M.Sc. Thesis

On the Computation of Fixpoints in Static Program Analysis with an Application to Analysis of AKL

ERIK SCHÖN [1]

October 15, 1995

[1] *E-mail:* f90-esc@nada.kth.se, *Address:* School of Engineering Physics, Royal Institute of Technology (KTH), S-100 44 Stockholm, Sweden.

Abstract

Static program analysis is a way of automatically deriving approximate semantic information about a program before actually running it. This information can then be used by a compiler for globally optimizing the generated code with respect to e.g. size, speed of execution and memory usage.

A fixpoint solver lies at the heart of most frameworks for static program analysis. Given a system of equations containing monotone/continuous functions on a complete lattice/complete partial order, the fixpoint solver iteratively computes the values of the functions and terminates with a solution when the equations are stable. This is then guaranteed to happen in a finite number of iteration steps. The thesis features an introduction to lattice- and fixpoint theory and a survey of methods and recent results for fixpoint computations in static program analysis.

A dependency graph is normally used as a guide to which function to compute in each iteration step of the fixpoint solver. This is a directed graph, possibly with cycles, where the vertices represent functions in the system of equations and an edge from vertex i to vertex j indicates that function j depends on the value of function i for its evaluation.

As the number of equations tend to be very large, it is extremely important to avoid unnecessary computations and hence to find an optimal order in which to compute the functions. It is also important to have an efficient method for detecting stability. An algorithm satisfying these two requirements is Bourdoncle's algorithm [Bou92b, Bou93] for finding a weak topological ordering of a directed graph which may contain cycles. This algorithm is a generalization of Tarjan's algorithm for finding the strongly connected components of a directed graph.

This thesis presents results from experiments with a fixpoint solver in the analysis framework for AKL (Agents Kernel Language, a concurrent constraint language developed at SICS) using the above ideas showing that a considerable speed-up (up to 3000%) can be achieved compared to that of a fixpoint solver based on strongly connected components. This is explained by showing how weak topological orders computed by Bourdoncle's algorithm incorporate many intuitively good heuristics.

The thesis also introduces *partially argument independent functions*, a class of functions particularly suited to the repetitive scheme of computations that arise in fixpoint computations. A function of this type needs to be re-evaluated only with respect to those of its arguments which have changed since its latest evaluation.

Descriptors

Static program analysis, fixpoint computations, dependency graphs, weak topological orders, topological sorting, strongly connected components, Tarjan's algorithm, partially argument-independent functions.

Acknowledgments

There are some people, without whom this M.Sc. Thesis would not have existed. First and foremost, many thanks go to Dan Sahlin, my supervisor at SICS, who inspiringly guided me into the wondrous world of program analysis, providing insights into everything from lattices to formulating definitions.

I would like to thank all the members of the Programming Systems Group at SICS for creating a stimulating research environment to work in. The following deserves special mentioning: Thomas Sjöland, Mats Carlsson, Joakim Hirsch, Khaled Shalaan, Kent Boortz, Johan Montelius and Per Brand.

I am also very grateful to Björn Lisper of the Department of Teleinformatics (IT) at the Royal Institute of Technology (KTH), who accepted my M.Sc. degree project proposal and has served as examiner, and to my supervisor at IT, Karl-Filip Faxén.

Last, but by no means least, I would like to thank my parents, Gerd and Erkki, who sparked my interest in computers and programming by giving me an exciting present way back in the early eighties: a BASIC programming course on the ABC-80 computer.

This M.Sc. degree project was conducted in and funded by the Programming Systems Group at the Swedish Institute of Computer Science (SICS). SICS is a non-profit research foundation, sponsored by the Swedish Board for Industrial and Technical Development (NUTEK), CelsiusTech Systems AB, Ellemtel Utvecklings AB, Ericsson, FMV (Defense Materials Administration), IBM Svenska AB, SUN Microsystems and Telia AB.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, the purpose and contribution of the thesis is given. It also includes some background information and an overview.

1.1 Background

The declarative programming languages, eg. Lisp and Prolog, represent a paradigm shift in that they make the programmer focus on stating the problem instead of specifying how to solve it in minute detail. Ideally, this leads to shorter development times and fewer programming errors. On the negative side, the increased level of abstraction also leads to slower, less memory efficient programs. Thus, to summarize the problem in one sentence: *What is stated declaratively must at some point, somehow be computed efficiently.*

Static program analysis is a way to bridge the gap between the declarative and the operational sides of programming, and hence to have the best of both worlds; by performing static program analysis, global information on properties of the program in question can be found, which can then be used by a compiler to generate faster and less memory consuming code or by a debugger to help the programmer find faults and inconsistencies.

Fixpoint computations lie at the heart of almost all frameworks for static program analysis. In order to find the properties of interest, an equation system is set up and the solution is usually found be performing an iterative computation leading to the least fixpoint of the system.

