Analysis of induction variables using chains of recurrences: extensions

Sébastian Pop
pop@icps.u-strasbg.fr

LSIIT UMR CNRS 7005
Université Louis Pasteur
Strasbourg, FRANCE

Reviewer: Mohamed Tajine

Advisors: Philippe Clauss,
Vincent Loechner

Résumé

L'analyse statique d'un programme consiste en l'approximation conservatrice des propriétés d'un programme sans l'exécuter. Plusieurs approches ont été proposées pour l'extraction d'informations de différentes représentations d'un programme. L'efficacité d'un analyseur est souvent déterminée par le langage source et par le langage d'extraction.

Dans ce mémoire, nous nous sommes intéressés à l'analyse statique (c'est-à-dire sans execution) de l'évolution des variables scalaires dans les boucles des programmes impératifs. Cette évolution a été modélisée par des chaînes de récurrences que nous avons étendu aux fonctions périodiques, et aux approximations de fonctions à l'aide des enveloppes d'évolution. L'analyse est décrite sous la forme d'un algorithme classique d'analyse de flot de données, puis l'algorithme a été raffiné en une version plus efficace qui utilise la représentation d'assignation unique d'un programme.

Rapporteur: Mohamed Tajine

Encadrement: Philippe Clauss,
Vincent Loechner

Remerciements:

Ce stage de DEA a été effectué du 3 février au 30 juin 2003 dans l'équipe Image et Calcul Parallèle Scientifique (ICPS) du Laboratoire des Sciences de l'Image, de l'Informatique et de la Télédétection (LSIIT) de l'Université Louis Pasteur de Strasbourg.

Je remercie Philippe Clauss et Vincent Loechner d'avoir encadré ce stage, Mohamed Tajine d'avoir rapporté ce mémoire, et toute l'équipe de l'ICPS pour son accueil.

Abstract

The static analysis of a program consists in safely approximating run time properties of a program without executing it. Many approaches have been proposed for extracting information from different representations of a program. The efficiency of an analyzer often stands in the source language and in the target language.

In this report we are interested in the static analysis of the evolution of scalar variables in the loop structures of imperative programs. The evolution is modeled with chains of recurrences. We give two extensions of the chains of recurrences, for handling the periodic functions, and for approximating the evolution of a variable via the envelopes of evolution. The analysis is described using a classic data flow algorithm, and then refined into a more efficient algorithm using the static single assignment intermediate representation of a program.

Contents

Chapter 1

Introduction

Determining the behavior of a software system before its execution is a challenge that has many applications in the software verification, certification and program optimization.

Program analysis provides formal techniques for safely approximating some of the run time properties. One of the difficult task that the program analyzers have to deal with, is the interaction of the analyzed program with a complex environment. This constant interaction introduces unpredictable behaviors, that the analyzers have to safely approximate. The program analysis is based on a large set of formal methods. We shortly expose a reduced number of possible program analyzers in chapter 2.

Determining the set of possible values that a variable can take during the execution of a program is one of the most interesting property. This analysis allows, for example, a program verifier to warn about a possible violation of the ranges of a variable. This kind of analyzers have been constructed from the early days of the computer engineering, and these days they are used in the verification of critical software. Another possible use of such an analyzer is the validation of a program optimization.

In this thesis we describe a program analyzer that allows to safely approximate the evolution of a variable in a program. The analysis of a variable evolution has received a large amount of attention from the research community, from the fact that it is a crucial component in the software development and in the program optimization. An extensive literature has been published on the analysis of variable evolution. This thesis work describes a combination of two previous methods, one that analyzes the monotonic evolution of variables with distance information, and the other technique that represents evolution functions using the chains of recurrences, as well as an extension of the method to some sub cases that were not handled by the previous works.

In chapter 3 we expose the extensions that we have added to the chains of recurrences representation, then in chapter 4 we give the new algorithms used for the analysis of imperative programs. In the rest of this chapter we introduce the data flow algorithms, the structures used for program analysis in an imperative based compiler (the GNU Compiler Collection), and the difficulties of the implementation of robust and efficient program analyzers and optimizers.

1.1 Data flow algorithms

Under the name of data flow analyzers are classified, in general, all the algorithms that study the behavior of a program with respect to the data that the program processes. Among the classical analyzers that are taken as examples in every compiler textbook [3, 1] we could cite for example the reaching definition, or the availability.

In order to manipulate the data in a program, the programming languages define a naming method that allows the programmer to assign an abstract symbol to a data element. However, the naming conventions in a programming language does not guarantee the uniqueness properties. The programmer can use several symbolic names for designating a same element of data, or accessing several different data elements via the same symbolic name.

Because of the multiple symbolic names that can make reference to a same data element, a data analyzer needs the services of an *alias analyzer*, that helps disambiguating, when possible, the relation between symbolic names. When the alias information cannot be computed at compile time, the alias analyzer will choose a conservative answer.

1.2 Control flow algorithms

Another important aspect, when considering the study of a data flow analyzer, is the underlying language that represents the instructions that either modify the data, or specify the control flow of the program. In general there are two main control flow representations over which the data flow analyzers are built:

- The control flow graph (CFG) is composed of basic blocks that contain instructions executed from the first to the last. The basic blocks are linked by edges that represent the flow of control.

- The loop hierarchy tree (LHT) is based on the control flow graph and stores, for each loop, the set of basic blocks that constitutes the loop's body, as well as a relation of inclusion over the loops that relates a loop to its father (*i.e.* the enclosing loop) in a loop nest.

The LHT intermediate representation contains the results of a control flow analysis that detects the strongly connected components (SCCs) on the CFG [7, 5] (these SCCs are also known as natural loops).

The LHT representation stores a relation between the loops and the CFG, but at the creation of the LHT, the structure does not contain other information related to the evolution of the data in the loops. Other data analyzers have to infer information such as the induction variables (*i.e.* the variables that are redefined at each iteration of the loop), the number of iterations in a loop, the data dependences of scalars or memory locations, ...

It is possible to consider the information of the main induction variable and the number of iterations in a loop as the result of a syntactic analysis of the program. Obviously in a Fortran program like the following one:

```
DO i = 1, 100
  ...
ENDDO
```

a quick analysis of the abstract syntax tree is enough for determining the iteration domain of the loop [10].

However in the general case where the program can contain goto statements, the simple syntactic analysis is not powerful enough for the study of the program. The analyzer can produce erroneous results when an irregular control flow is not correctly handled in a loop nest. As an example, I'll take the "toy loop analyzer" Frédéric Wagner and I have written (two years ago) for the Nestor compiler [9]. Suppose that we have to handle the following code:

```
DO i = 1, 100
  ...
  IF (i.GT.3) GOTO 20
  ...
ENDDO
20 ...
```

In this case our toy analyzer does not scan for irregular control flow in the loop's body, and just relies on the information it gathers from the loop headers. Obviously, in this case, the "toy analyzer" is broken when it initializes the number of iterations to 100 for the current loop. We have fixed the analyzer in a conservative way by not allowing any further analysis or optimization on a nest that contains an irregular control flow. But such an assumption is too conservative for an real analyzer in a compiler.

1.3 A first characterization of array access functions

The accesses to an array are given by its *access functions*. The array dependence analyzer expects a canonical representation of the access functions to be associated to each access in the test. In an abstract syntax tree that comes directly from the parser, this property is not satisfied. Consider the following example:

```
for (i = 0, j = 0; i < 100; i++)
  {
    A[i] = B[j + i + j]
    j = j + 1;
  }
```

The access function for the array B will be translated by the parser into an abstract syntax tree that will represent "j+i+j". Constructing an array dependence tester that study this kind of expressions is not realistic:

- first, the access function is not folded, and thus “i + 2*j” would be its canonical folded representation,
- then, the expression contains two variables, i and j, that follow exactly the same evolution: both variables start at zero and are incremented by one at each iteration in the loop. Thus the canonical representation of the access function for B would be “3*i”.

This gives the main reasons to consider the study of the access functions separately from the dependence tester.

An analyzer that canonicalize access functions reveals to be very difficult to write, as I have experienced in my first attempts. The role of this analyzer was to extend the standard folder of GCC that works only on constants, to reconstruct and to fold polynomial expressions. (The analyzer was called texpr.c and implemented the basic arithmetics on polynomials, as well as a very basic expression analyzer.)

The difficulty in analyzing abstract syntax trees produced by the parser, comes from the fact that a programming language allows the programmer to combine the expressions. Without this flexibility, a language would be very difficult to use. Thus, the compiler has to transform the original program in a representation that avoids complex analyzers. The pass that simplifies the complexity coming from combinations is called the expression lowering pass. This kind of lowering can be found in several compilers, such as McCAT, MIPS-Pro - Open64, GCC, . . .

1.4 Simplification of the intermediate representations

The work on the expressions simplification in GCC has been based on the previous seminal work of Henderen in the McCAT compiler [8]. They proposed a representation called SIMPLE, whose purpose is to decompose difficult combinations of expressions and implicit execution evaluation semantics (for example the truth values evaluation), into a three address code proper to analyze.

Diego Novillo (my first advisor on GCC, Diego works for Red Hat Canada) and I have implemented the original framework for the simplification, while another important work for abstracting the representation from the underlying parsed language was performed by Jason Merrill (maintainer of the C++ front end of GCC from Red Hat). The resulting representations are:

GENERIC a language independent representation, that is able to represent Fortran 95, C, C++, ObjectiveC, and Java trees [7],

GIMPLE a three address simplified representation.

Based on the GIMPLE representation, it becomes realistic to implement robust and efficient program analyzers and optimizers.

I’m often referring to a software engineering course by M.R. Woodward that I have followed during my studies at the University of Liverpool. In one of his lectures about the Jackson Structured Programming, M.R. Woodward has given an analogy of the JSP in terms of the *lego* game (I'm not sure but I think that he quoted one of the examples given by M. Jackson). The purpose is to build a lego castle (the report of the analysis) starting with a lego ship (the unstructured data). The first step consists in decomposing the ship into basic bricks (the structured data), then to assemble, or compose the bricks to build the castle.

In the case of GIMPLE, we are decomposing the unstructured constructs of expressions into basic structured expressions that are simpler to compose into higher level representations by further analyzers.

The Open64 compiler adopts a quite different approach by lowering the expressions progressively. In particular, its the Loop Nest Optimizer works on a high level of the WHIRL representation that has not yet lowered the expressions [11].

Chapter 2

Symbolic program analysis

An extensive literature has been published on various methods for determining the effects of a program on its data [1]. As an introduction, we describe shortly the foundations of some of the techniques, and illustrate each technique with an example.

As we will see in this section, one of the most crucial point in any analysis framework is the way the information is encoded, *i.e.* the symbolic intermediate representations. The other key point is the way the algorithms direct their analysis.

2.1 Data flow analysis

The data flow analysis is based on the computation of a set of properties satisfied by a program. A program is represented as a graph where the nodes are the basic blocks, and the edges describe the possible control paths between the nodes [1, 3].

2.1.1 Computing over lattices

The information extracted from the program is represented as elements of a complete lattice $L = (\mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$. The elements of the lattice are partially ordered by $\sqsubseteq$, such that all subsets of the lattice have least upper bounds, as well as greatest lower bounds. $\perp$ represents the least element, and $\top$ represents the greatest element. The *meet* operator $\sqcap$ is used to combine the information coming from different control paths.

The effects of a program on a value $l_1 \in L$ are computed using the transfer function of the program $F : L \rightarrow L$, yielding to another value $l_2 \in L$. The function F is monotone:

$$l_1 \sqsubseteq l_2 \Rightarrow F(l_1) \sqsubseteq F(l_2)$$

The monotonicity of F translates the fact that, by reevaluating the information on a more precise subset l_1, we do not lose the information we had before the reevaluation, *i.e.* we obtain $F(l_1)$ that is more precise or the same as $F(l_2)$.

2.1.2 The forward data flow analysis

In a forward analysis, the information are computed by propagation from the entry basic block to the exit basic block.

