolic expressions and k is the recurrence index with $k \geq 0$. Clearly, every instance of the recurrence is an element of $\mathbb{A}$. Without changing the semantics of an array recurrence, θ^* can be applied to simplify the recurrence relation.

Operator ρ needs to be extended for array recurrences such that arrays written inside of loops can be accessed, i.e. $\rho(a(z), i)$. The symbolic expression z is the number of loop iterations determined by the loop exit condition and i is the index of the accessed element. Furthermore, the recurrence index k is bound to $0 \leq k \leq z$. To determine a possible $\oplus$-function, where the accessed element is written, a *potential index set* $X_l(i)$ of the l-th $\oplus$-function is computed.

$$\forall 1 \leq l \leq m : X_l(i) = \{\, k |\, \beta_l(k) = i \wedge 0 \leq k \leq z\}$$

$X_l(i)$ contains all possible $\beta_l(k)$, equal to the index i $(0 \leq k \leq z)$. If an index set has more than one element, the array element i is written in different loop iterations by the l-th $\oplus$-function. Only the last iteration that writes array element i is of interest. Consequently, we choose the element with the greatest index. The *supremum* $x_l(i)$ of an index set $X_l(i)$ is the greatest index where

$$\forall 1 \leq l \leq m : x_l(i) = \max X_l(i).$$

Finally, we define operator ρ as follows.

$$\rho\,(a(z),i) = \begin{cases} \alpha_l(x_l(i)), \text{ if } \exists 1 \le l \le m : x_l(i) = \max_{1\le l\le m} x_l(i) \\ \rho(a(0),i), \text{ otherwise} \end{cases}$$

The maximum of the supremum indices $x_l(i)$ determines the symbolic value $\alpha_l(x_l(i))$. If no supremum index exists, ρ returns the access to the value before the loop.

Example 3.5.2 *Array example with read access and loop*

$[s_0 = \{M = m, I = \bot, A = \bot_m, W = \bot_m\}, t_0 = \text{true}, p_0 = 2 \le m]$
$[s'_0 = \delta(s_0; I = 1), t'_0 = t_0, p'_0 = p_0]$

ℓ_1: `do I=1,M-1`
$[s_1 = \delta(s_0; A = \underline{A}, I = \underline{I}), t_1 = \text{true}, p_1 = \underline{I} \le m - 1]$

ℓ_2: `A(I):=M-I`
$[s_2 = \delta(s_1; A = \underline{A} \oplus (m - \underline{I}), t_2 = t_1, p_2 = p_1]$
$[s'_2 = \delta(s_2; I = \underline{I} + 1, t'_2 = t_2, p'_2 = p_2]$

ℓ_3: `end do`
$[s_3 = \delta(s_0; I = m, A = \mu(A, s_0, [s'_2, t'_2, p'_2])), t_3 = t'_0, p_3 = p'_0]$
$[s'_3 = \delta(s_3; I = 1), t'_3 = t_3, p'_3 = p_3]$

ℓ_4: `do I=1,M-1`
$[s_4 = \delta(s_3; W = \underline{W}, I = \underline{I}), t_4 = t_3, p_4 = \underline{I} \le m - 1]$

ℓ_5: `W(A(I)):=I+1`
$[s_5 = \delta(s_4; W = \underline{W} \oplus (\underline{I} + 1, \rho(\mu(A, s_0, [s'_2, t'_2, p'_2]), \underline{I}))), t_5 = t_4, p_5 = p_4]$
$[s'_5 = \delta(s_5; I = \underline{I} + 1), t'_5 = t_5, p'_5 = p_5]$

ℓ_6: `end do`
$[s_6 = \delta(s'_0; I = M, W = \mu(W, s'_0, [s'_5, t'_5, p'_5])), t_6 = t'_0, p_6 = t'_6]$

Example 3.5.2 illustrates how to handle linear and non-linear array accesses. In the case of indirect array accesses such as $W(A(I))$ where an index array A is used to access a specific element of array W, our techniques commonly achieve better results (in terms of closed forms) for structural than for input dependent array accesses. Structural array references imply that array indices and index arrays are symbolic expressions whose domain is supplied by the program variables and integer constants. Array indices of input dependent array references depend on input data. The code excerpt in in Ex. 3.5.2 is an irregular code and consists of two loops. An array A is written in the first loop. The array element values are functions over the loop index variable and a loop invariant. In the second loop, A is used as an index array for array W.

Statement ℓ_2 assigns array `A` value `M-I` which is symbolically described by $A = \underline{A} \oplus (m - \underline{I}, \underline{I})$. The symbolic recurrence of `A` is described after the loop in statement ℓ_3. We rewrite the symbolic recurrence $\mu(A, s_0, [s'_2, t'_2, p'_2])$ in a mathematical notation as,

$$
\begin{aligned}
I(0) &= 1 \\
I(k+1) &= I(k) + 1, \quad k \geq 0 \\
A(0) &= \perp_m \\
A(k+1) &= A(k) \oplus (m - I(k)), I(k)), \quad k \geq 0 \\
z &= min\{k | I(k) > m - 1\}
\end{aligned}
$$

where the symbolic value of `A` is $A(z)$ since z determines the last iteration of the recurrence. In order to resolve the indirect access of Statement ℓ_4 we compute the closed form of I which is $I(k) = 1 + k$ and replace $I(K)$ with its closed form. Then, the index of the last iteration z can be rewritten to $z = m - 1$ and we now know that $A(m-1)$ represents the symbolic array description of array `A` after Statement ℓ_3. After simplifications the following recurrence expresses the value of variable `A`:

$$
\begin{aligned}
A(0) &= \perp_m \\
A(k+1) &= A(k) \oplus ((m - k - 1), k + 1), \quad k \leq 0
\end{aligned}
$$

In the next step we want to replace indirect array access `W` whose index expression is given by `A(I)` in statement ℓ_5. The symbolic recurrence of W is written in mathematical notation as follows,

$$
\begin{aligned}
I(0) &= 1 \\
I(k+1) &= I(k) + 1, \quad k \geq 0 \\
W(0) &= \perp_m \\
W(k+1) &= W(k) \oplus (I(k) + 1, \rho(A(m-1), I(K)), \quad k \geq 0 \\
z &= min\{k | I(k) > m - 1\}
\end{aligned}
$$

whereby the symbolic recurrence of `A` is already replaced by the result as given above. Similar to the first loop we replace the occurrences of $I(k)$ with its closed form and compute the last index. Then we obtain,

$$
\begin{aligned}
W(0) &= \perp_m \\
W(k+1) &= W(k) \oplus (k + 1, \rho(A(m-1), k+1)), \quad k \geq 0
\end{aligned}
$$

Based on the computation of the last iteration z we now know that k is in the range between 0 and $m-1$. We further deduce that for index $k+1$ the symbolic recurrence of $A(m-1)$ stores the symbolic value $m-k-1$. Based on the read operation ρ we rewrite the symbolic recurrence of W as,

$$
\begin{aligned}
W(0) &= \perp_m \\
W(k+1) &= W(k) \oplus (k + 1, m - k - 1), \quad k \geq 0
\end{aligned}
$$

By backward substitution of $I(k)$ we can rewrite the symbolic recurrence of W as,

$$[s_5 = \delta(s_4; W = \underline{W} \oplus (\underline{I} + 1, m - \underline{I})), \ldots .$$

$$
\begin{array}{ll}
 & \mathcal{F} : \texttt{<stmt>} \to C \to C \\
\text{(AA)} & \mathcal{F}[\![\texttt{<var> (<expr>}_1\texttt{) := <expr>}_2]\!] = \\
 & \quad \lambda[s,t,p] \in C.[\delta(s; \texttt{<var>} = \mathit{eval}(\texttt{<var>}, [s,t,p]) \\
 & \qquad \oplus(\mathit{eval}(\texttt{<expr>}_2, [s,t,p])\,, \\
 & \qquad\quad \mathit{eval}(\texttt{<expr>}_1, [s,t,p]))), t, p] \\
 & \\
 & \mathit{eval} : \texttt{<expr>} \times C \to E \\
\text{(E4)} & \mathit{eval}(\texttt{<var>(<expr>)}, [s,t,p]) = \\
 & \quad \rho(\mathit{eval}(\texttt{<var>}, [s,t,p]), \mathit{eval}(\texttt{<expr>}, [s,t,p]))
\end{array}
$$

Fig. 3.5. Symbolic evaluation of array accesses

Figure 3.5 indicates the denotational semantics of array accesses. Rule (AA) models array assignment and Rule (E4) read accesses. The described model is only valid for one-dimensional arrays. In order to symbolically analyze multi-dimensional arrays, the index must be replaced by an index vector. More complex simplification and read operations are required in order to cope with multi-dimensional arrays.

3.6 Procedures

In order to model procedure calls, we first symbolically evaluate the associated procedures by using the techniques described in the previous sections. Generic variable bindings are assumed for all variables used at the procedure start. At the end of the procedure we determine the variable values as expressions of the variable values at the start. For the sake of illustration we assume that each procedure has a single start and end which are respectively associated with a start and end context. Furthermore, our procedure model corresponds to Fortran where procedures are named *subroutines* or *functions* and all procedure parameters that are variables are passed by *call-by-reference*. Expressions other than variables are passed by *call-by-name*. The values of local procedure variables are set to $\perp$ at the procedure start. Global variables are inherently modeled by our symbolic analysis approach. These variables are always kept in the state of procedure contexts as additional variables. We assume that the call graph is acyclic which also excludes recursive procedure calls.

The program contexts of procedure `ADIFF` are computed as shown in Ex. 3.6.1. Contexts $[s_1, t_1, p_1]$ and $[s_6, t_6, p_6]$ are the start and end context of the procedure, respectively. At the time where ℓ_1 is evaluated nothing may be known about the variable values of the procedure parameters, therefore, generic values are assumed. Similar considerations account for t_1 and p_1 which are set to *true*.

Example 3.6.1 *Procedure*

ℓ_1: ***subroutine*** `ADIFF(A,B,S)`
$[s_1 = \{A = a, B = b, S = s\}, t_1 = \text{true}, p_1 = \text{true}]$
ℓ_2: ***if*** `(A ≥ B)` ***then***
$[s_2 = s_1, t_2 = t_1, p_2 = a \geq b]$
ℓ_3: `S:=A-B`
$[s_3 = \delta(s_2; S = a - b), t_3 = t_2, p_3 = p_2]$
ℓ_4: ***else***
$[s_4 = s_1, t_4 = t_1, p_4 = a < b]$
ℓ_5: `S:=B-A`
$[s_5 = \delta(s_4; S = b - a), t_5 = t_4, p_5 = p_4]$
ℓ_6: ***end if***
$[s_6 = \delta(s_1; S = u), t_6 = \gamma(a \geq b; u = a - b; u = b - a), p_6 = \text{true}]$
ℓ_7: ***end subroutine***

The code segment in Ex. 3.6.2 contains a call to procedure `ADIFF`. We refer to A, B and S in `subroutine ADIFF(A,B,S)` as the *formal procedure variables*. X, Y and Z in `call ADIFF(X,Y,Z)` are denoted the *actual procedure parameters*. Note that actual procedure parameters in principle can be arbitrary expressions whereas formal procedure variables are always variables. In the following we present an algorithm that computes the context after the statement `call ADIFF(X,Y,Z)` in ℓ'_1.

Example 3.6.2 *Procedure call*

$[s'_0 = \{X = m, Y = 1, Z = \bot, I = \bot, J = \bot, M = m, W = w, ...\},\ t'_0 = \text{true},\ p'_0 = m \geq 1]$
ℓ'_1: ***call*** `ADIFF(X,Y,Z)`
$[s'_1 = \{X = m, Y = 1, Z = u, I = \bot, ...\},\ t'_1 = \gamma(m \geq 1; u = m - 1; u = 1 - m),\ p'_1 = m \geq 1] = [s''_1 = \delta(s'_1; Z = m\text{-}1),\ t''_1 = \text{true},\ p''_1 = m \geq 1]$
ℓ'_2: ***do*** `I=1,M`
ℓ'_3: `W(M) := ...`
do `J=1,M`
...
$[s'_4 = \{..., Z = m - 1, J = \underline{J}, M = m, ...\}, t'_4 = ..., p'_4 = m \geq 1 \wedge ...]$
ℓ'_5: `W(M+J) := ...`
$[s'_5 = \{..., Z = m - 1, J = \underline{J}, M = m, ...\}, t'_5 = ..., p'_5 = m \geq 1 \wedge ...]$
ℓ'_6 `... := W(Z+J)`
end do
end do

In general, let $c_b^q = [s_b, t_b, p_b]$ and $c_a^q = [s_a, t_a, p_a]$ respectively specify the contexts before and after a call statement q to a procedure R. $(V_1, ..., V_k)$ is a vector of formal procedure variables of R and $(e_1, ..., e_k)$ the associated vector of actual procedure parameters of q. The associated formal procedure variable V_i of an actual procedure parameter e_i is denoted by *formal(e_i)* where $1 \leq i \leq k$. Let c^R be the context at the end of R. Then c_a^q is computed according to the following strategy:

1. In the first step, c^R is used to create a new context c'. The actual procedure parameter values are propagated into c^R. In other words, c^R is specialized for q by c_b^q which results in a new context $c' = [s', t', p']$:

   $$c' = c^R{}_{v_1 \to eval(e_1, c_b^q), ..., v_k \to eval(e_k, c_b^q)}$$

   where v_i is the generic value of V_i at the beginning of procedure R for all $1 \leq i \leq k$. The symbolic expression $\text{eval}(e_i, c_b^q)$ is the value of the i-th actual procedure parameter of q. $c_{v_1 \to e_1, ..., v_k \to e_k}$ specifies a simultaneous replacement of all v_i by e_i. $c^R{}_{v_i \to eval(e_i, c_b^q)}$ means that all occurrences of v_i are simultaneously substituted by $eval(e_i, c_b^q)$ for $1 \leq i \leq k$. After substitution a separate simplification step is performed.

2. In the second step, the values of the formal procedure variables in c' are returned to their associated actual procedure parameters in c_a^q. This step is only performed for those actual procedure parameters that are variables. Let $\{A_1, ..., A_l\}$ be the set of all vector components in $(e_1, ..., e_k)$ which are variables where $l \leq k$. Note that we exclude aliasing of variables; therefore, all actual parameters that are variables must be pairwise different.

   $$s_a := \delta(s_b; A_1 = eval(formal(A_1), c'), ..., A_l = eval(formal(A_l), c'))$$
   $$t_a := t_b \wedge t'$$
   $$p_a := p_b$$

Note that p_a and p_b are identical since the path condition at the start and at the end of a procedure is always *true*.

Our strategy for handling procedures requires a single evaluation of every procedure. Only the end context of each procedure is re-evaluated for specialization of a specific call to this procedure which is in contrast to a strategy that would require full evaluation of the entire procedure for each call to it. Furthermore, $[s_1', t_1', p_1']$ in the previous example can be rewritten as $[s_1'', t_1'', p_1'']$ according to simplification rule (2) of Fig. 3.2.

3.7 Dynamic Data Structures

Since Fortran 90 has evolved as a de-facto standard, dynamic records are part of the language and we need to symbolically analyze dynamically allocated records as well although it can become quite complex [97]. In general heap operations allocate and deallocate records and support reading and writing values from/to record fields. In our framework the heap is described by a heap algebra which is based on symbolic expressions and recurrences. The symbolic heap of a program point symbolically represents the current state of the heap and comprises all modifying operations performed on the heap. Moreover, in the semantic domain of symbolic analysis we use *symbolic pointers* to refer to dynamic records. Therefore, we tag all dynamically created records with a unique symbolic number.

A heap is a *symbolic chain* consisting of three functions **new**, **free** and **put**. Semantically, a function $\mathbf{new}(r, \texttt{<t>})$ allocates a new element with symbolic pointer r and type `<t>`. A function $\mathbf{free}(r)$ denotes a free operation where dynamic record r is freed. The $\mathbf{put}(r, \texttt{<q>}, e)$ assigns a new symbolic value of e to a dynamic field `<q>` of a dynamic record r. Pointer r of a pointer variable v is stored on the symbolic heap by $\mathbf{put}(\mathrm{v}, \bot, r)$. This additional heap function facilitates the detection of memory leaks [97]. The following table informally describes the heap functions by relating them to the constructs of the language.

Heap operation	Language Example
$\mathbf{new}(r, \texttt{<t>})$	`allocate (r)`
$\mathbf{free}(r)$	`deallocate (r)`
$\mathbf{put}(r, \texttt{<q>}, e)$	`r%q := e`
$\mathbf{put}(\mathrm{v}, \bot, r)$	`v := e`

More formally, the heap is modeled as a heap algebra $\mathbb{H}$. The algebra is inductively defined as follows.

1. $\bot$ is in $\mathbb{H}$.
2. If $h \in \mathbb{H}$, $r \in E$ is a symbolic pointer, and `<t>` is a type, then $h \oplus \mathbf{new}(r, \texttt{<t>}) \in \mathbb{H}$.
3. If $h \in \mathbb{H}$ and $r \in E$ is a symbolic pointer then $h \oplus \mathbf{free}(r) \in \mathbb{H}$.
4. If $h \in \mathbb{H}$, r is a symbolic pointer, `<q>` is a qualifier, and e is a symbolic expression then $h \oplus \mathbf{put}(r, \texttt{<q>}, e) \in \mathbb{H}$.
5. If $h \in \mathbb{H}$, v is pointer variable, and r is a symbolic pointer, then $h \oplus \mathbf{put}(\mathrm{v}, \bot, r) \in \mathbb{H}$.

An element h in $\mathbb{H}$ can be written as a $\oplus$-chain $\bot \bigoplus_{l=1}^{m} \sigma_l$ where σ_l is a **new**, **free**, or **put** function. In the further text, σ_l is also referred as a *heap function*. The length of chain a $|h|$ is the number of functions σ_l in chain h.

Example 3.7.1 *Simple dynamic record manipulation*

```
     type bin is record
       type(bin),allocatable,pointer::LEFT,RIGHT
     end type bin
     type(bin),allocatable, pointer :: P1,P2,P3
       [s_0 = {P1 = ⊥, P2 = ⊥, P3 = ⊥, hc = c, hp_0 = hp_i}, t_0, p_0]
ℓ1 : allocate(P1,P2,P3)
       [s_1 = δ(s_0; P1 = c + 1, P2 = c + 2, P3 = c + 3, hc = c + 3,
          hp_1 = hp_0 ⊕ new(c + 1, bin) ⊕ put(P1, ⊥, c + 1) ⊕ new(c + 2, bin)
            ⊕put(P2, ⊥, c + 2) ⊕ new(c + 3, bin) ⊕ put(P3, ⊥, c + 3)),
        t_1 = t_0, p_1 = p_0]
ℓ2 : P1%LEFT := P3
       [s_2 = δ(s_1; hp_2 = hp_1 ⊕ put(c + 1, LEFT, c + 3)), t_2 = t_1, p_2 = p_1]
ℓ3 : P1%RIGHT := P2
       [s_3 = δ(s_2; hp_3 = hp_2 ⊕ put(c + 1, RIGHT, c + 2)), t_3 = t_2, p_3 = p_2]
ℓ4 : P1%RIGHT%LEFT := P3
       [ s_4 = δ (s_3; hp_4 = hp_3 ⊕ put( get(hp_3,c+1,RIGHT), left,c+3)),
          t_4 = t_3, p_4 = p_3 ]
       [s'_4 = δ(s_4; hp'_4 = hp_3 ⊕ put(c + 2, LEFT, c + 3)), t'_4 = t_4, p'_4 = p_4]
ℓ5 : P3%RIGHT := P2
       [s_5 = δ(s'_4; hp_5 = hp'_4 ⊕ put(c + 3, right, c + 2)), t_5 = t'_4, p_5 = p'_4]
ℓ6 : P2:=nil
     P3:=nil
       [s_6 = δ(s_5; P2 = ⊥, P3 = ⊥, hp_6 = hp_5 ⊕ put(P2, ⊥, ⊥) ⊕ put(P2, ⊥, ⊥),
          t_6 = t_5, p_6 = p_5]
ℓ7 : free(P1);
       [s_7 = δ (s_6; hp_7 = hp_6 ⊕ free(c + 1)) , t_7 = t_6, p_7 = p_6]
```

We introduce a heap function **get**$(h, r, \texttt{<q>})$ to access field `<q>` from dynamic record r where h is a symbolically described heap and element of $\mathbb{H}$. The operator **get** seeks for the right-most function **put**$(r, \texttt{<q>}, e)$ in h whose symbolic pointer and qualifier matches the arguments of operator **get**. If such a function **put**$(r, \texttt{<q>}, e)$ exists, the result of operator **get** is e; otherwise it is $\perp$.

$$\textbf{get}\left(\perp \bigoplus_{l=1}^{m} \sigma_l, r, \texttt{<q>}\right) = \begin{cases} e_l, \text{ if } \exists l = \max\{l \mid 1 \le l \le m \wedge \sigma_l = \textbf{put}(r, \texttt{<q>}, e_l)\} \\ \perp, \text{ otherwise} \end{cases} \tag{3.19}$$

In general determining whether $\sigma_l = \textbf{put}(r, \texttt{<q>}, e_l)$ matches the arguments of operator **get** in a symbolic expression (or not) is undecidable. In practice a wide class of symbolic relations can be solved by techniques for comparing symbolic expressions [47]. If our symbolic analysis framework cannot prove that the result of **get** is `<q>` or $\perp$ then **get** is not resolvable and remains unchanged in a symbolic expression.

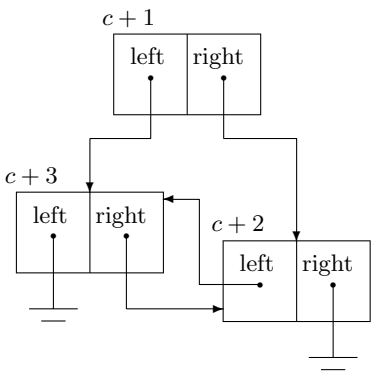

Fig. 3.6. Graphical representation of dynamic data structure of Ex. 3.7.1

For loops, we extend algebra $\mathbb{H}$ in order to model recurrences similar to arrays a shown in Section 3.5. A recurrence system over $\mathbb{H}$ consists of a boundary condition $h(0) = h$, $h \in \mathbb{H}$ and a recurrence relation

$$h(k+1) = h(k) \bigoplus_{l=1}^{m} \sigma_l(k), \quad k \geq 0 \tag{3.20}$$

where $\sigma_l(k)$ can be a function $\mathbf{new}\,(r_l(k), \texttt{<t>}_l)$, $\mathbf{free}\,(r_l(k))$ or $\mathbf{put}(r_l(k), \texttt{<q>}_l, e_l(k))$. The boundary condition represents the symbolic value before entering the loop. A symbolic value is computed for each loop iteration by the recurrence relation. The index k determines the loop iteration. Note that the parameters of the heap functions are functions over index k. Clearly, every instance of the recurrence is an element of $\mathbb{H}$. Moreover, for chains the operator **get** is not sophisticated enough to cope with general read accesses of recursively defined heaps. For this purpose the operator **get** must be extended for recurrences such that heap operations inside of loops can be accessed, e.g. $\mathbf{get}(h(z), r, \texttt{<q>})$. The symbolic expression z is the number of loop iterations determined by the loop exit condition and r is the symbolic pointer of the dynamic record. Furthermore, the recurrence index k has an upper and lower bound of $0 \leq k \leq z$. To determine a possible function **put** where the accessed element is written, a *potential index set* $X_l(r, \texttt{<q>})$ of the function σ_l is computed.

$$\forall 1 \leq l \leq m : X_l(r, \texttt{<q>}) = \{k \mid \sigma_l(k) = \mathbf{put}(r, \texttt{<q>}, e_l(k)) \wedge 0 \leq k \leq z\} \tag{3.21}$$

$X_l(r, \texttt{<q>})$ represents all possible indices of k such that function $\sigma_l(k)$ potentially describes the value of the read access. If the cardinality of X_l is zero, then the corresponding heap function is irrelevant to find the value of

the requested field of a dynamic record. If an index set $X_l(r, \texttt{<q>})$ has more than one element, the field `<q>` of record r is written more than once in different loop iterations. We are only interested in the right-most function of $h(z)$. Consequently, we choose the element with the largest index. The *supremum* $x_l(r, \texttt{<q>})$ of an index set $X_l(r, \texttt{<q>})$ is the largest index such that

$$\forall 1 \leq l \leq m : x_l(r, \texttt{<q>}) = \max X_l(r, < \texttt{q} >) \tag{3.22}$$

Finally, we define the operator **get** for recurrences as follows,

$$\textbf{get}\,(h(z), r, \texttt{<q>}) = \begin{cases} e_l(k), & \text{if } \exists 1 \leq l \leq m : x_l(r, \texttt{<q>}) = \max_{1 \leq k \leq m} x_k(r, \texttt{<q>}) \\ \textbf{get}(h(0), r, \texttt{<q>}), & \text{otherwise} \end{cases} \tag{3.23}$$

where $e_l(k)$ is the value of $\sigma_l(k) = \textbf{put}(r_l(k), \texttt{<q>}, e_l(k))$. The maximum of the supremum indices $x_l(r, \texttt{<q>})$ determines the symbolic value $e_l(x_l(r, \texttt{<q>}))$. If no supremum index exists then **get** returns the access to the value before the loop.

The program in Ex. 3.7.1 allocates three dynamic records. Three pointer variables `P1`, `P2`, `P3` of type `bin` are modified in the statements from ℓ_2 to ℓ_7. In context $[s_0, t_0, p_0]$ heap hp is set to an arbitrary state (hp_i) and symbolic counter hc is assigned symbolic value c which denotes an arbitrary heap configuration depending on the preceding statements of the program. Before statement ℓ_1 the values of variables `P1`, `P2`, and `P3` are set to $\perp$ which means that they do not have a valid value so far. In the first statement ℓ_1, three dynamic records are allocated and assigned to the variables `P1`, `P2`, and `P3`. The allocation is symbolically described by incrementing symbolic counter hc and by adding three **new** and three **put** functions to the heap hp. Note that the **put** functions are necessary due to the variable assignment. In the next two statements ℓ_2 and ℓ_3, field assignments are symbolically evaluated. For each statement a new **put** function is added to hp. In ℓ_4 a list of qualifiers is used. The picture in Fig. 3.6 shows the allocated records and its pointer values after statement ℓ_5. Nil-references are marked by a special symbol.

The last statement ℓ_7 frees the first dynamic record in the tree which results in a memory leak. Note that a memory leak occurs as it is no longer possible to access the dynamic records. Further explanations on how to detect memory leaks are given in [97].

Ex. 3.7.2 illustrates a program that supposedly builds a single linked list. However, the program fails to generate the list due to the fact that the reference to the first element is lost. Before evaluating the first statement, the heap variables hp and hc are initialized to $\perp$ and 0. The variables `I` and `P` do not have a value so far and variable `n` is set to n. The variables `i`, `P`, hc, and hp change their values in each iteration of the loop. Therefore, we have four recurrences induced by variable `I`, pointer variable `P`, symbolic counter hc and heap hp. Note that the result of operator **get** in statement

Example 3.7.2 *Dynamic records*

```
     type el
       integer :: ITEM
       type(el), pointer, allocatable :: NEXT
     end type el
     type(el), pointer, allocatable :: P
       [s_0 = {N = n, P = ⊥, I = ⊥, hc = 0, hp_0 = ⊥}, t_0 = true, p_0 = (n ≥ 2)]
ℓ1 : allocate (P)
       [s_1 = δ (s_0; P = 1, hc = 1, hp_1 = hp_0 ⊕ new(1, el) ⊕ put(p, ⊥, 1)) ,
          t_1 = t_0, p_1 = p_0]
ℓ2 : P%ITEM := 1
       [s_2 = δ (s_1; hp_2 = hp_1 ⊕ put(1, ITEM, 1)) , t_2 = t_1, p_2 = p_1]
       [s'_2 = δ (s_2; I = 1) , t_2 = t_1, p_2 = p_1]
ℓ3 : do I=2,N
          [s_3 = δ(s_2; I = I, P = P, hc = hc, hp_3 = hp), t_3 = true, p_3 = (I <= n)]
ℓ4 :    allocate (P%NEXT)
          [s_4 = δ(s_3; hc = hc + 1, hp_4 = hp ⊕ new(hc + 1, el)⊕
          put(P, NEXT, hc + 1)),    t_4 = t_3, p_4 = p_3]
ℓ5 :    P := P%NEXT
          [s_5 = δ(s_4; P = get(hp_4, P, NEXT)), t_5 = t_4, p_5 = p_4] =
          [s'_5 = δ(s_5; P = hc + 1), t'_5 = t_5, p'_5 = p_5]
ℓ6 :    P%ITEM := I
          [s_6 = δ(s'_5; hp_6 = hp_4 ⊕ put(hc + 1, ITEM, I)), t_6 = t'_5, p_6 = p'_5]
ℓ7 : end do
       [s_7 = δ(s_6; I = n + 1, P = n − 1, hc = n − 1, hp_7 = μ(P, s'_2, [s_6, t_6, p_6])),
          t_7 = t'_2,    p_7 = p_2]
```

ℓ_5 is given by $hc + 1$ since the right-most function **put** can be directly found in the recurrence relation of hp. After the loop terminates, the values of variables `I`, `P`, hc and hp are determined by their recurrences whereby the recurrence index z is derived from the loop exit condition. Closed forms can be found for `I`, `P`, hc. The heap hp in ℓ_7 symbolically describes all elements of the singly linked list. Figure 3.7 depicts the dynamically allocated records. Nil-references are marked by a special symbol.