AKL (Agents Kernel Language) is a declarative, multi-paradigm programming language which can benefit greatly from static program analysis. A framework for static program analysis of AKL, the goal of which is to support the AKL compiler, has been developed. A major bottle-neck in the framework is the execution speed and one way of overcoming this problem is to optimize the fixpoint solver.

1.2 Purpose

The purpose of this thesis is to investigate the methods used in computing the fixpoints that arise when solving the equation systems generated in static program analysis. A more specific goal is to enhance the fixpoint solver in the static program analysis framework for AKL.

1.3 Contributions

The main contributions are the following:

- A survey of methods used in fixpoint computations in static program analysis.
- The first major experimental evaluation of iteration strategies for fixpoint computations based on weak topological orders found by Bourdoncle's algorithm, showing behaviour which contradicts some of Bourdoncle's claims.
- An average speed-up of 300% and in some cases up to 3000%, in the fixpoint computation part of the analysis framework for AKL when weak topological orders are used.
- The introduction of the concept of *partially argument-independent functions* which decreases the cost of function applications.

1.4 Overview

The intention is for the thesis to be self-contained, hence Chapter 2 defines the basic concepts and notation to be used, whereas Part I deals with the theoretical foundations of fixpoint computations; Chapter 3 gives a brief introduction to lattice theory including fixpoints and the following chapters depend rather heavily on the concepts and results presented here. Chapter 4 presents some results from graph theory relevant in this context. Readers familiar with discrete mathematics can skip these chapters.

Part II relates static program analysis to fixpoint computations. Chapter 5 shows how static program analysis gives rise to an equation system which can be solved by means of a fixpoint procedure, and how the AKL analysis framework is set up. In Chapter 6 the problem of fixpoint computation in static program analysis is defined and a survey of methods for solving it is presented.

The last part is dedicated to fixpoint computations in practice. Two strategies for computing fixpoints, strongly connected components and weak topological orders, are presented and compared in Chapter 7, whereas Chapter 8 singles out the class of functions which do not need the values of all their arguments when evaluated.

Finally, the two appendices present AKL (Agents Kernel Language) and the full benchmark results, respectively.

Chapter 2

Preliminaries

This chapter defines the meaning of the basic symbols and modes of expression used throughout the text and is thus to be used as a reference.

2.1 Logic

The logical symbols and their meanings are given in Table 2.1. An arbitrary sequence of variables is denoted $\bar{x}$. A *definition* of a symbol is written $A := B$, where A is the *definiendum* (what is to be defined) and B is the *definiens* (the actual definition). An *assignment* of the value a to the variable x is denoted thus: $x \leftarrow a$.

$A \Rightarrow B$ expresses the fact that statement B follows from statement A. The expression $A \Leftrightarrow B$, or sometimes, A *iff* B, is defined by $A \Leftrightarrow B := A \Rightarrow B \wedge B \Rightarrow A$.

2.2 Sets

A *set* is a collection of objects without repetition. An *member* of a set is one of the objects. The finite set A consisting of members $a_1, \ldots, a_n$ is denoted $\{a_1, \ldots, a_n\}$ and that a_1 is a member of A is denoted $a_1 \in A$. A *set former* is a specification of the type $\{x \mid P(x)\}$ or $\{x \in A \mid P(x)\}$ where the first is read "the set of objects x such that $P(x)$ is true" and the second is read "the set of objects x in the set A such that $P(x)$ is true".

Symbol	*Description*
$\neg$	negation
$\wedge$	conjunction
$\vee$	disjunction
$\rightarrow$	implication
$\leftrightarrow$	equivalence
$\forall$	universal quantification, for all
$\exists$	existential quantification, at least one
$\exists!$	existential quantification, exactly one

Table 2.1: Logical connectives and quantifiers

The symbol $\emptyset$ denotes the set $\{x \mid x \neq x\}$, i.e. the empty set. A is a *subset* of B, denoted $A \subseteq B$, if $\forall x : x \in A \rightarrow x \in B$. The *power set* of B is defined by $\wp(B) := \{A \mid A \subseteq B\}$, i.e. the set of all subsets of B.

The *union* and *intersection* of A and B is defined by $A \cup B := \{x \mid x \in A \vee x \in B\}$ and $A \cap B := \{x \mid x \in A \wedge x \in B\}$, respectively.

2.3 Relations

An *ordered n-tuple* of objects $a_1, \ldots, a_n$ is defined by $\langle a_1, \ldots, a_n \rangle := \{a_1, \{\ldots, \{a_n\}\}\}$. A *pair* is an ordered 2-tuple. The *cartesian product* of A and B, is defined by $A \times B := \{\langle a, b\rangle \mid a \in A \wedge b \in B\}$.