On the entry of a basic block the information is computed using the meet operator over all the predecessors,

On the exit of a basic block the information is computed as the application of the transfer function on the entry initial conditions.

In other words, for each basic block we compute:

$$In(b) = \sqcap_{i \in Pred(b)} Out(i)$$

$$Out(b) = F_b(In(b))$$

where F_b represents the transfer function for the basic block b.

When the program contains loop structures, the evaluation of the properties in the basic block that represents the loop header depends on:

- the basic block that precedes the loop,
- and on the basic block that ends the loop's body.

The evaluation of the properties of the loop's body are computed as a fix point, *i.e.* the computation of the loop's transfer function will stop when the set of information gathered in the loop stabilizes.

The transfer function is monotone and is computed over a finite lattice, which guarantees the termination of the analysis for loop structures. The analysis can be stopped at any step of the iterative search of the fix point, since the computed values are conservative approximations of the program's real properties. This allows computations on infinite lattices as well.

In a classic data flow algorithm, the properties are usually stored in bit-arrays. The corresponding lattice is a hypercube whose dimension is the size of the bit-array.

The data flow algorithm is generally organized around a work list, that records the set of basic blocks for which the initial conditions have changed. The properties on exit of these basic blocks have to be evaluated again, and thus they are scheduled in the work list.

A symmetrical algorithm exists for backward data flow analysis, where the interpretation of the program starts from the exit basic block, and uses $\sqcup_{Succ(b)}$ for computing the information on exit of a basic block in function of the successors.

2.2 Constraints based analysis

The object of a constraint based analyzer is the determination of a system of constraints that describe the behavior of the program.

For example, during the verification of a program, one could be interested in the domain of possible values a variable is susceptible to take. One of the earliest works on the constraint based analysis is that of Cousot and Halbwachs [12] in 1978. They represent the evolution of a variable in the analyzed program by a system of linear constraints modeled by polyhedral domains. They describe a fix point iterative algorithm based on convex unions of polyhedra.

The constraint based analyzers are closely related to the *abstract interpretation* techniques.

2.3 Model checking techniques

In the model checking analysis of a program, the behavior of a program is modeled, for example, by a graph whose nodes are memory states of the program, and whose edges represent possible transitions between these states. Since the determination of the exact path of execution is sometimes an undecidable property, one has to study the evolution of the program over a possibly infinite number of worlds (in the case of program analysis, a world could be a memory state) reachable from the entry points [13]. (In general reactive systems $\mathcal{S} :< \Sigma, \Theta, \mathcal{R} >$ are described using Kripke models that are constituted of a set of worlds Σ, a set of entry worlds Θ, and a set of transitions between these worlds $\mathcal{R}$.)

Almost any code analysis can be described in terms of model checking, by searching the interesting properties over the abstract representation of the program. However, the main reason why they are not commonly used in compilers is that they are very costly in terms of memory space and execution time. Instead, the model checking is extensively used in verification tools that can afford huge latencies.

As an example of program analysis that could be implemented as a model checker, I will present the alias analysis, on which I have written some sketches some time ago, based on the course of "Knowledge Representation and Reasoning: Modal and Description Logics" taught by Ullrich Hustadt, and "Formal Methods" taught by Michael Fisher at the University of Liverpool.

As we shall see the alias analyzer is based on the model checking of the program, while the queries are written using the modal logic. Thus we shortly introduce the modal logic, followed by our algorithm for the alias analyzer.

2.3.1 Modal logics

The modal logics are a formal way of handling notions of necessity, possibility, knowledge, time, etc. The modal logics give an alternative to the first-order logic, in the sense that modal logics are different in expressive power, and are more natural to work with.

The modal logics are built over the formulae of the propositional logic and introduce two or more modal operators. In the case of more than two modal operators the logic is called multi-modal. Otherwise the modal operators are represented by:

a box $\Box$ whose meaning is: for all the worlds, the proposition quantified by $\Box$ is true,

a diamond $\Diamond$ whose meaning is: there exists a world in which the proposition quantified by $\Diamond$ is true.

If φ is a proposition, then the quantification could stand for:

$\Box\varphi$	$\Diamond\varphi$
φ is necessary	φ is possible
φ is known	$\neg\varphi$ is not known
φ is believed	$\neg\varphi$ is not believed
φ is obligatory	φ is permitted
φ will always be true	φ will be true at some future time

Depending on the intended meaning of the modal operators $\Box$ and $\Diamond$, we obtain a variety of modal logics.

We will show how to reason in terms of modal logics when analyzing the aliasing information in a program.

2.3.2 The alias analysis

The alias analysis is a classic technique [3] used in compilers for determining the set of symbolic names that may access the same storage location. In this analysis, we are interested in two main properties :

The must point-to sets record for two given variables the necessity that at any point in the program they must refer to the same storage location.

The may point-to sets record for two given variables the possibility that at a given point in the program they may refer to the same storage location.

Based on this definition, the *must point-to* propositions will be quantified by the $\Box$ modal operator, while the *may point-to* will be quantified by the $\Diamond$ operator.

In the classic alias analysis, the compiler records the alias information into lists of aliases, from which it extracts the information every time a client asks about the relation between two variables. In other terms, if we forget about the efficiency of the implementation, the compiler performs a proof for every question the client pass is asking. Thus given a question encoded under the form of modal formulae:

- "is it true that $\Box(a\ alias\ b)$?", the compiler has to prove that for all the possible states of the program during its execution, the property $(a\ alias\ b)$ is true,
- "is it possible that $\Diamond(a\ alias\ b)$?", the compiler has to prove that there exists at least one reachable state of the execution of the program in which the property $(a\ alias\ b)$ becomes true.

Now we should explain how the compiler directs its proofs based on the abstract syntax trees that represent the program to be analyzed.

2.3.3 Proving modal formulae over a program

A compiler transforms the original program from the abstract syntax tree representation into a set of *possible* execution traces. When the program contains iterative structures (*i.e.* loops), the set of possible traces can be infinite: that happens when the number of iterations in a loop is not computable. In such a case, only a finite number of traces are selected, and the analysis will be suboptimal, but conservative.

An interpreter uses the set of execution traces to build the set of possible worlds. Each possible world contains a set of relations under the form: (a *alias* b).

The model checker works on this set of possible worlds. Thus when studying a may alias proposition: $\diamond$(a *alias* b), the model checker verifies whether the property (a *alias* b) exists in a possible world, and stops when it finds one such world. When there are infinite number of worlds, the property could be undecidable, and thus the conservative answer is output: it is possible that *a may points-to b*.

The same scenario happens for the propositions quantified by $\Box$. It is possible to transform the property $\Box\varphi$ into $\neg\diamond(\neg\varphi)$, then verify that there does not exist a world in which the property φ does not hold.

In the model checking technique, we have seen that a compiler $\mathcal{C}$ transforms the Abstract Syntax Tree (AST) into sets of execution traces. This step consists in building a *model* of the program. Then, from this model, an interpreter $\mathcal{I}$ extracts the alias information into a set of possible worlds. A model checker uses this set of possible worlds to prove propositions quantified by modal operators, by enumeratively checking in every world whether the property holds or not.

Another approach to the alias analysis using a model checker is that of Martena and San Pietro [14]. They use a syntax based compiler to translate C programs into the Promela language, then they extract the alias information based on the results of a Spin simulation.

David A. Schmidt describes in [15] the relation between the data flow analysis and abstract interpretation with the help of model checking. The collecting semantics is expressed using formulae from the modal μ-calculus.

2.4 Abstract interpretation

Since the exact properties of a program are sometimes undecidable, the abstract interpretation defines a way to depict the exact semantics of a program using the semantics of an abstract set of symbols that represent a safe approximation of the properties of the program. The connections between the "real world" and the "sketch world" are defined in terms of an *abstraction* α and a *concretization* γ functions, and are known under the name of *Galois connections* [1]. A Galois connection is then represented by these two functions between the corresponding lattices: $(L_1, \alpha, \gamma, L_2)$.

The theory of abstract interpretation defines several automatic methods for constructing new program analyzers by compositions of Galois connections [1]. This is one of the most powerful techniques since it allows the reuse of basic analyzers in more complex ones by composition. The interface between two of these modules is the definition of a Galois connection.

Another promising technique is that of logic functors described by Sébastien Ferré and Olivier Ridoux in [34, 35]. They propose to construct customized logics by composition of basic logics with the help of a unique interface.

2.4.1 Evolution of variables by finite differentiations

Mohammad R. Haghighat has presented in [38] an algorithm based on abstract interpretation for detecting the exact evolution function for the variables in a program. He defines an abstract interpreter that transforms the semantics of a program into an abstract syntax. Strongly connected components of the control flow graph are detected as natural loops, and are processed apart. The analysis transforms loops into a system of recurrence relations (see Appendix C.1), that are solved by the method of finite differences and symbolic interpolation. The algorithm interprets the loop body a fixed number of times, and constructs a table of differences. Then based on this difference table, the algorithm deduces, by symbolic interpolation (see section 3.5), the exact characteristic function that describes the behavior of each variable in function of the main induction variable of the loop.

Robert van Engelen has shown in [17] that Haghighat's analysis algorithm could produce wrong results, if the number of interpretations of the loop body is set too low. He then presents an algorithm that represents the recurrence relations using the chains of recurrences. The chains of recurrences allows then the deduction of the closed form functions using a rewriting system (see Appendix C).

In this chapter we have seen a set of tools that could be used for the static analysis of a program. We have also stressed the importance of the intermediate languages used for representing a program, and for extracting the information from the program, as well as the importance of the way the analyzer conducts the information extraction.

In the next chapter, we shall extend the chains of recurrences for allowing the description of periodic functions, and for allowing to approximate evolutions in the case of uncomputable properties.

Then we give a modified version of the monotonic evolution algorithm [28] based on a more precise representation of the evolution functions using the envelope chains of recurrences.

Chapter 3

Chains of recurrences

Chains of recurrences (chrecs) were firstly proposed for evaluating closed form functions at regular intervals [23, 24]. The chains of recurrences were used by Robert van Engelen [17, 18, 16] in an algorithm that analyzes induction variables. In these previous works, there was no way to represent periodic functions, and thus it was impossible to express the properties of flip-flop operations in a loop.

In this chapter we extend the chains of recurrences to handle the case of periodic chains of recurrences. Then, we propose the envelope chains of recurrences for safely approximating the uncomputable properties of a program, such as the *zero-trip loop problem*, or the undecidable execution paths in a conditional expression.

The basic properties described by the previous works on this subject are left in Appendix C, the current chapter should present only the new contributions.

3.1 Periodic chains of recurrences: pchrecs

In terms of induction variables, periodicity is introduced in a loop by flip-flop operations, as described by Michael Wolfe in [21]. The usual operations over the chains of recurrences are extended to the periodic chains of recurrences.

3.1.1 Periodic functions

Definition 1 (periodic number) *A one dimensional periodic number*

$$u(n) = [u_0, u_1, \ldots, u_{p-1}]_n$$

is equal to the item whose rank is equal to n mod p. p is called the period of $u(n)$*.*

$$u(n) = \begin{cases} u_0 & \textit{if } n = 0 \textit{ mod } p \\ u_1 & \textit{if } n = 1 \textit{ mod } p \\ \ldots & \\ u_{p-1} & \textit{if } n = p-1 \textit{ mod } p \end{cases}$$

Definition 2 (pseudo-polynomial) *A pseudo-polynomial is a polynomial function whose coefficients are periodic numbers.*

Another representation of the pseudo-polynomials is where each value of the period is a polynomial:

$$Q(n) = \begin{cases} q_0 & \text{if } n = 0 \ mod \ p \\ q_1 & \text{if } n = 1 \ mod \ p \\ \dots & \\ q_{p-1} & \text{if } n = p-1 \ mod \ p \end{cases}$$

with $q_0, q_1, \dots, q_{p-1}$ polynomials.

We define the periodic chains of recurrences (pchrecs) based on these two representations of the pseudo-polynomials. As a shortcut we write:

$$pchrec = |chrec_0, chrec_1, ..., chrec_{p-1}|$$

for a periodic chain of recurrences of period p.