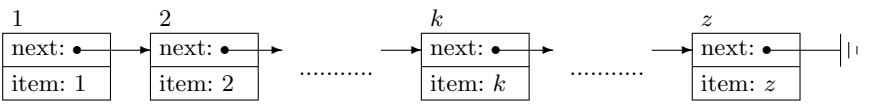

Fig. 3.7. Graphical representation of dynamic data structure of Ex. 3.7.2

In the following, we relate the syntax of heap operations to the semantics for allocating dynamic records, heap function **put** for setting fields, and

$$
\begin{array}{ll}
 & \mathcal{F} : \texttt{<stmt>} \to C \to C \\
\text{(D1)} & \mathcal{F}[\![\texttt{<ptr>}_1\texttt{\%<q> := <ptr>}_2]\!] = \\
 & \quad \lambda[s,t,p] \in C.[\delta(s, hp = \text{eval}(hp,[s,t,p]) \oplus \\
 & \qquad \mathbf{put}(\mathcal{F}[\![\texttt{<ptr>}_1]\!]([s,t,p]), \texttt{<q>}, \mathcal{F}[\![\texttt{<ptr>}_2]\!]([s,t,p]))), t, p] \\
\text{(D2)} & \mathcal{F}[\![\texttt{<ptr>\%<q> := <value>}]\!] = \\
 & \quad \lambda[s,t,p] \in C.[\delta(s, hp = \text{eval}(hp,[s,t,p]) \oplus \\
 & \qquad \mathbf{put}(\mathcal{F}[\![\texttt{<ptr>}]\!]([s,t,p]), \texttt{<q>}, \text{eval}(\texttt{<value>},[s,t,p]))), t, p] \\
\text{(D3)} & \mathcal{F}[\![\texttt{allocate(<ptr>\%<q>)}]\!] = \\
 & \quad \lambda[s,t,p] \in C.[\lambda r \in E.\delta\,(s, hc = r, \\
 & \qquad hp = \text{eval}(hp,[s,t,p]) \oplus \mathbf{new}(r, \texttt{<t>}) \oplus \\
 & \qquad\quad \mathbf{put}(\mathcal{F}[\![\texttt{<ptr>}]\!]([s,t,p]), \texttt{<q>}, r))\,(\text{eval}(hc,[s,t,p]) + 1), t, p] \\
\text{(D4)} & \mathcal{F}[\![\texttt{<var> := <ptr>}]\!] = \\
 & \quad \lambda[s,t,p] \in C.[\lambda r \in E.\delta(s, \texttt{<var>} = r, hp = \text{eval}(hp,[s,t,p]) \oplus \\
 & \qquad \mathbf{put}(\text{eval}(\texttt{<var>},[s,t,p]), \bot, r)) \\
 & \qquad (\mathcal{F}[\![\texttt{<ptr>}]\!]([s,t,p])), t, p] \\
\text{(D5)} & \mathcal{F}[\![\texttt{allocate(<var>)}]\!] = \\
 & \quad \lambda[s,t,p] \in C.[\lambda r \in E.\delta(s, \texttt{<var>} = r, hc = r, hp = \text{eval}(hp,[s,t,p]) \oplus \\
 & \qquad \mathbf{new}(r, \texttt{<t>}) \oplus \mathbf{put}(\texttt{<var>}, \bot, r)) \\
 & \qquad (\text{eval}(hc,[s,t,p]) + 1), t, p] \\
\text{(D6)} & \mathcal{F}[\![\texttt{deallocate(<ptr>)}]\!] = \\
 & \quad \lambda[s,t,p] \in C.[\delta(s, hp = \text{eval}(hp,[s,t,p]) \oplus \\
 & \qquad \mathbf{free}(\text{eval}(\texttt{<ptr>},[s,t,p])), t, p] \\
 & \\
 & \textit{eval} : \texttt{<expr>} \times C \to E \\
\text{(E5)} & \textit{eval}(\texttt{<nil>},[s,t,p]) = \bot \\
\text{(E6)} & \textit{eval}(\texttt{<ptr>\%<q>},[s,t,p]) = \\
 & \quad \mathbf{get}(\textit{eval}(hp,[s,t,p]), \textit{eval}(\texttt{<ptr>},[s,t,p]), \texttt{<q>})
\end{array}
$$

Fig. 3.8. Symbolic evaluation of heap operations

heap function **free** for freeing dynamic records. Figure 3.8 lists the symbolic evaluation rules of the heap operations. Nonterminal `<stmt>` denotes a heap operation, nonterminal `<ptr>` a pointer value, and nonterminal `<value>` a value of a basic type.

For symbolically evaluating a program symbolic counter hc is employed and it is incremented each time when a new dynamic record is allocated. The symbolic value of the counter is used to generate a label for the newly allocated record. The symbolic pointer of the new record is the value of hc after incrementing it. The modifications on the heap are symbolically described by a heap element hp of $\mathbb{H}$. For both, hc and hp, we introduce two new pseudo variables in state s. In the first program context, heap hp is set to $\bot$ and hc to 0.

In Fig. 3.8 the denotational semantic rules (D1)-(D6) and (E5)-(E7) are given. We distinguish between two different pointer assignments, (D1) and (D4). Rule (D1) deals with a pointer assignment for a field of a dynamic record. An assignment for a pointer variable (D4) generates a new entry

put(<var>, $\perp, r$) in the chain. This redundant entry facilitates subsequent analyses. Rule (D2) assigns a new symbolic value to a non-pointer field of a dynamic record. Rule (D3) allocates a new dynamic record and assigns it to a pointer field. Rule (D5) allocates a new dynamic record and assigns it to a program variable. The semantics of the free operation is given by (D6). Rules (E5) and (E6) extend the semantics of function *eval* for read accesses.

3.8 Summary

In this chapter, we have described a symbolic analysis framework in order to statically determine the values of variables and symbolic expressions, assumptions about and constraints between variable values, and the condition under which control flow reaches a program statement. Our symbolic analysis is based on a novel representation for comprehensive and compact control and data flow analysis information, called program context. Computations are represented as algebraic expressions defined over a program's problem size and are an amenable data structure for computer algebra systems. Our symbolic analysis framework can cope with accurate modeling of assignment and input/output statements, branches, loops, recurrences, arrays, procedures, and dynamic records. Note that in this chapter the formalism for symbolic analysis is styled as a functional semantics model with strong resemblances to denotational semantics [86].

4. Generating Program Contexts

4.1 Introduction

In the previous chapter we have introduced a new data structure named program contexts. Moreover, we have explained how to compute program contexts for a variety of program statements. However, the denotational semantic approach as used in the previous chapter cannot deal with more complex control flow such as GOTO statements. Although there are algorithms [4] that translate a program containing GOTOs to a well-structured program, the relationship between analyzed program and analysis information would be hidden. Hence, we introduce an algorithm that computes program contexts for programs that are represented by *control flow graphs(CFG)*. By using CFGs the control flow of a program is abstracted as a graph and syntactical details such as GOTOs, IFs, etc. are encoded in the graph.

For the sake of demonstration, we keep the computation of program contexts simple by restricting our algorithm to reducible control flow graphs. Algorithms for mapping an irreducible control to reducible control flow graphs are given in [78]. The algorithm consists of two steps. In the first step the control flow graph is transformed to an *extended control flow graph*. Artificial nodes are inserted to mark loop entry and exit points. In the second step, program contexts are propagated through the extended control flow graph.

4.2 Extended Control Flow Graph

We consider only reducible control flow graphs and assume that there are no jumps into the loop body from outside (which is implied by reducibility). The extended control flow graph only contains natural loops that are detected by using dominator information [2]. Each loop has a unique *header* which dominates all nodes in the loop body. Nevertheless, a loop may have several exits. To simplify the computation of program contexts we use an extended CFG similar to that proposed in [35, 56, 71] by inserting several artificial nodes in the original CFG without side-effects.

This includes pre-header, post-body and post-exit nodes (see Figure 4.1). A *pre-header* node, *PH*, is inserted such that it has only the header as successor and all edges which formerly entered the header from outside the loop

T. Fahringer and B. Scholz: Advanced Symbolic Analysis, LNCS 2628, pp. 41–47, 2003.

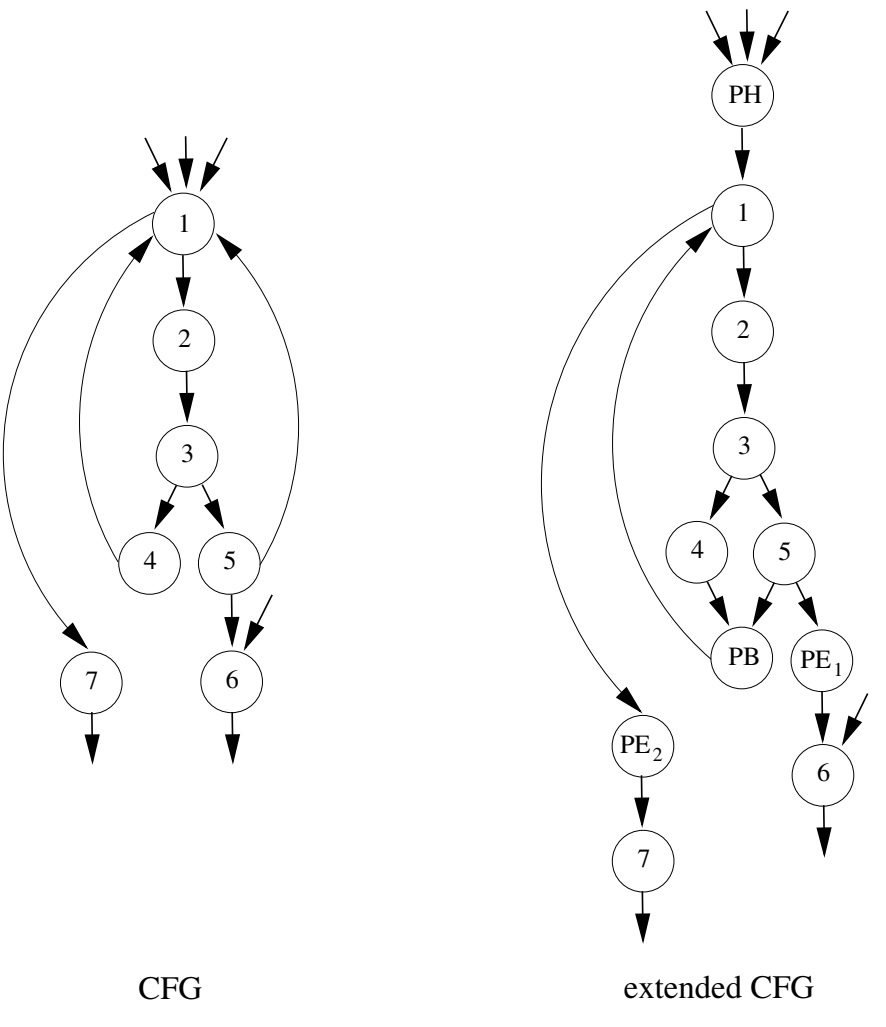

Fig. 4.1. CFG and extended CFG

instead enter the pre-header. A *post-body* node, *PB*, is inserted such that it provides a target for all loop back edges. An edge is inserted from the pre-header to the header and another edge from the post-body to the header. For each edge exiting the loop, a *post-exit* node, *PE*, is inserted outside the loop, between the source (loop exit node within the loop body) of the exit edge and the target (outside the loop). These changes cause threefold growth in a CFG at worst. In the remainder of this chapter a CFG refers to the extended CFG if not mentioned otherwise.

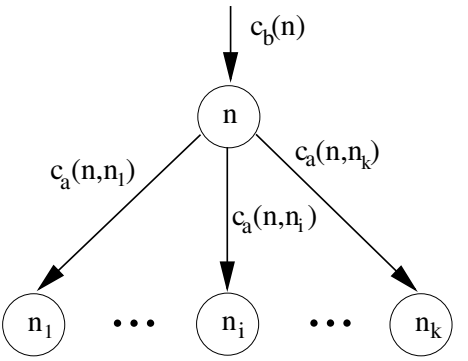

Fig. 4.2. Exit and entry contexts for a generic node n with k ($1 \leq i \leq k$) successor nodes.

4.3 Algorithm

The algorithm computes program contexts for all nodes in the CFG. Every node $n \in N$ has a unique *entry context* $c_b(n) = [s_b, t_b, p_b]$ that describes the variable bindings and state and path conditions immediately before node n. Note that every node n has only a single entry context although node n may have multiple predecessors[1]. The entry context $c_b(x)$ of the start node x includes the initial variable bindings and state and path condition of the CFG as described in Section 3.2.

Furthermore, every CFG node n has a unique *exit context* $c_a(n, n')$ for each of its successor nodes $n' \in succs(n)$. $c_a(y, undef)$, the exit context of the CFG end node y, describes the context at the end of the CFG. Note that y does not have a successor node; therefore, the second parameter of c_a is set to undef. Let *cond_expr(n,n′)* denote the conditional expression that governs traversal of the edge (n,n′). If n has only a single successor node n', then cond_expr(n,n′) = true. If n has two or more successor nodes, then every time n is executed at execution time, exactly one conditional expression of all outgoing edges from n is true, all others are false. It is assumed that at the beginning of our algorithm all contexts are undefined (initialized with $\perp$). If symbolic evaluation can prove that a node n is never reached during execution of the program, then all entry and exit contexts of n are set to $\top$.

Figure 4.2 illustrates the entry and exit contexts for a generic CFG node n.

In the first step of our algorithm we initialize the exit context of all post-body nodes to $\top$ which specifies an undefined context. This is important for

[1] The folding operator $\odot$ is used to generate $c_b(n)$ based on the exit contexts of all the predecessor nodes of n.

```
procedure gen_context(CFG G)
1:    for every n ∈ N of G in reverse post-order traversal do
2:          gen_context_entry(n)
3:          gen_context_exit(n)
4:    endfor
endprocedure
```

Fig. 4.3. Traverse CFG in reverse post-order to generate contexts

computing the entry context of loop header nodes. Note that according to the previous section all post-body nodes have only a single successor node (loop header).

Figure 4.3 shows our algorithm that generates the entry and exit contexts for every node $n \in N$ by traversing the CFG in reverse post-order. For every node n we call two procedures *gen_context_entry* and *gen_context_exit* that, respectively, generate the entry and exit contexts of n. Note that according to [73] the reverse post-order sorts a reducible flow graph in topological order which enables the algorithm to generate the entry and exit contexts of all CFG nodes – except for pre-header and procedure end nodes – through a single traversal of the CFG. In other words, every time our algorithm traverses a node $n \in N$ it is guaranteed that the exit contexts of all nodes in $preds(n)$ are already generated. Only loop pre-header and procedure end nodes must be traversed more than once.

We can generate contexts for an entire program (including main program and all procedures) by using gen_context. Every procedure (including the main program) is represented through a unique CFG. Based on the assumption of an acyclic procedure call graph we invoke gen_context for every procedure exactly once while traversing the call graph bottom up. This guarantees that the contexts of a procedure have already been generated at the time where gen_context encounters the associated procedure call. Moreover, according to Section 3.6 for every call statement q to a procedure R, our algorithm accesses the end context of R in order to compute the context after q.

In order to describe gen_context_entry and gen_context_exit we define for every node $n \in N$ a predicate *type(n)* which specifies the type of the statement (ASSIGNMENT, GOTO, READ, CONDITIONAL, CALL, PREHEADER, POSTEXIT and POSTBODY) associated with node n. Furthermore, for every loop exit node n, we introduce a predicate *recurrence_cond(n)* that specifies the (recurrence) condition under which the associated loop is exited at n.

Procedure *gen_context_entry* (see Figure 4.4) describes how to create the entry context of a node n. First, it is examined whether n is the start node of the CFG whose entry context is initialized by function *init_start_node* according to Section 3.2. Otherwise, node n has at least one predecessor. In

```
procedure gen_context_entry(n)
1:    if |preds(n)| = 0 then
2:        init_start_node(n);
3:    else
4:        c_b(n) := ⊙_{n'∈preds(n)} c_a(n', n)
5:    endif
6:    if type(n) = PREHEADER then
7:        init_preheader(n)
8:    elseif type(n) = POSTEXIT then
9:        update_postexit(n)
10:   endif
endprocedure
```

Fig. 4.4. Generate context for node entry

order to generate the entry context of n, the folding operator is applied to the exit contexts of all nodes in $preds(n)$. Note that if n corresponds to a loop header, the folding operation is defined due to initialization of the associated post-body's exit context. Figure 4.5 shows the properties that hold for the folding operator. In case of a loop header node, we apply *prep_header(n)* which updates $c_b(n)$ with additional information to model recurrences.

$$
\begin{aligned}
c_1 \odot c_2 &= c_2 \odot c_1 \\
(c_1 \odot c_2) \odot c_3 &= c_1 \odot (c_2 \odot c_3) \\
c \odot c &= c \\
c \odot \top &= c
\end{aligned}
$$

Fig. 4.5. Properties of the folding operator $\odot$ (c, c_1, c_2 and c_3 are contexts).

If a node n is traversed that corresponds to the pre-header node of a loop L, then function *init_preheader(n)* is invoked which initializes the recurrence variables v of L in context $c_b(n)$ with the value of $\underline{v}$ (see Section 3.4).

If a postexit node n of a loop L is traversed, then a function *update_postexit* is invoked which tries to compute closed forms for all recurrence variables based on the recurrence system according to the strategy outlined in Section 3.4. The recurrence relations and boundary conditions are derived from the pre-header of L and the recurrence condition from $p_b(n)$. The values of recurrence variables are then replaced with their associated closed forms in $c_b(n)$. If a closed form for a recurrence variable v cannot be computed, then the value of v associated with the corresponding μ-function as described in Section 3.4. Reasoning about recurrence variables that are represented by a recurrence

```
procedure gen_context_exit(n)
1:    for every n' ∈ succs(n) do
2:        c_a(n, n') := eval_exit_context(n, c_b(n), cond_expr(n, n'))
3:    endfor
4:    if type(n) = POSTBODY then
5:        update_preheader(n)
6:    endif
endprocedure

function eval_exit_context(n,c=[s,t,p],cond)
1:    switch type(n)
2:        case ASSIGNMENT | READ | CALL:
3:            c' := side_effect(n,c)
8:        default: /* PREHEADER, POSTEXIT, POSTBODY, GOTO, COND. */
5:            s' := s
6:            t' := t
7:            if p ⇒ eval(cnd, c) then
8:                p' := p
9:            elseif p ⇒ ¬eval(cond, c) then
10:               p' := false
11:           else
7:                p' := p ∧ eval(cond, c)
11:           endif
10:   endswitch
11:   return c'
endfunction
```

Fig. 4.6. Generate contexts for node exits

system is a complex task. Our current implementation introduces new value symbols for every recurrence variable for which no closed form can be determined. These value symbols are linked with their associated μ-function. Evaluation of recurrence variables for which no closed form can be found is done based on their value symbols instead of their recurrence system.

Procedure *gen_context_exit* (see Figure 4.6) generates the exit contexts of a node n based on the entry context of n and the semantics of the statement associated with n. If n corresponds to the post-body node of a loop L, then function gen_context_exit has traversed the entire loop body of L. A function *update_preheader(n)* is invoked to insert the recurrence relations – obtained from $c_b(n)$ – for all recurrence variables in the pre-header node of L. Furthermore, the loop invariant is computed – based on the recurrence relations and the boundary conditions – and stored in the pre-header node of L.

A function *eval_exit_context* is invoked in *gen_context_exit* in order to compute the exit context $c_a(n, n')$ of n with respect to every specific $n' \in succs(n)$. Function *side_effect(n, c)* evaluates all nodes n associated with an ASSIGNMENT, READ or CALL statement that may imply a side-effect. Note that we can adopt the denotational semantic rules as described in Chapter 3 for these statements to deduce the resulting program context. The default case considers all statements without side-effects. For unconditional statements the exit context is given by the entry context. For branch statements we have to distinguish three cases: First, if the path condition of

the entry context implies that *cond* is true, then the path conditions of both entry and exit contexts are identical. Second, p' is set to false if we can statically determine that the branch from n to n' can never be taken. Third, if it is not possible to statically determine whether a branch is taken or not, then we have to add the associated branch condition to p' through a conjunction.

In order to improve the efficiency of symbolic analysis and to reduce the associated memory requirement, our implementation only stores symbolic analysis information in a node n that reflects the semantics of n. Information that has been propagated from predecessor nodes to n is accessible from n through linked lists. Let c_j be a predecessor and c_k a successor context of a context c_i and assume that the program does not change the state between the associated nodes of c_j and c_k. Then, if s_j is not defined as a constant, s_i is a link to s_j (s_i is identical with s_j) and s_j is not defined as a constant ($\top$, $\bot$, true, false), then s_k is directly linked with s_j but not with s_i. If, however, s_i refers to s_j and s_j is defined as a constant, then s_i is set to the same constant as s_j instead of a link to s_j. The same accounts for state and path conditions. For the sake of demonstration, the code examples shown in this book do not follow these implementation strategies as they make the resulting analysis information less readable.

The overall algorithm traverses every node and all its predecessors once and each loop pre-header at a maximum three times (once each by the reverse post-order traversal, update_postexit and update_preheader). Furthermore, every end context of a procedure is accessed once for each call to it.

In general, all symbolic analyses – about which the authors are aware of – that compute exact values of variables, manipulate and simplify symbolic expressions, and resolve recurrences, require exponential complexity. This accounts for our own approach as well. However, the complexity to generate the contexts for a CFG is given by $O(|N| + |E|)$, if we assume an eval function with linear complexity (that does only simple but exact evaluations) and exclude the overhead for generating extended control flow graphs, computing closed forms for recurrences and simplifying symbolic expressions and constraints.

4.4 Summary

We have described an algorithm for generating program contexts which have been introduced in Chapter 3. Our algorithm works on extended control flow graphs which are control flow graphs extended with artificial control flow nodes that mark loop pre-headers, loop headers, loop exit nodes, and post body nodes. An extended control flow graph can only be generated if the underlying control flow graph is reducible. Moreover, we have shown that program contexts are propagated through the extended control flow graph in post-order traversal. Artificial nodes are processed differently in order to set up recurrence relations and to model loops as described in Section 3.4.

5. Symbolic Analysis Algorithms and Transformations

5.1 Introduction

In the previous chapters we have introduced the notation for program contexts and a general framework for generating these contexts. In order to compute contexts and develop useful applications that use context information, we frequently need to examine generic symbolic expressions as part of program contexts. Besides that, numerous researchers [20, 98, 44, 18, 102, 67, 71, 29] have reported on the occurrence of complex and even non-linear symbolic expressions in practical codes and the need for effective techniques to analyze such expressions. Non-linear symbolic expressions are commonly caused by induction variable substitution, linearizing arrays, parameterizing parallel programs with symbolic machine and problem sizes, tiling with symbolic block sizes, recurrences, and so forth. Symbolic expressions can seriously hamper crucial compiler and performance analyses including testing for data dependences, optimizing communication, array privatization, expression simplification, dead code elimination, detecting zero-trip-loops, performance prediction, etc. The inability to effectively handle symbolic expressions and constraints (e.g. symbolic conditions contained in program context information) commonly results in worst case assumptions, or program analysis is moved into execution time and consequently may cause considerable performance losses.

Consider the loop nest with linearized and indirect array referencing in Ex. 5.1.1 which is very similar to a code excerpt as found in the TRFD code of the Perfect Benchmarks [12]. In loop L_1 array `IA` is initialized with a non-linear array subscript expression $(\mathtt{J1} * (\mathtt{J1} - 1))/2$, which is then used in statement $\mathtt{l}_{10}$ to read $\mathtt{X}(\mathtt{IJ}, \mathtt{KL})$ and write $\mathtt{X}(\mathtt{KL}, \mathtt{IJ})$ with $\mathtt{IJ} = (\mathtt{J1} * (\mathtt{J1} - 1))/2 + \mathtt{J2}$ and $\mathtt{KL} = (\mathtt{J3} * (\mathtt{J3} - 1))/2 + \mathtt{J4}$, respectively. In order to examine whether loop L_3 can be executed in parallel, we must determine whether statement l_{10} implies a true dependence (array elements are written before they are read) with respect to L_3. Conventional dependence tests cannot examine non-linear array subscript expressions and, therefore, assume a data dependence that leads to sequentializing loop L_3.

In this chapter we will describe an algorithm that can compare non-linear array subscript expressions such as `KL` and `IJ`, and consequently determines

T. Fahringer and B. Scholz: Advanced Symbolic Analysis, LNCS 2628, pp. 49–74, 2003.

Example 5.1.1

```
ℓ1:  L1 |  do J1 = 1, N
ℓ2:     |     IA(J1) = (J1 * (J1 - 1))/2
ℓ3:     |  end do
        |  ...
ℓ4:  L2 |  do J1 = 1, N
ℓ5:     |     do J2 = 1, J1
ℓ6:     |           IJ = IA(J1) + J2
        |           ...
ℓ7:     |           do J3 = 1, J1
        |                 ...
ℓ8:     |  L3 |           do J4 = 1, J2 - 1
ℓ9:     |     |                 KL = IA(J3) + J4
        |     |                 ...
ℓ10:    |     |                 X(KL, IJ) = X(IJ, KL) + VAL
ℓ11:    |     |           end do
ℓ12:    |           end do
ℓ13:    |     end do
ℓ14:    |  end do
```

the absence of a true dependence, which enables a compiler to parallelize loop L_3.

For the loop nest in Ex. 5.1.2 a compiler may determine whether the condition of statement ℓ_3 ever evaluates to true. If not then we can simply eliminate the entire conditional statement (dead code elimination). The detection of zero-trip-loops [67] corresponds to a similar problem in which the loop body of a given loop nest is examined whether it is ever executed. Counting the number of loop iterations has been shown [44, 40, 67, 100] to be crucial for many performance analyses such as modeling work distribution [43], data locality [44] and communication overhead [42]. All of these problems can be formulated as queries based on a set of linear and non-linear constraints I. The constraints of I are defined over loop *variables* and *parameters* (loop invariants) which are commonly derived from loop bounds, conditional statements, data declarations, data and work distribution specifications [74, 11], etc. For instance, I is given by $\{1 \leq \mathtt{J1} \leq \mathtt{N}, \mathtt{N}/(2 * \mathtt{J1}) \leq \mathtt{J2} \leq \mathtt{N}, \mathtt{J1} * \mathtt{J2} \leq \mathtt{N}\}$ for Ex. 5.1.2. The number of solutions of I can then be used as a basis for the solution of all problems mentioned above.

Furthermore, many analyses based on systems of constraints I are significantly facilitated if I is simplified by eliminating redundant constraints and tautologies, substituting equations, detecting contradictions, etc.

In this chapter we will describe a variety of techniques for effective examining of symbolic expressions and counting the number of solutions to a system of constraints. First, we describe an algorithm for computing lower and upper bounds of wide classes of linear and non-linear symbolic expressions based

Example 5.1.2

```
ℓ1:    do J1 = 1, N
ℓ2:       do J2 = N/(2 * J1), N
ℓ3:          if (J1 * J2 ≤ N) then
ℓ4:             A(J1, J2) = ...
ℓ5:          end if
             ...
ℓ6:       end do
ℓ7:    end do
```

on a set of constraints. Among others, this algorithm supports comparing symbolic expressions for equality and inequality relationships, simplifying systems of constraints, examining non-linear array subscript expressions for data dependences, and optimizing communication [49].

Second, we present a symbolic sum algorithm that estimates the number of integer solutions to a system of constraints. This algorithm is used to support the detection of zero-trip loops, elimination of dead code, and performance prediction of parallel programs.

Third, based on the algorithm for computing lower an upper bounds for symbolic expressions and the symbolic sum algorithm, we describe how to simplify systems of constraints.

Moreover, several experiments will be shown that demonstrate the effectiveness of algorithms introduced in this chapter.

The ideas underlying our approach are not restricted to compilers and tools for parallel architectures. They can be equally important for sequential program analysis.

This chapter is organized as follows: Basic definitions and notations are given in Section 5.2. In Section 5.3 we present a novel algorithm for computing lower and upper bounds of symbolic expressions and how this algorithm is applied to compare symbolic expressions and examine data dependences of non linear array subscript expressions. Our algorithm for counting the number of solutions to a system of linear and non-linear constraints is described in Section 5.4. The next section shows how to simplify a set of symbolic constraints. Thereafter, we present experiments that demonstrate the effectiveness of our symbolic analysis. A summary is given in Section 5.7.

5.2 Preliminaries

The following notations and definitions are used in the remainder of this chapter:

- Let I be a set of non-linear and linear constraints (equalities and inequalities) defined over loop *variables* and *parameters* (loop invariants) which are commonly derived from loop bounds, array subscript expressions, conditional statements, data declarations, and data and work distribution specifications of a program. V is the set of variables and P the set of parameters appearing in I.
- *low(e)* and *up(e)* are two functions, which define the minimum and maximum value of a symbolic expression e for all values of variables and parameters appearing in e. These two functions (*low* and *up*) will be referred to as *φ-functions*. $\varphi(e)$ which is either $low(e)$ or $up(e)$ is denoted as *φ-expression*. Note that e may again contain φ-expressions.
- A *symbolic (integer-valued) expression* e may consist of arbitrary division, multiplication, subtraction, addition, exponentiation, maximum, minimum, and φ-functions. Its operands can be variables, parameters, integer constants and infinity symbols $(\infty, -\infty)$. Symbolic expressions, therefore, describe both linear as well as non-linear expressions. Note that our algorithm for counting solutions to a system of constraints (see Section 5.4) is restricted to a sub-class of these expressions.

To provide an effective framework for manipulating symbolic expressions and constraints, we have implemented an interface [51] to *Mathematica*^{TM}[1] via *MathLink*^{TM} [105]. Mathematica is a commercial package that provides a range of standard manipulation and simplification techniques, including multiplying out products and powers, reducing products of factors, putting all terms over a common denominator, separating into terms with simple denominators, cancelling common factors between numerators and denominators, etc. Our framework has been developed primarily based on Mathematica's support for pattern matching and transforming rules to rewrite expressions. Note that Mathematica does not provide explicit support for manipulating integer-valued symbolic expressions. Therefore, we have implemented our own techniques [95] for manipulating such expressions on top of Mathematica.

5.3 Symbolic Expression Evaluation

In this section we present our techniques to evaluate symbolic expressions for equality and inequality relationships which are based on the following Lemma:

Definition 1: *Let e_1, e_2 be two symbolic expressions defined in $V \cup P$. Then $e_1 \geq e_2 \overset{Def.}{\Longleftrightarrow} low(e_1) \geq up(e_2)$.*

Lemma 1: *Comparison of Expressions Let e_1, e_2 be two symbolic expressions defined in $V \cup P$ then $e_1 \geq e_2$ iff $low(e_1 - e_2) \geq 0$.*

[1] Mathematica and MathLink are registered trademarks of Wolfram Research, Inc.

Proof:

$$
\begin{aligned}
e_1 \geq e_2 &\overset{Def.\ 1}{\Rightarrow} && low(e_1) - up(e_2) \geq 0 \\
&\overset{Rule\ 5.6\ of\ Table\ 5.1}{\Rightarrow} && low(e_1 - e_2) \geq 0 \\
&\overset{*(-1)}{\Rightarrow} && -low(e_1 - e_2) \leq 0 \\
&\overset{Rule\ 5.4\ of\ Table\ 5.1}{\Rightarrow} && up(e_2 - e_1) \leq 0
\end{aligned}
$$

Note that $low(e_1 - e_2) \geq 0$ is identical with $up(e_2 - e_1) \leq 0$.