A *relation* from A to B is a set of pairs R such that $R \subseteq A \times B$. A relation R is *on* the set A, if $R \subseteq A \times A$. The fact $\langle a, b\rangle \in R$ is written aRb. A relation $R \subseteq A \times A$ is *reflexive*, if $aRa, \forall a \in A$; *transitive*, if $aRb \wedge bRc \rightarrow aRc, \forall a, b, c \in A$; *symmetric*, if $aRb \rightarrow bRa, \forall a, b \in A$, and *anti-symmetric*, if $aRb \wedge bRa \rightarrow a = b, \forall a, b \in A$.

Let R_1 be a relation from A to B and let R_2 be a relation from B to C; the *composition* of R_1 and R_2 is defined by $R_1 \circ R_2 := \{\langle a, c\rangle \mid a \in A, c \in C, \exists b \in B : aR_1b \wedge bR_2c\}$. The *transitive closure* of a relation R is defined by $R^+ := R \cup R^2 \cup R^3 \cup \ldots$, where $R^{n+1} := R^n \circ R$ and $R^1 := R$. The *reflexive and transitive closure* of a relation R on the set A is defined by $R^+ \cup \{\langle a, a\rangle \mid a \in A\}$.

A relation R on a set A is an *equivalence relation*, if R is reflexive, symmetric and transitive. For any $a \in A$, the *equivalence class* of a is defined by $[a] := \{a' \in A \mid a'Ra\}$, where R is an equivalence relation. An equivalence relation R on a set A partitions A into a number of disjoint non-empty equivalence classes.

2.4 Functions

A relation from a set X to a set Y is a *function*, denoted $f : X \rightarrow Y$, if $\forall x \in X : \exists! y \in Y : xfy$. The set X is the *domain* and Y is the *range* of the function f. The relation xfy, where f is a function, is written $y = f(x)$. Functions are sometimes called maps or mappings.

Definition 2.1 (λ-abstraction) If E is an expression taking values in a domain D, then

$$\lambda x_1, \ldots, x_n.E$$

denotes the function $f : D' \times \ldots \times D' \rightarrow D$ such that $f(x_1, \ldots, x_n)$ for any $x_i \in D'$ may be obtained by evaluating E using the values for $x_1, \ldots, x_n$. □

The *extension* by $\langle x', y'\rangle$ of a function $f : X \rightarrow Y$ is defined by $f[x' \mapsto y'] := \lambda x.$ if $x = x'$ then y' else $f(x)$. The *restriction* of the function $f : X \rightarrow Y$ to values in a certain domain X' is defined by $f\,|_{X'} := \{\langle x', f(x')\rangle \mid x' \in X' \subset X\}$. The *composition* of two functions $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ is the function from X to Z defined by $(f \circ g)(x) := g(f(x)), x \in X, g(f(x)) \in Z$. A relation from a set X to a set Y is a *functional*, denoted $f : X \rightarrow Y$, if X and Y are sets of functions.

A function $f : X \times X \rightarrow Y$ is *commutative*, if $f(x, x') = f(x', x), \forall x, x' \in X$; *associative*, if $f(x, f(x', x'')) = f(f(x, x'), x''), \forall x, x', x'' \in X$. A function $f : X \rightarrow X$ is *idempotent*, if $f(f(x)) = f(x), \forall x \in X$.

Definition 2.2 (order of magnitude) Let f, g be functions of $x_1, \ldots, x_n$. f is of the same *order of magnitude* as g, denoted

$$f(x_1, \ldots, x_n) \in O(g(x_1, \ldots, x_n)),$$

if

$$\exists k_0, k_1 : |f(x_1, \ldots, x_n)| \leq k_0 + k_1 |g(x_1, \ldots, x_n)|, \forall x_1, \ldots, x_n$$

□

A function $f(x)$ is *linear*, if $f(x) \in O(x)$; *quadratic*, if $f(x) \in O(x^2)$, and *cubic*, if $f(x) \in O(x^3)$.

Part I

Foundations of Fixpoint Computations

Chapter 3

Lattice Theory and Fixpoints

This chapter gives a very brief introduction to the theory of ordered sets, lattices and fixpoints. The material is well-known and the presentation relies heavily on Chapters 1–4 of [DP90] and, to some extent, on Chapter 1 of [LLo87]. Further information can be found in [DP90], which is a modern textbook, or in the more advanced treatment [Grä78].

3.1 Ordered Sets

In lattice theory the first objective is to impose a certain structure on sets. This is done by means of a binary relation, relating elements to each other.

Definition 3.1 (order, ordered set) Let P be a set. A relation $\sqsubseteq$ is an *order* on P, if $\sqsubseteq \subseteq P \times P$ is reflexive, anti-symmetric and transitive. A set P is an *ordered set* ordered by $\sqsubseteq$, denoted $\langle P;\ \sqsubseteq\rangle$, if $\sqsubseteq$ is an order on P. □

An order and an ordered set is sometimes called a partial order and a partially ordered set or poset, respectively.