3.1.2 Operations on pchrecs

The operations on the pchrecs are defined as on the pseudo-polynomials [27]: the result of a binary operation on periodic functions has a period equal to the least common multiple of their periods. In the general case a binary operation is defined as follows:

$$|a_0, a_1, ..., a_{p-1}| \odot |b_0, b_1, ..., b_{q-1}| = |c_0, c_1, ..., c_{m-1}|$$

with $m = lcm(p, q)$, $\odot$ a binary operation, and $c_k = a_{k \ mod \ p} \odot b_{k \ mod \ q}$. The elements a_i and b_j are chains of recurrences, and thus it is possible to have again, at this level, pchrecs, or any type of chrec.

A unary operation is defined as follows:

$$f(|a_0, a_1, ..., a_{p-1}|) = |f(a_0), f(a_1), ..., f(a_{p-1})|$$

with f a unary operation, and the elements a_i are chains of recurrences.

3.2 Envelopes of chains of recurrences: echrecs

The algorithm on monotonic evolution by Peng Wu and Albert Cohen [28] used what they call minimal and maximal distance information. The minimal distance information records the minimal evolution of a variable along a path from one use of the variable to another use. They extended their original algorithm with the distance information for being able to represent conditional expressions, as well as loop level jumps, such as the continue statements. If we look more closely, their new version of the algorithm uses the arithmetics on intervals in their abstract interpreter.

If we want to use the chains of recurrences interpreter in their algorithm, we would need a similar mechanism for handling intervals of evolutions. In other words we would like to write for $\lhd_1^8$:

$$\{[-\infty, +\infty], +, [1, 8]\}$$

that is a chain of recurrence whose initial condition is unknown, and that follows an increasing undetermined evolution that could vary between 1 and 8.

This observation was the basis for considering the chains of recurrences over the intervals, instead of over the classic commutative rings [25] (Appendix B gives a definition of the lattice of intervals).

3.2.1 A formal definition

Definition 3 (Envelope of chains of recurrences) *An envelope of chains of recurrences is a chain of recurrence over the intervals.*

Envelope chains of recurrences (echrecs) are enough descriptive to represent a set of possible evolutions.

The operations over the echrecs are not different of the operations over classical chrecs, and, at the elements level, the operations use the arithmetic on intervals.

3.2.2 Application of echrecs to program analysis

Based on the echrecs, it is possible to make the monotonic evolution algorithm work on the general case which comprise polynomials, exponentials, and periodic functions. Note that the monotonic evolution algorithm extended to minimal and maximal distance information, proposed by Peng Wu and Albert Cohen [28, 30], does not register initial conditions, and does not record the exact evolution function of a variable. However the initial value of a variable at the entry point in a loop can determine its variation in the case where its evolution is exponential.

As an illustration of echrecs for the program analysis, consider the following example (the expressions between sharp signs constitute the abstract information gathered by the analyzer):

```
i = 3
loop
  if (volatile_variable != 0)
    i = i + 2
  else
    i = i + 1
  endif
endloop
```

```
First propagation:
#i = [-oo, +oo]#
i = 3
#i = {[3, 3]}#
loop
  #i = {[3, 3], +, [0, 0]}#
  if (volatile_variable != 0)
    #i = {[3, 3], +, [0, 0]}#
    i = i + 2
    #i = {[3, 5], +, [0, 2]}#
  else
    #i = {[3, 3], +, [0, 0]}#
    i = i + 1
    #i = {[3, 4], +, [0, 1]}#
  endif
  #i = {[4, 5], +, [1, 2]}#
endloop
----
Second propagation:
loop
  #i = {[3, 3], +, [1, 2]}#
  if (volatile_variable != 0)
    #i = {[3, 3], +, [1, 2]}#
    i = i + 2
    #i = {[3, 5], +, [1, 2]}#
  else
    #i = {[3, 3], +, [1, 2]}#
    i = i + 1
    #i = {[3, 4], +, [1, 2]}#
  endif
  #i = {[4, 5], +, [1, 2]}#
endloop
```

Note that if the exit condition depends on the evolution of i, the analyzer would not be able to determine at compile time the exact number of iterations the loop would run. This problem is known under the name of *zero-trip loop problem*. In this case it is possible, as in the Peng Wu's monotonic evolution, to approximate with the help of an interval, the number of iterations. The use of an approximation for the number of iterations solves the problems of finding the exit value of a variable from a loop, and thus it makes possible to approximatively analyze a loop nest even when the loop bounds are not known.

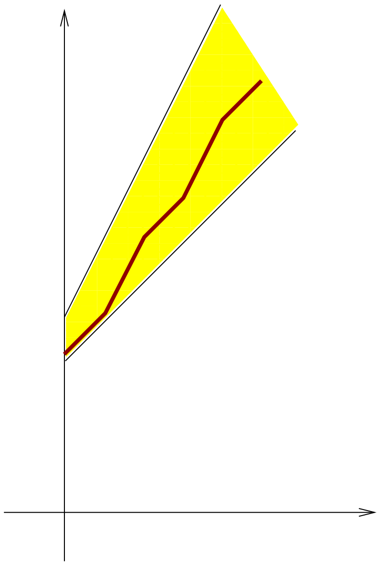

Figure 3.1: Envelope and a possible evolution path for $\{[4, 5], +, [1, 2]\}$.

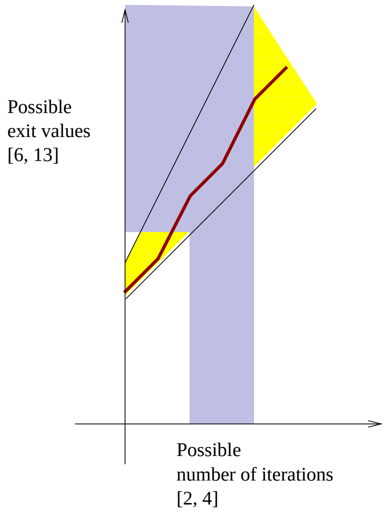

Figure 3.2: Possible exit values for possible number of iterations.

3.3 Using chains of recurrences in program analysis

When analyzing a program, some of the properties could not be computable, and thus the compiler has to safely approximate these properties. For this reason it is important to take the elements of the chains of recurrences over a complete lattice (see section 2.1.1). If we take the chains of recurrences over the classic commutative rings [25], we need two more symbols to represent

- the tautology: $\top$ for the leak of information, and
- the contradiction: $\perp$ for contradictory information.

This constitutes a three level lattice as the one used in the constant propagation algorithm proposed by Mark Wegman and Kenneth Zadeck [32], (see Figure 3.3).

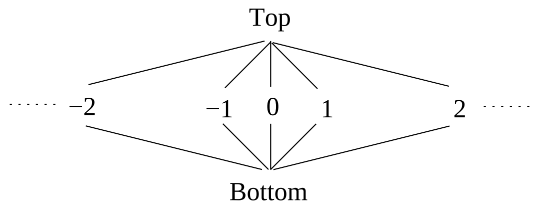

Figure 3.3: Lattice of integer constants

3.4 Factorization of chains of recurrences over the integers

In this section we consider only the case of univariate chains of recurrences over the integers. We start by considering the symbolic multiplication, and deduce a factorization algorithm based on the syntactic properties of the chains of recurrences.

Kaltofen has given in 1982 a survey of the factorization of integer polynomials [33].

Factorization of polynomial functions over the integers has several applications in program analysis. One of them is to detect from a linearized array access function the possible access functions of the original multi dimensional array.

3.4.1 Integer factorization

The simplest case of the multiplication is when both operands are integer constants. The corresponding case in the factorization algorithm is when we have to factor an integer constant.

One of the algorithms for factoring integers is to test the divisibility by all the prime integers less or equal to the square root. This algorithm can be found in the section 4.5.4 of TAOCP [2].

3.4.2 One factor is an integer constant

Another simple case is when one of the multiplication operands is an integer constant:

$$a * \{b, +, c\} \rightarrow \{a * b, +, a * c\}$$

in which case the factorization algorithm is given an expression under the form $\{x, +, y\}$. The algorithm is as follows:

```
Input: A chrec {x, +, y}
Output: A constant chrec {a}, and a chrec {b, +, c}
        such that {a} * {b, +, c} = {x, +, y}.

Step1: Factorize the coefficients x, and y,
       and keep their factors into the lists lx, and ly.

Step2: for each factor a in lx
         if a belongs to ly
           then return {a}, and {x/a, +, y/a}.
         end if
       end for
       if none of the factors of x is a factor of y,
         then return {1}, and {x, +, y}.
```

3.4.3 Two chains of recurrences

In the case where both operands are chains of recurrences, the symbolic evaluation of the multiplication gives:

$$\{a, +, b\} * \{c, +, d\} \rightarrow$$

$$\rightarrow \{a * c, +, a * d + b * c + b * d, +, 2 * b * d\}$$

This constitutes the basis for a syntactic algorithm to factorize the chain of recurrences that are under the form $\{x, +, y, +, z\}$. The procedure is as follows:

```
Input: A chrec {x, +, y, +, z}
Output: Two chrecs {a, +, b} and {c, +, d}
        such that {a, +, b} * {c, +, d} = {x, +, y, +, z},
        or {a} and {b, +, c, +, d}
        such that {a} * {b, +, c, +, d} = {x, +, y, +, z}.

Step1: if 2 is a factor of z,
         then z := z / 2
         else go to Step4
       end if

Step2: Factorize the coefficients x, and z,
       and keep their factors into the lists lx, and lz.

Step3: for each factor b in lz
         d := z / b
         for each factor a in lx
           c := x / a
           if y is equal to a*d + b*c + b*d
             then return {a, +, b} and {c, +, d}
           end if
         end for
       end for

Step4: Factorize the coefficient y,
       and keep its factors into the list ly.
       for each factor a in lx
         if a belongs to ly,
         and a belongs to lz,
           then return {a} and {x/a, +, y/a, +, z/a}
         end if
       end for
       return {1} and {x, +, y, +, z}
```

The algorithm factors all the coefficients of the chain of recurrence, and then matches against the template expression given by the symbolic evaluation of the multiplication. Thus the lists of factors lx, ly, and lz are constructed recursively by calling the same algorithm.

The complexity in the worst case is $O(n^3)$ in the number of factors of the coefficients.

3.5 From the finite differences to the chains of recurrences

In this section we show the relation between the finite differences technique and the chains of recurrences.

The finite differences technique for interpolating a polynomial function from a sequence a_n of integers consists in building a difference table. The elements of the forward difference table are:

- at the first differentiation $b_n = \Delta_n = a_{n+1} - a_n$,
- at the second differentiation $c_n = \Delta_n^2 = b_{n+1} - b_n$, and so on.

For example, for the sequence $a_n = 96, 726, 2556, 6216, 12336, 21546$, the corresponding difference table is:

k	0	1	2	3	4	5
Δ_k^0	96	726	2556	6216	12336	21546
Δ_k^1	630	1830	3660	6120	9210	
Δ_k^2	1200	1830	2460	3090		
Δ_k^3	630	630	630			
Δ_k^4	0	0				

The function that generates the a_n sequence is interpolated using the Newton's forward formula:

$$f(n) = \sum_{i=0}^{p} \Delta_0^i \binom{n}{i} = a_0 + b_0 n + c_0 \frac{n(n-1)}{2} + d_0 \frac{n(n-1)(n-2)}{2 \cdot 3} + \ldots$$

Applied in the previous example, we get:

$$f(n) = 96 + 630n + 1200\frac{n(n-1)}{2} + 630\frac{n(n-1)(n-2)}{2 \cdot 3}$$

$$f(n) = 96 + 240n + 285n^2 + 105n^3$$

Now if we write this function under the chain of recurrences syntax, we get:

$$\{96, +, 630, +, 1200, +, 630\}$$

that is exactly the first column in the difference table. This property of the pure-sum chains of recurrences has been described by Zima in [25], under the form:

$$\{\phi_0, +, \phi_1, +, \ldots, +, \phi_p\} = \phi_0 + \phi_1 \binom{n}{1} + \phi_2 \binom{n}{2} + \ldots + \phi_p \binom{n}{p}$$

This tight relation between the pure-sum chains of recurrences and the finite difference technique shows that the foundations of the algorithms for induction variables described by Mohammad Haghighat in [38], and Robert Van Engelen in [17] are common. The differences between these two algorithms are based only on the way the analyzers proceed in the interpretation of the program.