In order to compare symbolic expressions we developed an algorithm that computes the lower and upper bound of symbolic expressions based on a system of constraints.

Inherently, computing bounds of a symbolic expression is in general undecidable and depends mostly on the set of constraints given. Our objective is to cover wide classes of both linear and non-linear symbolic expressions for which we selectively compute lower and/or upper bounds.

In Chapter 4 we described an algorithm that effectively collects constraints about variables in program contexts and propagates them through the program. Our algorithm for computing lower and upper bounds of symbolic expressions can simplify program contexts and thus symbolic analysis defined over program contexts.

5.3.1 Compute Lower and Upper Bounds of Symbolic Expressions

Figure 5.1 shows algorithm *expr_bound* which computes the lower and/or upper bound of a symbolic expression e defined in $V \cup P$. The input parameters of *expr_bound* are given by e, B, and Q. B is the set of known upper and lower bound expressions for each variable and parameter. These expressions are derived from I, the set of constraints of the underlying problem.

Assume that the cardinality of V and P is respectively n and m, then Q is given by a set of triples as follows:

$$Q=\{(v_1,v_1^l,v_1^u),...,(v_n,v_n^l,v_n^u),\ (p_1,p_1^l,p_1^u),...,(p_m,v_m^l,v_m^u)\}.$$

The triple (v_i, v_i^l, v_i^u) – where $1 \leq i \leq n$ – specifies a variable $v_i \in V$ with its associated maximum lower bound v_i^l and minimum upper bound v_i^u. The triple (p_j, p_j^l, p_j^u) – where $1 \leq j \leq m$ – specifies a parameter $p_j \in P$ with its associated maximum lower bound p_j^l and minimum upper bound p_j^u. The algorithm maintains the maximum lower and minimum upper bound for each variable and parameter in Q_{new}, which is an output parameter of *expr_bound*. For instance, if a new lower bound for a variable is deduced by the algorithm that is larger than the current maximum lower bound as stored in Q, then this new bound is updated in Q_{new} accordingly. We similarly determine the minimum upper bound.

Let us continue Ex. 5.1.1 of Section 5.1. The set of constraints I for this problem as derived from the corresponding loop nest of L_2 yields:

expr_bound(e,B,Q)

- **INPUT:**
  - e: φ-expression defined in $V \cup P$
  - B: set of lower and upper bound expressions for variables and parameters defined in $V \cup P$.
  - Q: set of single lower and upper bound for each variable and parameter in e.
- **OUTPUT:**
  - Q_{new}: updated set of single lower and upper bound for each variable and parameter in e updated by *expr_bound*.
  - e_b: bound for e which is a constant, ∞, or -∞.
- **METHOD:**

```
1:  V := set of variables appearing in e.
2:  P := set of parameters appearing in e.
3:  e_orig := e
4:  Q_new := Q
5:  repeat
6:      e := e_orig
7:      Q_old := Q_new
8:      while variables appear in e do
9:          choose a φ-expression in e for rewriting
10:         rewrite φ-expression of 9: in e
11:         simplify e
12:         update Q_new
13:     end while
14:     while parameters appear in e do
15:         choose a φ-expression in e for rewriting
16:         rewrite φ-expression of 15: in e
17:         simplify e
18:         update Q_new
19:     end while
20: until Q_new = Q_old
21: e_b := determine bound for e
```

Fig. 5.1. A Demand driven algorithm to compute the lower and/or upper bound of a symbolic expression defined in $V \cup P$

$$I = \{1 \leq \texttt{J1} \leq \texttt{N}, 1 \leq \texttt{J2} \leq \texttt{J1}, 1 \leq \texttt{J3} \leq \texttt{J1}, 1 \leq \texttt{J4} \leq \texttt{J2} - 1\} \tag{5.1}$$

This yields the following sets of variables, parameters, lower and upper bounds: $V = \{\texttt{J1}, \texttt{J2}, \texttt{J3}, \texttt{J4}\}$, $P = \{\texttt{N}\}$, $L_{J1} = \{1, \texttt{J2}, \texttt{J3}\}$, $U_{J1} = \{\texttt{N}\}$, $L_{J2} = \{1, \texttt{J4} + 1\}$, $U_{J2} = \{\texttt{J1}\}$, $L_{J3} = \{1\}$, $U_{J3} = \{\texttt{J1}\}$, $L_{J4} = \{1\}$, $U_{J4} = \{\texttt{J2} - 1\}$, $L_N = \{\texttt{J1}\}$, $U_N = \{\infty\}$, $B = L_N \cup U_N \cup \bigcup_{1 \leq i \leq 4} \left(L_{Ji} \cup U_{Ji}\right)$

Initially all lower and upper bounds in Q are set to $-\infty$ and ∞, respectively.

The algorithm traverses e and replaces all φ-expressions of e using bound expressions in B and Q_{new}, and rewrite-rules of Table 5.1 until e is a constant, $-\infty$, or ∞. During a single iteration of the repeat-loop, the bounds in Q may become tighter as compared to the previous loop iteration. Tighter bounds in

Table 5.1. Rules for rewriting symbolic expressions containing *low* and *up* functions. x and y are symbolic integer-valued expressions. x^- and x^+ respectively refer to the set of negative and positive integer values (neither contains 0) of x. x_0^+ is defined by x^+ including also 0. c is an integer constant. Expression x has a lower $\{lb_1^x, ..., lb_a^x\}$ and b upper $\{ub_1^x, ..., ub_b^x\}$ bounds with $a, b \geq 1$

$$up(x) \rightarrow min(up(ub_1^x), \ldots, up(ub_b^x)) \quad : x \text{ has } b \text{ upper bounds}$$

$$up(c) \rightarrow c \quad : c \text{ is an integer constant} \tag{5.1}$$

$$low(x) \rightarrow max(low(lb_1^x), \ldots, low(lb_a^x)) \quad : x \text{ has } a \text{ lower bounds}$$

$$low(c) \rightarrow c \quad : c \text{ is an integer constant} \tag{5.2}$$

$$low(-x) \rightarrow -up(x) \tag{5.3}$$

$$up(-x) \rightarrow -low(x) \tag{5.4}$$

$$low(x + y) \rightarrow low(x) + low(y) \tag{5.5}$$

$$low(x - y) \rightarrow low(x) - up(y) \tag{5.6}$$

$$up(x + y) \rightarrow up(x) + up(y) \tag{5.7}$$

$$up(x - y) \rightarrow up(x) - low(y) \tag{5.8}$$

$$low(x * y) \rightarrow \begin{cases} low(x) * up(y) & : \ x \leq 0 \text{ and } y > 0 \\ up(x) * low(y) & : \ x \geq 0 \text{ and } y < 0 \\ low(x) * low(y) & : \ x, y \geq 0 \\ up(x) * up(y) & : \ x, y < 0 \\ min(low(x) * up(y), up(x) * low(y), & : \\ \quad low(x) * low(y), up(x) * up(y)) & : \ \text{otherwise} \end{cases} \tag{5.9}$$

$$low(\tbinom{x}{y}) \rightarrow \begin{cases} \binom{up(x)}{up(y)} & : \ x \geq 0 \text{ and } y < 0 \\ \binom{up(x)}{low(y)} & : \ x, y < 0 \\ \binom{low(x)}{up(y)} & : \ x \geq 0 \text{ and } y > 0 \\ \binom{low(x)}{low(y)} & : \ x < 0 \text{ and } y > 0 \\ min(\binom{up(x^-)}{low(y)}, \binom{low(x)}{low(y^+)}, \binom{up(x)}{up(y^-)}, \binom{low(x^+)}{up(y)}) & : \ \text{otherwise} \end{cases} \tag{5.10}$$

$$low(x^c) \rightarrow \begin{cases} (low(x))^c & : \ (x \geq 0 \text{ and } c \geq 0) \text{ or} \\ & : \ (x \leq 0 \text{ and } c \geq 0 \text{ and } c \text{ odd}) \text{ or} \\ & : \ (x < 0 \text{ and } c < 0 \text{ and } c \text{ even}) \\ (up(x))^c & : \ (x > 0 \text{ and } c < 0) \text{ or} \\ & : \ (x \leq 0 \text{ and } c \geq 0 \text{ and } c \text{ even}) \text{ or} \\ & : \ (x < 0 \text{ and } c < 0 \text{ and } c \text{ odd}) \\ min((low(x))^c, (up(x))^c, (low(x_0^+))^c) & : \ \text{otherwise} \end{cases} \tag{5.11}$$

Q_{new} may induce tighter and consequently more precise bounds for e as well. Therefore, the algorithm iterates as long as Q_{new} changes between any pair of consecutive repeat-loop iterations. At the end of a specific iteration, e might be a constant or an infinity symbol which cannot be changed by additional iterations of the algorithm as no variables or parameters are available for substitution. For this reason e is set to the original expression at the beginning of each iteration of the repeat-loop.

5.3.2 Rewrite φ-Expressions

Rewriting $\varphi(e)$ – where e is a symbolic expression – describes the process of applying a rewrite rule (see Table 5.1) to $\varphi(e)$. For instance, a φ-expression $low(e_1+e_2)$ is rewritten as $low(e_1)+low(e_2)$ according to rule 5.5 of Table 5.1: $low(e_1 + e_2) \rightarrow low(e_1) + low(e_2)$. e_1 and e_2 are symbolic expressions. This means that $low(e_1 + e_2)$ is textually *replaced* (semantically equivalent) by $low(e_1)+low(e_2)$. If e in $\varphi(e)$ is a variable or a parameter then we frequently use the term *substituting* instead of rewriting.

Continuing Ex. 5.1.1 of the previous section, for which we try to prove that $\mathtt{IJ} \geq \mathtt{KL}$ in statement ℓ_{10}: $\mathtt{X}(\mathtt{KL}, \mathtt{IJ}) = \mathtt{X}(\mathtt{IJ}, \mathtt{KL}) + \mathtt{VAL}$ for all loop iterations of L_3 which implies that L_3 can be executed in parallel. Note that this may still imply an anti dependence (array elements are read before written). If array $\mathtt{X}$ is block distributed then statement ℓ_{10} still causes communication. However, all communication can be hoisted outside of L_3 through communication vectorization and aggregation [50]. As $\mathtt{IJ} = \frac{\mathtt{J1}*(\mathtt{J1}-1)}{2} + \mathtt{J2}$ and $\mathtt{KL} = \frac{\mathtt{J3}*(\mathtt{J3}-1)}{2} + \mathtt{J4}$ according to statements ℓ_2, ℓ_6 and ℓ_9 of Ex. 5.1.1, we have to show that

$$\tfrac{\mathtt{J1}*(\mathtt{J1}-1)}{2} + \mathtt{J2} \geq \tfrac{\mathtt{J3}*(\mathtt{J3}-1)}{2} + \mathtt{J4} \Longleftrightarrow \tfrac{\mathtt{J1}*(\mathtt{J1}-1)}{2} + \mathtt{J2} - \tfrac{\mathtt{J3}*(\mathtt{J3}-1)}{2} - \mathtt{J4} \geq \mathtt{0}.$$

The set of constraints I for this problem is given by (5.1). In order to prove the absence of a true dependence we have to show that $low(\frac{\mathtt{J1}*(\mathtt{J1}-1)}{2} + \mathtt{J2} - \frac{\mathtt{J3}*(\mathtt{J3}-1)}{2} - \mathtt{J4}) \geq \mathtt{0}$:

$$\begin{aligned} & low(\frac{\mathtt{J1}*(\mathtt{J1}-1)}{2} + \mathtt{J2} - \frac{\mathtt{J3}*(\mathtt{J3}-1)}{2} - \mathtt{J4}) \\ & \rightarrow low(\frac{\mathtt{J1}*(\mathtt{J1}-1)}{2}) + low(\mathtt{J2}) - up(\frac{\mathtt{J3}*(\mathtt{J3}-1)}{2}) - up(\mathtt{J4}) \end{aligned} \quad (5.2)$$

which is based on rewrite rules (7) and (8) of Table 5.1. Note that Table 5.1 displays primarily rewrite rules for *low* functions. Rewrite rules for *up* functions are similar. Rewrite rules for expressions containing infinity symbols have not been included, since they are easy to determine.

The aim of rewriting φ-expressions is to reduce all sub-expressions, variables, and parameters of e until e is a constant or an infinity symbol which is then returned as the result of the algorithm. In order to find the correct rewrite rule to be applied to a multiplication, division or exponentiation expression, the algorithm may have to determine the value ranges (signs) of the associated sub-expressions. This is done by recursively calling *expr_bound*.

It may actually occur that variables are induced again by replacing parameters, as the bounds of parameters (in B) may contain variables. In this case we recursively call the algorithm for the variable bounds required.

In order to guarantee that *expr_bound* terminates we use the following *termination condition (TC)*: If the algorithm is trying to rewrite *low(expr)* in a recursive call of *expr_bound* while trying to rewrite *low(expr)* in a previous

recursion, then *low(expr)* is replaced by $-\infty$. The same holds for *up(expr)* which is replaced by ∞. Furthermore, if there are no lower or upper bounds given for a variable or a parameter, then the associated lower and upper bound is assumed to be $-\infty$ and ∞, respectively.

Whenever *expr_bound* deduces a new tighter lower or upper bound for a variable, it is stored (statements 12 and 18 in Fig. 5.1) in Q_{new}.

Some of the rewrite rules for φ-functions may contain φ-functions applied to x^- (all negative integers of x) or x^+ (all positive integers of x). If the algorithm is unable to apply rewrite rules to such expressions because there is no information available for the value range (sign) of x^+ and x^-, then the following rules – which enforce conservative and, therefore, correct bounds – are used:

$$\begin{aligned} low(x^-) &\rightarrow -\infty \\ low(x^+) &\rightarrow 1 \\ up(x^-) &\rightarrow -1 \\ up(x^+) &\rightarrow \infty \end{aligned}$$

5.3.3 Rewrite Policies

In statement 9 of Fig. 5.1 it is first tried to find sub-expressions of e containing variables for rewriting. Each variable corresponds to a loop variable with a unique loop nest level. Those φ-expressions that contain the variable with the highest loop nest level (loop variable appearing in innermost loop) have to be rewritten first; the one with the second innermost loop variable second, ..., the one with the outermost loop variable will be rewritten at the very end. If there are several φ-expressions which contain a variable with highest loop nest level then choose any of these φ-expressions for rewriting.

If there are no more variables in e then rewrite (statements 10 and 16) those φ-expressions in e first that contain parameters which depend (their lower or upper bounds contain other parameters) on other parameters.

Substituting variables – with their lower or upper bounds – before parameters enables us in many cases to find tighter bounds for φ-expressions. Each variable is in fact a loop variable in the underlying problem. Loop variables may directly or indirectly depend on constants or parameters (loop invariants). Overly conservative expression bounds may be computed, if we reduce all variables to constants, $-\infty$, and ∞ without ever trying to simplify intermediate expressions. See also Section 5.3.4.

Continuing rewriting (5.2) of the previous section, we determine:

$$\begin{aligned} & low(\frac{\texttt{J1} * (\texttt{J1}-1)}{2}) + low(\texttt{J2}) - up(\frac{\texttt{J3} * (\texttt{J3}-1)}{2}) - up(\texttt{J4}) \\ & \rightarrow low(\frac{\texttt{J1} * (\texttt{J1}-1)}{2}) + low(\texttt{J2}) - up(\frac{\texttt{J3} * (\texttt{J3}-1)}{2}) - up(\texttt{J2}-1) \end{aligned} \quad (5.3)$$

$$\rightarrow low(\frac{\texttt{J1} * (\texttt{J1}-\texttt{1})}{\texttt{2}}) - up(\frac{\texttt{J3} * (\texttt{J3}-\texttt{1})}{\texttt{2}}) + \texttt{1} \tag{5.4}$$

$$\rightarrow low(\frac{\texttt{J1} * (\texttt{J1}-\texttt{1})}{\texttt{2}}) - up(\frac{\texttt{J1} * (\texttt{J1}-\texttt{1})}{\texttt{2}}) + \texttt{1} \tag{5.5}$$

Firstly, we have to rewrite the term that contains the innermost loop variable J4 which is up(J4). By replacing J4 with its upper bound (J2 − 1) we obtain (5.3). Simplifying (5.3) yields (5.4) which is explained in the next section. Thereafter, we search again for the term with the highest loop nest level. Only J1 and J3 appear in (5.4). J3 is substituted next as it has a higher loop nest level than J1 which implies (5.5).

5.3.4 Simplify Expressions

In statements 11 and 17 of Fig. 5.1 we try to simplify e by first saving the outermost φ-function of e in φ', then replacing all $\varphi(y)$ in e by y – where y is a symbolic expression – which yields e'. Thereafter, we simplify e' using standard symbolic expression simplification. Finally, e is replaced by $\varphi'(E')$ which is the new simplified e. Note that $\varphi'(E')$ is guaranteed to represent either the same or a tighter bound for e. For instance, let $e = \mathbf{low}(\frac{up(x^2)+up(x)}{low(x)} - 1)$ where x is a symbolic expression, then the algorithm first saves the outermost φ-function of e – which is **low** – in φ'. Replacing all $\varphi(y)$ in e by y yields $e' = \frac{x^2+x}{x} - 1$ which can be further simplified to $e' = x$. We then re-apply φ' to e' which yields $low(x)$, the new simplified e.

We have observed that we achieve better simplification results by substituting first those variables and parameters – by their corresponding bounds – that depend on variables and parameters, as opposed to an arbitrary replacement order. Our strategy, therefore, is to replace variables in the opposite order as appearing in a loop nest (loop variable of innermost loop first, loop variable of second innermost loop second, etc.). Thereafter, we substitute parameters that depend on other parameters first. Parameters that do not depend on any other parameter are replaced at the very end.

If we would simultaneously substitute all variables and parameters by their corresponding bounds, then we may get very conservative bounds. Consider an example where $e = low(x) - up(x) + 1$, $low(x) = 1$, and $up(x) = \infty$. If we simplify e according to the method described above, then we obtain $x - x + 1$ which is further reduced to 1. On the other hand, we may reduce e as follows: $low(x) - up(x) + 1 \rightarrow 1 - \infty + 1 \rightarrow -\infty$. Clearly, the first result represents a much tighter bound for e than the second result.

We continue with Ex. 5.1.1 by simplifying (5.5) as follows:

$$\begin{aligned} & low(\frac{\texttt{J1} * (\texttt{J1} - \texttt{1})}{\texttt{2}}) - up(\frac{\texttt{J1} * (\texttt{J1} - \texttt{1})}{\texttt{2}}) + \texttt{1} \\ & \rightarrow \frac{\texttt{J1} * (\texttt{J1} - \texttt{1})}{\texttt{2}} - \frac{\texttt{J1} * (\texttt{J1} - \texttt{1})}{\texttt{2}} + \texttt{1} \\ & \rightarrow \texttt{1} \end{aligned} \tag{5.6}$$

5.3.5 Determine Result

Finally, in statement 21 we reach a point where e is an expression consisting only of constants and infinity symbols. If e is a constant, $-\infty$ or ∞ then e is returned as a result in e_b (output parameter of algorithm *expr_bound*). In all other cases the algorithm makes a worst-case assumption and respectively returns ∞ or $-\infty$ depending on whether the algorithm has been called for an upper or lower bound.

In continuation of Ex. 5.1.1 we determine that (5.6) has already been simplified to an integer constant, which is the lower bound of the original expression $low(\frac{\texttt{J1}*(\texttt{J1}-1)}{2} + \texttt{J2} - \frac{\texttt{J3}*(\texttt{J3}-1)}{2} - \texttt{J4})$. This enables us to conclude the following: As $low(\frac{\texttt{J1}*(\texttt{J1}-1)}{2} + \texttt{J2} - \frac{\texttt{J3}*(\texttt{J3}-1)}{2} - \texttt{J4}) \geq \texttt{1}$ and, therefore, $\texttt{IJ} \geq \texttt{KL}$, we can safely parallelize loop L_3 of Ex. 5.1.1. Most existing dependence analysis techniques assume a true dependence for statement ℓ_{10} in this code and consequently serialize loop ℓ_8 as they fail to evaluate non-linear array subscript expressions.

5.4 Count Solutions to a System of Constraints

Counting the number of integer solutions to a set of constraints has been shown to be a key issue in performance analysis of parallel programs. Numerous applications [41, 100, 67, 98, 91] include estimating statement execution counts, branching probabilities, work distribution, number of data transfers and cache misses. Even compiler analysis can be supported, for instance, by detecting and eliminating dead code such as loops that never iterate (zero-trip loops).

Consider the following loop nest with a statement ℓ_4 included in a conditional statement.

```
ℓ1:  do J1 = 1, N1
ℓ2:      do J2 = 1, N2 * J1
ℓ3:          if (J1 ≤ N2) then
ℓ4:          A = A + ...
ℓ5:          end if
ℓ6:      end do
ℓ7:  end do
```

Computing how many times statement ℓ_4 is executed is equivalent to counting the number of integer solutions of $I = \{1 \leq \texttt{J1} \leq \texttt{N1}, 1 \leq \texttt{J2} \leq \texttt{N2} * \texttt{J1}, \texttt{J1} \leq \texttt{N2}\}$. J1 and J2 are (loop) *variables* and N1, N2 are *parameters* (loop invariants).

Note that we consider $\texttt{J2} \leq \texttt{N2} * \texttt{J1}$ to be non-linear, although N2 is loop invariant. The statement execution count for ℓ_4 is given by:

$$\sum_{\mathtt{J1}=1}^{min(\mathtt{N1},\mathtt{N2})} \sum_{\mathtt{J2}=1}^{\mathtt{N2}*\mathtt{J1}} 1 = \begin{cases} \frac{\mathtt{N1}^2*\mathtt{N2}}{2} + \frac{\mathtt{N1}*\mathtt{N2}}{2} & : \text{ if } 1 \leq \mathtt{N1} \leq \mathtt{N2} \\ \frac{\mathtt{N2}^3}{2} + \frac{\mathtt{N2}^2}{2} & : \text{ if } 1 \leq \mathtt{N2} < \mathtt{N1} \end{cases}$$

In general, every loop implies at least two constraints on its loop variable, one for its upper and one for its lower bound. Additional constraints on both parameters and variables can be implied, for instance, by conditional statements, minimum and maximum functions, data declarations, etc.

Algorithms [41] that count the number of integer solutions of arbitrary large sets of linear constraints over loop variables have already been implemented and successfully used in the context of performance analysis and parallelizing compilers. The drawback with these algorithms, however, is that program unknowns (*parameters*) must be replaced by constant integers. In order to overcome this disadvantage M. Haghighat and C. Polychronopoulos [67] described an algebra of conditional values and a set of rules for transforming symbolic expressions in order to compute symbolic sums. However, they did not present an algorithm that decides which rule to apply when. N. Tawbi [100] developed a symbolic sum algorithm which handles loop nests with parameters. Her approach is restricted to sums based on linear inequalities implied by loop header statements. W. Pugh [91] improved Tawbi's algorithm by extending it to techniques that count the number of integer solutions for an arbitrary set of linear constraints.

In this section we describe a symbolic sum algorithm that goes beyond previous work by estimating the number of integer solutions to linear as well as a class of non-linear constraints.

5.4.1 Symbolic Sum Computation

In this section we describe a symbolic algorithm which computes the number of integer solutions of a set of linear and non-linear constraints I defined over $V \cup P$. Every $c \in I$ is restricted to be of the following form:

$$p_1(\boldsymbol{p}) * v_1 + \cdots + p_k(\boldsymbol{p}) * v_k \;\; REL \;\; 0 \tag{5.7}$$

where REL $\in \{\leq, \geq, <, >, =, \not=\}$ represents an equality or inequality relationship. $\boldsymbol{p}$ is a vector defined over parameters of P. $p_i(\boldsymbol{p})$ are linear or non-linear expressions over P, whose operations can be addition, subtraction, multiplication, division, floor, ceiling, and exponentiation. Constraints with minimum and maximum functions are substituted where possible by constraints free of minimum and maximum functions which is shown in Section 5.4.4. Not all variables in $\{v_1, \ldots, v_k\}$ are necessarily pairwise different where $v_i \in V$ $(1 \leq i \leq k)$. Then

$$W = \{\boldsymbol{v} | \boldsymbol{v} \text{ satisfies } I\}$$

is the set of integer solutions over all variables in V which satisfy the conjunction of constraints in I. Every $\boldsymbol{v}$ is a vector over variables in V, which represents a specific solution to I. Figure 5.2 shows the algorithm for counting

the number of elements in W, given I, P, V, e, and R. A symbolic expression e is an intermediate result for a specific solution e_i of the symbolic sum algorithm. R and the output parameter R_{new} are a set of tuples (C_i, e_i) where $1 \leq i \leq k$. Each tuple corresponds to a conditional solution of the sum algorithm. C_i is a set of linear or non-linear constraints (satisfying (5.7)) defined over P which must be interpreted as a conjunction of these constraints. Note that the constraints in C_i of the solution tuples are not necessarily disjoint. The result has to be interpreted as the sum over all e_i under the condition of C_i as follows:

$$\sum_{1 \leq i \leq k} \varrho(C_i) * e_i \tag{5.8}$$

where ϱ is defined based on a set of constraints $C = \{c_1, ..., c_n\}$:

$$\varrho(C) = \begin{cases} 1 & : \text{ if } \left[\bigwedge_{1 \leq i \leq n} c_i \right] = \text{true} \\ 0 & : \text{ otherwise} \end{cases} \tag{5.9}$$

R and e must be, respectively, set to ϕ (empty set) and 1 at the initial call of the algorithm.

5.4.2 The Algorithm

In each recursion the algorithm is eliminating one variable $v \in V$. First, all lower and upper bounds of v in I are determined. Then the maximum lower and minimum upper bound of v is searched by generating disjoint subsets of constraints based on I. For each such subset I', the algebraic sum (see Section 5.4.3) of the current e over v is computed. Then the sum algorithm is recursively called for I', the newly computed e, $V - \{v\}$, P, and R_{new}. Eventually at the deepest recursion level, V is empty, then e and its associated I represent one solution tuple defined solely over parameters.

In statement 2 we simplify I, which includes detection and elimination of tautologies, equalities and redundant inequalities. Our simplification techniques and algorithms are described in detail in Section 5.5. If I contains a contradiction then statement 4 returns without adding a solution tuple to the output parameter R_{new}. In statement 6 the algorithm checks for an empty set of variables. If so, then there are no variables remaining for being eliminated. The algorithm adds the current solution tuple (I, e) to R_{new} and returns.

Statements 10–20 describe the core of the algorithm, which addresses several issues:

- In statement 10, a variable is chosen which is being eliminated in this recursion. A heuristic is used which selects the variable appearing in as few constraints of I as possible. Note that an inequality which contains more than one variable is a (lower or upper) bound for all of these variables.

sum(I,V,P,e,R)

- **INPUT:**
  - I: set of linear and non-linear constraints defined over $V \cup P$
  - V: set of variables
  - P: set of parameters
  - e: symbolic expression defined over $V \cup P$
  - R: set of solution tuples (C_i, e_i) where $1 \leq i \leq k$. C_i is a set of linear or non-linear constraints defined over P. e_i is a linear or non-linear symbolic expression defined over P.
- **OUTPUT:**
  - R_{new}: R updated with additional solution tuples.
- **METHOD:**

```
1:  R_new := R
2:  Simplify I
3:  if I is inconsistent (no solution) then
4:      return
5:  end if
6:  if V = φ then
7:      R_new := R_new ∪ (I, e)
8:      return
9:  end if
    // Split I
10: Choose variable v ∈ V for being eliminated
11: I'' := subset of I not involving v
12: L := {l_1,...,l_a}                    // set of lower bounds of v in I
13: U := {u_1,...,u_b}                    // set of upper bounds of v in I
14: a := |L|
15: b := |U|
16: for each (l_i, u_j) ∈ L × U do
17:     I'_{i,j} := I'' ∪ { l_1 < l_i, .., l_{i-1} < l_i, l_i ≤ l_{i+1}, .., l_a ≤ l_i }
            ∪ {u_1 > u_j, .., u_{j-1} > u_j, u_j ≤ u_{j+1}, .., u_b ≥ u_j} ∪ {l_i ≤ u_j}
18:     e_{i,j} := Σ_{v=l_i}^{u_j} E
19:     R_new := R_new ∪ sum(I'_{i,j}, V − {v}, P, e_{i,j}, R_new)
20: end for
```

Fig. 5.2. Symbolic sum algorithm for computing the number of solutions of a set of constraints I

- A variable $v \in V$ may have several lower and upper bounds in I. In order to compute the sum of all integer solutions for v over all constraints in I, we need to know the maximum lower bound and minimum upper bound of v. In general we may not know these bounds, therefore, we have to generate disjoint subsets of constraints based on I which is done in the for-loop. For each such subset $I'_{i,j}$ it is assumed that l_i and u_j is the associated maximum lower and minimum upper bound for v, respectively. Clearly, in this step we may create sets of constraints $I'_{i,j}$ that contain contradictions

which are tried to be detected in statement 3 (see also Section 5.5). Choosing v according to the previous heuristic implies a minimum number of different $I'_{i,j}$ created at this step.
- For all of the previously created subsets $I'_{i,j}$, we compute the algebraic sum $e_{i,j}$ (see Section 5.4.3) based on the current e by iterating over v which is bound by l_i and u_i.
- The algorithm is recursively called with $I'_{i,j}$, $V - \{v\}$, P, $e_{i,j}$, and R_{new}.

We further extended the sum algorithm of Fig. 5.2 to a larger class of constraints that goes beyond I as described for (5.7). In the following we assume that I represents a set of constraints that satisfy the conditions mentioned by (5.7). Let I_1 be a set of constraints defined over $V \cup P$ which is given by

$$p_1(\boldsymbol{p})f_1(\boldsymbol{V_1}) + \cdots + p_k(\boldsymbol{p})f_k(\boldsymbol{v_k}) \;\; REL \;\; 0 \tag{5.10}$$

$p_i(\boldsymbol{p})$ and REL have the same meaning as described by (5.7). $\boldsymbol{v_i}$ describes a vector of variables over V_i (subset of variables in V) with $1 \leq i \leq k$. $f_i(\boldsymbol{v_i})$ are linear or non-linear expressions defined over V of the following form (transformed to their simplest form):

$$\frac{v_1 * \cdots * v_j}{w_1 * \cdots * w_l}$$

where all variables in $\{v_1, \ldots, v_j, w_1, \ldots, w_l\} \subseteq V$ must be pairwise different after simplification. Furthermore, any variable $v \in V$ may appear in several f_i with the restriction that v must always appear either in the denominator or in the numerator of all f_i in which v appears. For instance, v may appear in the numerator of all f_i. However, v cannot appear in the numerator of f_i and in the denominator of f_j with $i \neq j$.

Clearly, $I \subseteq I_1$. Unfortunately, in general the algorithm cannot handle constraints of the form described by (5.10), as for certain I_1 the algorithm may create a new set of constraints in the for-loop of the symbolic sum algorithm that does not honor the constraints of (5.10). We have not yet found a way to narrow down the constraints of I_1, such that the algorithm never creates a new constraint that does not satisfy (5.10). Nevertheless, we extended the symbolic sum algorithm to handle a larger class of constraints as described in (5.7) by simply checking at the beginning of the algorithm whether I satisfies (5.10). If yes, then the algorithm proceeds, otherwise, the algorithm returns and indicates that the original I cannot be processed for the reasons mentioned above. For instance, this extension of the symbolic sum algorithm enables us to count the integer solutions to the following set of constraints which is not covered by (5.7):

$$\texttt{1} \leq \texttt{J1} \leq \texttt{N1},\ \texttt{1} \leq \texttt{J1} * \texttt{J2} \leq \texttt{N2} * \texttt{J1},\ \texttt{N1} * \texttt{N2} \leq \texttt{J1} * \texttt{J2}$$

5.4.3 Algebraic Sum

In the following we describe the computation of the algebraic sum $\sum_{v=l}^{u} E$ over a variable $v \in V$, where l and u have been extracted (for-loop in Fig. 5.2) from constraints of I. e is a symbolic expression over $V \cup P$, which has been transformed to its simplest form by using our own symbolic manipulation package [95]. Computing the algebraic sum is then reduced to solving:

$$\sum_{v=l}^{u} e_0 + e_1 * v^{\pm 1} + e_2 * v^{\pm 2} + \cdots + e_q * v^{\pm q} \tag{5.11}$$

e_i is a linear or non-linear expression over $P \cup V - \{v\}$. $\pm q$ (q is an integer constant) means that the sign of q can either be positive or negative. The problem, therefore, amounts to computing the following: $\sum_{v=1}^{n} v^i$ where $1 \leq i \leq q$. In the following we show how to approximate all terms in (5.11) by an upper or lower bound or an approximate expression (average between upper and lower bound):

For the case where $i \geq 0$ we can use standard formulas for sums of powers of integers. They are described in [63] and reviewed in [100]. For $i < 0$ there are no closed forms known. However, for $i = -1$ and $2 \leq n$ it has been shown [63] that (ln is the natural logarithm):

$$\begin{aligned} ln(n) < \sum_{v=1}^{n} \tfrac{1}{v} &< ln(n) + 1 \\ -ln(n) - 1 < \sum_{v=-1}^{-n} \tfrac{1}{v} &< -ln(n) \end{aligned}$$

and also [25] that $\sum_{v=1}^{\infty} \frac{1}{v^2} = \frac{\pi^2}{6} \approx 1.65$. Based on this approximation it can be shown that for all $i \geq 2$ the following holds: $\sum_{v=1}^{\infty} \frac{1}{v^2} \geq \sum_{v=1}^{\infty} \frac{1}{v^i}$, and finally

$$\begin{aligned} 1 < \sum_{v=1}^{\infty} \tfrac{1}{v^i} &\leq \tfrac{\pi^2}{6} \text{ if } i \geq 2 \\ 1 < \sum_{v=-1}^{-\infty} \tfrac{1}{v^i} &\leq \tfrac{\pi^2}{6} \text{ if } i \geq 2 \text{ and } i \text{ is even} \\ -\tfrac{\pi^2}{6} \leq \sum_{v=-1}^{-\infty} \tfrac{1}{v^i} &< -1 \text{ if } i \geq 2 \text{ and } i \text{ is odd} \end{aligned}$$

5.4.4 Miscellaneous

In this section we briefly describe how to handle equalities, $\neq$ inequalities, min/max functions, and fractional expressions in our symbolic sum algorithm.

- Equalities are eliminated by our simplification techniques according to Section 5.5.
- If I contains a "$\not=$" inequality$c$, for instance, $\texttt{J1} * \texttt{J2} \not= \texttt{N}$, then we split I into two disjoint subsets I_1 and I_2 such that $I_1 := I - c \cup \{\texttt{J1} * \texttt{J2} > \texttt{N}\}$ and $I_2 := I - c \cup \{\texttt{J1} * \texttt{J2} < \texttt{N}\}$. The sum algorithm is then called for both subsets I_1 and I_2 and the combined result is equivalent to the original problem including the "$\not=$" inequality.
- In many cases we can eliminate *min* and *max* functions appearing in inequalities by rewriting them. For instance, let $c \in I$ be an inequality $\texttt{J1} \leq \texttt{min}(\texttt{J2}, \texttt{J3})$, then we replace c by the conjunction of two inequalities $\texttt{J1} \leq \texttt{J2}$ and $\texttt{J1} \leq \texttt{J3}$ in I. If, however, c is given by $\texttt{J1} \leq \texttt{max}(\texttt{J2}, \texttt{J3})$, then we have to split I into two disjoint subsets I_1 and I_2 such that $I_1 := I - c \cup \{\texttt{J1} \leq \texttt{J2}, \texttt{J2} \geq \texttt{J3}\}$ and $I_2 := I - c \cup \{\texttt{J1} \leq \texttt{J3}, \texttt{J2} < \texttt{J3}\}$. The algorithm is then called for both subsets I_1 and I_2 and the combined result is equivalent to the original problem including the *max* function. $\geq$ inequalities are handled similarly.
- There are several options to handle fractional expressions (floor and ceiling functions) in constraints and general symbolic expressions:
  - The simplest way to handle floor and ceiling operations is to assume that all operations are defined over rational numbers. This is clearly an approximation of the integer sum computation which may induce high estimation errors. However, we also observed that in many cases this approach achieves very reasonable estimates.
  - If a fractional contains only parameters and constants, then we can replace it by a newly introduced parameter and proceed as if there were no fractional. In the result we will substitute all such parameters back to their original fractional expression. This alternative may imply quite complex inequalities and result expressions (even though exact) as the dependencies among such fractional parameters can only partially be exploited by our simplification techniques. Of course, simplification within fractionals can still be done.
  - Fractional expressions can be replaced by their corresponding lower or upper bounds based on the following observation:
    $$\frac{e_1}{e_2} - 1 < \lfloor \frac{e_1}{e_2} \rfloor \leq \frac{e_1}{e_2} \leq \lceil \frac{e_1}{e_2} \rceil < \frac{e_1}{e_2} + 1$$
    The difference between the lower and upper bound may be used to specify the estimation accuracy. Also the average between lower and upper bound can be used to replace a fractional expression.

A detailed example of how to apply our symbolic sum algorithm is given in Section 5.6.2.

5.5 Simplify Systems of Constraints

The complexity of many compiler analyses as well as the associated results which are based on a set of constraints I can be considerably reduced if we can detect contradictions and eliminate tautologies, equalities and redundant constraints. Simplifying a set of constraints has been noted [44, 84] to be critical in order to simplify performance as well as compiler analysis. We have implemented the following simplification techniques:

- Tautology inequalities (e.g. $2 > 0$, $N \leq N$, and $N \leq N * N$) in I are detected and removed by using standard symbolic expression simplification and algorithm *expr_bound* (see Section 5.3).
- Search for two inequalities $I_1, I_2 \in I$ which induce an equality. For instance, let I_1 be $2 * J1 * J2 - N \leq 0$ and I_2 is $2 * J1 * J2 - N \geq 0$, which implies I_3 : $2 * J1 * J2 - N = 0$. If so, then we search for a variable v with a coefficient as simple as possible (e.g. constant) in the resulting equality, solve it for v and substitute in I for all occurrences of v. In general, searching for a coefficient with small complexity in a non-linear symbolic expression can be quite a difficult task to accomplish. We apply a heuristic that proceeds as follows: First search for constant coefficients. If there are no constant coefficients, then search for a coefficients that is a single parameter. If there are no single parameter coefficients then search for coefficients that are linear symbolic expressions. Otherwise take any coefficient. Only if there are no variables in the equality we will try to solve it for a parameter and proceed as for variables. Eliminating variables before parameters has higher priority as many symbolic compiler analyses try to eliminate variables in a set of constraints and provide results as symbolic expressions defined over parameters. Once we have successfully substituted equality I_3 we can delete I_1, I_2, and I_3 in I.
- Examine whether two inequalities of I contradict each other (e.g. $2 * J1 * J2 + N > 0$ and $2 * J1 * J2 + N + 3 \leq 0$), which means that I has no solution. If contradictions are detected most compiler analyses can be immediately terminated.
- Try to detect and eliminate redundant inequalities in I. This is in particular important for inequalities involving variables, as the number of inequalities in I is in many cases directly proportional to the complexity of finding a solution to I. Furthermore, in many cases we can drastically simplify results of symbolic analysis by removing redundant constraints. The following Lemma states a condition which allows us to eliminate redundant inequalities in a set of constraints.

Lemma 2: *Redundant Inequalities*
Let e_1, e_2 be two symbolic expressions defined in $V \cup P$ and $q \in V \cup P$. c_1, c_2 are two inequalities of I such that c_1 is defined by e_1 REL q and

c_2 by e_2 REL q. REL is an inequality relationship in $\{\leq, <\}$. Then c_2 can be eliminated from I, iff $low(e_1 - e_2) \geq 0$.

Proof:

1. *$e_1 > e_2$ and $e_1 \leq q \Rightarrow e_2 < q$*
2. *$e_1 = e_2$ and $e_1 \leq q \Rightarrow e_2 \leq q$*
   Based on 1. and 2. we can conclude that if $e_1 \geq e_2$ and $e_1 \leq q \Rightarrow e_2 \leq q \Rightarrow c_2$ is redundant and can be eliminated. See also the proof on page 53 which demonstrates that $e_1 \geq e_2 \Rightarrow low(e_1 - e_2) \geq 0$ and $up(e_2 - e_1) \leq 0$.
3. *$e_1 < e_2$ and $e_2 \leq q \Rightarrow e_1 < q$ Based on 2. and 3. we can conclude that if $e_1 \leq e_2$ and $e_2 \leq q \Rightarrow e_1 \leq q \Rightarrow c_1$ is redundant and can be eliminated.*

Note that $low(e_1 - e_2) \geq 0$ is identical with $up(e_2 - e_1) \leq 0$.

The lemma for inequality relationships $>$ and $\geq$ is similar. Therefore, algorithm *expr_bound* can also be used to detect and eliminate redundant inequalities.

5.6 Experiments

All algorithms and techniques as described in this book have been implemented and integrated with *VFC* (Vienna High Performance Compiler – see Section 6.4) and P^3T ([51] a performance estimator for parallel and distributed programs). In what follows, we describe three experiments to measure the potential benefits of our techniques for improving the analysis and performance of parallel programs.

5.6.1 Eliminate Redundant Constraints

In this section we continue Ex. 5.1.2 of Section 5.1 which demonstrates how algorithm *expr_bound* can be used to eliminate a redundant constraint on a variable and as a consequence simplifies other compiler analyses significantly. Consider the following set of inequalities as obtained from the code of Ex. 5.1.2.

$$I = \{1 \leq \texttt{J1} \leq \texttt{N},\ \tfrac{\texttt{N}}{2*\texttt{J1}} \leq \texttt{J2} \leq \texttt{N},\ \texttt{J1} * \texttt{J2} \leq \texttt{N}\}$$

This yields the following sets of variables, parameters, lower and upper bounds:
$V = \{\texttt{J1}, \texttt{J2}\}, P = \{\texttt{N}\}, L_{J1} = \{1, \frac{\texttt{N}}{2*\texttt{J2}}\}, U_{\texttt{J1}=\{N, \frac{N}{J2}\}}, L_{J2} = \{\frac{\texttt{N}}{2*\texttt{J1}}\}$,
$U_{J2} = \{\texttt{N}, \frac{\texttt{N}}{\texttt{J1}}\}, L_N = \{\texttt{J1}, \texttt{J2}, \texttt{J1} * \texttt{J2}\}, U_N = \{2 * \texttt{J1} * \texttt{J2}\}$,
$B = L_N \cup U_N \cup \bigcup_{1 \leq i \leq 2} (L_{Ji} \cup U_{Ji})$
where L_r and U_r are respectively the sets of lower and upper bounds for

Table 5.2. Example that demonstrates the usage of *expr_bound* to prove that $\mathtt{N} \geq \frac{\mathtt{N}}{\mathtt{J2}}$

R	S	e	Q	Comments
1	1	low($\mathtt{N} - \frac{\mathtt{N}}{\mathtt{J2}}$)		
	2	$\rightarrow$ low(N) - up($\frac{\mathtt{N}}{\mathtt{J2}}$)		rule (8)
2	1	low(J2)		
	2	$\rightarrow$ low($\frac{\mathtt{N}}{2*\mathtt{J1}}$)		rule (3)
3	1	low(J1)		
	2,4	$\rightarrow$ max(low($\frac{\mathtt{N}}{2*\mathtt{J2}}$),1)	(J1,1,∞)	rule (3)
	4		(J2,$-\infty$,∞)	TC for low(J2)
4	1	up(J2)		
	2	$\rightarrow$ min(up(N),up($\frac{\mathtt{N}}{\mathtt{J1}}$))		rule (3)
5	5	low(N)		
	6	$\rightarrow$ max(low(J1), low(J2),low(J1*J2))		rule (3)
	6,8	$\rightarrow$ max(1,low(J2),min(low(J1)*up(J2), up(J1)*low(J2),low(J1)*low(J2), up(J1)*up(J2))	(N,1,∞)	rule (9)
6	1	up(J1)		
	2	$\rightarrow$ min(up(N),up($\frac{\mathtt{N}}{\mathtt{J2}}$))		rule (1)
	2	$\rightarrow$ min(up(N),∞)		TC for up($\frac{\mathtt{N}}{\mathtt{J2}}$)
	3,4	$\rightarrow$ up(N)	(J1, 1, N)	
5	6	$\rightarrow$ max(1,$-\infty$,min(1 $*$ ∞,up(N)*-∞, 1*$-\infty$,up(N)*∞))		TC for up(N)
	7	$\rightarrow$ max(1,$-\infty$, min(∞,$-\infty$,$-\infty$,∞))		simplify
	7	$\rightarrow$ max(1,$-\infty$,$-\infty$)		simplify
	7	$\rightarrow$ 1		simplify
4	2	$\rightarrow$ min(up(N),$\frac{\text{up(N)}}{\text{low(J1)}}$))		rule (10)
	2	$\rightarrow$ min(up(N),$\frac{\text{up(N)}}{1}$)		subst. low(J1)
	3,4	$\rightarrow$ up(N)	(J2,$-\infty$,N)	
3	2	$\rightarrow$ max(min($\frac{\text{up(N}^-\text{)}}{\text{low(2*J2)}}$,$\frac{\text{low(N)}}{\text{low(2*J2}^+\text{)}}$, $\frac{\text{up(N)}}{\text{up(2*J2}^-\text{)}}$, $\frac{\text{low(N)}}{\text{up(2*J2)}}$), 1)		rule (10)
	2	$\rightarrow$ max(min($\frac{\text{low(N)}}{2}$, $\frac{\text{up(N)}}{2*(-1)}$, $\frac{\text{low(N)}}{2*\text{up(N)}}$),1)		subst.
	3	$\rightarrow$ max(-$\frac{\text{up(N)}}{2}$,1)		simplify
	3	$\rightarrow$ 1		simplify
2	2	$\rightarrow$ $\frac{\text{low(N)}}{2*\text{up(J1)}}$		rule (10)
	2	$\rightarrow$ $\frac{\text{low(N)}}{2*\text{up(N)}}$		subst. up(J1)
	3	$\rightarrow$ $\frac{1}{2}$		simplify
	3,4	$\rightarrow$ 1	(J2,1,N)	J2 integer
1	2	$\rightarrow$ low(N) - $\frac{\text{up(N)}}{\text{low(J2)}}$		rule (10)
	2	$\rightarrow$ low(N) - $\frac{\text{up(N)}}{1}$		subst. low(J2)
	3	$\rightarrow$ low(N) - up(N)		simplify
	9	$\rightarrow$ 0		proved $\mathtt{N} \geq \frac{\mathtt{N}}{\mathtt{J2}}$

a variable or parameter r which are derived from I. An interesting question arises whether one of the two upper bounds for J1 in I is redundant. If we can show that $low(\mathtt{N} - \frac{\mathtt{N}}{\mathtt{J2}}) \geq \mathtt{0}$ for all possible values of variables and parameters

in $V \cup P$ according to Lemma 2, then $\texttt{J2} \leq \texttt{N}$ is redundant in I. For this purpose we invoke *expr_bound*($e = low(\texttt{N} - \frac{\texttt{N}}{\texttt{J2}})$,B,Q={(J1,$-\infty$,$\infty$),(J2,$-\infty$,$\infty$),(N,$-\infty$,$\infty$)}). Table 5.2 shows the most interesting steps for each recursion of *expr_bound*. Note that there are no rule numbers for *up* functions as they are not shown in Table 5.1.

R specifies the current recursion of the algorithm, and S the algorithm statement number (as shown in Table 5.2) within a given recursion. The current symbolic expression as processed by *expr_bound* is given by e. *TC* refers to the termination condition of *expr_bound*. The current contents of Q at a specific table entry is the union of the most recent table entries for each different variable and parameter. E.g. $Q = \{(\texttt{J1}, 1, \texttt{N}), (\texttt{J2}, 1, \texttt{N}), (\texttt{N}, 1, \infty)\}$ at recursion 2 statement 3.

An item $i.j$ of the following itemized list corresponds to recursion i and statement j of algorithm *expr_bound*.

1.2 In order to identify the rewrite rule to apply to up($\frac{\texttt{N}}{\texttt{J2}}$), the value range for J2 needs to be known. Therefore, the algorithm is recursively (recursion 2) called for $e =$ low(J2) to find out whether J2 can ever be negative.

3.2 Allows us to deduce that low(J1) $\geq$ 1 which is used to update the lower bound of J1 in Q. Then the algorithm is evaluating $low(\frac{\texttt{N}}{2*\texttt{J2}})$ to investigate whether this expression is larger than 1. In order to determine the correct rewrite rule to apply to $low(\frac{\texttt{N}}{2*\texttt{J2}})$, we need to determine the value range for J2. As recursion 2 already tried to deduce low(J2), we add $-\infty \leq$ J2 into Q based on the termination condition (see Section 5.3.2) of *expr_bound*. Then the algorithm is recursively called for $e = up$(J2) to determine whether J2 can ever be positive.

5.8 As the maximum function for low(N) contains 1, we can add 1 $\leq$ N into Q. We must determine the sign of up(J1) in order to further rewrite the minimum function.

6.2 As in recursion 1 statement 2 the algorithm already tried to evaluate $up(\frac{\texttt{N}}{\texttt{J2}})$, we substitute $up(\frac{\texttt{N}}{\texttt{J2}})$ by ∞ according to the termination condition of *expr_bound*. After statement 4 of this recursion, the algorithm returns to recursion 5.

2.3 The expression after recursion 2 statement 2 has been simplified according to statement 10, which yields $\frac{1}{2}$. This means that $low(\texttt{J2}) \geq \frac{1}{2}$, and consequently, $low(\texttt{J2}) \geq 1$ as low(J2) must be an integer.

1.3 Simplifies $low(\texttt{N}) - up(\texttt{N})$ according to statement 10, which yields $\texttt{N} - \texttt{N} = 0$.

1.9 Proves that $\texttt{N} \geq \frac{\texttt{N}}{\texttt{J2}}$ for all values of J1, J2, and N. Therefore, the inequality J2 $\leq$ N is redundant in I according to Lemma 2.

Other important compiler analyses such as detecting zero-trip-loops, dead code elimination, and performance prediction commonly examine whether I has a solution at all. If I contains program unknowns (parameters) then these analyses may yield guarded solutions. In Section 5.4 we have described a symbolic sum algorithm that computes the number of solutions of a set of

constraints which are defined over variables and parameters. For instance, by detecting and eliminating the redundant inequality $\texttt{J2} \leq \texttt{N}$ in I, the number of guarded solutions to I as determined by our symbolic sum algorithm is reduced by 50 %.

Note that Table 5.2 shows an overly conservative application of our algorithm for the sake of illustrating the most interesting aspects of *expr_bound*. The actual implementation of *expr_bound* detects at the beginning of the first recursion that $\texttt{J1}, \texttt{J2}, \texttt{N} \geq 1$ by simply checking the lower bound expressions of B and initializes Q with this information. Based on Q the following rewrite rules are applied: $low(\texttt{N} - \frac{\texttt{N}}{\texttt{J2}}) \rightarrow low(\texttt{N}) - up(\frac{\texttt{N}}{\texttt{J2}}) \rightarrow low(\texttt{N}) - \frac{up(\texttt{N})}{low(\texttt{J2})} \rightarrow low(\texttt{N}) - \frac{up(\texttt{N})}{1} \rightarrow low(\texttt{N}) - up(\texttt{N}) \rightarrow 0$. Therefore, our implementation can prove in 5 steps of a single iteration of *expr_bound* that $\texttt{N} \geq \frac{\texttt{N}}{\texttt{J2}}$ under the constraints of I. A step corresponds to the application of a single rewrite rule of Table 5.1 to a symbolic expression.

5.6.2 Counting Solutions to a System of Constraints

In what follows we present an experiment that demonstrates how the symbolic sum algorithm can be used to estimate the amount of work to be processed by every processor of a data parallel program. The following code shows a High Performance Fortran - HPF code excerpt with a processor array PR of size P.

```
   integer :: A(N2)
   !hpf$ processors :: PR(P)
   !hpf$ distribute (block) onto PR :: A
l1:  do J1=1,N1
l2:       do J2 = 1, J1 * N1
l3:            if ( J2 ≤ N2 ) then
l4:              A(J2) = ...
l5:            end if
l6:       end do
l7:  end do
```

The loop contains a write operation to a one-dimensional array A which is block-distributed onto P processors. Let k $(1 \leq k \leq \texttt{P})$ denote a specific processor of the processor array. Computations that define the data elements owned by a processor k are performed exclusively by k. For the sake of simplicity we assume that P evenly divides N2. Therefore, a processor k is executing the assignment to A based on the underlying block distribution if $\frac{\texttt{N2}*(k-1)}{\texttt{P}} + 1 \leq \texttt{J2} \leq \frac{\texttt{N2}*k}{\texttt{P}}$. The precise work to be processed by a processor k is the number of times k is writing A, which is defined by *work(k)*.

The problem to estimate the amount of work to be done by processor k can now be formulated as counting the number of integer solutions of I which is given by:

$$\begin{aligned} 1 \leq \texttt{J1} &\leq \texttt{N1} \\ 1 \leq \texttt{J2} &\leq \texttt{J1} * \texttt{N1} \\ \texttt{J2} &\leq \texttt{N2} \\ \frac{\texttt{N2}*(k-1)}{\texttt{P}} + 1 \leq \texttt{J2} &\leq \frac{\texttt{N2}*k}{\texttt{P}} \end{aligned}$$

In the following we substitute $\frac{\mathtt{N2}*(\mathtt{k}-1)}{\mathtt{P}}+1$ by L and $\frac{\mathtt{N2}*\mathtt{k}}{\mathtt{P}}$ by U. By using our simplification techniques of Section 5.5 we can determine that $1 \le \mathtt{J2}$ is made redundant by $\mathtt{L} \le \mathtt{J2}$ and $\mathtt{J2} \le \mathtt{N2}$ by $\mathtt{J2} \le \mathtt{U}$. Therefore, the simplified I with all redundant inequalities removed is given by

$$\begin{aligned} 1 \le \mathtt{J1} &\le \mathtt{N1} \\ \mathtt{L} \le \mathtt{J2} &\le \mathtt{U} \\ \mathtt{J2} &\le \mathtt{J1}*\mathtt{N1} \end{aligned}$$

In the first recursion we can either choose variable J1 or J2 for being eliminated according to the heuristic of statement 9 of our sum algorithm (see Fig. 5.2). Both variables appear in 3 inequalities of the simplified I. By selecting J2 we obtain two upper bounds for J2 which are given by $\{\mathtt{J1}*\mathtt{N1}, \mathtt{U}\}$. Based on statement 15 we split the inequalities of I into I_1 and I_2.

- $I_1 = \{1 \le \mathtt{J1} \le \mathtt{N1},\ \mathtt{J1}*\mathtt{N1} \le \mathtt{U},\ \mathtt{L} \le \mathtt{J1}*\mathtt{N1}\}$ with $\sum_{\mathtt{J2}=\mathtt{L}}^{\mathtt{J1}*\mathtt{N1}} 1 = \mathtt{J1}*\mathtt{N1} - \mathtt{L}$.

  In statement 2 of the algorithm (see Fig. 5.2) we detect that $1 \le \mathtt{J1}$ is made redundant by $\frac{\mathtt{L}}{\mathtt{N1}} \le \mathtt{J1}$. We now eliminate J1 based on two upper bounds $\{\frac{\mathtt{U}}{\mathtt{N1}}, \mathtt{N1}\}$ for this variable. This yields two solutions (c_1,e_1) and (c_2,e_2). Note that $e_1(k)$, $e_2(k)$, and $e_3(k)$ are processor specific.
  - $c_1 = \{\frac{\mathtt{U}}{\mathtt{N1}} \le \mathtt{N1},\ \mathtt{L} \le \mathtt{U}\} = \{\mathtt{U} \le \mathtt{N1}^2,\ \mathtt{P} \le \mathtt{N2}\}$ with
    $e_1(k) = \sum_{\mathtt{J1}=\frac{\mathtt{L}}{\mathtt{N1}}}^{\frac{\mathtt{U}}{\mathtt{N1}}} \mathtt{J1}*\mathtt{N1} - \mathtt{L} = \frac{(\mathtt{N1}+\mathtt{U}-\mathtt{L})*(\mathtt{L}-2*\mathtt{N1}+2*\mathtt{L}*\mathtt{N1}+\mathtt{U})}{2*\mathtt{N1}^2}$.
  - $c_2 = \{\frac{\mathtt{U}}{\mathtt{N1}} > \mathtt{N1},\ \frac{\mathtt{L}}{\mathtt{N1}} \le \mathtt{N1}\}$ with $e_2(k) = \sum_{\mathtt{J1}=\frac{\mathtt{L}}{\mathtt{N1}}}^{\mathtt{N1}} \mathtt{J1}*\mathtt{N1} - \mathtt{L} = (\mathtt{N1} - \frac{\mathtt{L}}{\mathtt{N1}} + 1)*$ $(\frac{\mathtt{N1}^2}{2} - \frac{\mathtt{L}}{2} + 1)$.
- $I_2 = \{1 \le \mathtt{J1} \le \mathtt{N1},\ \mathtt{J1}*\mathtt{N1} > \mathtt{U},\ \mathtt{L} \le \mathtt{U}\} = \{1 \le \mathtt{J1} \le \mathtt{N1},\ \mathtt{J1} \ge \frac{\mathtt{U}+1}{\mathtt{N1}},\ \mathtt{L} \le \mathtt{U}\}$ with $\sum_{J2=\mathtt{L}}^{\mathtt{U}} 1 = \frac{N2}{P}$.

  The algorithm detects in statement 1 that $1 \le \mathtt{J1}$ is made redundant by $\frac{\mathtt{U}+1}{\mathtt{N1}} \le \mathtt{J1}$. Therefore, we can eliminate J1 for $c_3 = \{\mathtt{N2} \ge \mathtt{P},\ \mathtt{N1}^2 \ge \mathtt{U}+1\}$ which yields $e_3(k) = \sum_{\mathtt{J1}=\frac{\mathtt{U}+1}{\mathtt{N1}}}^{\mathtt{N1}} \frac{\mathtt{N2}}{\mathtt{P}} = \frac{\mathtt{N2}}{\mathtt{P}} * (\mathtt{N1} - \frac{\mathtt{U}+1}{\mathtt{N1}} + 1)$.

  Finally, statement l_4 in the example code is approximately executed

  $$\overline{work(k)} = \sum_{1 \le i \le 3} \varrho(c_i) * e_i(k)$$

times by a specific processor k ($1 \le k \le \mathtt{P}$) for the parameters N1, N2 and P. $\varrho(c_i)$ is defined by (5.9). Note that most conventional performance estimators must repeat the entire performance analysis whenever the problem size or the number of processors change. However, our symbolic performance analysis provides the solution of the above problem as a symbolic expression of the program unknowns (P,N1,N2, and k). For each change in the value of any program unknown we simply re-evaluate the result instead of repeating the

Table 5.3. Measured versus estimated values for the amount of work to be done by all processors for $\mathbf{P = 4}$

N1	N2	$\sum_{1 \le k \le P} work(k)$	$\sum_{1 \le k \le P} \overline{work(k)}$	error in %
100	100	10000	10025	0.25
200	100	20000	20238	1.19
200	200	40000	40450	1.12
400	200	80000	80474	0.59
400	400	160000	160900	0.56
800	400	320000	320950	0.29
800	800	640000	641800	0.28

entire performance analysis. Table 5.3 shows an experiment which compares measured against estimated values for version of the example code with 4 processors (P = 4) by varying the values for N1 and N2. The measurements have been done by executing the example code on an iPSC/860 hypercube system. For each processor we enumerate how often it writes array A in the loop of the example code. It can be seen that the estimates ($\overline{work(k)}$) are very close to the measurements ($work(k)$). In the worst case the estimates are off by 1.19 % for a relative small problem size (N1 = 200, N2 = 100). For larger problem sizes the estimation accuracy consistently improves.

5.6.3 Optimizing FTRVMT

In this section we study the potential performance benefits as implied by our advanced symbolic analysis to FTRVMT which is the core subroutine of the OCEAN perfect benchmark which simulates a 2-dimensional fluid flow – [12]. FTRVMT accounts for close to 50 % of the overall execution time of OCEAN. FTRVMT contains non-linear array subscript expressions and 5 DO-loop nests which iterate over a one-dimensional array DATA. We created two parallel program versions for varying problem sizes (ranging from 16 to 56) in the OCEAN benchmark:

Firstly, a version (solid line curve in Fig. 5.3) as created by *VFC* without symbolic analysis. A true dependence that affects array DATA for all 5 loops of FTRVMT is assumed which prevents communication optimization.