For the analysis of exponential functions (*i.e.* pure product chains of recurrences), instead of constructing a difference table, we have to construct a division table. If we allow the mix of difference and division lines in the table, then we obtain an expressive power equivalent to the general chains of recurrences.

3.6 Possible extensions

3.6.1 Derivative of a chrec

Under the coefficients of a chrec stands the property that a function is completely determined by its integration coefficients.

After having observed that the last coefficient of a pure sum chain of recurrence of degree n is always equal to its n-th derivative (*i.e.* its integration constant for the polynomial of degree 0), and that the first coefficient of the chrec is the integration constant for the degree n, I started to investigate on the meaning of the coefficients of a chrec. This work is still not finished, and thus a not so formal description of the properties of the chains of recurrences are described below.

The original question was, is it possible to extract the integration coefficients from the coefficients of a chrec? By answering this question it is possible to deduce an algorithm for deriving and integrating the chains of recurrences.

For example, the evolution of the polynomial

$$x^7 + 2x^6 + 3x^5 + 4x^4 + 5x^3 + 6x^2 + 7x + 8$$

is embedded into the initial conditions of integration

$$5040, 1440, 360, 96, 30, 12, 7, 8$$

The same polynomial is represented as

$$\{8, +, 28, +, 438, +, 3510, +, 12336, +, 20760, +, 16560, +, 5040\}$$

under the chrec notation. The first coefficient of the chrec, 8, is the integration coefficient for the degree 7, while the right hand side represent a combination of the other integration coefficients. The last coefficient, 5040, represent the integration coefficient for the degree 0 of the polynomial.

The successive derivatives of the above polynomial are:

$$\begin{array}{ccccccccccccccc}
\{8 & ,+, & 28 & ,+, & 438 & ,+, & 3510 & ,+, & 12336 & ,+, & 20760 & ,+, & 16560 & ,+, & 5040\} \\
 & & \{7 & ,+, & 77 & ,+, & 1130 & ,+, & 6216 & ,+, & 14160 & ,+, & 14040 & ,+, & 5040\} \\
 & & & & \{12 & ,+, & 240 & ,+, & 2556 & ,+, & 8820 & ,+, & 11520 & ,+, & 5040\} \\
 & & & & & & \{30 & ,+, & 726 & ,+, & 4740 & ,+, & 9000 & ,+, & 5040\} \\
 & & & & & & & & \{96 & ,+, & 1920 & ,+, & 6480 & ,+, & 5040\} \\
 & & & & & & & & & & \{360 & ,+, & 3960 & ,+, & 5040\} \\
 & & & & & & & & & & & & \{1440 & ,+, & 5040\} \\
 & & & & & & & & & & & & & & \{5040\}
\end{array}$$

The derivative of a chain of recurrence $\Phi = \{\phi_0, +, \Phi_1\}$ is computed by determining the coefficients of the derivative from the right hand side Φ_1.

The integration of a chain of recurrence consists in determining the coefficients for one level above, and this requires an integer as a parameter: the left hand side. This integration coefficient does not perturb the evolution of the rest of the chain.

For a chain of recurrence $\{\phi_0, +, \phi_1, +, \ldots, +, \phi_n\}$, the coefficients are given by:

$$\begin{aligned}
\phi_0 &= \{int(n)\}, \\
&\vdots \\
\phi_{n-k} &= \{int(k), +, \sum_{i=1}^{k} \frac{int(k-i)}{(i+1)!}, +, \ldots, +, \frac{int(0)}{2^k}\}, \\
&\vdots \\
\phi_{n-3} &= \{int(3), +, \frac{12int(2) + 2int(1) + int(0)}{24}, +, \frac{3int(1) + int(0)}{6}, +, \frac{int(0)}{8}\}, \\
\phi_{n-2} &= \{int(2), +, \frac{3int(1) + int(0)}{6}, +, \frac{int(0)}{4}\}, \\
\phi_{n-1} &= \{int(1), +, \frac{int(0)}{2}\}, \\
\phi_n &= \{int(0)\}
\end{aligned}$$

where $int(k)$ is the k-th integration coefficient. Each coefficient of a chain of recurrence is a function of the integration coefficients.

The general term $\sum_{i=1}^{k} \frac{int(k-i)}{(i+1)!}$ looks like the Taylor series:

$$T(x) = \sum_{i=0}^{\infty} \frac{f^{(i)}(0)}{i!} x^i$$

however the other coefficients are more difficult to characterize, this is still a work in progress.

The proof for the general term of the coefficients of a chrec could be performed by finite differentiation of the function $\Phi = \{\phi_0, +, \phi_1, +, \ldots, +, \phi_n\}$ and its successive derivatives $\Phi^{(k)}$ for which we have shifted k times to the right the coefficients.

In the rest of this subsection, we shortly give a sketch of the integration algorithm, with the above description of the coefficients of a chrec being the key of the algorithm.

The first step before computing the integral or the derivative of a chain of recurrence, the initial conditions of integration have to be determined. These initial conditions are determined by firstly computing the $int(1)$ from the before last coefficient ϕ_{n-1} of the chrec, by evaluating

$$int(1) = \{\phi_{n-1}, +, -\frac{int(0)}{2}\}(n-1) = \phi_{n-1} - (n-1) \cdot \frac{int(0)}{2}$$

then determining the $int(2)$ from the chain of recurrence

$$int(2) = \{\phi_{n-2}, +, -\frac{3int(1) + int(0)}{6}, +, -\frac{int(0)}{4}\}(n-2)$$

and so on.

Derivation of chrecs

The derivation of a chain of recurrence $\{\phi_0, +, \phi_1, +, \ldots, +, \phi_n\}$ is performed by taking the right hand side of the chrec, $\{\phi_1, +, \ldots, +, \phi_n\}$, and then evaluating for each coefficient ϕ_k its instantiation at the $k-1$ level: *i.e.* $\phi_k(k-1)$, for $k = 1, 2, \ldots, n$.

Integration of chrecs

The integration of a chain of recurrence $\{\phi_0, +, \phi_1, +, \ldots, +, \phi_n\}$ is performed by evaluating for each coefficient, $\phi_k(k+1)$ for $k = 0, 1, 2, \ldots, n$, and then by adding the integration coefficient to the left side of the chrec.

3.6.2 Mixing operations on polynomial and exponential CRs

There are some difficult operations that were not addressed in the previous papers on chrecs. What is the result of the multiplication of a polynomial with an exponential function? Is it still a function that we can express using the chrecs?

A possible solution is to use a Taylor decomposition of the exponential function into an infinite sum of polynomials. Then split this infinite sum in two parts: one that gathers all the terms with a degree less or equal to the degree of the polynomial function the other containing all the terms with a degree strictly greater. The operation is then performed only on this first part. The result is the sum of a finite term polynomial function with the second part that is an infinite sum for which we have the general term expression.

I have not tried to implement this extension, but the general idea would be to express the result of the addition again as an exponential function.

Chapter 4

Determining the evolution of scalar variables

Now that we have settled the landscape of the possible techniques for analyzing a program, we describe in more details an algorithm for detecting the evolution functions for scalar induction variables.

Determining the evolution of a variable is of paramount importance in a compiler. Based on this information it is possible to deduce properties such as the number of iterations of a loop, the dependence relations between memory accesses, or more aggressively, express the evolution of a variable in function of the main induction variable of the loop, exposing more parallelism in the vectorization pass.

In the first time, I give a simple algorithm that is able to determine the evolution of a variable in some cases. I discuss the weaknesses of this basic algorithm, and deduce a more efficient algorithm. Then, I describe the minimal algorithm that allows the detection of the main induction variable, making it possible to determine the number of iterations, the aim being to reduce the compilation time spent in the analysis. Finally, I will expose the much more difficult cases as a theoretical extent in the last section.

4.1 Determining the evolution with a basic data flow algorithm

This algorithm uses the loop hierarchy tree (LHT) for detecting the loop nests, and then focuses on the study of the induction variable for a loop nest.

The algorithm follows the lines of the algorithm described by Peng Wu and Albert Cohen [29, 30, 31, 28] for analyzing the affine monotonicity of variables' evolution, and the intermediate representation of the chains of recurrences described by Eugene Zima [23].

First step the interpretation of the loop nest begins from the outermost loop level, and propagates the information in the loop's body following the control flow graph (CFG) in which all the back edges were excluded. The algorithm associates with each variable (could be a version name in the case of the static single assignment representation) a symbolic function following the envelope chains of recurrence syntax. At the entry point of a loop, the right hand side of the chain of recurrence grows by one in the new dimension indexed by the number of the loop, and is initialized (*i.e.* $\{left, +, [0, 0]\}$, or $\{left, *, [1, 1]\}$). The $left$ part could be a multivariate chain of recurrence that represent all the parent loops indexes.

At each step, the interpreter keeps a list of variables for which the evolution has changed (*i.e.* the right hand side of the chain of recurrence has been modified). The left hand side is modified for keeping track of the modifications on the variable from the loop header to the current point. This left hand side information is not merged during the reevaluation of the loop.

Second step if we have modified the evolution part for one of the variables, then merge the information obtained in end of the loop with the initial conditions in entry of the loop, and restart the first step. Otherwise the algorithm has reached its stability point.

If the evolution of a variable changes at a step i, then at the step $i + 1$, we have to update the information of all the variables that were defined as a function of the first one. This inefficiency of the basic algorithm is exposed in Example 4.

For a more efficient algorithm we should direct the analysis to first analyze the evolution of the variables that does not depend on any other, then successively study the variables that depend only on the variables for which the evolution is known. As we will see, the schedule of the analyzer can be produced by topologically sorting (see Appendix A.2) the scalar dependence graph (see Appendix A.1) for the entire loop nest. The schedule constitutes the basis of the more efficient algorithm exposed in the next section.

Also to be noted is that if we had to analyze programs with cyclic scalar dependences with diameters larger than two (that could represent in some cases flip-flop operations, or mixers, see sections 4.5 and 4.6), it would be a quite tedious task based on the interpreter we had described in this section. This is one more reason to separate the scalar dependence analysis from the current interpreter.

In the case of the algorithm proposed by Robert van Engelen [18, 17, 16], the topological sort of the scalar dependence graph is embedded in the work list of assignments that are analyzed. The idea of topologically sorting the SDG comes from these previous works. However the separation of the SDG from the list of assignments to be analyzed is important in the case we want to detect more difficult cases, such as the flip-flop operations, or the difficult cases that we expose in section 4.6.

4.2 The improved data flow algorithm

We have given a rough idea of how the algorithm could be improved by directing the analyzer to analyze the variables of the loop nest. The improved algorithm that we obtain by including the scheduler is:

First step Produce an analysis schedule for the entire loop nest. The scalar dependence graph (SDG) is built and is topologically sorted. The result may be either a list of variables that should be sequentially analyzed, or a list of sets of variables that can be analyzed in parallel due to the independence of their evolutions. If the code is not under a static single assignment (SSA) form, then the extraction of parallel plans from the SDG could render the evolution analyzer more efficiently, by requiring less iterations over the loop nest. Under SSA, the one variable at a time schedule is acceptable since the evolution of a variable is studied by following the def-use links.

When the SDG contains cycles, they are extracted with all the variables that depend on these cycles. When a cycle belongs to a same loop level, the cycle is produced by a flip-flop operation, and is analyzed by the periodic specific algorithm. It could be the case that the cycle spans over multiple loop levels, in which case the evolution is much more difficult to characterize.

Second step Following the schedule, analyze all the levels of the scheduling list. At a given level, analyze only the evolution of the variables that belong to the level. Thus, the analyzer augments its knowledge about the program by following the schedule, and it stops when all the variables in the schedule are analyzed.

Third step Analyze the variables from the strongly connected components (SCC) of the SDG. We suppose that the first step postpones the analysis of the cyclic components until here. Thus the first thing to do is to determine the SCCs that does not depend on other variables than those for which we already know the evolution from the second step. This constitutes again a scheduling list, that is produced by topologically sorting the rest of the graph after having collapsed all the cycles into a single node: the in and out dependence edges are all linked to the symbolic node, while the cyclic dependence edges are saved and used in the next step of the analysis.

It could be the case that a SCC contains one or more SCCs, in which case the evolution of any variable of the SCC is difficult to describe. See section 4.6 for a short discussion on these strange objects.

The skeleton of the analyzer is as follows:

```
detect evolutions in a loop nest N =
  construct_SDD (body (N))
  (AG, CG) = extract_SCCs (SDD)
  For each variable V in AG, in topological order,
    analyze_evolution (V)
  if CG contains cycles of higher order, then the algorithm fails.
  For each cycle C in CG, in topological order
  (i.e. starting with the cycles with all dependences analyzed)
    construct_periodic_chrec (C)

extract_SCCs (SDD) =
  detect all SCCs in the SDD
  extract all the variables in cyclic dependences
  as well as all the variables that depend on them.
  Return
    an acyclic graph: AG,
    and a graph with cycles: CG.
```

4.3 The light weight version

One of the issues in any industrial compiler is the compilation speed. We will focus on this constraint in this section. The aim is to give a minimal algorithm for determining, when possible, the iteration domain, at a low compile time cost.

4.3.1 Selecting the candidate loops

A well formed loop has a single exit edge. If this condition is not verified, the analyzer avoids the analysis of the loop. This drastic condition discards all the loops that contains a break, or a "user goto" (*i.e.* a goto not introduced by the compiler) in their body, and that cannot be reduced to the normal form. This kind of loop are much more difficult to analyze correctly, and thus the early elimination of these loops will avoid the later difficult cases.

Then, the analyzer focus on the exit condition. If the exit condition makes use of unusual components, such as function calls, volatile variables, etc., then the loop is discarded as well. The ideal case is an exit condition that uses a combination of integer or real variables and constants.

The loops that have filtered until this stage are more seriously analyzed: we compute the scalar dependence graph (SDG) for the loop's body, and store it in the loop structure for later uses. If the underlying representation is the static single assignment, then it is possible to avoid the construction of the entire SDG by constructing a partial SDG that focuses only on the exit condition variables and their dependences. By following the def-use chains, the algorithm avoids the walking of the whole loop nest body.

Based on the SDG, it is possible to compute an analysis schedule (by partially topologically sorting the list of relations $a \to b$). If there exists cyclic components in the SDG that conflicts with the determination of a proper analysis schedule, then the loop is discarded. This belongs to one of the difficult cases that should not be analyzed by the fast algorithm. The schedule will allow the analysis of the variables whose evolution does not depend on any other, as well as the minimal set of variables to be analyzed in the loop for determining the evolution of the variables involved in the exit condition.

Consider for example the following GIMPLE code:

```
i = 3;
while (1)
  {
    if (i > 100)
      goto end;
    j = i + 3;
    i = i + 1;
  }
end:;
```

The first step is to compute the scalar dependence graph, that will contain the relation $j \rightarrow i$. Since the exit condition is based on the evolution of i, the only variable that the analyzer should take care of for determining the number of iterations the loop will run is i.

4.3.2 Determining the evolution of a variable

Starting the analysis with the first variable of the schedule guarantees that it does not depend on any other variable, then picking up the next variable of the schedule guarantees that this second variable will probably depend on the first variable, but on no other variable whose evolution hasn't been computed yet. Since we already know the evolution for the first variable, it is possible, in some cases, to compute an approximation of the evolution for the second variable, and so on.

The last variable in the schedule is a component of the exit condition. Thus, after having determined the evolution of all the variables in the schedule, we can try to compute the number of iterations for the current loop. The number of iterations is computed as the smallest number of iterations after which the main induction variable satisfies the exit condition.

4.3.3 An overview of the algorithm

The skeleton of the analyzer is as follows:

```
for each loop nest N in the LHT
  if N is a good candidate, then
    construct_SDD (body (N))
    for each loop L in N
      determine the exit conditions of L,
    end for
    determine the reduced analysis schedule
    for each variable V in the schedule
      analyze the evolution of V
    end for
    for each loop L in the loop nest N
      determine the number of iterations
    end for
  end if
end for
```

4.4 An SSA based algorithm

The static single assignment (SSA) representation (described for example in [3]) is also susceptible to improve the compile time with respect to the classic data flow analysis. We shortly describe the SSA specificities of the algorithm in this section.

The SSA version of the chains of recurrences analysis works on a variable at a time model. This is a characteristic that occurs in many conversions of iterative data flow algorithms to SSA (for example in the SSAPRE: the partial redundancy elimination based on SSA [36, 37]).

In the rest of this section, I describe the algorithm that analyze a variable V in the loop nest N:

Initialization step Retrieve the loop ϕ-node of the variable V in the current loop. The loop ϕ-node represents a merge point between the evolution of V in the outer loop (the constant part with respect to the current loop), and the evolution of V in the current loop nest.

If the loop does not contain a loop ϕ-node for the variable V, then V is constant in this loop, but can vary in the inner loops. Thus at this loop level, the evolution determined outside the loop is associated to the variable V. The exploration of the loop nest continues with the analysis of the next inner loop nest, and then with the siblings of the inner loop nest. For each of these loop nests, the *Initialization step* is reexecuted.

Analysis step When the loop contains a loop ϕ-node, it is possible to use the information from the SSA representation to navigate to the definitions of the variable.

The initial conditions are the first analyzed: *i.e.* the definition that is not contained in the current loop nest.

Then the update edge is taken to the bottom of the loop nest, and step after step the SSA-edges are followed until either:

- reaching again the ϕ-node from which the analysis has started,
- or reaching a node for which the evolution has already been determined.

This constructs a reverse order path, that is then walked from the last analyzed node to the bottom of the current loop. The last exit condition means also that it could be the case that some of the versions of the variable are not walked in the current loop nest (see Example 5). However, the evolution of these versions does not determine the evolution of the variable in the loop nest. These versions of the variable are analyzed only when another variable depends on one of them.

During the construction of the path there are two possible types of nodes that we have to analyze:

- an assignment, in which case if one of the right hand side operands is the use of the same variable, then the analysis follows the link to its definition,
- a ϕ-node, in which case only one of the arguments is taken, that does not belong to an inner loop, *i.e.* we follow the SSA chain towards the original ϕ-node. The other arguments of the ϕ-node are followed and analyzed during the back walking of the path.

At each step of this path, a new version of the variable is analyzed. A new initial condition is associated to each version of the variable, while the evolution part of the chain of recurrence is associated to the real variable in the current loop. Thus for a given analysis of a definition, the chain of recurrence is not completely instantiated. The initial condition is in fact the result of a simple range propagation in the loop's body [39], while the evolution part is still susceptible to change during the analysis of the remaining code in the loop.

Construction of chains of recurrences During the analysis of an assignment expression, the evolution of the right hand side is first determined. Then the initial condition is associated to the left hand side version, and the evolution part updates the evolution of the real variable.

When analyzing a ϕ-node that stands at the merge point of two or more branches of a conditional expression, the analyzer has to approximate the information coming from all the edges. Since only one of the branches have been analyzed for the moment, the analyzer has to gather the information coming from the other branches as being an alternate possible evolution. It then launches the analysis on these other branches, and reminds of the fact that the evolution that it analyzes is an alternate branch of control. This step introduces envelopes of evolutions as an approximation of the real evolution function. Example 6 shows the analyzer on a loop containing a conditional expression.

When analyzing a loop ϕ-node, the initial condition in the outer loop has been determined. The only thing that remains to analyze is the evolution of the variable in the inner loop. This step is performed by calling recursively the same algorithm on the inner loop. On end of the algorithm, the number of iterations is computed, and the evolution function is instantiated to the exit value of the main induction variable. Example 8 shows the analysis of a loop nest containing two loops.

4.5 Analysis of flip-flop operations

The strongly connected components (SCCs) are analyzed for determining whether they produce a periodic behavior in the loop or not. If every variable in the SCC contains exactly one dependence in the SCC, then the SCC is regular, and its behavior is periodic. Else the SCC is a more complex object described in section 4.6.

The SCCs that produce periodic behaviors are selected, and the evolution of the whole cycle is determined by analyzing the evolution of each component taken without the SCC dependences. Then, the evolutions are merged, as well as the initial conditions.

```
construct_periodic_chrec (C) =
  for each variable V in the cycle C
    determine the initial conditions of V
    determine the evolution of V
  end for
  merge initial conditions
  merge evolutions
  associate with each variable in the cycle a pchrec
```

4.6 Bizarre mixer objects

After the transformation of the cyclic part of the scalar dependence graph into an acyclic graph by collapsing SCCs, it is possible that the resulting graph still contains SCCs.

In this section we're investigating on the properties of these multi-cyclic graphs. Their periodic behavior is still difficult to explain. The difficulty comes from the fact that the cycles exchange information, and their evolution depends on the evolution of their sibling cycle. This results in a mixing of the information contained in all these cycles.

Figure 4.1 illustrates the case of a mixer with two cycles.

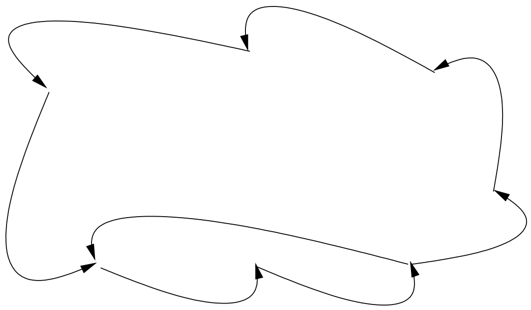

Figure 4.1: A mixer.

4.6.1 The Fibonacci system of recurrence relations

The *Fibonacci sequence*, discovered by Leonardo Pisano Fibonacci, begins

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots$$

The nth Fibonacci number is generated by adding the previous two. Thus, the Fibonacci sequence has the recurrence relation

$$f_n = f_{n-1} + f_{n-2}$$

with $f_0 = 0$ and $f_1 = 1$, that can be modeled by the following program:

```
a = 0
b = 1
loop
  fib = a + b
  a = b
  b = fib
endloop
```

The scalar dependence graph of this program is represented in Figure 4.2. This graph contains two strongly connected components, and is in some way the minimal object that follows the mixers pattern.

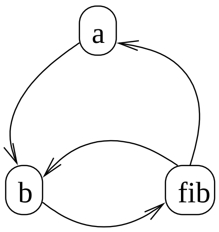

Figure 4.2: The scalar dependence graph for the Fibonacci system of recurrences.

We can safely transform the first version of the Fibonacci program into:

```
a = 0
b = 1
fib = b
loop
  fib = fib + a
  a = b
  b = fib
endloop
```

This program has a scalar dependence graph represented in Figure 4.3.

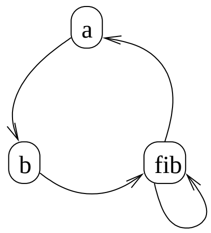

Figure 4.3: The scalar dependence graph for the modified Fibonacci program.

The two SDGs of the Fibonacci programs are similar to the SDG of a flip-flop with a period of two, but they contain one more dependence in the cycle. The behavior of these cycles is no more periodic.

The Fibonacci recurrence relation can be solved into the closed form [6]:

$$f(n) = \frac{1}{\sqrt{5}} \left(\phi^n - \phi'^{\,n}\right)$$

Where ϕ is the golden ratio $\frac{1+\sqrt{5}}{2}$ and $\phi' = \frac{1-\sqrt{5}}{2}$.