Secondly, the *VFC* code was hand-instrumented (dashed line curve in Fig. 5.3) to reflect the communication optimizations that are enabled through a symbolic dependence test. Non-linear symbolic expressions are compared by a symbolic dependence test which is based on algorithm *expr_bound* (see Section 5.3.1). For the first 3 loops of FTRVMT our algorithm can be used to detect the absence of a true dependence on DATA which enables communication vectorization and aggregation. For both program versions we applied induction variable substitution [11] and block-distributed array DATA.

Figure 5.3 shows the execution times of both versions for varying problem sizes. Clearly, the version based on a symbolic dependence test outperforms the original *VFC* code. Our techniques for comparing symbolic expressions

allows us to hoist communication out of loop nests. The current *VFC* without symbolic analysis prevents communication from being hoisted to an outer loop which drastically increases the number of network messages. Although our symbolic analysis detects the absence of a true dependence with respect to array DATA for only three loop nests, it makes a significant difference in the overall execution time ranging by a factor of roughly 2–3 for the problem sizes measured and consistently improves for increasing problem sizes.

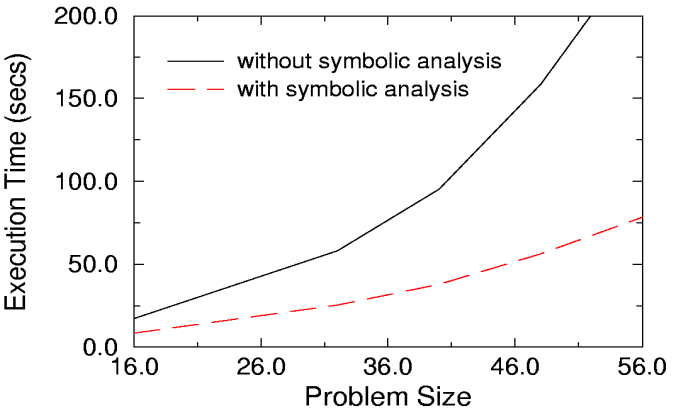

Fig. 5.3. Measured execution times of FTRVMT with and without symbolic analysis for varying problem sizes on a MEIKO CS-2 with 16 processors

5.7 Summary

Numerous researchers have shown the importance of symbolic analysis [67, 20, 84, 71, 18, 98, 100, 44, 102, 29] for optimizing and analyzing performance of parallel programs. A crucial problem of many compiler analyses is to determine the relationship between symbolic expressions. We have described a novel algorithm for computing lower and upper bounds of wide classes of linear and non-linear symbolic expressions based on a set of constraints defined over loop variables and loop invariants. This algorithm supports comparing symbolic expressions for equality and inequality relationships, simplifying systems of constraints, examining non-linear array subscript expressions for data dependences, and optimizing communication [49, 52]. Furthermore, we presented a symbolic sum algorithm that estimates the number of integer solutions to a system of constraints. This algorithm supports detection of zero-trip loops, elimination of dead code, and performance prediction of parallel programs.

Previous methods are based on algorithms that are either restricted to a smaller class of symbolic expressions and constraints, are more expensive in

terms of complexity, or produce less accurate and more complex (due to the lack of aggressive simplification techniques) results.

We have implemented all techniques described in this chapter as part of a parallelizing compiler and a performance estimation tool. Preliminary results demonstrate the effectiveness of our techniques by proving the absence of a data dependence in a loop with non-linear array references that enables the compiler to parallelize this loop. We also showed how to use our algorithm to detect redundant inequalities in a set of non-linear constraints which as a result significantly reduces the complexity of other important compiler and performance analyses. Furthermore, it has been demonstrated that our symbolic sum algorithm can estimate the work distribution of a parallel program with high accuracy. Finally, we applied our techniques to a code taken from the perfect benchmarks which significantly reduced communication costs and overall execution time.

The techniques described in this chapter are used in our symbolic evaluation framework described in Chapter 3 to simplify symbolic constraints and to compare symbolic expressions for a large variety of program analyses covering both parallel and sequential architectures.

6. Symbolic Analysis for Parallelizing Compilers

6.1 Introduction

High performance computers (HPCs) are of paramount importance in science and engineering. They are widely used in large-scale applications that simulate complex processes. Examples of such applications include weather modeling and forecasting, crash-tests, optimizing combustion engines, aerodynamic behavior of airplanes, study of chemical phenomena, and financial modeling [8, 39, 72, 13, 38]. Currently, HPCs offer a peak performance of several hundred TFlops and memory capacities of several TBytes. Even new systems achieving PFlops are in development. The two major sources contributing to the power of HPCs are (1) powerful hardware technology and (2) parallelism. Due to physical limits of hardware technology, parallelism plays a more important role in meeting the ever increasing demand of computational power.

Parallel computers can be characterized by their control mechanism, memory organization, and interconnection network. Flynn[54] has introduced three categories which include SISD, SIMD, and MIMD architectures. Conventional computer systems have a *single instruction stream and single data stream* (SISD). Computers with single control unit and multiple processing units are referred to as *single instruction stream and multiple data stream* (SIMD) computers. Typical representatives of SIMD parallel computers are Illiac IV, CM-2, MasPar MP-1, and MasPar MP-2. Computers with multiple processing nodes and the ability to execute different programs are called *multiple instruction stream, multiple data stream* (MIMD) computers. MIMD computers such as SGI/Cray Origin 2000, SGI/Cray T3E, Fujitsu AP3000, Hitachi SR8000 system, NEC SX-5 have a substantial share on the market. Parallel computers are also characterized by their memory organization such as *shared memory*, *distributed memory*, *virtual shared memory* and *cluster of shared memory* architectures. In a shared memory architecture, all nodes access the same memory. Every node in a distributed-memory computer owns a private *local memory*. These nodes communicate to exchange non-local data items which is usually done by *message passing*. Virtual shared-memory systems feature a physically distributed-memory with operating system support to exchange data, i.e. no explicit message passing is required. Virtual shared-memory systems can be further classified by their memory access behavior.

T. Fahringer and B. Scholz: Advanced Symbolic Analysis, LNCS 2628, pp. 75–97, 2003.

For *uniform memory access*(UMA) machines the access time to non-local and local data does not vary – for *non-uniform memory access* (NUMA) machines access time may vary. SMP (symmetric multiprocessing) cluster architectures consist of SMP units with several CPUs that share a common memory. Communication is required to exchange non-local data from one shared-memory unit to another. SMP clusters are currently some of the most promising architectures. In this book we focus on parallel high performance computers with a MIMD architecture and a distributed-memory system which are often referred to as *distributed memory multiprocessing system* (DMMP).

Programs on HPCs can only exploit the provided computational power by taking advantage of two kinds of parallelism: (1) *data parallelism* and (2) *task parallelism*. Data parallelism is characterized by distributing data onto processing nodes and all processing nodes perform identical computations on its local data. In contrast, task parallelism allows the execution of different instruction streams in parallel. It is important to note that both kinds of parallelism may occur in an application. *Parallel programming models* have been evolved in order to program parallel computers. In general, two models can be distinguished: (1) *implicit parallel programming* and (2) *explicit parallel programming*. Explicit parallel programming takes the machine model into account whereas implicit parallel programming provides a high-level view of parallelism for the programming and delegates parallelization on the compiler. A representative for explicit parallel programming are message-passing libraries for conventional programming languages such as PVM[99] and MPI[87]. Although this approach implies high costs in terms of software development and maintenance, it often yields the best computational performance of an application. Examples of implicit parallel programming are given by *data parallel programming languages* such as HPF[74], Fortran D [55], and Vienna Fortran[108]. Data parallel programs are translated into a *single program, multiple data* (SPMD) code. In that model each processing node executes the same (parameterized) code in a *loosely synchronized* fashion. Synchronization and exchange of data is implicitly generated by the compiler. In this book we develop analysis and transformations techniques based on symbolic analysis (see Chapter 3) in the context of the Vienna High Performance Compiler (VFC) [10].

6.2 Programming DMMPs

Although DMMPs have an appealing architecture since they are scalable in size, severe difficulties are to be expected without parallelizing compilers. The message passing code of DMMPs has a high potential of errors and is inflexible. It takes considerable time to adapt existing sequential codes for DMMPs, and it is questionable that new applications developed for a specific DMMP can easily migrate to another parallel architecture.

Fig. 6.2.1 shows an excerpt of a sequential Fortran program. It consists of a simple loop that computes new values for elements of array `A`. As described in Chapter 3, all statements are annotated with their program contexts. Let us assume that the values of `N` and `M` are unknown at compile time and we only know that they are greater than or equal to two which is reflected in the path condition p_0.

Example 6.2.1 *Sequential Fortran program*

real::*A(ASIZE)*
$[s_0 = \{N = n, M = m, MIN = \bot, MAX = \bot, A = \bot_{ASIZE}\},$
$t_0 = \text{true}, p_0 = n \geq 2 \wedge m \geq 2]$
ℓ_1 : ***if*** *(N > M)* ***then***
$[s_1 = s_0, t_1 = t_0, p_1 = p_0 \wedge n > m]$
ℓ_2 : $MIN = M$
$[s_2 = \delta(s_1; MIN = m, t_2 = t_1, p_2 = p_1]$
ℓ_3 : $MAX = N$
$[s_3 = \delta(s_2; MAX = n, t_3 = t_2, p_3 = p_2]$
ℓ_4 : ***else***
$[s_4 = s_0, t_4 = t_0, p_4 = p_0 \wedge n \leq m]$
ℓ_5 : $MIN = N$
$[s_5 = \delta(s_4; MIN = n, t_5 = t_4, p_5 = p_4]$
ℓ_6 : $MAX = M$
$[s_6 = \delta(s_5; MAX = n, t_6 = t_5, p_6 = p_5]$
ℓ_7 : ***end if***
$[s_7 = \delta(s_0; MIN = x, MAX = y),$
$t_7 = t_0 \wedge \gamma(n > m; x = m, y = n; x = n, y = m), p_7 = p_0]$
$[s'_7 = \delta(s_7; I = 2), t'_7 = t_7, p'_7 = p_7]$
ℓ_8 : ***do*** *I=2,N-1*
$[s_8 = \delta(s'_7; I = \underline{I}, A = \underline{A}), t_8 = t_7, p_8 = p'_7 \wedge \underline{I} \leq n - 1]$
ℓ_9 : *A(N-I)=A(M-I+MAX)- A(N-I+MIN)-A(N-I+1)*
$[s_9 = \delta(s_8; A = \underline{A} \oplus (\rho(\underline{A}, m - \underline{I} + y) - \rho(\underline{A}, n - \underline{I} + y) -$
$\rho(\underline{A}, n - \underline{I} + 1), n - \underline{I})), t_9 = t_8, p_9 = p_8]$
$[s'_9 = \delta(s'_9; I = \underline{I} + 1, t'_9 = t_9, p'_9 = p_9]$
ℓ_{10} : ***end do***
$[s_{10} = \delta(s_8; A = \mu(A, s'_7, [s'_9, t'_9, p'_9]), I = n), t_{10} = t'_7, p_{10} = p'_7]$

When this code is parallelized by hand, it is left to the programmer to distribute the workload and data among the processors. Numerical codes feature a certain regularity and it makes sense to employ the SPMD model for implementing the parallel code. With this approach, arrays are partitioned and mapped to the processors [110]. A processor *owns* data assigned to it and these data items are stored in its local memory. After assigning data items to processors, the work is distributed according to the data distribution. A processor which ,,owns" a data item is responsible for its computation. This is known as the *owner computes* paradigm whereby all processors execute the same (parameterized) code in parallel. Non-local data accesses must be

explicitly handled by the programmer who has to insert communication statements, i.e. *send* and *receive* operations to transfer data.

To insert communication statements, the programmer must be aware of deadlocks and live-locks. Consequently, the resulting message passing code may become quite complex which can be a serious problem for debugging. Another key problem of manually hand-coding message passing code is that data distributions must be hard-coded which results in a hard-coded work distribution as well. Therefore, the code is inflexible and it can be a very time consuming task to change the data and work distribution which is required to optimize the performance of the program.

To illustrate hand-coded message passing code of the program in Ex. 6.2.1 we have inserted and modified the code to run it on a set of `NP` processors which is shown in Ex. 6.2.2. We have simplified matters by assuming that array A is *block-wise distributed*[110] in `NP` contiguous blocks and `K` is a multiple of ASIZE. Variable `K` gives the processor number and the block size is a constant for all processors which is computed by `BS=ASIZE/K`. The lower bounds of distributed blocks are stored in array `LB(K)=1+(K-1)*BS` for each processor.

Note that the parallelized code is inefficient since the whole iteration space of the loop is executed by all processors, and inside the loop it is examined whether array element `A(N-I)` is owned by the executing processor `K`. A more optimized code has to remove the `if`-statement inside the loop by restructuring the iteration space to the data elements owned for every processor.

Moreover, the array indices of array assignment in ℓ_{15} are shifted by `-L(K)+1` in order to map the original array indices to local array indices. Array accesses on the right-hand side may induce non-local accesses. We avoided explicit communication for each array access since after a thorough investigation we can deduce that there is never a write before a read situation for array `A`. If we know that there is no data dependence[109] which prohibits the parallelization of the loop, we can hoist and aggregate communication outside of the loop. Data which is not owned by the processor, but is needed for computation is stored in an extra space known as *overlap area*. Before entering the loop non-local data (which is needed for computation) is received from the owning processor of the non-local data. This is reflected in the statement ℓ_{12} where non-local data is received and stored into the overlap area between `BS(K)+1` and `BS(K)+1+OVERLAP` of the distributed array `A`. Before receiving non-local data the processor must send its own data to the processor which needs this data in its own overlap area (see statement ℓ_{11}).

As shown in the example, manually parallelizing a program can be a non-trivial task since upper and lower bounds of blocks have to be computed, overlap areas must be considered, and special masks as employed in statement ℓ_{14} have to be inserted in order to obtain a semantically correct parallel program. Moreover, without knowing that the loop can be parallelized, local

Example 6.2.2 *Manually parallelized Fortran code*

```
      /* Processor structure PROC(K) is assumed. Lower block bounds and block */
      /* sizes of distributed array A are stored in L(K) and BS(K), respectively. */
      real,allocatable::A(:)
      /* Determine values of M,N and the overlap area */
ℓ1    if (N > M) then
ℓ2       MIN = M
ℓ3       MAX = N
ℓ4       OVERLAP = M
ℓ5    else
ℓ6       MIN = N
ℓ7       MAX = M
ℓ8       OVERLAP = 2*M-N
ℓ9    end if
      /* Allocate array with overlap area */
ℓ10   allocate A(BS + OVERLAP)
      /* Send data to other processors */
ℓ11   if (K > 1) SEND(A(1:OVERLAP)) TO PROC(K-1)
      /* Receive data from other processors */
ℓ12   if (K > 1) RECEIVE(A(BS+1:BS+1+OVERLAP) TO PROC(K-1)
      /* distributed loop */
ℓ13   do I=2,N-1
ℓ14      if owned(A(N-I)) /* Mask computation */
ℓ15         A(N-I-L(K)+1)=A(M-I+MAX-L(K)+1)-
            A(N-I+MIN-L(K)+1)-A(N-I+1-L(K)+1)
ℓ17   end do
```

data has to be exchanged inside the loop that would result in a poorly performing parallel program. Therefore, the programmer must be fully aware of the program semantics for efficiently parallelizing the sequential code.

6.3 Data Dependence Analysis

The notion of dependence captures the most important property of a program for efficient execution on parallel computers. The dependence structure of a program provides sufficient information to exploit the available parallelism. Although powerful dependence analysis techniques have been employed in commercial and experimental systems[61, 7, 109, 104], all these approaches suffer by a lack of accurate dependence analysis information in the presence of complex symbolic expressions. Therefore, symbolic analysis is a necessary prerequisite to overcome the deficiencies of classical techniques.

Sequential programs imply a linear order of statement execution. In most cases this order is too restrictive. A parallelizing compiler tries to find a new execution order for the program to exploit inherent parallelism. This new execution order must preserve program semantics, and a parallelizing com-

piler commonly invokes complex analyses [109, 7] for finding relations that characterize the semantic constraints on the execution order of a program. These *data dependence relations* are deduced by read/write statements that modify and access memory locations and the control flow of the sequential program. In a program, a statement can be executed more than once. Each execution of a statement is a *statement instance*. In a sequential program we refer to a statement ℓ_B as *flow dependent* on a statement ℓ_A if in some execution of the program, an instance of ℓ_B could read from a memory location which is previously written to by an instance of ℓ_A. We say that a statement ℓ_B is *anti-dependent* on statement ℓ_A if an instance of ℓ_B could rewrite to a memory location which was previously read by ℓ_A. Similarly, statement ℓ_B is *output dependent* on statement ℓ_A if an instance of ℓ_B could rewrite a previously written statement of ℓ_A.

The existence of data dependences prevents the parallel execution of statements ℓ_A and ℓ_B and constrains the execution order for parallel programs to preserve program semantics. Note that anti- and output-dependences only exist in imperative program languages which permit the programmer to reuse the same memory locations via accesses to variables. Therefore, this dependences are also known as *memory-related dependences*. In theory anti- and output- dependences can be eliminated by introducing new variables for different instances. One of the objective of this book is to capture data dependences for programs. If there are no flow dependences we can parallelize loops and can optimize communication.

In a straight line code it is easy to determine data dependences for scalar variables. A more difficult task is to state data dependences in more complex code, e.g. for array assignment ℓ_9 in Fig. 6.2.1. If we prove that the index expression on the left-hand side (`N-I`) is less or equal than the index expressions on the right-hand side (`N-I+MIN`, `M-I+MAX`, and `N-I+1`), we can parallelize the code since there is no write before read situation for array elements of `A`. In principle we have to show the following relations,

$$\begin{aligned}
&\texttt{N-I} \leq \texttt{N} - \texttt{I} + \texttt{1} \\
&\texttt{N-I} \leq \texttt{N} - \texttt{I} + \texttt{MIN} \\
&\texttt{N-I} \leq \texttt{M} - \texttt{I} + \texttt{MAX}
\end{aligned}$$

If we subtract the left-hand side from the right-hand side and reverse the relation, we obtain the following result,

$$\begin{aligned}
\texttt{1} &\geq 0 \\
\texttt{MIN} &\geq 0 \\
\texttt{M+MAX-N} &\geq 0
\end{aligned}$$

The first relation $1 \geq 0$ is a trivial one which is true for all possible bindings of `N` and `I`. Therefore, no flow dependent data dependence between left-hand side `A(N-I)` and right-hand side `A(N-I+1)` exists which would prohibit parallelization. Contrary, the last two dependence relations cannot be resolved

easily since there is no local information about variables `MIN` and `MAX`. For the example in Fig. 6.2.1 classical data dependence tests [61, 7, 109, 104] would fail since there is no information available about `MIN` and `MAX`. Without knowing the symbolic values of `MIN` and `MAX` we are unable to automatically parallelize the code of Ex. 6.2.2. Therefore, we apply our symbolic analysis techniques as follows:

- For the first data dependence relation `N-I`$\leq$`N-I+MIN` we symbolically evaluate the difference between both expressions and examine whether or not the difference is positive.

$$\begin{aligned} eval((N-I+MIN)-(N-I),[s_9,t_9,p_9]) &= eval(MIN,[s_9,t_9,p_9]) \\ &= x \end{aligned}$$

  The difference is given by symbol x which is bound in $\gamma(n > m; x = m; x = n)$. Moreover, path condition p_9 constrains symbols m and n by a value that is greater than or equal to two. Based on p_9 we can conclude that $x \geq 0$ and consequently there is no data dependence which would prohibit parallelization.
- For the second data dependence relation `N-I`$\leq$`M-I+MAX` we symbolically subtract the left-hand side from the right-hand side and obtain the following result,

$$\begin{aligned} eval((M\text{–}I\text{+}MAX)\text{–}(N\text{–}I),[s_9,t_9,p_9]) &= eval(M\text{+}MAX\text{–}N,[s_9,t_9,p_9]) \\ &= m+y-n \end{aligned}$$

  where symbol y is bound in $\gamma(n > m; y = n; y = m)$ according to t_9 of Ex. 6.2.1. Consequently, if $n > m$, then $m - n + y = m$; otherwise $n \leq m$ and $m - n + y = 2 * m - n$. We can deduce that $m \geq 0$ and $2 * m - n \geq 0$ based on p_9.

As shown in the example, symbolic analysis can be of paramount importance to explore data dependences in programs. Without symbolic analysis, classical data dependence analyses are too pessimistic and commonly fail to give precise answers.

6.4 Vienna High Performance Compiler

To overcome some disadvantages of hand-coded message passing code, high-level approaches have been developed. Namely, data parallel languages such as High Performance Fortran (HPF) [74] require the user to specify the distribution of the program's data to a set of available processors. The programmer does not need to face technical issues such as inserting message passing code, determining overlap areas, masking statements for the owner computes paradigm and other cunning problems. Data distribution information is used

solely by the compiler to guide the process of restructuring the code into a SPMD program. Since the distribution of data is crucial for the performance of the resulting parallel program, HPF provides a variety of methods to distribute data and changing data distributions in an HPF program.

HPF defines a distribution as a mapping of elements of an array to a set of processors, including total as well as partial replication of data. E.g., Ex. 6.4.2(a) shows the original program of Ex. 6.2.1 written as an HPF-program which distributes array `A` clockwise onto `NP` processors. The first directive determines the data distribution of A. The second directive organizes the processors [74, 10] as processor array `PROC`. The rest of the program is identical to the original one and no additional work is required to transform the sequential code to a parallel program. HPF statements are written as comments and, therefore, it is possible to compile HPF program with a sequential Fortran compiler. One code base is sufficient for the sequential Fortran and parallel HPF program.

The work distribution for this code under HPF is based on the owner computes rule which means that the processor that owns a datum will perform the computations that make an assignment to this datum. Non-local data referenced by a processor are buffered in overlap areas that extend the memory allocated for the local segment (owned by a processor according to the data distribution strategy). For many regular computations, the precise pattern of non-local accesses can be computed; this information can then be used both to determine the storage requirements for the array and to optimize communication. The updating of the overlap areas is automatically organized by the compiler via explicit message passing. Updating overlap areas is the result of a series of optimization steps [10].

Let A be an array that is distributed onto a set of processors Υ, $\zeta^A(\upsilon)$ is the set of array elements owned by $\upsilon \in \Upsilon$, and $\eta^A(\upsilon)$ the set of all non-local elements (owned by processors in Υ other than υ) of A accessed in υ. Then for each distributed array A and processor $\upsilon \in \Upsilon$, $\zeta^A(\upsilon)$ is allocated in the local memory associated with υ. In a parallel program, communication is inserted each time an element of $\eta^A(\upsilon)$ is referenced, and private variables are created to hold copies of the original non-local values. Overlap analysis is used to allocate memory space for these non-local values in locations adjacent to the local segment.

Let A be a distributed array and $\upsilon \in \Upsilon$, then the **overlap area**, *OA(A,υ)*, is defined as the smallest rectilinear contiguous area around the local segment of a processor υ, containing all non-local variables accessed. The union of the local segment and the overlap area of array A with respect to processor υ is called the extension segment, *ES*(A,υ).

To illustrate these concepts, consider the HPF code excerpt in Ex. 6.4.1. If we distribute arrays `U`, `UNEW` and `F` block-wise to a 2×2 processor array `PROC`, then the array index mapping for arrays `UNEW` and `F` is equal to array `U`. In

Example 6.4.1 *HPF JACOBI relaxation code*

```
      REAL U(100,100),UNEW(100,100),F(100,100)
      !hpf$ processors :: PROC(2,2)
      !hpf$ distribute (block,block) :: UNEW, U, F
      INIT(U, F, 100)
      ...
      do J = 2, 99
         do J = 2, 99
l:         UNEW(I,J) = 0.25 * (F(I,J)+U(I-1,J)+U(I+1,J)+U(I,J-1)+U(I,J+1))
         end do
      end do
```

Table 6.1 we illustrate the corresponding local segments for every processor in PROC.

The overlap area for array A is specified by its **overlap description**, *OD(A)*, which is determined by the maximum offsets for every dimension of the local segments, over the set of all processors to which A is distributed. If n is the dimension of A, this takes the form

$$OD(A)=[dl_1 : du_1, \ldots , dl_n : du_n]$$

where dl_i and du_i denote the offsets with respect to the lower and upper bound of dimension i. If dl_i or du_i is equal to a $<$ or $>$ which is known as *total overlap*, then the offset extends the overlap area to the array segment boundaries in the lower or upper bound of dimension i, respectively.

Finally, consider a statement l with a read access *ref* to array A in l. The overlap description of l with respect to *ref*, *OD(l,ref)*, is defined as the contribution of l and *ref* to *OD(A)*.

The overlap description can be used to significantly improve the organization of communication; it facilitates memory allocation and the local addressing of arrays. The relevant analysis is described in detail in [57, 58, 59, 60].

Table 6.1. Local segments of processors in PROC

processor	$\zeta^U(v)$
PROC(1,1)	U(1:50,1:50)
PROC(2,1)	U(51:100,1:50)
PROC(1,2)	U(1:50,51:100)
PROC(2,2)	U(51:100,51:100)

For illustration of the overlap techniques we continue Ex. 6.4.1 by analyzing statement l of the JACOBI relaxation code. Based on the data dis-

tribution strategy of this example the overlap descriptions associated with the different elements in *USE(l)* are computed as shown in Fig. 6.1. Thus *OD*(F)=[0:0,0:0], and *OD*(U)=[1:1,1:1].

The segment of U on PROC(1,2) is U(1:8,9:16). Its extension segment is given by *ES*(A,PROC(1,2))= U(0:9,8:17), and the overlap area is U(0,9:16) ∪ U(9,9:16) ∪ U(1:8,8) ∪ U(1:8,17). It consists of an area of depth 1 around the local segment.

Note that the overlap area may contain variables which do not correspond to elements of the original arrays. They will never be accessed, as they are not related to non-local uses.

OD(*l*,F(I,J))=[0:0,0:0]
OD(*l*,U(I-1,J))=[1:0,0:0]
OD(*l*,U(I+1,J))=[0:1,0:0]
OD(*l*,U(I,J-1))=[0:0,1:0]
OD(*l*,U(I,J+1))=[0:0,0:1]

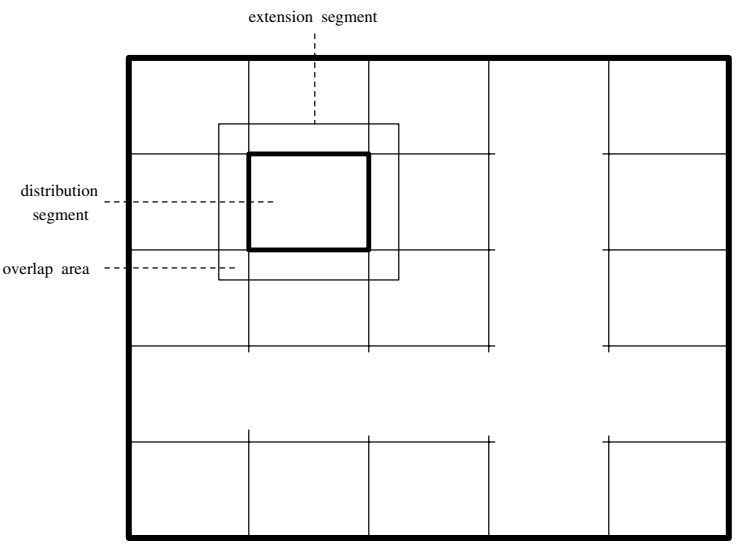

Fig. 6.1. Overlap area of array U in the JACOBI code

In the following we describe the code generation of VFC system [10], which is a source-to-source translator that translates an HPF program into a Fortran 90 program. VFC optimizes communication[50] by extracting single element messages from a loop and combining them to vector-form (*communication vectorization*), removing redundant communication (*communication*

coalescing), and aggregating different communication statements (*communication!aggregation*). VFC generates a communication (EXSR) statement for each right-hand side array reference in assignment statements. An `EXSR` [10] statement is syntactically described as $EXSR\ A(I_1, ..., I_n)\ [l_1/u_1, ..., l_n/u_n]$, where $v = A(I_1, ..., I_n)$ is the array element inducing communication and $[l_1/u_1, ..., l_n/u_n]$ is the overlap area for array A. For each i, the expressions l_i and u_i, respectively, specify the left and right extension of dimension i in the local segment. The dynamic behavior of an *EXSR* statement can be described as follows:

```
IF (executing processor p owns v) THEN
   send v to all processors p', such that
   (1) p' reads v and
   (2) p' does not own v
ELSEIF (v is in the overlap area of p) THEN
   receive v
ENDIF
```

After analyzing the data dependences of the program, VFC determines the size of overlap areas. E.g., the overlap area of the array access `A(N-I+1)` is simply one element since the difference of `(N-I+1) - (N-I)` is one. Without symbolic analysis VFC cannot vectorize large portions of the communication inside of the loop which is reflected in Ex. 6.4.2(b) since it is unable to determine whether `A(M-I+MAX)` and `A(N-I+MIN)` are read before they are written by the assignment to `A(N-I)` as described in Section 6.3.

E.g., the statements `EXSR A(N-I+MIN) [</>]` in Ex. 6.4.2(b) imply that the processor which owns `A(N-I+MIN)` sends its element to the processor which owns `A(N-I)`. Statement `EXSR A(M-I+MAX) [</>]` behaves similarly. Both statements cause a large communication overhead, as they are executed inside a loop. For the array access `A(N-I+1)` on the right-hand side we know that there is an artificial data dependence, which means that we can vectorize communication. This is reflected by the statement `EXSR A(:)[0/1]`.