4.6.2 Other strange objects

Consider the SDG containing two SCCs:

$$a \rightarrow b \rightarrow c \rightarrow a \rightarrow d \rightarrow e \rightarrow f \rightarrow d$$

represented in Figure 4.4.

It could have been extracted from the program:

```
loop
  d = e
  e = f
  f = d
  a = b op d
  b = c
  c = a
endloop
```

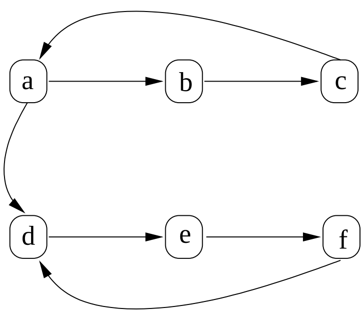

Figure 4.4: Two flip-flop operations, one dependent on the other.

The first SCC $a \to b \to c \to a$ depends on the second one $d \to e \to f \to d$, and thus by collapsing the cycles we obtain an acyclic graph with a single edge: $abc \to def$. The evolution of the variables in this loop can be modeled by the periodic chains of recurrences. The periodicity of the cycle $d \to e \to f \to d$ is not altered by the existence of the dependence $a \to d$, and thus the period of this cycle is equal to two. The periodicity of the other cycle, $a \to b \to c \to a$ depends on the period of the cycle $d \to e \to f \to d$, and the period is computed by the least common multiple of the periods of the cycles taken without the dependence. Example 3 shows the evolution of the variables of the above program for a given initial context.

Now suppose that these two flip-flop cycles exchange information: *i.e.* there exists a dependence from the second cycle to the first cycle: for example $e \to b$. The resulting graph is represented in Figure 4.5.

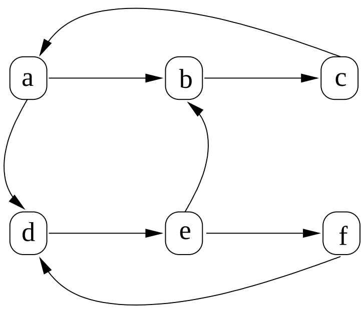

Figure 4.5: Two interdependent flip-flop operations.

```
loop
  d = e
  e = f op b
  f = d
  a = b op d
  b = c
  c = a
```

```
endloop
```

After having collapsed both the cycles $a \to b \to c \to a$ and $d \to e \to f \to d$, we still have a SCC, namely $abc \to def \to abc$. The analyzer doesn't know how to handle this kind of evolution, and thus classifies it as unknown.

Chapter 5

Conclusion

The detection of induction variables is a well known analysis that has been previously addressed by a large number of techniques. In this thesis we have extended two of the previous techniques: the monotonic evolution, and the chains of recurrences.

Together, the monotonic evolution and the chains of recurrences techniques completes each other. The approximation of the evolution by intervals is used for determining some of the uncomputable properties of the program, and the chains of recurrences extend the previous works on monotonic evolution by allowing to gather more precise information on the initial conditions and evolutions.

We have given two major extensions of the chains of recurrences: the first allows the analyzers to handle the periodic functions, and the second allows the approximation of the evolution. We have also compared the algorithms used by Haghighat and Van Engelen starting from the observation that under the chains of recurrences stands the Newton's interpolating method.

We have implemented the extension of the chains of recurrences algorithms from chapter 3 in a library of basic functions for constructing and manipulating the chains of recurrences. This library is not flexible enough for its integration in a compiler, and thus the chains of recurrences has also been implemented using the internal representation of trees of the GCC. This implementation uses the internal tree expression folder for the arithmetic operations on integers. An interesting development would be to extend the current analyzer to the floating point arithmetics. But this is a much more difficult task for obtaining conservative approximations.

The algorithms on the detection of the monotonic evolution described in chapter 4 have been adapted and implemented using the experimental branch of development *tree-ssa* of the *GNU Compiler Collection*. We hope that our contribution will be included in this branch of GCC after some more work.

Thanks — Remerciements

I would like to thank the following people for their help and for their useful advice. Je voudrais remercier les personnes suivantes pour leur aide, et leurs précieux conseils.

- Daniel Berlin,
- Philippe Clauss (*ICPS/LSIIT*),
- Albert Cohen (*INRIA Rocquencourt*),
- Zdeněk Dvořák (*SuSE*),
- Jan Hubička (*SuSE*),
- Jeff Law (*Red Hat*),
- Vincent Loechner (*ICPS/LSIIT*),
- Andrew MacLeod (*Red Hat*),
- Benoît Meister (*ICPS/LSIIT*),
- Jason Merrill (*Red Hat*),
- Diego Novillo (*Red Hat*),
- Frédéric Wagner (*Loria*),
- the members of the *ICPS* labs at *LSIIT*.

Appendix A

Misc. Algorithms

A.1 Scalar Dependence Graph

The aim of this preliminary analysis is to extract the information that will direct the variable evolution analyzer. This analyzer extracts from the instruction stream the order in which the variables were defined and used.

We're interested exclusively in finding the scalar dependences for the integer variables in the loop's body. We restrict the action field of the analyzer to the integers induction variable for avoiding the floating point arithmetic evaluations. The extension of the analyzer to handle floating point arithmetics is possible, but needs infinite precision evaluations and a rounding of the result as described in the ANSI/IEEE-754 standard [19].

The algorithm is based on a loop nest at a time analysis and constructs the scalar dependence graph by excluding any extensions of the dependence graph out of the loop nest's bounds.

Example 1 (Scalar Dependence Analysis) *As an example of construction of the scalar dependence graph, we will see the action of the compiler on the following program:*

```
i_0 = 3;
loop
  i_1 = phi (i_0, i_2);
  j_0 = i_1 + 1;
  k_0 = i_1 + j_0;
  i_2 = k_0 + 2;
endloop
```

The analyzer walks the loop's body from the first instruction to the last, and translates the assignment instructions into a set of tuples that constitute the oriented edges of the scalar dependence graph.

```
i_0 = 3;
loop
  {}
  i_1 = phi (i_0, i_2);
  {}
  j_0 = i_1 + 1;
  {(j->i)}
  k_0 = i_1 + j_0;
  {(j->i), (k->i), (k->j)}
  i_2 = k_0 + 2;
  {(j->i), (k->i), (k->j), (i->k)}
endloop
```

If we consider the original code in the SSA form, each variable has its version number. The SDG analyzer has to record only the variable's name. If the graph contains the version number, the graph becomes acyclic and the detector of flip-flop variables that is based on the discovery of strongly connected components will not work.

The ϕ-node does not introduce edges in the graph since it represents inter-iterations dependences. An assignment to a scalar variable introduces edges between the left hand side of the assignment and the scalar variables of the right hand side. Note that the operations dealing with memory accesses are not handled by this analyzer, neither are the operations involving function calls.

It is possible to improve the algorithm based on the SSA form by avoiding to walk over all the loop's body. For a given variable to be analyzed, it is possible to construct a partial SDG by following the def-use links. This improves the compilation speed by avoiding to fetch all the underlying representation of the loop's body into the caches.

A.2 Topological order

The previous analysis has extracted a set of dependences over the variables definitions that will direct the study of the variables evolution. In order to respect all these dependences we have to order them into a linear task list using a topological sort.

The topological sorting of a directed acyclic graph satisfies the property [2, 5] that if $m_i \to m_j$ is an edge of the graph, then $m_i \prec m_j$.

The topological order can be extracted only when the graph represents a partial order. For a graph with cycles, we extract the acyclic part of the graph. This acyclic part results, after sorting, in an analysis schedule, while the cyclic parts are decomposed into strongly connected components (SCC) and variables that depend on the SCCs. The following section describes how to detect and extract strongly connected components from the scalar dependence graph.

The algorithm for "embedding the partial order into a linear order" has been taken from the section 2.2.3 of TAOCP [2]. This algorithm stops when all the nodes not involved or dependent on a cycle have been sorted. The residue graph contains all the cycles and the variables dependent on a cycle.

When we are interested only on the evolution of a single variable, it is possible to partially sort the graph: *i.e.* start the analysis from the target, and then gather the dependence relations downward. This algorithm can be used in the light weight version of the monotonic evolution algorithm, that focuses the search on the variables involved in the exit condition.

A.3 Detecting Strongly Connected Components

After having extracted a topologically sorted list from the SDG, the set of variables not processed are either a component of a strongly connected region (SCR), or dependent on a variable in a SCR.

The strongly connected components (SCC) are detected by the Tarjan's algorithm [20]. This algorithm has been used by Wolfe and Gerlek in *Beyond Induction Variables* [21].

In the same time as the SCC detection, we determine the length of each SCC. Based on the length of a SCC we determine the period of the variables involved in the SCC.

A.4 Detecting the period of a SCC

In a strongly connected component the variables exchange their information with their neighbors in the scalar dependence graph. After a complete tour they inherit from their initial information that could have been altered by some operations at some point in this cycle. The periodicity of a variable in the cycle is determined by the length of a tour, and is given by the following formula:

$$period = length(SCC) - 1$$

The -1 is due to the fact that in a flip-flop, one of the variables is the one that stores the intermediate result until all the other variables are updated.

Example 2 (A simple cycle) *Every variable in a cyclic dependence chain of length three has a period two.*

```
loop
  a = b
  b = c
  c = a
endloop
```

Let's execute the first two iterations of the loop: the information gathered by the interpreter are between sharp signs.

```
#(a = undef), (b = ib), (c = ic)#
loop
  #(a = undef), (b = ib), (c = ic)#
  a = b
  #(a = ib), (b = ib), (c = ic)#
  b = c
  #(a = ib), (b = ic), (c = ic)#
  c = a
  #(a = ib), (b = ic), (c = ib)#
endloop

loop
  #(a = ib), (b = ic), (c = ib)#
  a = b
  #(a = ic), (b = ic), (c = ib)#
  b = c
  #(a = ic), (b = ib), (c = ib)#
  c = a
  #(a = ic), (b = ib), (c = ic)#
endloop
```

After two iterations, the variables b and c contain the same values as at the starting point of the program. Thus, the values taken by the variables b and c vary periodically, and can be described by the following notation:

$$b = |ib, ic|$$

$$c = |ic, ib|$$

where the first value of the expression is the value at the entry point in the loop, and the second value is that after one execution of the loops body.

The period of a SCC A that depends on another SCC B is given by the least common multiple of the period of A taken without dependences on B, and of the period of B.

$$period(A) = lcm(length(A) - 1, length(B) - 1)$$

Example 3 gives an illustration of this case.

Example 3 (A dependence between two cycles) *In this example we show the evolution of the variables, in the initial context* #a = undef, b = ib, c = ic, d = undef, e = ie, f = if#, *for the following program:*

```
loop
  d = e
  e = f
  f = d
  a = b + d
```

```
  b = c
  c = a
endloop
```

The scalar dependence graph for this example is illustrated in Figure 4.4.

```
#a = undef, b = ib, c = ic, d = undef, e = ie, f = if#
loop
  #a = undef, b = ib, c = ic, d = undef, e = ie, f = if#
  d = e
  #a = undef, b = ib, c = ic, d = ie, e = ie, f = if#
  e = f
  #a = undef, b = ib, c = ic, d = ie, e = if, f = if#
  f = d
  #a = undef, b = ib, c = ic, d = ie, e = if, f = ie#
  a = b + d
  #a = ib + ie, b = ib, c = ic, d = ie, e = if, f = ie#
  b = c
  #a = ib + ie, b = ic, c = ic, d = ie, e = if, f = ie#
  c = a
  #a = ib + ie, b = ic, c = ib + ie, d = ie, e = if, f = ie#
endloop

loop
  #a = ib + ie, b = ic, c = ib + ie, d = ie, e = if, f = ie#
  d = e
  #a = ib + ie, b = ic, c = ib + ie, d = if, e = if, f = ie#
  e = f
  #a = ib + ie, b = ic, c = ib + ie, d = if, e = ie, f = ie#
  f = d
  #a = ib + ie, b = ic, c = ib + ie, d = if, e = ie, f = if#
  a = b + d
  #a = ic + if, b = ic, c = ib + ie, d = if, e = ie, f = if#
  b = c
  #a = ic + if, b = ib + ie, c = ib + ie, d = if, e = ie, f = if#
  c = a
  #a = ic + if, b = ib + ie, c = ic + if, d = if, e = ie, f = if#
endloop
```

Using the chains of recurrences notation, we write:

$$b = |<ib, +, ie>, <ic, +, if>|$$

$$c = |<ic, +, if>, <ib, +, ie>|$$

$$e = |<ie>, <if>|$$

$$f = |<if>, <ie>|$$

Appendix B

Computing over the lattice of intervals

This is a well known technique that we have used in several sections of this thesis. There are many possible presentations of the arithmetic of intervals, and we have chosen the one exposed in [34]. The proof of the theorem is given in [35].

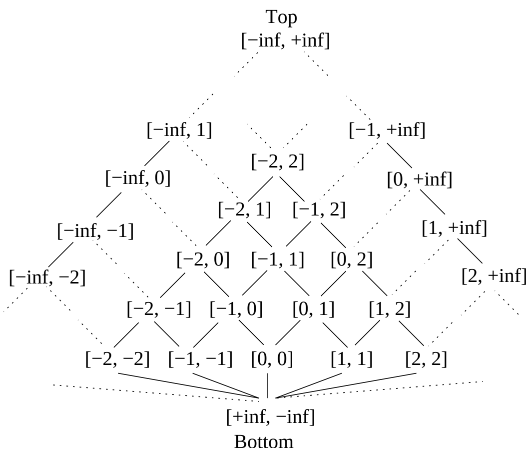

Figure B.1: The complete lattice **Interval**

Definition 4 (Syntax of Intervals) $AS_{Interval} = \{[x, y] \mid x, y \in \mathbb{Z} \biguplus \{-\infty, +\infty\}\}$.

Definition 5 (Semantics of Intervals) $S_{Interval}$ *is* $(I, \models)$ *where* $I = \mathbb{Z}$ *and* $i \models [x, y] \Leftrightarrow x \leq i \leq y$.

Definition 6 (Implementation of Intervals) $P_{Interval}$ *is* $(\sqsubseteq, \sqcap, \sqcup, \bot, \top)$*, where for every* $[x_1, y_1]$*,* $[x_2, y_2] \in AS_{Interval}$

- $[x_1, y_1] \sqsubseteq [x_2, y_2]$ *iff* $x_2 \leq x_1$ *and* $y_1 \leq y_2$*,*

- $[x_1, y_1] \sqcap [x_2, y_2] = [max(x_1, x_2), min(y_1, y_2)]$,
- $[x_1, y_1] \sqcup [x_2, y_2] = \begin{cases} [min(x_1, x_2), max(y_1, y_2)] & if\ x_2 \leq y_1\ and\ x_1 \leq y_2 \\ undef & otherwise \end{cases}$,
- $\top = [-\infty, +\infty]$,
- $\perp = [+\infty, -\infty]$.

Theorem 1 (Completeness/consistency) *$P_{Interval}$ is consistent and complete in $\sqsubseteq, \sqcap, \sqcup, \perp, \top$ wrt. $S_{Interval}$.*

Appendix C

Chains of recurrences

We introduce chains of recurrences by repeating some known definitions for the recurrence relations and closed form functions [6], then we give the formal definitions of chains of recurrences as described by Zima [23, 24, 25], finally we give some implementation details for dealing with the rewriting rules for chains of recurrences described by Van Engelen [18, 17, 16].

C.1 Recurrence relations

A *recurrence relation* is a function which gives the value of a sequence at some position based on the values of the sequence at previous positions and the position index itself. If the current position n of a sequence s is denoted by s_n, then the next value of the sequence expressed as a recurrence relation would be of the form

$$s_{n+1} = f(s_1, s_2, \ldots, s_{n-1}, s_n, n)$$

Where f is any function. An example of a simple recurrence relation is

$$s_{n+1} = s_n + (n + 1)$$

which is the recurrence relation for the sum of the integers from 1 to $n + 1$.

C.2 Closed form functions

A *closed form* function which gives the value of a sequence at index n has only one parameter, n itself. This is in contrast to the recurrence relation form, which can have all of the previous values of the sequence as parameters.

The benefit of the closed form is that one does not have to calculate all of the previous values of the sequence to get the next value. It is very useful to get the value of the sequence at some index n. In the case where the recurrence relation is computed in a loop, the closed form function removes loop carried scalar dependences.

There are many techniques used to find a closed-form solution for a recurrence relation. Some are

- Repeated substitution. Replace each s_k in the expression of s_n (with $k < n$) with its recurrence relation representation. Repeat again on the resulting expression, until some pattern is evident.
- Estimate an upper bound for s_n in terms of n. Then, solve for the unknowns (say there are r unknowns) by finding the first r values of the recurrence relation and solving the linear system formed by them and the unknowns.
- Find the characteristic equation of the recurrence relation and solve for the roots.

C.3 The chains of recurrences

The chains of recurrences were introduced by Eugene Zima for evaluating the closed form of functions at regular intervals: *i.e.* we need the values of $F(i)$ for the values $i = 0, 1, \ldots, n$. For this purpose, the function $F(i)$ is transformed into a system of recurrence relations, defined as follows:

$$\begin{aligned} f_j(i) &= \begin{cases} \phi_j & \text{if } i = 0, \\ f_j(i-1) \odot_{j+1} f_{j+1}(i-1) & \text{if } i > 0, \end{cases} \\ j &= 0, 1, \ldots, k-1, \end{aligned}$$

where $\phi_0, \ldots, \phi_{k-1}$ are constant expressions, $\odot_j \in \{+, *\}$, and $f_k(i)$ is a closed form function that is either a constant, or defined again as a chain of recurrence. When $\odot_j = +$ for $j = 0, 1, \ldots, k-1$, the chain of recurrence is called a pure sum, and when $\odot_j = *$ for $j = 0, 1, \ldots, k-1$, the chain of recurrence is called a pure product.

The simplified notation for this system of recurrence relations is:

$$f_j(i) = \{\phi_j, \odot_{j+1}, f_{j+1}\}_i$$

As an example taken from [25]:

$$F(i) = \frac{i!(n-i)!}{n!}, i = 0, 1, \ldots, n,$$

that is represented by a chain of recurrence:

$$\begin{aligned} f_0(i) &= \begin{cases} 1 & \text{if } i = 0, \\ f_0(i-1) * f_1(i-1) & \text{if } i > 0, \end{cases} \\ f_1(i) &= \frac{g_0(i)}{h_0(i)} \\ g_0(i) &= \begin{cases} 1 & \text{if } i = 0, \\ g_0(i-1) + 1 & \text{if } i > 0, \end{cases} \\ h_0(i) &= \begin{cases} n & \text{if } i = 0, \\ h_0(i-1) + (-1) & \text{if } i > 0. \end{cases} \end{aligned}$$

Using the linear notation, this system can be written as:

$$\begin{aligned} F(i) &= \{1, *, \frac{\{1, +, 1\}}{\{n, +, -1\}}\}(i), \\ i &= 0, 1, \ldots, n. \end{aligned}$$

C.4 Simplification and interpretation of chrecs

The operations over the chains of recurrences are described in several publications [17, 25]. We expose here, for reference, the reduced set of rules given in [25].

S_1**:** $c + \{\varphi_0, +, \Phi_1\} = \{c + \varphi_0, +, \Phi_1\}$

S_2**:** $c\{\varphi_0, +, \Phi_1\} = \{c\varphi_0, +, c\Phi_1\}$

S_3**:** $c\{\varphi_0, *, \Phi_1\} = \{c\varphi_0, *, \Phi_1\}$

S_4**:** $c^{\{\varphi_0, +, \Phi_1\}} = \{c_0^{\varphi}, *, c_1^{\Phi}\}$

S_5**:** $\{\varphi_0, +, \Phi_1\} \pm \{\psi_0, +, \Psi_1\} = \{\varphi_0 \pm \psi_0, +, \Phi_1 \pm \Psi_1\}$

S_6**:** $\{\varphi_0, *, \Phi_1\} * / \{\psi_0, *, \Psi_1\} = \{\varphi_0 * / \psi_0, *, \Phi_1 * / \Psi_1\}$

S_7**:** $\{\varphi_0, +, \Phi_1\} * \{\psi_0, +, \Psi_1\} = \{\varphi_0 * \psi_0, +, \Phi\Psi_1 + \Phi_1 E(\Psi)\}$

where E is the shift operator, $E(\Psi) = \Psi + \Psi_1$.

Appendix D

Examples

In this chapter the information gathered by an analyzer are included between the lines of the program and are marked by sharp signs.

D.1 The basic monotonic evolution algorithm

Example 4 (Inefficiency of the basic monotonic evolution algorithm) *In the following code, the evolution of j will be known only after the evolution of i has been established. The computation of the variable j during the first iteration is not necessary.*

```
i = 3
j = 4
loop
  j = i + 2
  i = i + 1
endloop
----
First propagation:
#i = {[-oo, +oo]}, j = {[-oo, +oo]}#
i = 3
#i = {[3, 3]}, j = {[-oo, +oo]}#
j = 4
#i = {[3, 3]}, j = {[4, 4]}#
loop
  #i = {[3, 3], +, [-oo, +oo]}, j = {[4, 4], +, [-oo, +oo]}#
  j = i + 2
  #i = {[3, 3], +, [-oo, +oo]}, j = {[5, 5], +, [-oo, +oo]}#
  i = i + 1
  #i = {[4, 4], +, [1, 1]}, j = {[5, 5], +, [-oo, +oo]}, i changed#
endloop
----
```

```
Second iteration:
loop
  #i = {[3, 3], +, [1, 1]}, j = {[4, 4], +, [-oo, +oo]}#
  j = i + 2
  #i = {[3, 3], +, [1, 1]}, j = {[5, 5], +, [1, 1]}, j changed#
  i = i + 1
  #i = {[4, 4], +, [1, 1]}, j = {[5, 5], +, [1, 1]}, j changed#
endloop
----
Third iteration:
loop
  #i = {[3, 3], +, [1, 1]}, j = {[4, 4], +, [1, 1]}#
  j = i + 2
  #i = {[3, 3], +, [1, 1]}, j = {[5, 5], +, [1, 1]}#
  i = i + 1
  #i = {[4, 4], +, [1, 1]}, j = {[5, 5], +, [1, 1]}#
endloop
```

D.2 Illustrations of the SSA based algorithm

Example 5 (Not walked versions) *This example shows that some of the versions of the analyzed variable (the code is in SSA form) could not interfere with the evolution of the real variable.*

```
i_0 = 2
loop
  i_1 = phi (i_0, i_2)
  ...
  i_3 = i_1 + 3
  ...
  i_4 = 4
  loop
    i_2 = phi (i_4, i_5)
    ...
    i_5 = i_2 + 5
  end loop
end loop
```

In this case the analysis follows the SSA-chain $i_1 \rightarrow i_2 \rightarrow i_4$ and stops on i_4 since its evolution is already determined. The version i_3 is not walked during the analysis, since the evolution of the real variable i does not depend on the evolution of the version i_3.

Example 6 (A conditional expression in the loop) *The conditional expressions are often difficult to analyze in loops, because they introduce sometimes conditions that cannot be analyzed at compile time. One such example is illustrated in the following program:*

```
i_0 = 2
loop
  i_1 = phi (i_0, i_2)
  ...
  if (volatile_variable)
    i_3 = i_1 + 1
  else
    i_4 = i_1 + 2
  end if
  i_2 = phi (i_3, i_4)
  ...
end loop
```

The volatile_variable *is a variable whose value cannot be computed at compile time. A static analysis cannot predict the path that will be taken at run time, and thus the analysis has to approximate the evolution of the variable.*

The analyzer determines the loop's ϕ-node, and then analyzes the initial condition out of the current loop. It initializes the real variable to: $i = i_1 = i_0 = \{[2, 2]\}$.