As deduced in Section 6.3 we know that the symbolic differences of the array indices are positive and, therefore, no flow dependent data dependence prohibits the parallelization of the code. The overlap areas u_1 and u_2 of array accesses `A(N-I+MIN)` and `A(M-I+MAX)` are given as,

$$u_1 = x$$
$$u_2 = m + y - n$$

where symbols x and y are bound in $\gamma(n > m; x = m, y = n; x = n, y = m)$. After applying simplification techniques as described in Chapter 3 we obtain overlap area $U_1 = u_1$ with γ(n>m;u_1=m;u_1=n) and overlap area $U_2 = u_2$ with γ(n>m;u_2=m;u_2=2*m-n). This yields the following communication statements which are placed immediately outside the loop (not shown in Ex. 6.4.2 (c)) in the first step of communication optimization (vectorization):

- ℓ_1 : EXSR A(:) [0/1]
- ℓ_2 : EXSR A(:) [0/U_1]
- ℓ_3 : EXSR A(:) [0/U_2]

In order to eliminate redundant communication by communication coalescing [50] we have to verify whether the data transferred by one communication statement is already implied by another. By comparing the lower and upper bounds [47] of the associated overlap areas our framework can determine that ℓ_1 and ℓ_2 are made redundant by ℓ_3 since u_2 is greater than one and greater than u_2.

Ex. 6.4.2(c) shows the final code as generated by VFC with symbolic analysis. The computation of the overlap area was placed inside of the first if-statement. Depending on the values of n and m, the appropriate value of U_2 is calculated. In the code fragment U_2 was replaced by U because there is only one overlap area left. The resulting code implies 2 different communication patterns depending on whether or not $n > m$ which is resolved during runtime.

Table 6.2 tabulates and Fig. 6.2 visualizes the execution times for the codes shown in Ex. 6.4.2(b) (VFC w/o symbolic analysis) – and Ex. 6.4.2(c) (VFC with symbolic analysis) on a Meiko CS-2 machine for various problem sizes and a fixed machine size (8 processors). The latter code has been created by VFC and hand-instrumented to reflect communication vectorization and coalescing that can be performed based on symbolic analysis. Our framework automatically applies symbolic analysis to the code of Ex. 6.4.2(c) which includes automatic generation of contexts and invokes the *eval* functions at the array assignment inside the loop. Moreover, our framework compares the lower and upper bounds of the overlap areas which is necessary to determine whether communication vectorization and coalescing is legal as described above.

Table 6.2. Meiko CS-2 (8 processors) execution times for codes of Ex. 6.4.2(b) and 6.4.2(c)

		Execution Time (secs)			
M	N	VFC (w/o SA)	VFC (with SA)	Improvement (factor)	Overlap Size
256	256	0.11	0.00423	26	256
	512	0.22	0.00461	47	256
	768	0.31	0.00507	62	256
512	256	0.10	0.02210	4	768
	512	0.21	0.01051	20	512
	768	0.30	0.01255	23	512
768	256	0.11	0.03229	3	1280
	512	0.22	0.02918	7	1024
	768	0.33	0.02212	15	768

Example 6.4.2 *VFC source code translation*

```
      !hpf$ processors :: PROC(NP)
      !hpf$ distribute (block) :: A
l1:   if (N > M) then
l2:      MIN = M
l3:      MAX = N
l4:   else
l5:      MIN = N
l6:      MAX = M
l7:   end if
l8:   do I = 2, N − 1
l9:      A(N−I) = A(M− I+MAX)−
l10:        A(N−I+MIN)−A(N−I+1)
l11:  end do
```

(a) Source code

```
l1:   if (N > M) then
l2:      MIN = M
l3:      MAX = N
l4:   else
l5:      MIN = N
l6:      MAX = M
l7:   end if
l8:   EXSRA(:)[0/1]
l9:   do I = 2, N − 1
l10:     EXSR A(N-I+MIN) [< / >]
l11:     EXSR A(M-I+MAX) [< / >]
l12:     A(N−I) = A(M− I+MAX)−
l13:        A(N−I+MIN)−A(N−I+1)
l14:  end do
```

(b) Unoptimized code

```
l1:   if (N > M) then
l2:      MIN = M
l3:      MAX = N
l4:      U = M
l5:   else
l6:      MIN = N
l7:      MAX = M
l8:      U = 2 ∗ M − N
l9:   end if
l10:  EXSR A(:) [0/U]
l11:  do I = 2, N − 1
l12:     A(N−I) = A(M− I+MAX)−
l13:        A(N−I+MIN)−A(N−I+1)
l14:  end do
```

(c) Optimized code

The resulting improvement (execution time of column 3 in Table 6.2 divided by the execution time of column 4) that is gained by using symbolic analysis varies from 3 to 62 and is shown in column 5. N significantly impacts the performance for the codes produced by VFC w/o symbolic analysis as the loop iteration count is directly proportional to N and large portions of

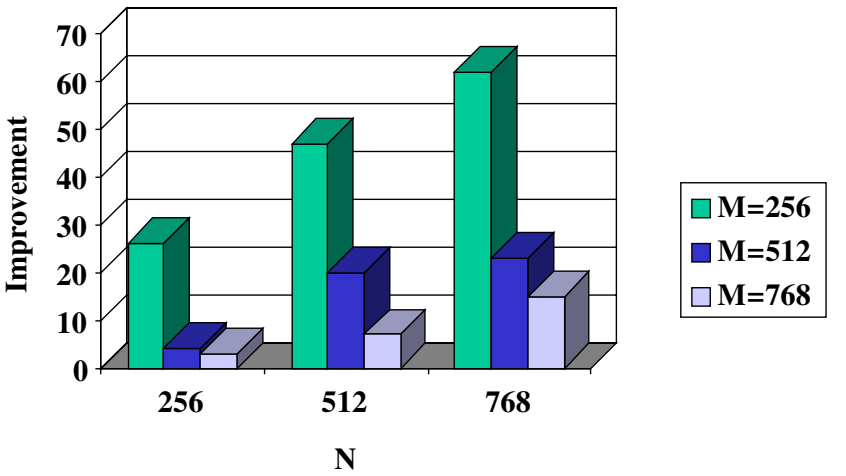

Fig. 6.2. Execution time improvement (as factor) for the codes of Ex. 6.4.2(b) and 6.4.2(c)

the communication remain inside the loop. This is also the reason why VFC w/o symbolic analysis does not achieve any speedup for this code. For the code versions that are based on symbolic analysis, it is the size of the overlap area (last column) depicted in Fig. 6.3, that dominates the performance. For instance, the longest execution time has been measured for $M = 768$ and $N = 256$ which also implies the largest overlap area $2 * 768 - 256 = 1280$. The speedup figures range from 1.5 - 7 depending on the size of the overlap area. Note that the overlap column (Overlap Size) of Table 6.2 reflects only the overlap for EXSR statement in Ex. 6.4.2(c). The computation time as part of the total execution time for the codes that are optimized based on symbolic analysis varies between 1 - 70 %. The smaller the overlap area the larger the percentage of computation time as part of the total execution time. Communication takes by far the largest portion (close to 99 %) of the total execution time for the VFC codes generated without symbolic analysis.

6.5 Implementation

We have implemented a prototype of our symbolic analysis framework which is used as part of the *VFC*, a parallelizing compiler, and P^3T, a performance estimator for parallel programs. Our system for manipulating symbolic expressions and constraints, as well as our techniques for simplifying expressions and constraints, comparing symbolic expressions and computing the number of solutions to a system of constraints have been fully implemented. The current implementation of our recurrence solver handles recurrences of the following kind: linear recurrence variables (incremented inside a loop by a

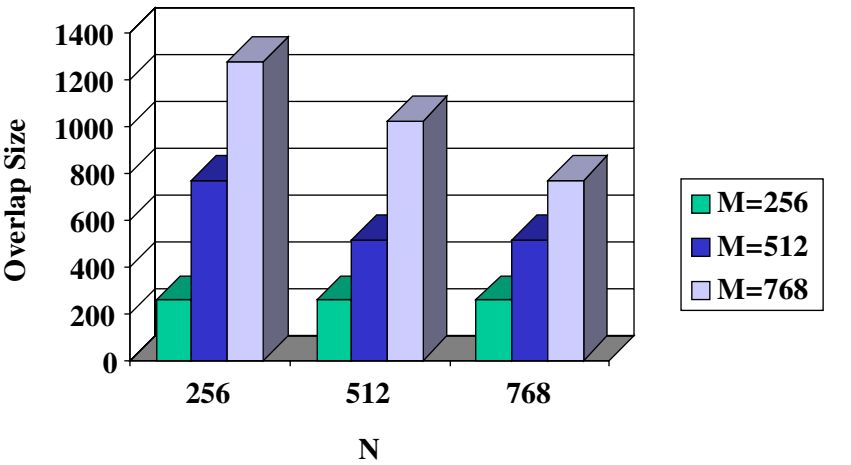

Fig. 6.3. Overlap size for the codes of Ex. 6.4.2(b) and 6.4.2(c)

symbolic expression defined over constants and invariants), polynomial recurrence variables (incremented by a linear symbolic expression defined over constants, invariants and recurrence variables), and geometric recurrence variables (incremented by a term which contains a recurrence variable multiplied by an invariant). The algorithm [48] for computing lower and upper bounds of symbolic expressions based on a set of constraints is used to detect whether a recurrence variable monotonically increases or decreases. Even if no closed form can be found for a recurrence variable, monotonicity information may be useful, for instance, to determine whether a pair of references can ever touch the same address.

We present empirical results to demonstrate the need for more advanced symbolic analysis techniques. For this study we used several Fortran programs that have been taken from the Perfect Benchmarks, RiCEPS (Rice Compiler Evaluation Program Suite), and Genesis benchmark suites [12, 1] as well as from the WIEN95 software package [13]. The latter code has been developed by the chemistry department of the Vienna University of Technology and is used for quantum mechanical calculations of solids.

Table 6.3 provides a brief description, and the number of lines and subroutines for each program. In Table 6.4 we report on the frequencies of some important program constructs/features that are currently not adequately handled by most existing compiler techniques. The data is presented as a percentage of the total number of references, loops, variables, or expressions. For instance, 14 % of all array references in the BDNA code are non-linear. The two columns about "advanced dep. test" reflect the need of advanced dependence testing.

Furthermore, many of the codes contain linear references with symbolic coefficients (reflected by the third column in Table 6.4). Without additional

Table 6.3. Program description

code	benchmark	type	lines	subr.
BDNA	molecular dynamics of DNA	Perfect Benchm.	3980	43
MDG	molecular dynamics of water	Perfect Benchm.	1238	16
MG3D	depth migration	Perfect Benchm.	3038	28
OCEAN	2-dim ocean simulation	Perfect Benchm.	3198	32
TRFD	2 electron integral transform	Perfect Benchm.	485	7
TRACK	missile tracking	RiCEPS	3735	34
HNS	quantum mechanical calculations	Wien95	495	5
PDE2	2-dim multigrid Poisson solver	GENESIS	551	14

information about these coefficients conventional dependence analyses imply worst-case assumptions (assume a dependence). However, most of these symbolic coefficients in linear references in the codes of Table 6.4 – except for HNS and PDE2 – can be replaced with constants by using interprocedural constant propagation. The fraction of array references for which advanced dependence tests are required ranges from 7 - 78 % across all codes as shown in Table 6.4. Our techniques for comparing both linear and non-linear symbolic expressions and the ability of our symbolic analysis to propagate constraints through a program enables us to handle the majority of these references. Note that there are a few sophisticated dependence tests [93, 9, 83, 21] that handle both linear and non-linear symbolic expressions. Although some of these approaches appear to be more powerful than our techniques, they also have some drawbacks associated with them, for instance, restricted program models (no GOTOs and procedure calls) [9], and introduction of a large number of linear constraints to model non-linear constraints [83]. We believe that all of these techniques could benefit by interprocedural analysis and collection and propagation of constraints as provided by our symbolic analysis.

The next two columns ("indirect array refs.") in Table 6.4 summarize the frequencies of indirect array references which can go up to 15 %. For other codes of the perfect benchmarks, such as *spice* (not included in our study) up to 60 % of all array subscripts correspond to indirect array references as reported in [56]. All array accesses are precisely modeled by our symbolic analysis techniques, even in the case where we cannot find closed forms. For most array references excluding input dependent indirection arrays, we can determine useful results in terms of closed forms. Although the number of indirection arrays appear to be small, they may be significant in terms of performance. For instance, a single loop nest in the TRFD code, that contains an indirect array reference, if not parallelized, accounts for close to 25 % of the overall execution time. Our symbolic analysis techniques actually enable dependence testing of most non input dependent indirect array references as shown in Table 6.4. Similar accounts for all array references with linear subscript expressions. The two columns about "closed forms" tabulate for

Table 6.4. Program characteristics and effectiveness of symbolic analysis. All figures are given as percentage values.

code	advanced dep. test		indirect array refs.		closed forms	
	non-linear refs.	symbolic coeff.	non input depent.	input depend.	loop variables	other variables
BDNA	14	23	14	0	95	55
MDG	3	15	0	0	100	71
MG3D	1	22	1	0	97	26
OCEAN	23	8	0	0	100	40
TRFD	3	4	3	0	100	57
TRACK	3	10	2	0	100	43
HNS	67	11	0	15	100	80
PDE2	18	8	12	0	100	84

code	bounds/subscript	loops	
	expr. as a function of problem size	with procedure calls	with multiple exits
BDNA	95	11	1
MDG	91	10	0
MG3D	90	8	5
OCEAN	83	10	0
TRFD	99	3	1
TRACK	95	10	3
HNS	80	15	0
PDE2	90	12	0

each program the percentage of all recurrences (loop and other variables) for which our symbolic analysis finds a closed form. "Other variables" correspond to scalar variables that cannot be eliminated by scalar forward substitution and whose value changes inside of loops. In most cases it is very critical to find closed forms for recurrences as they commonly appear in array subscript expressions which are subject of dependence analysis.

The next column ("bounds/subscript expr. as a function of problem size") demonstrates that our symbolic analysis techniques can express most variables appearing in loop bounds (lower and upper bound and stride expressions) and array subscript expressions as functions of the problem size.

Finally, the last two columns tabulate the percentage of loops with procedure calls or multiple exits. Although the number of these loops is small, in most cases they significantly impact the overall execution time. For instance, all loops with multiple exits of the TRACK code amount for approximately 40 % of the overall sequential execution time. D. Wonnacott [106] shows how to handle breaks from loops and gives rules for interprocedural analysis but apparently has not yet implemented these techniques. Except for Won-

nacott's work and our symbolic analysis framework, we are not aware of any symbolic analysis technique that actually copes with multi-exit loops. We believe that modeling procedure calls is very important in order to detect and exploit sources of parallelism in particular in the case of procedure calls inside of loops.

We have examined the simplification techniques (for instance, those shown in Chapter 3 of our symbolic analysis framework when applied to the codes of Table 6.4. Without simplification techniques our framework's ability to

- conduct dependence tests is reduced by at least 56 % for non-linear references, by 35 % for symbolic coefficients, and by 68 % for non-input dependent indirect array references.
- compute closed forms is reduced by at least 20 % for loop variables and 57 % for other variables.
- specify variables appearing in loop bounds and array subscript expressions as functions of the problem size, is reduced by at least 45 %.

Therefore, adding simplification techniques is crucial to further improve our symbolic analysis framework.

In the following we report on several experiments conducted with existing codes such as OLDA and HNS.

6.6 Symbolic Analysis to Optimize OLDA

In this experiment, we use OLDA which is the dominant routine in the TRFD program of the Perfect Benchmarks [12] in order to demonstrate more general improvements obtained by using symbolic analysis. The OLDA code contains conditional assignments and GOTOs, recurrences, non-linear loop bounds and array index expressions. Table 6.5 displays the execution times as obtained on a Meiko CS-2 for various problem (N) and machine sizes (number of processors NP) for two different parallelization strategies. Firstly, we optimized and parallelized OLDA by using VFC without symbolic analysis (column 3). Secondly, we compiled the codes by using VFC with symbolic analysis (column 4) which in particular reduced communication and also partially improved dependence analysis.

We ran our tests on 4, 8, 16 and 32 processors on a Meiko CS-2 for two different problem sizes $N = 4$ and $N = 8$.

Optimizations made possible by symbolic analysis reduced the total execution time by up to 27 %. By resolving recurrences, symbolic analysis eliminates several dependences that cannot be handled by VFC without symbolic analysis. By far the largest performance improvement obtained by our symbolic analysis is caused by a reduced communication overhead. The ability to compare non-linear loop bound and index expressions allowed us to eliminate redundant communication which is not done by VFC without symbolic analysis. Symbolic analysis reduces only a small portion of OLDA's computation

Table 6.5. Meiko CS-2 execution times for OLDA

		Execution Time in Seconds		
N	*NP*	VFC (w/o SE)	VFC (with SE)	Improvement (in %)
4	1	0.412	0.393	4.7
	4	0.165	0.145	13.8
	8	0.141	0.118	19.5
	16	0.127	0.101	25.7
	32	0.122	0.096	27.1
8	1	2.114	1.980	6.2
	4	0.961	0.889	8.1
	8	0.801	0.711	12.6
	16	0.703	0.596	18.0
	32	0.654	0.546	19.8

time. OLDA's computational overhead is of the order $O(N^4)$ and the communication overhead is of the order $O(N^2)$. Therefore, by increasing the problem size N, the communication takes a smaller portion of the total execution time and the influence of our symbolic analysis becomes less apparent. Similarly, when increasing the number of processors for a fixed problem size, the percentage of reducing execution time increases which is caused by an increased communication overhead whereas the computation remains unchanged.

6.7 Symbolic Analysis to Optimize HNS

The HNS code is used for quantum mechanical calculations of solids as part of the WIEN95 software package [13]. A crucial part of the material science code WIEN95 is to solve the Schroedinger equation using a variational method (Raleigh-Ritz). The eigenvalues of the corresponding eigenvalue problem correspond to energy levels of electronic orbitals and the eigenvectors are used to compute the electron density in a given crystal. To solve the eigenproblem a symmetric (hermitian) matrix (the Hamiltonian) has to be defined. The subroutine HNS is part of the computation of the matrix elements. Radial and angular dependent contributions to these elements are pre-computed and condensed in a number of vectors which are then applied in a series of rank-2 updates to the symmetric (hermitian) Hamilton matrix. HNS has 17 one-dimensional arrays, 14 two-dimensional arrays, 5 three-dimensional arrays, and 6 four-dimensional arrays. The computational overhead of HNS is of the order $O(N^2)$. All floating point operations are on double (eight bytes). All important arrays have been distributed CYCLIC [10, 74] due to non-linear array references and triangular loop iteration spaces.

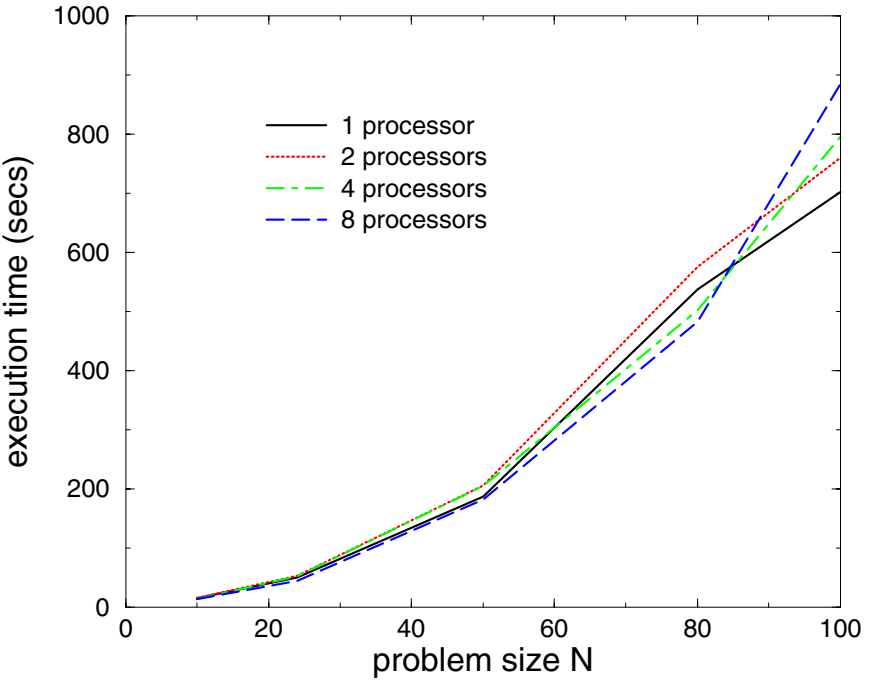

Fig. 6.4. Execution time for parallel HNS code without symbolic analysis for various problem and machine sizes on a Meiko CS-2

Figures 6.4 and 6.5 plot the execution times of the HNS code for various problem (N) and machine sizes (number of processors) on a Meiko CS-2 for two different parallelization strategies. Firstly, HNS has been parallelized and optimized by using VFC without symbolic analysis (see Fig. 6.4). Due to many non-linear array references the compiler is unable to determine the absence of data dependences which results in sequentializing the core loops of HNS and as a consequence in a performance slowdown for increasing number of processors. Moreover, VFC inserts a mask for every array assignment statement in order to enforce the "owner computes rule". In addition, the compiler introduces for every right-hand-side array reference a function call that examines whether or not this reference must be communicated for a particular instance of the array assignment. The extra computational overhead implied by masks and function calls is the first order performance effect for smaller problem sizes. Communication becomes the predominated factor for increasing problem sizes.

Secondly, we used our symbolic analysis to improve the parallelization and optimization techniques of VFC. Based on symbolic analysis the compiler can

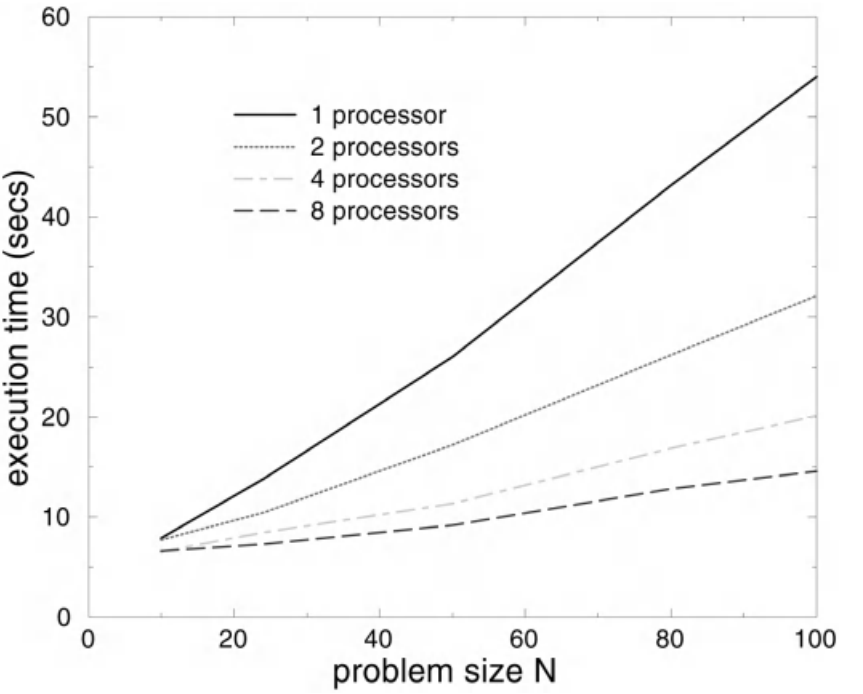

Fig. 6.5. Execution time for parallel HNS code with symbolic analysis for various problem and machine sizes on a Meiko CS-2

parallelize all core loops of HNS without communication. Symbolic analysis also enables the compiler to generate more efficient code as compared to compiling the program without symbolic analysis. Instead of including function calls to ensure the "owner computes rule", loop bounds can now be parameterized [41] such that every processor is restricted to those loop iterations where local data (owned by the processor) is written. VFC with symbolic analysis results in a performance gain (see Fig. 6.5) that scales reasonably well for increasing processor numbers and, therefore, clearly outperforms the version without symbolic analysis.

The execution time and memory requirement to compile the HNS code by VFC without code generation and symbolic analysis on a Sparc 10 workstation is 4.36 seconds and 2 MBytes, respectively. Whereas compiling HNS with symbolic analysis takes 7.26 seconds and requires 3.1 MBytes memory space.

More experiments that show the effective usage of our symbolic analysis framework can be found in [50, 17, 97].

6.8 Summary

It is widely accepted [67, 102, 101, 19, 20, 98, 84] that current parallelizing compilers are not very effective in optimizing parallel programs that fully utilize the target multiprocessor system. The poor performance of many compiler analyses can be attributed to ineffective parallelization of programs (for instance, High Performance Fortran *HPF* [74] and Fortran 90 [85]) that have a strong potential for unknowns such as number of processors and sizes of allocateable arrays.

The quality of many optimizations and analyses of parallelizing compilers significantly depends on the ability to evaluate symbolic expressions and on the amount of information available about program variables at arbitrary program points. All of our techniques target both linear as well as non-linear expressions and constraints. Efficiency of symbolic analysis is highly improved by aggressive simplification techniques. A variety of examples, including program verification, dependence analysis, array privatization, communication vectorization and elimination of redundant communication has been used to illustrate the effectiveness of our approach. We have presented results from a preliminary implementation of our framework which is used as part of a parallelizing compiler that demonstrate the potential performance gains achievable by employing symbolic evaluation to support program parallelization.

Previous approaches on symbolic compiler analysis frequently have several drawbacks associated with them:

- restricted program models (commonly exclude procedures, complex branching and arrays)
- analysis that covers only linear symbolic expressions
- insufficient simplification techniques
- memory and execution time intensive algorithms
- unstructured, redundant and inaccurate analysis information, and complex extraction of analysis information
- additional analysis is required to make the important relationship between problem sizes and analysis results explicit
- recurrences are frequently not supported at all or separate analysis and data structures are required to extract, represent, and resolve recurrence systems.

Furthermore, for parallelizing compilers there is a need to express values of variables and expressions as well as the (path) conditions under which control flow reaches a program statement. Some approaches [71, 68, 27] do not consider path conditions for their analysis. Without path conditions the analysis accuracy may be substantially reduced. More recent work [101] is based on an existing representation of analysis information. For instance, G-SSA form [5] commonly requires additional algorithms to determine variable values and path conditions. G-SSA form does not represent this information in a unified data structure. We are not aware of an approach that combines

all important information about variable values, constraints between them, and path conditions in a unified and compact data representation. Moreover, it has been realized [98] that systems with interfaces for off-the-shelf software are critical for future research tool and compiler development. Approaches that rely on specialized representations for their analysis are commonly forced to re-implement standard symbolic manipulation techniques which otherwise could be taken from readily available software packages.

Empirical results based on a variety of benchmark codes demonstrate among others the need for advanced analysis to handle non-linear and indirect array references, procedure calls and multiple exit loops.

We have implemented a prototype of our symbolic evaluation framework which is used as part of the Vienna High Performance Compiler (VFC) [10] – a parallelizing compiler for distributed memory architectures – and P^3T [40, 41] – a performance estimator – to parallelize and optimize High Performance Fortran programs [74, 10] for distributed memory architectures.

7. Related Work

P. and R. Cousot [31] pioneered abstract interpretation as a theory of semantic approximation for systematic data and control flow analysis of sequential programs. In abstract interpretation a program denotes computations in some universe of objects. According to [31] abstract interpretation can be considered as a mathematical theory to unify many program analyses. Moreover, abstract interpretation commonly achieves only useful results if substantially customized for particular applications.

Despite its fundamentally incomplete results (for instance, due to approximations) abstract interpretation may find solutions to problems which tolerate an imprecise results. Many program analysis systems are based on abstract interpretation. Blume and Eigenmann [22] use abstract interpretation to extract and propagate constraints about variable ranges in shared memory parallel programs. Their way of computing ranges for program values is based on data-flow analysis that requires to iterate to a fixed point. Widening operations are used to reduce the number of data-flow iterations which may result in conservative results for recurrences. For instance, $x = x + 1$ implies $x \leq \infty$ independent of whether this statement is placed inside a loop with a given lower and upper bound. Also narrowing operations tries to compensate for the loss of accuracy, according to [31], just discards infinite bounds but makes no improvement on finite bounds. P. Cousot and N. Halbwachs [32] compute and propagate constraints through a sequential program based on abstract interpretation. Constraints between program variables are represented as n-dimensional convex polyhedrons and are restricted to linear inequality relationships. Neither interprocedural analysis nor effective techniques to eliminate redundant constraints are included in their approach.

M. Haghighat and C. Polychronopoulos [67] describe a variety of symbolic analysis techniques based on abstract interpretation. The information of all incoming paths to a statement is intersected at the cost of analysis accuracy. Their approach excludes an explicit model for arrays which prevents handling of array references as part of symbolic expressions (although array subscript expressions can be compared for dependence analysis). They do not propagate predicates guarding the conditional values of variables through a program. Haghighat's symbolic differencing method recognizes *generalized induction variables* that form polynomial and geometric progressions through

T. Fahringer and B. Scholz: Advanced Symbolic Analysis, LNCS 2628, pp. 99–108, 2003.

read(A) A=A+1 B=A+2	read(A_1) A_2=A_1+1 B_1=A_2+2	read(A) [s_1={A=▽,B=b},t_1=true,p_1=true] A=A+1 [s_2=φ(s_1;A=▽+1),t_2=t_1,p_2=p_1] B=A+2 [s_3=φ(s_2;B=▽+3),t_3=t_2,p_3=p_2]
(a) Example 1	(b) (G-)SSA form	(c) Symbolic Analysis

Fig. 7.1. Comparison (G)SSA vs. Symbolic Analysis

loop iterations. For resolving generalized induction variables the compiler has to iterate the loop a fixed number of times and the number of iterations has to be provided by the user. This means, that the user must be knowledgeable about the type of program to make a wise decision, otherwise the analysis information is not safe. In contrast, our approach computes the recurrence relations of induction variables and, then, an algebraic tool (1) tests if the recurrence relation is in the class of solvable recurrence systems and (2) solves the recurrence system.

The main differences between abstract interpretation and our symbolic analysis are as follows: First, our symbolic analysis framework precisely represents the values of program variables whereas abstract interpretation commonly approximates a program's computations. Second, path conditions or predicates guarding conditional variable values are not included in abstract interpretation. Third, applications of abstract interpretation are faced with a trade-off between the level of abstraction and the precision of the analysis. Although abstract interpretation may have a better computational complexity, its approximated information may not be accurate enough to be useful. Fourth, the underlying analysis representation is fundamentally different. Whereas we introduced program contexts (state, state condition, and path condition) as a novel basis for our symbolic analysis, abstract interpretation is primarily based on state (function from variables to abstract symbolic values) information.

J. King [80] introduced symbolic execution of programs where a program is actually executed based on symbolic input. The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols. In [30] symbolic execution has been used for software specialization.

T. Cheatham, G. Holloway and J. Townley [27] discovered symbolic evaluation as a means of static program analysis to support verification of sequential EL1 programs. They also introduced some techniques to solve recurrence relations as well as simplifying system of constraints both of which are based on linear problems. They consider all paths that reach a program

point which implies exponential complexity and use a complex and costly program representation for their analysis. For instance, program statements, predicates of conditional assignments, and variable values are all separately represented and linked together via context numbers and time labels. The list of all values ever assigned to a variable is explicitly stored and retrieved via time stamps. Path conditions are not employed by their method. Our approach is integrated in a control flow graph and presents a compact association of each program statement with a state, state condition and path condition. Furthermore, our techniques to simplify symbolic constraints as described in Chapter 5 are superior to their methods.

R. Cytron et al. [34] developed a new program representation called static single-assignment form (SSA) where each variable is written only once and which encodes DEF/USE information. Although it is not a target of our symbolic analysis, DEF/USE information is implicitly contained in the program contexts. SSA form has been used by numerous researchers [56, 94] to statically examine programs at compile time.

R. Ballance et al. [5] extended SSA to gated SSA (G-SSA) form by introducing special gating functions to include predicates of conditional branches which is a major deficiency of SSA form. Extensive renaming of program variables in SSA and G-SSA form makes it more difficult to express symbolic values as functions over a program's problem size. Our analysis information representation does not change the code and can be optionally displayed together with the code. A key reason why symbolic analysis is getting increasingly popular to use for many program analysis tools, (compilers, performance tools, debuggers, etc.) is its expected ability to make the crucial relationship between a program's problem sizes (program input data, number of processors, array sizes, etc.) and the analysis information transparent. In contrast to our symbolic analysis approach, SSA and G-SSA form do not directly preserve this problem size/analysis information relationship. Explicit analysis is required to determine input data relationships which can be expensive in the general case. Consider the code in Fig. 7.1(a) where the value of B is computed as a function of the input value of A. SSA and G-SSA form (Fig. 7.1(b)) require explicit analysis to determine that the value of B depends on the input value of the read statement. By contrast the code based on our symbolic analysis (Fig. 7.1(c)) directly presents the value of B as a function of the input value $\bigtriangledown$ without additional analysis. For this reason G-SSA and SSA forms seem to be less amenable for program analysis where the resulting analysis information should be expressed as a function over problem sizes. E.g., for performance estimators it is very desirable to provide performance information as parameterized functions over the number of processors, array sizes and input data. These parameterized functions can then be employed to compute the optimal number of processors and array sizes to be used for a specific parallel machine [66, 26].

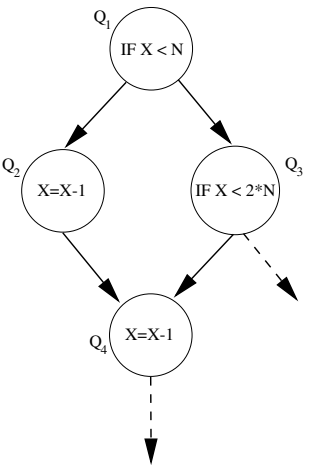

Fig. 7.2. Example 2

It has been shown [71] that G-SSA form can imply undefined and, therefore, redundant components in G-SSA gating functions. For instance, γ(X<N,X-2,γ(X<2*N,X-1,$\perp$)) is the G-SSA form of the value of X at the end of ℓ_4 in Fig. 7.2 where $\perp$ is an undefined value implied by a path that can never be taken. Ballance et al. [5] need this information to drive the insertion of switches, creating a data-driven form of their PDW (Program Dependence Web). However, in most other G-SSA applications, $\perp$ in a γ function implies redundant information and unnecessary overhead for analysis, simplification, and manipulation of such terms. Our symbolic analysis technique does not imply redundancy for the context of ℓ_4 which is [s_4=φ(s_0;X=x_1), t_4=(x_0<N$\wedge x_1$=x_0-2)$\vee$($x_0\geq$N$\wedge x_0$<2*N$\wedge x_1$=x_0-1), p_4=x_0<2*N]. The state immediately before ℓ_1 is s_0 and x_0 is the value of X in s_0. Although a method has been found to eliminate $\perp$ in G-SSA forms for well-structured IF-constructs [71], in general, it is not possible to eliminate all $\perp$ in G-SSA forms.

P. Havlak [71] used G-SSA form to build a global value graph and conduct interprocedural analysis to support symbolic dependence testing, array section analysis, and test elision. He does not address the issue of modeling loops with multiple exits.

P. Tu and D. Padua [102, 101] developed a system for computing symbolic values of expressions using a demand-driven backward analysis based on G-SSA form. Their backward-driven analysis can be faster and less memory intensive than our approach, if local analysis information suffices to obtain a result. However, if local information is not sufficient, they may have to examine large portions of a program. An additional algorithm and analysis

```
ℓ1:    if (A < B) then
ℓ2:       X = V
ℓ3:    else
ℓ4:       X = V + 1
ℓ5:    endif
ℓ6:    if (X < B) then
ℓ7:       Z = X + 2
ℓ8:    else
ℓ9:       Z = X
ℓ10:   endif
```

Fig. 7.3. Example 3

is required to determine the path condition as G-SSA form does not explicitly incorporate this information. Our symbolic analysis approach directly represents path conditions in the context information. Furthermore, whereas in symbolic analysis values of symbolic expressions and variables are determined by using local information of a program context, Tu's and Padua's method requires an additional backward-driven substitution algorithm based on G-SSA form.

In contrast to our representation, describing symbolic expression values based on G-SSA can imply significant redundancy. Consider the code in Fig. 7.3. The value of Z in G-SSA form with substitution after ℓ_{10} – assuming that substitution of the value of X is required in order to determine whether the outcome of the conditional expression of ℓ_6 can be statically determined – is given by $\gamma(\gamma(\text{A<B,V,V+1})<\text{B},\gamma(\text{A<B,V,V+1})+2, \gamma(\text{A<B,V,V+1}))$. Note that the term $\gamma(\text{A<B,V,V+1})$ appears three times in this expression. The symbolic context after ℓ_{10} is specified by $[s_{10}{=}\varphi(s_0;\text{X=x,Z=z}), t_{10}{=}\gamma(a{<}b;x{=}v;\ x{=}v{+}1) \wedge \gamma(x{<}b;z{=}x{+}2;z{=}x), p_{10}{=}\text{true}]$ where a, b, and v are respectively the values of variable A, B and V in state s_0 immediately before ℓ_1. G-SSA form may imply several identical gating functions in a symbolic expression each of which has to be explicitly considered by potential manipulation techniques. Redundant gating functions can, therefore, impact the overall computational complexity. Whereas our symbolic analysis tries to avoid redundancy through the use of n-order logic, the definition of our specific γ function, and the simplifications for program contexts with γ functions. For the example given in Fig. 7.3 the conditional expressions are represented exactly once for each different IF-construct. Clearly, this reduces the overhead to analyse, simplify, and compare symbolic expressions. Furthermore, both backward-driven analysis and G-SSA tend towards hiding the crucial relationship of problem size and resulting analysis information. Consider the code of Fig. 3.3.1 of Chapter 3. Our symbolic analysis represents the values of X and Y after statement ℓ_{10} by $[s_{10}{=}\varphi(s_7;Y{=}2 * \bigtriangledown_1), t_{10}{=}\gamma(\bigtriangledown_1{<}0;\ x_2{=}{-}2 * \bigtriangledown_1; x_2{=}2 * \bigtriangledown_1), p_{10}{=}\text{true}]$ which can be rewritten

as [s_{10}={X=x_2,Y=$2 * \bigtriangledown_1$},t_{10}=$\gamma(\bigtriangledown_1$<0; x_2=$-2 * \bigtriangledown_1$;x_2=$2 * \bigtriangledown_1$),p_{10}=true]. G-SSA form with substitution yields y = $\gamma(\gamma(Y_0 < 0, 2{*}Y_0, Y_0) \geq 0, 2{*}\gamma(Y_0 < 0, 2{*}Y_0, Y_0), \gamma(Y_0 < 0, 2{*}Y_0, Y_0))$ and x=$\gamma(Y_0 < 0, -2{*}Y_0, 2{*}Y_0)$ where X_0 and Y_0, respectively, correspond to the variables of X and Y immediately after ℓ_1. Clearly, G-SSA form implies redundancy compared to symbolic analysis as the gating function $\gamma(Y_0 < 0, 2{*}Y_0, Y_0)$ occurs three times and the gating condition $Y_0 < 0$ four times in the G-SSA form for X and Y.

For recurrences Tu and Padua cannot directly determine the corresponding system of recurrences from a given G-SSA form. Instead they need to apply a sequence of substitution rules to G-SSA form, interpret the result as a λ function (lambda calculus), rewrite the λ function in the terms of recursive sequence in combinatorial mathematics before they finally use standard techniques to resolve the system of recurrences. Our approach enables us to directly extract the system of recurrences from our data representation without additional analysis. Tu and Padua do not explain how symbolic expressions without gating functions are simplified and compared which is non-trivial. In [102, 101] they describe that two expressions e_1, e_2 without gating functions can be compared by their method iff $e_1 - e_2$ is a constant. Our implemented framework uses efficient techniques introduced in Chapter 5 [47, 48] for comparing symbolic expressions, computing bounds of symbolic expressions, and simplifying systems of symbolic constraints which are considerably more effective. For instance, comparing symbolic expressions is done based on a system of constraints (contained in the context information). If the difference of $e_1 - e_2$ is not a constant it may still be possible to determine a useful result based on the constraints given and the underlying algorithm employed. In addition, their approach to compare symbolic expressions with gating functions can be substantially improved. For instance, for the purpose of comparing two expressions their techniques transform $i > \gamma(i > j, j, i - 1)$ into $i > j$ and thus do not incorporate the predicate in the γ function to determine that the result of this expression is *true*. Techniques introduced in [47, 48] can easily determine that the result of this expression is *true* by considering the condition of the gamma function. Moreover, in Tu's and Padua's system, expressions of the form $e(c)$ – expression value e at a statement with path condition c – are rewritten to e if e does not contain gating functions. Again important information is lost by their expression manipulation system which can imply critical approximations that may no longer be useful. They do not state how to handle division, multiplication, and exponentiation as part of operations defined over symbolic expressions which has been addressed by expression manipulation and simplification system [47, 48]. Tu's and Padua's approach avoids modeling of procedures by full in-line expansion.

The complexity to generate G-SSA form cannot be directly compared with the creation of program contexts as introduced in this book. A program context fully represents the symbolic values of variables and path conditions whereas G-SSA requires additional analysis (such as backward-driven substi-

tution) to determine the same information. In general, symbolic expressions can grow with the program size and manipulating and simplifying symbolic expressions is known to be of exponential complexity.

M. Gerlek, et al. [56] developed a recurrence solver based on a classification system of linear induction variables as part of the Nascent compiler. We partially use their techniques for resolving linear, polynomial and geometric recurrence variables. Their method uses SSA form without employing path conditions and propagation of constraints about program variables. Gerlek et al. cannot handle loops with multiple exits, conditional recurrences, array references as part of recurrences (for instance, I = I + A(I)), and indirect array references. They also have problems with interprocedural analysis. For instance, variables as procedure parameters cannot be handled in recurrences. An important difference between their work and our approach is the way in which recurrences are detected and represented. Our approach enables us to represent all recurrence systems implied by recurrence variables. Gerlek et al. classify recurrences without representing them. They couple detecting and resolving recurrences which makes their system more difficult to extend for new recurrence classes. Except for simple cases Gerlek does not state how to determine the monotonic behavior for unresolved variables (no closed form can be found) whereas we have used a very effective algorithm introduced in Section 5.3 for this problem.

There is a variety of symbolic analyses that focus on array region analysis. For instance, W. Pugh and D. Wonnacott [92] use a set of constraints to describe linear array data flow problems and solve them by the Fourier-Motzkin variable elimination method. B. Creusillet and F. Irigoin [33] introduced an interprocedural array flow analysis for array privatization. Linear array subscript expressions and array regions are described by a single convex region. J. Gu, et al. [65] substantially improved array region analysis by conditional array regions which models control flow of programs and is used for array privatization. They also describe array regions by a set of convex regions which is more precise than using only a single convex region. This analysis can be applied both intra- and inter-procedurally. We believe that our symbolic analysis framework can support all of these techniques by collecting constraints about program unknowns throughout the program.

Our symbolic analysis framework has a promising potential for sparse representation of analysis information and demand driven analysis. For instance, analysis information can be stored at selected program statements such as at the entry and exit of loops and procedures. Analysis can be restricted to code segments, by assigning unknown symbolic values to all variables at the entry of the code segment selected. Modeling arrays, recurrences, procedures, etc. can be selectively turned on and off. Furthermore, if the values of a subset of variables should be computed at a specific statement, we can use conventional USE/DEF information to reduce the list of variables to be maintained in a program context. Finally, analysis information is implemen-

ted as a linked list of contexts. Only the semantic information implied by a statement ℓ is actually included in the context of ℓ. A link is included to the predecessor contexts that hold the remaining information valid at ℓ.

7.1 Advanced Symbolic Analysis Algorithms

In this Section we compare our techniques for advanced symbolic analysis algorithms (see Section 5) with related work.

Blume [18] developed an algorithm to compare two symbolic expressions based on symbolic ranges. A range $[a : b]$ for a variable x defines a single lower and upper bound on x, which can also be written as $a \leq x \leq y$. The main difference between our algorithm and the one of Blume is that Blume's techniques are based on manipulation of symbolic expressions containing variable ranges, whereas our method applies φ-functions to symbolic expressions. Furthermore, Blume assumes for the range $[a : b]$ of a variable v that $a \leq b$. This restriction makes it difficult for some important compiler analyses to apply his algorithm. For instance, loop variables whose lower bounds can be larger than their upper bounds for some loop iterations must be modeled in order to compute loop iteration counts or detect zero-trip-loops [67]. Blume does not discuss how to handle multiple lower and upper bounds for a variable v, which is a critical issue for symbolic sum algorithms or dead code elimination.

M. Haghighat and C. Polychronopoulos [67] described techniques to prove that a symbolic expression is strictly increasing or decreasing, which enables them to analyze codes for data dependences. Our approach goes beyond their techniques by also considering symbolic expressions as exponents. P. Tu and D. Padua [102] present an algorithm to determine the relationship between two symbolic expressions which is based on gated static single assignment (G-SSA) form. V. Maslov and W. Pugh [84] propose a method to simplify polynomial constraints over integers to make dependence analysis more precise. They transform classes of polynomial constraints into a conjunction of affine constraints by using factorization and affinization techniques.

P. Havlak [71] uses G-SSA for comparing symbolic expressions and for data dependence testing as part of interprocedural compiler analysis. His techniques are efficient for linear expressions but are limited in case of nonlinear expressions.

E. Su et al. [98] implemented an interface between a commercial symbolic manipulation package and a parallelizing compiler. They developed several compiler analyses and transformations based on symbolic analysis including integer area estimation, computation partitioning and cyclic array distribution.

Our symbolic sum algorithm as shown in Fig. 5.2 of Section 5.4 is an extension to what has been sketched by W. Pugh as the convex sum for linear inequalities in [91]. Pugh has not implemented his techniques according to

[91]. Whereas we have presented a precise definition for the symbolic sum algorithm, fully implemented it, and integrated our techniques with P^3T to apply them to performance prediction applications. Pugh described his algorithm for a set of linear constraints, whereas we extended the algorithm to cover also non-linear constraints. His techniques are based on simplifications for linear inequalities, whereas our simplification techniques (see Section 5.5) handle also classes of non-linear constraints and expressions. Pugh's algebraic sum algorithm is restricted to variables with positive exponents, whereas our method also handles negative exponents.

7.2 Parallelizing Compilers

The Parafrase-2 compiler [89] is a source to source code restructurer for Fortran and C supports analysis, transformation and code generation. It allows automatic vectorization and parallelization. Haghighat's symbolic analysis framework [67] is part of the Parafrase-2 compiler. The implemented symbolic analysis framework supports symbolic forward substitution, generalized induction variables, and some algorithms for solving a class of recurrences and symbolic integer division algorithms.

The PARADIGM compiler [6] is a source to source translator based on Parafrase-2 which accepts sequential Fortran 77 annotated with High Performance Fortran [74] directives. The compiler produces optimized message-passing Fortran 77 parallel programs.

The Polaris compiler [23] is a source to source restructurer developed for Fortran 77. It performs interprocedural analysis for automatic parallelization of programs, targeting shared-memory architectures. Blume and Eigenmann [22] have implemented their symbolic analysis techniques in the context of the Polaris compiler.

The SUIF compiler [81] is another parallelizing compiler. The compiler uses the *Stanford University Intermediate Form* that is a low level representation with some high level additions that keep track of the program structure such as loops, conditions and array references. The input language is C and Fortran through the f2c translator. The compiler outputs machine code for different architectures. The compiler initial design and the current distribution of SUIF does not incorporate interprocedural analysis.

The Fortran D compiler [75] converts Fortran D programs to F77 SPMD message-passing programs. Data distributions are restricted to single array dimensions and the number of processors has to be a constant at compile time.

The Fortran 90D compiler [107] is a source-to-source compiler for SIMD and MIMD parallel architectures. The Fortran90D compiler only exploits the parallelism inherent in the array constructs and makes no attempts to parallelize sequential constructs, e.g. loops. Data distributions are restricted to static block and cyclic distributions and the number of processors must

be known at compile time. The output of the compiler is a Fortran SPMD program with calls to a communication library.

The ADAPTOR system [24] developed at the GMD is a prototype system for interactive source-to-source transformations for Fortran Programs. The input language is Fortran 77 with Fortran 90 extensions and some data distribution directives. The output is a Fortran 77 SPMD program with explicit message passing calls. The system has no facilities for automatic parallelization.

The PIPS [77] developed at Ecole des mines de Paris is an HPF compiler which translates Fortran 77 with HPF directives to a Fortran 77 SPMD program. It supports a variety of compiler analyses such as array privatization, array section privatization, and array element regions.

The VFC compiler [10] is a source to source HPF compiler, which translates Fortran 95/HPF+ programs to Fortran 90 message-passing programs. VFC implements most features of HPF+ and provides languages features that can deal with irregular programs. Our symbolic analysis framework has been implemented as part of the VFC. We have shown significant performance improvements in Chapter 6 by applying our techniques.

8. Conclusion

Program analysis tries to automatically determine properties of the run-time behavior of a program. Conventional analyses commonly fail to deal with complex (e.g. non-linear) expressions with unknowns, to compare symbolic expressions, to express analysis information as a function over program problem sizes, to analyze non-linear terms in subscript expressions, and to determine under which condition the control flow reaches a certain program point. Ineffective approaches to gather and propagate sufficient data and control flow information through the program continues to have a detrimental impact on many compiler analyses and optimizations. As a consequence worst case assumptions are frequently made or program analysis is shifted to run-time which increases the execution overhead. Sophisticated symbolic analysis that can cope with program unknowns, evaluate symbolic expressions, deal with complex symbolic expressions, determine variable values and control flow conditions, and propagate symbolic data and control flow information through a program can simplify many compiler deficiencies.

In this book we have described a novel symbolic analysis framework for sequential, parallel, and distributed programming languages that statically determines the values of variables and symbolic expressions, assumptions about and constraints between variable values and the condition under which control flow reaches a program statement. The computed program analysis information is described by program contexts, a new representation for comprehensive and compact control and data flow analysis information. Program contexts are described as logic formulas which allows us to use computer algebra systems for standard symbolic manipulation. Our symbolic evaluation techniques comprise accurate modeling of assignment and input/output statements, branches, loops, recurrences, arrays, dynamic records and procedures. All of our techniques target both linear as well as non-linear expressions and constraints. Efficiency of symbolic evaluation is highly improved by aggressive simplification techniques.

The most important features and contributions of our symbolic analysis framework are:

- a novel representation for comprehensive and compact control and data flow analysis information, called *program context*,

T. Fahringer and B. Scholz: Advanced Symbolic Analysis, LNCS 2628, pp. 109–111, 2003.

- program context components that can be separately accessed at well-defined program points and processed by computer algebra systems,
- an algorithm for generating program contexts based on control flow graphs including accurate modeling of assignment and input/output statements, branches, loops, recurrences, arrays, dynamic records, and procedures,
- an approach which targets linear and non-linear symbolic expressions, and the representation of program analysis information as symbolic expressions defined over the program's problem size.
- sophisticated algorithms to compare symbolic expressions and to compute the number of solutions to a set of symbolic constraints which among others can be employed to simplify symbolic expressions and constraints.

We do not know of any other system that models a similarly large class of program constructs based on a comprehensive and unified analysis representation that explicitly captures exact information about variable values, assumptions about and constraints between variable values, path conditions, recurrence systems, and side-effect information of procedure calls. Although we have examined our framework for Fortran programs, the underlying techniques are equally applicable to any similar imperative programming language.

Many parallelizing compilers are unable to effectively parallelize and optimize programs on today's multiprocessor architectures primarily due to the lack of advanced and accurate compiler analysis. We have shown empirical results based on a variety of benchmark codes that demonstrate among others the need for advanced analyses to handle non-linear and indirect array references, procedure calls, and multiple exit loops. In order to simplify or even overcome these program analysis complexities, we have applied our techniques, analyses and algorithms for symbolic program analysis to sequential, parallel and distributed programming languages. We have developed a symbolic analysis framework that has been mostly used in the context of a parallelizing compiler and a performance estimator for parallel Fortran programs. Our system has been employed to support symbolic dependence testing and various optimizations (including communication vectorization and elimination of redundant communication) which can result in significant performance improvements for parallel and distributed programs. Moreover, we have used our symbolic analysis framework to predict the symbolic work distribution of data parallel programs as part of a performance estimator.

The most important features and contributions of our symbolic analysis framework for parallelizing compilers are:

- a more precise data dependence analysis based on our symbolic analysis framework,
- optimizations for communication [50] by extracting single element messages from a loop and combining them to vector-form (*communication vectorization*), removing redundant communication (*communication coalescing*),

and aggregating different communication statements (*communication aggregation*),
- implementation of our symbolic analysis framework in the context of the Vienna High Performance compiler (VFC) [10],
- demonstration of effectiveness of our framework for programs that have been taken from the Perfect, RiCEPS (Rice Compiler Evaluation Program Suite), and Genesis benchmark suites [12, 1] as well as from the WIEN95 software package [13].
- integration of our symbolic analysis framework with a state-of-the-art performance estimator [51] to predict the work distribution of parallel programs as a parameterized function defined over unknown problem and machine sizes.

Currently, we are extending several compiler optimizations for distributed memory architectures to exploit the prototype implementation of our symbolic evaluation framework under VFC. We are also working on modeling recursive procedures and adding alias and pointer analysis. In the future, we plan to conduct more extensive experiments, to study the performance impact of other optimizations captured by our framework such as reducing the number of cache misses and optimizing parallel input/output. Furthermore, our techniques are not restricted to compilers and tools for parallel architectures. They can be equally important to compiler analysis and tools for sequential architectures, program testing [80], program verification [27], software specialization [30], software reuse [28], pattern matching and concept comprehension [37], etc. We plan to investigate the application of our framework to some of these areas.

9. Appendix

9.1 Control Flow Graphs

A control flow graph (*CFG*) of a program is defined by a directed flow graph $G = (N, E, x, y)$ with a set of nodes N and a set of edges $E \subseteq N \times N$. A node $u \in N$ represents a program instruction (statement), an edge $u \to v \in E$ indicates transfer of control between instructions $u, v \in N$. Node $x \in N$ and node $y \in N$ are the unique *start* and *end* node of G, respectively. Consider the control flow graph in Fig. 9.1. We have a set of nodes $N = \{x, 1, \ldots, 7, y\}$ and a set of edges. The start node is denoted by x and the end node by y.

If $u \to v \in E$ is an edge, then u is the *source* node and v is the target node of the edge $u \to v$. The set $succs(u)$ denotes the set of successors of a node $u \in N$ and is defined by $succ(u) = \{v \mid u \to v \in E\}$. The set $preds(v)$ correspond to the set of predecessors and is given by $pred(v) = \{u \mid u \to v \in E\}$. Note that the set of successors of the start node x is empty and the set of successors of the end node y is empty as well. In general a *path* in G is a sequence of nodes $[u_1, ..., u_k]$, where $\forall 1 \leq i < k$ the following holds: $u_{i+1} \in succs(u_i)$.

A CFG may not be connected, that is, some nodes may not be reachable from the start node x which can be caused by compiler transformation, e.g. dead code elimination where parts of the control flow graph are eliminated. Therefore, whenever we refer to a CFG we mean the subgraph of a CFG such that all nodes in the sub graph are reachable from the start node x and all nodes can reach the end node y, i.e. every node is assumed to reside on a path from x to y.

Let S be a set, we will use the notation $|S|$ to denote the number of elements in the set. E.g., $|N|$ denotes the number of nodes in the control flow graph. For our example in Fig. 9.1 we have $|N| = 9$ and $|E| = 11$ since we have 9 nodes (start and end node included) and 11 edges.

In a control flow graph, a node u *dominates* another node v iff all paths from the start node x to v always pass through u.

CFGs are classified into the class of *reducible* and the class of *irreducible* graphs. One of the most important properties of reducible CFGs is that there are no jumps into the middle of the loops from outside; the only entry to a loop is through a header [2]. More formally, a control flow graph is reducible

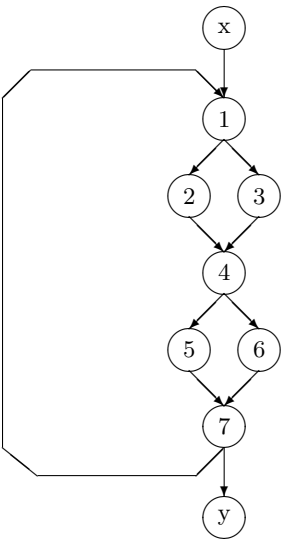

Fig. 9.1. Control flow graph

if and only if we can partition the edges into two disjoint groups, often called the *forward edges* and *back edges* with the following two properties:

1. The forward edges form an acyclic graph in which every node can be reached from the start node x.
2. The back edges consist only of edges whose targets dominate their sources.

If the control flow graph is not reducible, it is called an *irreducible* control flow graph.

The control flow graph of Fig. 9.1 is reducible since there is only one back edge from node 7 to node 1 and 1 dominates 7. All other edges are forward edges and form an acyclic graph as given in the conditions for reducible control flow graphs above.

9.2 Denotational Semantics

There are two main aspects of a computer language — its syntax and its semantics. The syntax defines the correct form for legal programs and the semantics determines what they compute. While the syntax of a language is always formally specified in a variant of BNF, the more important part of defining its semantics is mostly left to natural language, which is ambiguous and leaves many questions open. Hence, methods were developed to describe the semantics of computer languages. Denotational semantics is a methodology to define the precise meaning of a computer language. In denotational semantics, a computer language is given by a valuation function that maps programs into mathematical objects considered as their denotation, i.e. meaning. Thus the valuation function of a computer program reveals the meaning of computer programs while neglecting its syntactic structure.

As mentioned above the syntax of a computer language is concerned only with the structure of programs. The syntax treats a language as a set of strings over an alphabet of symbols. The syntax is usually given by a grammar with *production rules* for generating strings of symbols using auxiliary *nonterminal* symbols (see Context-Free grammars [76]). An alternative representation is an abstract syntax which treats a language as a set of trees. In contrast to grammars, trees have a compositional structure which is inherently unambiguous and there is only one way of constructing a particular tree [86].

Consider the syntax of binary numerals as follows,

Binary-Numeral
B::= 0 | 1 | B 0 | B 1

We have one nonterminal symbol B which has 4 productions. The grammar above can generate all possible binary numerals. Now the question is how can we translate a binary number to a mathematical object.

Denotational semantics relates semantic objects to productions. The semantic objects specified for productions are called *denotations*. Semantic functions map productions to their actual denotations. The semantics of a computer language may be specified by defining a semantic function for each nonterminal in the grammar. For our previous example we introduce a semantic function $\mathcal{B}$ in order to translate a string of 0 and 1 to a number in $\mathbb{Z}$. This function maps a production of the grammar to an element of $\mathbb{Z}$. By successively applying the semantic rule we compute the numerical value of the binary number as follows,

Binary-Numeral
$\mathcal{B} : \mathrm{B} \mapsto \mathbb{Z}$
$\mathcal{B}[\![0]\!] = 0$

$\mathcal{B}[\![1]\!] = 1$
$\mathcal{B}[\![\mathrm{B0}]\!] = 2 \cdot (\mathcal{B}[\![\mathrm{B}]\!])$
$\mathcal{B}[\![\mathrm{B1}]\!] = 2 \cdot (\mathcal{B}[\![\mathrm{B}]\!]) + 1$

In the example we describe the semantics of the right-hand side α of an production by $\mathcal{B}[\![\alpha]\!]$. The denotations are given on the left-hand side such as $2 \cdot (\mathcal{B}[\![\mathrm{B}]\!])$ in the third rule of the example. Here we recursively process a string and apply a simple number conversion schema.

In this work we use denotations for the lambda calculus as explained in [86]. The lambda calculus allows us to extend the denotational semantic functions with additional parameters. These extension are necessary to deduce semantics for more complex languages with variable bindings. For variable bindings we need to establish a concept of a relation between program variables and their values. Variable bindings can be expressed by additional parameters and the denotations of statements can change the variable binding.