Then, the analyzer follows the SSA edge $i_1 \to i_2$ and since i_2 is a ϕ-node, it follows only one of the branches: $i_2 \to i_3$. It determines that the right hand side contains i_1 and that by following the edge $i_3 \to i_1$ it ends on the starting ϕ-node. It decides then to stop the recursive walk of the SSA edges. The assignment "$i_3 = i_1 + 1$" is analyzed: the initial condition for the version i_3 is set to $\{[3, 3]\}$, and the evolution of the real variable i is set to $\{[2, 2], +, [1, 1]\}$. The back walking continues, and the analyzer ends on the ϕ-node "$i_2 = \phi(i_3, i_4)$".

The second argument of the ϕ-node is followed $i_2 \to i_4$, and since the right hand side contains i_1 that reaches the starting ϕ-node, the recursion stops, and the assignment "$i_4 = i_1 + 2$" is analyzed: $\{[4, 4]\}$ is associated to i_4, and the evolution of the real variable is updated to $\{[2, 2], +, [1, 2]\}$.

Finally the initial condition for version i_2 is computed as the interval $\{[3, 4]\}$.

Example 7 (Extended example) *We extend the previous example to the analysis of the variable k that depends on i.*

```
i_0 = 2
loop
  i_1 = phi (i_0, i_2)
  k_1 = phi (i_0, k_2)
  ...
  if (volatile_variable)
    i_3 = i_1 + 1
    k_3 = k_1 + 5
  else
    i_4 = i_1 + 2
  end if
  i_2 = phi (i_3, i_4)
  k_4 = phi (k_1, k_3)
  ...
  k_2 = k_4 + i_2
  ...
end loop
```

The initial value of k is determined: $k = k_1 = i_0 = \{[2, 2]\}$. Then the edge $k_1 \rightarrow k_2 \rightarrow k_4 \rightarrow k_1$ is analyzed. Since k_4 is a ϕ-node, the second argument is analyzed as an alternative: the edge $k_4 \rightarrow k_3 \rightarrow k_1$ is analyzed. "$k_3 = k_1 + 5$" is analyzed as an alternative: $k_3 \leftarrow \{[7, 7]\}$, and $k \leftarrow \{[2, 2], +, [0, 5]\}$.

Then, $k_4 \leftarrow \{[2, 7]\}$, and the walk on the path reaches the definition of "$k_2 = k_4 + i_2$". At this point, the previous evolution of $i = \{[2, 2], +, [1, 2]\}$ and $i_2 = \{[3, 4]\}$ is retrieved and merged into: $(i@i_2) \leftarrow \{[3, 4], +, [1, 2]\}$. The version of k_2 is set to $\{[2, 7]\} + \{[3, 4]\} = \{[5, 11]\}$, and the evolution of k is updated to $\{[2, 2], +, ([0, 5] + [3, 4]), +, [1, 2]\} = \{[2, 2], +, [3, 9], +, [1, 2]\}$.

Example 8 (Analysis of a loop nest) *The analysis of a loop nest uses the multivariate extension of the chains of recurrences. The loop indexes are taken for naming the dimensions.*

```
i_0 = 2
loop_A
  i_1 = phi (i_0, i_2)
  if (i_1 > 100)
    goto end_A
  ...
  j_0 = 0
  loop_B
    j_1 = phi (j_0, j_2)
    i_4 = phi (i_1, i_3)
    if (j_1 > 10)
      goto end_B
    ...
    i_3 = i_4 + 5
    j_2 = j_1 + 1
  endloop
  end_B
  i_2 = i_4 + 1
endloop
end_A
```

The analyzer detects that the inner loop has its exit condition on j, and after having determined from the scalar dependence graph that j does not depend on other variables, it starts the analysis with the variable j. Since j has no ϕ-node in the loop A, the analysis is started again on the inner loop B. As in the previous examples, the initial value is determined: $j = j_1 = j_0 = \{[0,0]\}$, and then the edge $j_1 \rightarrow j_2 \rightarrow j_1$ is analyzed. The final evolution of the variable j is determined as $\{[0,0],+,[1,1]\}$. From this evolution function, the number of iterations is determined: the least iteration number that satisfies the condition is 11.

The variable i is then analyzed: initial value is $i \leftarrow i_1 \leftarrow i_0 \leftarrow \{[2,2]\}$, and after having followed the edge $i_1 \rightarrow i_2 \rightarrow i_4 \rightarrow i_1$, the analyzer comes back on the ϕ-node $i_4 = \phi(i_1, i_3)$. Since i_4 is a loop ϕ-node, the analyzer reinitializes a new pass of the same algorithm on this loop: the initial condition for the variable i is $i \leftarrow i_4 \leftarrow i_1 \leftarrow \{[2,2]\}$, and the edge $i_4 \rightarrow i_3 \rightarrow i_4$ is analyzed, from which the analyzer determines that the variable i follows the evolution $i \leftarrow \{[2,2],+,[5,5]\}_B$. The analysis is finished for the loop B. The chain is followed backward from i_4 to i_2. At this point, the initial condition of i_2 is computed as the value of i on end of the loop B: $i_2 \leftarrow \{[2,2],+,[5,5]\}_B(11) = \{[57,57]\}$, and the evolution of i is updated to: $i \leftarrow \{\{[2,2],+,[5,5]\}_B,+,[1,1]\}_A$.

Bibliography

[1] F. Nielson, H.R. Nielson, C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[2] D. E. Knuth. *The Art of Computer Programming.* Addison-Wesley Publishing Company, 1969.

[3] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[4] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1995.

[5] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* AddisonWesley, 1986.

[6] *Planet Math.* http://planetmath.org

[7] *The First Annual GCC Developers' Summit.* http://www.gccsummit.org/2003/

[8] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. *Designing the McCAT compiler based on a family of structured intermediate representations.* In Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing, pages 406-420. Lecture Notes in Computer Science, no. 457, Springer-Verlag, August 1992.

[9] G.A. Silber. *The Nestor library: source to source transformations of Fortran programs.* http://www.cri.ensmp.fr/people/silber/nestor/index.html

[10] F. Wagner, S. Pop. *Optimisation temporelle des nids de boucles.* Rapport de Travail d'Etude et de Recherche, LSIIT-ULP, 2001.

[11] S. Pop. *Interface and Extension of the Open Research Compiler.* Internship Report, http://www-rocq.inria.fr/~pop/, INRIA 2002.

[12] P. Cousot and N. Halbwachs. *Automatic Discovery of Linear Restraints Among Variables of a Program.* In Proc. of POPL 1978, pages 84-97. ACM Press, 1978.

[13] T. E. Uribe. *Combinations of Model Checking and Theorem Proving.* In Frontiers of Combining Systems: Third International Workshop, FroCoS 2000:

Nancy, March 2000: Lecture Notes in Artificial Intelligence, vol. 1794 (Springer-Verlag, 2000), pp. 151–170.

[14] V. Martena, P. San Pietro. *Alias Analysis by Means of a Model Checker.* 10th International Conference on Compiler Construction (CC 01), LNCS vol. 2027, 2001.

[15] D.A. Schmidt. *Data Flow Analysis is Model Checking of Abstract Interpretation.* In Proc. of POPL '98, ACM Press, 1998.

[16] R.A. van Engelen, K.A. Gallivan. *An Efficient Algorithm for Pointer-to-Array Access Conversion for Compiling and Optimizing DSP Applications*

[17] R.A. van Engelen. *Efficient Symbolic Analysis for Optimizing Compilers.* In proceedings of the International Conference on Compiler Construction, ETAPS 2001, LNCS 2027, pp. 118-132

[18] R.A. van Engelen. *Symbolic evaluation of chains of recurrences for loop optimization.* Technical report, TR-000102, Computer Science Deptartment, Florida State University, 2000.

[19] American National Standards Institute, Inc. *IEEE standard for binary floating point arithmetic.* Technical Report 754-1985, ANSI/IEEE, 1985. http://grouper.ieee.org/groups/754/

[20] R. E. Tarjan. *Depth-first search and linear graph algorithms.* SIAM Journal on Computing, 1:146, 1972.

[21] M. Wolfe. *Beyond induction variables.* In Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, pages 161–174, 1992.

[22] E.V. Zima. *Automatic Construction of Systems of Recurrence Relations.* USSR Comput. Maths. Math. Phys., Vol.24, N 6, 1984, pp. 193–197.

[23] O. Bachmann, P. S. Wang, E. V. Zima. *Chains of Recurrences - a method to expedite the evaluation of closed-form functions.* In Proceedings of the International Symposium on Symbolic an Algebraic Computation - ISSAC'94, pages 242–249, Oxford, England, United Kingdom, July 1994. ACM Press.

[24] Kislenkov, Mitrofanov, Zima. *Multidimensional chains of recurrences.* ISSAC 98, Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation, Rostock, Germany, 13-15 Aug. 1998.

[25] E.V. Zima. *On computational properties of chains of recurrences.* ISSAC 01, Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation, Canada 2001.

[26] B. Franke, M. O'Boyle. *Compiler Transformation of Pointers to Explicit Array Accesses in DSP Applications.*

[27] B. Meister. *De l'utilisation de périodiques dans les problèmes polyédriques en nombres entiers.* Rapport de recherche (à paraître), LSIIT, 2003.

[28] Peng Wu. *Analyses of Pointers, Induction Variables, and Container Objects for Dependence Testing.* Ph.D Thesis, University of Illinois at Urbana-Champaign, DCS TR-2001-2209, May 2001.

[29] Albert Cohen and Peng Wu. *Dependence Testing without Induction Variable Substitution.* The 9th Workshop on Compilers for Parallel Computers (CPC'01), Edinburgh, Scotland, UK, June, 2001.

[30] Peng Wu, Albert Cohen, and David Padua. *Induction Variable Analysis without Idiom Recognition: Beyond Monotonicity.* 14th International Workshop of Languages and Compilers for Parallel Computing (LCPC'01), Cumberland Fall, Kentucky, Aug, 2001.

[31] Peng Wu, Albert Cohen, Jay Hoelfinger, and David Padua. *Monotonic Evolution: an Alternative to Induction Variable Substitution for Dependence Analysis.* ACM Int'l. Conf. on Supercomputing (ICS'01), Sorrento, Italy, June, 2001.

[32] Mark Wegman and Kenneth Zadeck. *Constant propagation with conditional branches.* In Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, pages 291– 299. ACM SIGACT and SIGPLAN, January 1985.

[33] E. Kaltofen. *Polynomial factorization.* In B. Buchberger, G. Collins, and R. Loos, editors, Computer Algebra, 2nd ed., pages 95–113. Springer Verlag, Vienna, 1982.

[34] Sébastien Ferré and Olivier Ridoux. *A Framework for Developing Embeddable Customized Logics.* Logic Based Program Synthesis and Transformation, 11th International Workshop, LOPSTR 2001, Paphos, Cyprus, November 28-30, 2001.

[35] Sébastien Ferré. *Incremental Concept Formation Made More Efficient by the Use of Associative Concepts.* RR-4569, Inria, 2002, http://www.inria.fr/rrrt/rr-4569.html.

[36] F. Chow, S. Chan, R. Kennedy, S-M. Liu, R. Lo, and P. Tu. *A New Algorithm for Partial Redundancy Elimination based upon SSA Form.* ACM SIGPLAN Conf. on Programming Language Design and Implementation, pages 273-286, Las Vegas, Nevada, June 1997.

[37] R. Kennedy, S. Chan, S.-M. Liu, R. Lo, P. Tu and F. Chow. *Partial redundancy elimination in SSA form.* ACM Transactions on Programming Languages and Systems, vol. 21, 1999.

[38] M. R. Haghighat and C. D. Polychronopoulos. *Symbolic Program Analysis and Optimization for Parallelizing Compilers.* In Conference Record of the 5th

Workshop on Languages and Compilers for Parallel Computing, Yale University, Department of Computer Science, 1992.

[39] W. Blume and R. Eigenmann. *Demand-driven, Symbolic Range Propagation.* Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing, Columbus, OH, pages 141–160, August 1995.