9.3 Notation

Symbol	Description
$\cup$	union of sets
$\|\|$	set cardinality
$\rightarrow$	rewrite operator for expressions
iff if and only if	
s	state of a program context
t	state condition of a program context
p	path condition of a program context
c	program context $c = [s, t, p]$
ℓ_i	program label
$\overline{c}$	strongest post condition of context c
$\perp$	undefined value
$\delta(s; v_1 = e_1, \dots)$	changes variable binding of a state
$\bigtriangledown$	numbered input symbol of a read statement
ι	input counter
C	set of contexts
E	set of symbolic expressions
$low(e)$	minimum value of a symbolic expression e
$up(e)$	maximum value of a symbolic expression e
$eval(e, [s, t, p])$	evaluates expression e under $[s, t, p]$
F	denotational semantic function for programs
$\gamma(cnd; \dots; \dots)$	conditional variable binding
$\odot$	confluence operator
$\mu(v, s, c)$	multivariate symbolic recurrence
$\mathbb{A}$	array algebra
$\oplus$	chain operator
ρ	read operator for symbolic chains
θ	simplification operator for symbolic chains
$\mathbb{H}$	heap algebra
hp	heap state
$\mathbf{put}(r, \texttt{<q>}, v)$	heap function for field assignment
$\mathbf{free}(r)$	heap function for de-allocation
$\mathbf{new}(r, < \texttt{t} >)$	heap function for allocation
tr	symbolic tracefile
$\mathbb{T}$	chain algebra for symbolic tracefiles
$\mathbf{s}(r, nb)$	write reference r of length nb in symbolic tracefile
$\mathbf{l}(r, nb)$	read reference r of length nb in symbolic tracefile
$\mathbf{a}(r, nb)$	read or write reference r of length nb in symbolictracefile
$\otimes$	cache evaluation operator
H	hit set for symbolic cache evaluation
ca	symbolic cache contents
h	number of hits as symbolic expression
$\widetilde{r}$	start address of corresponding memory block of reference r
$\chi(r)$	cache placement function
$S(\iota)$	slot with index ι
$\nu(r)$	replacement strategey for reference r
$\tau^{\max}$	symbolic reference counter
φ	upper/lower bounds for symbolic expressions
$\varrho(c)$	checks for a condition c to be true
υ	processor
Υ	set of processors
$\zeta^A(\upsilon)$	set of local elements of array A owned by processor υ
$\eta^A(\upsilon)$	set of non-local accesses in processor υ for array A

9.4 Denotational Semantic: Notation

$$\mathcal{F} : \texttt{<stmt>} \to C \to C$$

(AS) $\mathcal{F}[\![\texttt{<var> := <expr>}]\!] = \lambda[s,t,p] \in C.\,[\delta\,(s; \texttt{<var>} = eval\,(\texttt{<expr>}, [s,t,p]))\,, t, p]$

(SQ) $\mathcal{F}[\![\texttt{<stmt>}_1\ \texttt{<stmt>}_2]\!] = \lambda[s,t,p] \in C.\,(\mathcal{F}[\![\texttt{<stmt>}_2]\!](\mathcal{F}[\![\texttt{<stmt>}_1]\!][s,t,p]))$

(RE) $\mathcal{F}[\![\texttt{read <v>}]\!] = \lambda[s,t,p] \in C.\,[((\lambda\alpha.\delta\,(s; v = \bigtriangledown_\alpha, \iota = \alpha + 1))eval(\iota, [s,t,p])), t, p]$

(I1) $\mathcal{F}[\![\texttt{if <cond> then <stmt> end if}]\!] =$
$\lambda[s,t,p] \in C.\,(\mathcal{F}[\![\texttt{<stmt>}]\!]\,[s,t,p \wedge eval\,(\texttt{<cond>}, [s,t,p])])\odot$
$[s,t,p \wedge \neg eval\,(\texttt{<cond>}, [s,t,p])]$

(I2) $\mathcal{F}[\![\texttt{if <cond> then <stmt>}_1\ \texttt{else <stmt>}_2\ \texttt{end if}]\!] =$
$\lambda[s,t,p] \in C.\,(\mathcal{F}[\![\texttt{<stmt>}_1]\!]\,[s,t,p \wedge eval\,(\texttt{<cond>}, [s,t,p])])\odot$
$(\mathcal{F}[\![\texttt{<stmt>}_2]\!]\,[s,t,p \wedge \neg eval\,(\texttt{<cond>}, [s,t,p])])$

(WI) $\mathcal{F}[\![\texttt{while <cond> do <stmt> end do}]\!] =$
$\lambda[s,t,p] \in C.[loopeval(\texttt{<cond>}, \texttt{<stmt>}, [s,t,p]), t, p]$

(AA) $\mathcal{F}[\![\texttt{<var> (<expr>}_1\texttt{) := <expr>}_2]\!] =$
$\lambda[s,t,p] \in C.[\delta(s; \texttt{<var>} = eval\,(\texttt{<var>}, [s,t,p])$
$\oplus(eval\,(\texttt{<expr>}_2, [s,t,p])\,, eval\,(\texttt{<expr>}_1, [s,t,p]))), t, p]$

(D1) $\mathcal{F}[\![\texttt{<ptr>}_1\texttt{\%<q> := <ptr>}_2]\!] =$
$\lambda[s,t,p] \in C.[\delta(s, hp = \text{eval}(hp, [s,t,p])$
$\oplus\mathbf{put}(\mathcal{F}[\![\texttt{<ptr>}_1]\!]([s,t,p]), \texttt{<q>}, \mathcal{F}[\![\texttt{<ptr>}_2]\!]([s,t,p]))), t, p]$

(D2) $\mathcal{F}[\![\texttt{<ptr>\%<q> := <value>}]\!] =$
$\lambda[s,t,p] \in C.[\delta(s, hp = \text{eval}(hp, [s,t,p]) \oplus$
$\mathbf{put}(\mathcal{F}[\![\texttt{<ptr>}]\!]([s,t,p]), \texttt{<q>}, \text{eval}(\texttt{<value>}, [s,t,p]))), t, p]$

(D3) $\mathcal{F}[\![\texttt{allocate(<ptr>\%<q>)}]\!] =$
$\lambda[s,t,p] \in C.[\lambda r \in E.\delta\,(s, hc = r, hp = \text{eval}(hp, [s,t,p]) \oplus \mathbf{new}(r, \texttt{<t>}) \oplus$
$\mathbf{put}(\mathcal{F}[\![\texttt{<ptr>}]\!]([s,t,p]), \texttt{<q>}, r))\,(\text{eval}(hc, [s,t,p]) + 1), t, p]$

(D4) $\mathcal{F}[\![\texttt{<var> := <ptr>}]\!] =$
$\lambda[s,t,p] \in C.[\lambda r \in E.\delta(s, \texttt{<var>} = r, hp = \text{eval}(hp, [s,t,p])$
$\oplus\mathbf{put}(\text{eval}(\texttt{<var>}, [s,t,p]), \bot, r))$
$(\mathcal{F}[\![\texttt{<ptr>}]\!]([s,t,p])), t, p]$

(D5) $\mathcal{F}[\![\texttt{allocate(<var>)}]\!] =$
$\lambda[s,t,p] \in C.[\lambda r \in E.\delta(s, \texttt{<var>} = r, hc = r, hp = \text{eval}(hp, [s,t,p])$
$\oplus\mathbf{new}(r, \texttt{<t>}) \oplus \mathbf{put}(\texttt{<var>}, \bot, r))$
$(\text{eval}(hc, [s,t,p]) + 1), t, p]$

(D6) $\mathcal{F}[\![\texttt{deallocate(<ptr>)}]\!] =$
$\lambda[s,t,p] \in C.[\delta\,(s, hp = \text{eval}(hp, [s,t,p]) \oplus \mathbf{free}(\text{eval}(\texttt{<ptr>}, [s,t,p]))\,, t, p]$

$$eval : \texttt{<expr>} \times C \to E$$

(E1) $eval\,(\texttt{<const>}, [s,t,p]) = \texttt{<const>}$

(E2) $eval\,(\texttt{<var>}, [s,t,p]) = e_i$, where $s = \{v_1 = e_1, \ldots, \texttt{<var>} = e_i, \ldots\}$

(E3) $eval\,(\texttt{<expr>}_1\ op\ \texttt{<expr>}_2, [s,t,p]) = eval(\texttt{<expr>}_1, [s,t,p])\ op\ eval(\texttt{<expr>}_2, [s,t,p])$

(E4) $eval\,(\texttt{<var>(<expr>)}, [s,t,p]) = \rho(eval(\texttt{<var>}, [s,t,p]), eval(\texttt{<expr>}, [s,t,p]))$

(E5) $eval(\texttt{<nil>}, [s,t,p]) = \bot$

(E6) $eval(\texttt{<ptr>\%<q>}, [s,t,p]) = \mathbf{get}(eval(hp, [s,t,p]), eval(\texttt{<ptr>}, [s,t,p]), \texttt{<q>})$

References

1. C. Addison, J. Allwright, N. Binsted, N. Bishop, B. Carpenter, P. Dalloz, D. Gee, V. Getov, A. Hey, R. Hockney, M. Lemke, J. Merlin, M. Pinches, C. Scott, and I. Wolton. The GENESIS distributed-memory benchmarks. Part 1: Methodology and general relativity bnechmark with results for the SUPRENUM computer. *Concurrency Practice and Experience (Ed. Geoffrey Fox), Wiley*, 5(1):1–22, 1993.
2. A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools.* Series in Computer Science. Addison Wesley, 1988.
3. T. M. Austin, S. E. Breachand, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of SIGPLAN'94 Conference on Programming Languages Design and Implementation*, volume 29 of *ACM SIGPLAN Notices*, pages 290–301, Orlando, FL, June 1994. ACM Press.
4. B. S. Baker. An algorithm for structuring flowgraphs. *J. ACM*, 24(1):98–120, Jan. 1977.
5. R. Ballance, A. Maccabe, and K. Ottenstein. The Program Dependence Web: a Representation Supporting Control-, Data-, and Demand Driven Interpretation of Imperative Languages . In *Proceedings of the SIGPLAN 90 Conference on Program Language Design and Implementation*, pages 257–271, White Plains, New York, June 1990.
6. P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers. In *Proceedings of the First International Workshop on Parallel Processing*, pages 322–330, Bangalore, India, December 1994.
7. U. Banerjee. *Dependence analysis for supercomputing.* Kluwer Academic, Boston, MA, USA, 1988.
8. S. R. M. Barros, D. Dent, L. Isaksen, G. Robinson, G. Mozdzynski, and F. Wollenweber. The IFS model: A parallel production weather code. *Parallel Computing*, 21(10):1621–1638, November 1995.
9. D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40(2):210–226, Feb. 1997.
10. S. Benkner. VFC: The Vienna Fortran Compiler. *Scientific Programming, IOS Press, The Netherlands*, 7(1):67–81, 1999.
11. S. Benkner, S. Andel, R. Blasko, P. Brezany, A. Celic, B. Chapman, M. Egg, T. Fahringer, J. Hulman, Y. Hou, E. Kelc, E. Mehofer, H. Moritsch, M. Paul, K. Sanjari, V. Sipkova, B. Velkov, B. Wender, and H. Zima. *Vienna Fortran Compilation System – Version 2.0 – User's Guide*, October 1995.

12. M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orzag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, pages 3(3): 5–40, 1989.
13. P. Blaha, K. Schwarz, P. Dufek, and R. Augustyn. Wien95, a full-potential, linearized augmented plane wave program for calculating crystal properties. Institute of Technical Electrochemistry, Vienna University of Technology, Vienna, Austria, 1995.
14. J. Blieberger. Data-Flow Frameworks for Worst-Case Execution Time Analysis. (accepted for Real Time Systems Journal), 2001.
15. J. Blieberger, B. Burgstaller, and B. Scholz. Interprocedural Symbolic Analysis of Ada Programs with Aliases. In *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*, pages 136–145, June 1999.
16. J. Blieberger, B. Burgstaller, and B. Scholz. Symbolic Data Flow Analysis for Detecting Deadlocks in Ada Tasking Programs. In *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*, Potsdam, Germany, June 2000.
17. J. Blieberger, T. Fahringer, and B. Scholz. Symbolic Cache Analysis for Real-Time Systems. *Real-Time Systems Journal*, 18(2/3):181–215, May 2000.
18. W. Blume. *Symbolic Analysis Techniques for Effective Automatic Parallelization*. PhD thesis, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, June 1995.
19. W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the perfect benchmark programs. *IEEE Transactions on Parallel and Distributed Systems*, pages 3(6):643–656, November 1992.
20. W. Blume and R. Eigenmann. An Overview of Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Benchmarks. In *Proceedings of the 1994 International Conference on Parallel Processing*, St. Charles, IL, 1994.
21. W. Blume and R. Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. CSRD (Center for Research on Parallel Computation) Report No. 1345, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, April 1994.
22. W. Blume and R. Eigenmann. Demand-driven, Symbolic Range Propagation. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing, Columbus, OH*, August 1995.
23. W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. A. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The Next Generation in Parallelizing Compilers. . In *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing, Ithaca, New York*, pages 10.1–10.18, August 1994.
24. T. Brandes. Efficient data parallel programming without explicit message passing for distribute memory multiprocessors. Technical Report AHR-92-4, GMD, 1992.
25. I. Bronstein and K. Semendjajev. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, Frankfurt, Germany, ISBN 3 87144492 8, 1989.
26. B. Chapman, T. Fahringer, and H. Zima. *Automatic Support for Data Distribution*, pages 184–199. Springer Verlag, Lecture Notes in Computer Science: Languages and Compilers for Parallel Computing, 6th International Workshop, (Ed.) U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Portland, Oregon, Aug. 1993.

27. T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic Evaluation and the Analysis of Programs. *IEEE Transactions on Software Engineering*, 5(4):402–417, 1979.
28. A. Cimitile, A. D. Lucia, and M. Munro. Qualifying Reusable Functions Using Symbolic Execution. In *Proceedings of the 2nd Working Conference on Reverse Engineering, IEEE Comp. Society*, Toronto, Canada, August 1995.
29. M. Clement and M. Quinn. Symbolic Performance Prediction of Scalable Parallel Programs. In *Proc. of 9th International Parallel Processing Symposium*, St. Barbara, CA, April 1995.
30. A. Coen-Porisini, F. D. Paoli, C. Ghezzi, and D. Mandrioli. Software Specialization Via Symbolic Execution. *IEEE Transactions on Software Engineering*, 17(9):884–899, Sept. 1991.
31. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, January 1977.
32. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 84–97, Tucson, AZ, January 1978.
33. B. Creusillet and F. Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6):513–546, Dec. 1996.
34. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
35. R. Cytron, J. Ferrante, and V. Sarkar. Experience using control dependence in PTRAN. In D. Gelernter, A. Nicolau, and D. P. (Editors), editors, *Languages and Compilers for Parallel Computing. The MIT Press*, 1990.
36. E. Dijkstra. *A discipline of programming.* Prentice Hall, New Jersey, 1976.
37. B. DiMartino. Algorithmic Concept Recognition Support for Automatic Parallelization: A Case Study for Loop Optimization and Parallelization. *Journal of Information Science and Engineering, Special Issue on Compiler Techniques for High-Performance Computing*, to appear in March 1998.
38. E. Dockner and H. Moritsch. Pricing Constant Maturity Floaters with Embeeded Options Using Monte Carlo Simulation. Technical Report AuR_99-04, AURORA Technical Reports, University of Vienna, January 1999.
39. B. Elsner, H. G. Galbas, B. Görg, O. Kolp, and G. Lonsdale. A parallel, multilevel approach for contact problems in crashworthiness simulation. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and D. Trystram, editors, *Parallel Computing: State-of-the-Art and Perspectives, Proceedings of the Conference ParCo'95, 19–22 September 1995, Ghent, Belgium*, volume 11 of *Advances in Parallel Computing*, pages 157–164, Amsterdam, Feb. 1996. Elsevier, North-Holland.
40. T. Fahringer. Estimating and Optimizing Performance for Parallel Programs. *IEEE Computer*, 28(11):47–56, November 1995.
41. T. Fahringer. *Automatic Performance Prediction of Parallel Programs.* Kluwer Academic Publishers, Boston, USA, ISBN 0-7923-9708-8, March 1996.
42. T. Fahringer. Compile-Time Estimation of Communication Costs for Data Parallel Programs. *Journal of Parallel and Distributed Computing, Academic Press*, 39(1):46–65, Nov. 1996.

43. T. Fahringer. On Estimating the Useful Work Distribution of Parallel Programs under P^3T: A Static Performance Estimator. *Concurrency, Practice and Experience (Ed. Geoffrey Fox)*, 8(4):261–282, May 1996.
44. T. Fahringer. Toward Symbolic Performance Prediction of Parallel Programs. In *IEEE Proc. of the 1996 International Parallel Processing Symposium*, pages 474–478, Honolulu, Hawaii, April 15–19, 1996. IEEE Computer Society Press.
45. T. Fahringer. Effective Symbolic Analysis to Support Parallelizing Compilers and Performance Analysis. In *Proc. of the International Conference and Exhibition on High-Performance Computing and Networking (HPCN'97), Vienna, Austria*. Lecture Notes in Computer Science, Springer Verlag, 1997.
46. T. Fahringer. Symbolic Expression Evaluation to Support Parallelizing Compilers. In *IEEE Proc. of the 5th Euromicro Workshop on Parallel and Distributed Processing, London, UK*, pages 22–24. IEEE Computer Society Press, January 1997.
47. T. Fahringer. Efficient Symbolic Analysis for Parallelizing Compilers and Performance Estimators. *Journal of Supercomputing, Kluwer Academic Publishers*, 12(3):227–252, May 1998.
48. T. Fahringer. Symbolic Analysis Techniques for Program Parallelization. *Journal of Future Generation Computer Systems, Elsevier Science, North-Holland*, 13(1997/98):385–396, March 1998.
49. T. Fahringer and E. Mehofer. Buffer-safe communication optimization based on data flow analysis and performance prediction. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques (PACT'97)*, pages 189–200, San Francisco, California, Nov. 10–14, 1997. IEEE Computer Society Press.
50. T. Fahringer and E. Mehofer. Buffer-Safe and Cost-Driven Communication Optimization. *Journal of Parallel and Distributed Computing, Academic Press*, 57(1):33–63, April 1999.
51. T. Fahringer and A. Požgaj. P^3T+: A Performance Estimator for Distributed and Parallel Programs. *Scientific Programming, IOS Press, The Netherlands*, 8(2), November 2000.
52. T. Fahringer and B. Scholz. Symbolic Evaluation for Parallelizing Compilers. In *Proc. of the 11th ACM International Conference on Supercomputing*, pages 261–268, Vienna, Austria, July 1997. ACM Press.
53. T. Fahringer and B. Scholz. A Unified Symbolic Evaluation Framework for Parallelizing Compilers . *IEEE Transactions on Parallel and Distributed Systems*, 11(11):1106–1126, 2000.
54. M. Flynn. Some Computer Organisations and theis Effectiveness. *Trans. Computers*, 21:948–960, 1972.
55. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.
56. M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-Driven SSA Form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 17(1):85–122, January 1995.
57. H. Gerndt. *Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Bonn, December 1989.
58. H. Gerndt. Updating Distributed Variables in Local Computations. *Concurrency: Practice and Experience*, Vol. 2(3):171–193, September 1990.
59. H. M. Gerndt. Work Distribution in Parallel Programs for Distributed Memory Multiprocessors. In *Proceedings of the International Conference on Supercomputing*, pages 96–104, Cologne, June 17–21 1991.

60. M. Gerndt and H. Zima. Optimizing Communication in SUPERB. In *Technical Report ACPC/TR 90-3*. Austrian Center for Parallel Computation, March 1990. Also: Proc. CONPAR 90-VAPP IV (Zurich, Sep 1990) Lecture Notes in Computer Science LNCS 457, 300–311.
61. G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. *ACM SIGPLAN Notices*, 26(6):15–29, June 1991.
62. R. Gordon. *The Denotational Description of Programming Languages* . Springer Verlag, New York, 1975.
63. R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics – A foundation for Computer Science*. Addison-Wesley Publishing Company, ISBN 0-201-14236-8, 1988.
64. D. H. Greene and D. E. Knuth. *Mathematics for the Analysis of Algorithms*. Birkhäuser Boston, 1982.
65. J. Gu, Z. Li, and G. Lee. Experience with efficient array data flow analysis for array privatization. . In *Proc. Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 157–167, Las Vegas, Nevada, June 18–21 1997.
66. M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, 1992.
67. M. Haghighat and C. Polychronopoulos. Symbolic Analysis for Parallelizing Compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 18(4):477–518, July 1996.
68. M. R. Haghighat. *Symbolic Analysis for Parallelizing Compilers*. Kluwer Academic Publishers, 1995.
69. M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler . *IEEE Computer*, 29(12):84–89, December 1996.
70. D. Harper, C. Wooff, and D. Hodgkinson. *A Guide to Computer Algebra Systems*. John Wiley & Sons, 1991.
71. P. Havlak. *Interprocedural Symbolic Analysis*. PhD thesis, Department of Computer Science, Rice University, 1994.
72. K. A. Hawick and G. C. Fox. Exploiting high performance Fortran for computational fluid dynamics. *Lecture Notes in Computer Science*, 919:413–419, 1995.
73. M. S. Hecht. *Flow Analysis of Computer Programs* . Elsevier, North-Holland, 1977.
74. High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.
75. S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In *Proc. of the 4th Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, Aug 1991.
76. J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, N. Reading, MA, 1980.
77. F. Irigoin, P. Jouvelot, and R. Triolet. Semantical Interprocedural Parallelization: An Overview of the PIPS Project. Technical Report Report A/201, Ecole des mines de Paris, 1991.
78. J. Janssen and H. Corporaal. Making graphs reducible with controlled node splitting. *ACM Transactions on Programming Languages and Systems*, 19(6):1031–1052, November 1997.

79. K. Kennedy, K. McKinley, and C.-W. Tseng. Interactive parallel programming using the ParaScope editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.
80. J. C. King. Symbolic Execution and Program Testing . *Communications of the ACM*, 19(7):385–394, July 1976.
81. S.-W. Liao, A. Diwan, R. P. Bosch, A. Ghuloum, and M. S. Lam. SUIF Explorer: an interactive and interprocedural parallelizer. *ACM SIGPLAN Notices*, 34(8):37–48, Aug. 1999.
82. G. S. Lueker. Some Techniques for Solving Recurrences. *ACM Computing Surveys*, 12(4):419–435, December 1980.
83. V. Maslov. Delinearization: An efficient way to break multiloop dependence equations. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, volume 27, pages 152–161, New York, NY, July 1992. ACM Press.
84. V. Maslov and W. Pugh. Simplifying Polynomial Constraints Over Integers to Make Dependence Analysis More Precise. UMIACS-TR-93-68.1 and CS-TR-3109.1, Univ. of Maryland, Dept. of Computer Science, February 1994.
85. M. Metcalf and J. Reid. *Fortran 90/95 explained*. Oxford Science Publications, 1996.
86. P. D. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Sience*, volume B, chapter 11, pages 575–631. The MIT Press/Elsevier, P.O. Box 211, 1000 AE Amsterdam, The Netherlands, 1990.
87. M. P. I. F. MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, January 1996.
88. C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. Parafrase-2: An Environment for Parallelizing, Paritioning, Synchronizing, and Scheduling Programs on Multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume II, pages 39–48. Pennsylvania State University Press, August 1989.
89. C. D. Polychronopoulous, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume II – Software, pages II–39–II–48, University Park, Penn, Aug. 1989. Penn State. U. Ill.
90. W. Pugh. The Omega Test: a Fast and Practical Integer Programming Algorithm for dependence Analysis. *Communications of the ACM*, 8:102–114, August 1992.
91. W. Pugh. Counting Solutions to Presburger Formulas: How and Why. In *ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 121–134, Orlando, FL, June 20–24 1994.
92. W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. *Lecture Notes in Computer Science*, 768:546ff., 1994.
93. W. Pugh and D. Wonnacott. Nonlinear array dependence analysis. In *In 3rd Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Troy, New York, May 1995.
94. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 12–27, San Diego, CA, January 1988.
95. M. Scheibl, A. Celic, and T. Fahringer. Interfacing Mathematica from the Vienna Fortran Compilation System. Technical Report, Institute for Software Technology and Parallel Systems, Univ. of Vienna, December 1996.

96. B. Scholz. Symbolische Verifikation von Echtzeitprogrammen. Master's thesis, Vienna University of Technology, Austria, 1997.
97. B. Scholz, J. Blieberger, and T. Fahringer. Symbolic Pointer Analysis for Detecting Memory Leaks. In *Proc. of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 104–113, Boston, MA, USA, January 2000. ACM Press.
98. E. Su, A. Lain, S. Ramaswamy, D. Palermo, E. Hodges, and P. Banerjee. Advanced Compilation Techniques in the Paradigm Compiler for Distributed-Memory Multicomputers. In *Proc. of the 9th ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.
99. V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: practice and experience*, 2(4), December 1990.
100. N. Tawbi. Estimation of nested loop execution time by integer arithmetic in convex polyhedra. In *Proc. of the 1994 International Parallel Processing Symposium*, April 1994.
101. P. Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.
102. P. Tu and D. Padua. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. In *9th ACM International Conference on Supercomputing*, pages 414–423, Barcelona, Spain, July 1995.
103. M. N. Wegman and K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.
104. M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, Reading, Mass., 1996.
105. S. Wolfram. *Mathematica – A System for Doing Mathematics by Computer*. The Advanced Book Program. Addison Wesley, 1991.
106. D. Wonnacott. *Constraint-Based Array Dependence Analysis*. PhD thesis, University of Maryland, 1995.
107. M. Wu and G. Fox. A Test Suite Approach for Fortran90D Compilers on MIMD Distributed Memory Parallel Computers. In *Proc. of the Scalable High Performance Computing Conference*, Williamsburg, USA, April 1992.
108. H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – a language specification. Technical report, ICASE, Hampton,VA, 1992. ICASE Internal Report 21.
109. H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press Frontier Series. Addison-Wesley, 1990.
110. H. Zima and B. Chapman. Compiling for distributed memory systems. In *Proceedings of the IEEE*, pages 264–287, February 1993.

Index

Υ, 86
$\bigoplus$, 26
$\bigtriangledown$, 2
$\bigtriangledown$, 15–20
$\oplus$, 26
– chain, 26
φ, 54
P^3T , 101
loopeval, 24

abstract syntax, 120
algebraic sum, 66
algorithm
– algebraic sum, 66
– lower and upper bound, 56
– symbolic sum, 63, 64, 73
algorithm:simplify constraints, 68
application
– large-scale, 79
array, 25
– algebra, 25
– block-wise distributed, 82
– distributed, 81
– multi-dimensional, 32
– one-dimensional, 25
– overlap area, 82
– privatization, 10
assignment, 14
associativity, 3

back edges, 118
benchmark
– BDNA, 94
– Genesis benchmark suites, 93
– HNS, 94
– MDG, 94
– MG3D, 94
– OCEAN, 94
– PDE2, 94
– Perfect Benchmarks, 93, 96
– RiCEPS, 93
– TRACK, 94
– TRFD, 94
BNF
– Backus Naur Form, 120
boundary condition, 22

CFG, 118
closed forms, 4
codes
– FTRVMT, 75
communication
– aggregation, 89
– coalescing, 89
– vectorization, 88
commutativity, 3
computer
– cluster, 79
– control mechanism, 79
– distributed memory, 79
– distributed memory multiprocessing system DMMP, 80
– high performance, 79
– interconnection network, 79
– local memory, 79
– memory organization, 79
– MIMD, 79
– non-uniform memory access NUMA, 80
– shared memory, 79
– SIMD, 79
– SISD, 79
– uniform memory access UMA, 80
– virtual shared memory, 79
computer algebra systems
– CAS, VIII
computer language
– semantics, 120
– syntax, 120
concept comprehension, 8
conditional statements
– if-statement, 17
confluence, 18
constraints, 9

– linear constraints, 54
– nonlinear constraints, 54
– tautologies, 69
context-free grammar, 120
control flow domination, 118

data dependence, 83
– analysis, 83
– relation, 84
data parallelism, 80
dead code elimination, 118
deadlock, 8
δ, 15
denotational semantics, 120
dependence
– anti-dependence, 84
– flow dependence, 84
– memory-related, 84
– output dependence, 84
difference equation, 4
distributivity, 3
dynamic data structures, 34
dynamic record, 14, 34

equalities
– simplify, 67
eval
– function, 16
EXSR, 89

$\odot$, 20
Fortran 90, 7
forward edges, 118
forward substitution, 3
Fourier transforms, 5
fractional expressions, 68
FTRVMT, 75
functions
– *varphi*, 54
– up, 54

G-SSA, 100
γ, 19
generating functions, 5
graph
– irreducible, 118
– reducible, 118

heap, 34
– free, 35
– functions σ, 35
– new, 35
– operations, 34
– put, 35
HPF, 7, 80

identity, 3
inequalities
– eliminate, 70
– redundant, 69, 70
– simplify, 67, 69
initialization, 14
Input/Output Operations, 14
inverse, 3

lambda calculus, 121
local segment, 86
loop, 21
loop invariant, 23, 53, 54, 60, 62, 77, 93
low function, 54

Meiko CS-2, 96
memory leak, 8, 35
message passing, 79
monotonicity, 93
μ, 22

Nascent, 10
nil-pointer dereferences, 8
non-local
– access, 86
– elements, 86

overlap
– analysis, 86
– area, 86
– description, 87
– size, 92
– total, 87
owner computes paradigm, 81

Paradigm, 10
Parafrase-2, 10
parallel programming model, 80
– explicit, 80
– implicit, 80
– MPI, 80
– PVM, 80
parallelizing compiler, 7, 79
Parascope, 10
path, 118
pattern matching, 8
Perfect Benchmarks, 96
performance prediction
– P^3T , 101
Polaris, 10
potential index set, 29, 37
preds(u), 118

procedure, 32
– actual variable, 33
– call-by-reference, 32
– formal variable, 33
program context, VIII, 4, 13
– path condition, 14
– state, 13
– state condition, 13
program verification, 8
properties
– symbolic computations, 3

read statement, 2
receive, 82
recurrence, 4, 21, 92, 95, 96, 100
– boundary condition, 4
– closed form, 22–25
– condition, 22, 25
– geometric recurrence variable, 93
– polynomial recurrence variable, 93
– relation, 4
ρ, 27

send, 82
simplify
– γ expressions, 19
– $\oplus$-chains, 27
simplify (in)equalities, 67
simplify constraints, 68
software reuse, 8
software specialization, 8
source code translation, 91
statement instance, 84
sub graph, 118
subroutine, 33
substitution
– backward, 31
– forward, 16
succs(u), 118
SUIF compiler, 10
suprenum, 29, 37
symbolic
– program testing, 8
symbolic expression, 54
– comparison, 55
– compute lower/upper bound, 56
– integer-valued expression, 54
– lower bound, 55
– rewrite, 57, 58
– simplify, 60
– upper bound, 55
symbolic index, 27
symbolic pointer, 34
symbolic simplifications, 3
symbolic sum, 63, 64, 73
symbolic variable binding, 1, 2
system of constraints, 61
– count solutions, 61

task parallelism, 80
θ, 27

undefined
– array state $\perp_n$, 26
up function, 54

Vienna Fortran, 80
Vienna High Performance Compiler
– VFC, 80, 85

WIEN95, 93
work distribution, 73