Example 3.1 (Order and ordered set) Let X be a set and form the powerset $\wp(X)$; then the set inclusion-operator, $\subseteq$, is easily seen to be an order on $\wp(X)$, and the ordered set is thus $\langle \wp(X);\ \subseteq\rangle$. □

Two elements, $x, y \in P$ are *incomparable*, symbolically $x \| y$, if neither $x \sqsubseteq y$ nor $y \sqsubseteq x$. Incomparable elements are sometimes said to be unrelated.

Let $\langle P;\ \sqsubseteq\rangle$ be an ordered set and let Q be a subset of P. Then Q can inherit the order relation $\sqsubseteq$ from P; given $x, y \in Q$, $x \sqsubseteq y$ in Q, iff $x \sqsubseteq y$ in P. A new ordered set $\langle Q;\ \sqsubseteq\rangle$ is thus *induced from* P.

The following definition gives two quite special kinds of ordered sets; the first in which any two elements are comparable and, the second, in which no two elements are comparable.

Definition 3.2 (chain, anti-chain) The ordered set $\langle P;\ \sqsubseteq\rangle$ is a *chain*, if $\forall x, y \in P$, either $x \sqsubseteq y$ or $y \sqsubseteq x$. The ordered set $\langle P;\ \sqsubseteq\rangle$ is an *anti-chain*, if $\forall x, y \in P : x \sqsubseteq y$ iff $x = y$. □

Two alternative names for a chain are linearly ordered set and totally ordered set.

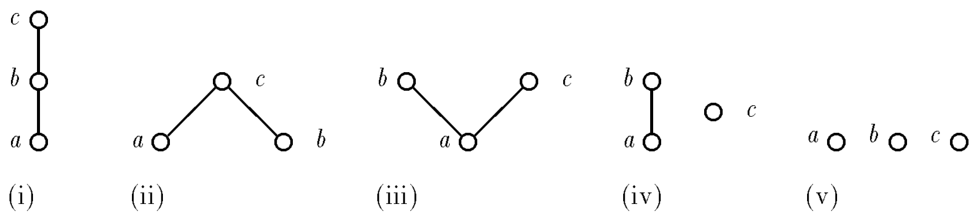


Figure 3.1: Every possible ordered set generated from the set containing a, b, c.

Example 3.2 (Chain) The set of natural numbers, $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$ endowed with the order in which $0 < 1 < 2 \ldots$ becomes the chain $\langle \mathbb{N}_0;\ < \rangle$. □

The order relation $\sqsubseteq$ on P gives rise to the more strict order relation $\sqsubset$: $x \sqsubset y$ in P, iff $x \sqsubseteq y$ and $x \neq y$. The cartesian product $P^n := P_1 \times \cdots \times P_n$ of the ordered sets $P_1, \ldots, P_n$ can be made into an ordered set by imposing the coordinatewise ordering defined by:

$$\langle x_1, \ldots, x_n \rangle \sqsubseteq \langle y_1, \ldots, y_n \rangle \Longleftrightarrow \forall i \in \{1, \ldots, n\} : x_i \sqsubseteq y_i \text{ in } P_i$$

The notion of order is generalized to sets of maps in the following way. An order on the set of maps $Y^X = \{\phi \mid \phi : X \rightarrow Y\}$ is given by putting $\phi \sqsubseteq \phi'$ in Y^X, iff $\phi(x) \sqsubseteq \phi'(x)$ in $Y, \forall x \in X$. When X is a set of n elements, then Y^X collapses into Y^n.

Definition 3.3 (covering) Let $\langle P;\ \sqsubseteq \rangle$ be an ordered set and let $x, y \in P$. y *covers* x, if $x \sqsubset y$ and there is *no* $a \in P$ such that $x \sqsubset a \sqsubset y$. □

It is often useful to visualize ordered sets as (Hasse) diagrams. The *diagram* of an ordered set $\langle P;\ \sqsubseteq \rangle$ represents the elements by small circles. Two elements $x, y \in P$ are connected by a straight line if one covers the other; if x covers y , then the circle representing x is higher than the circle representing y

Example 3.3 (Diagrams) In Figure 3.1 diagrams every possible ordered set generated from $\{a, b, c\}$ (modulo renaming) are given. Diagram (i) shows the chain $a \sqsubseteq b \sqsubseteq c$; in (ii) the order $a \sqsubseteq c$, $b \sqsubseteq c$, $a \| b$ is shown; (iii) represents $a \sqsubseteq b$, $a \sqsubseteq c$, $b \| c$; (iv) shows how $a \sqsubseteq b$, while $c \| a$, $c \| b$; and the anti-chain $a \| b$, $a \| c$, $b \| c$ is depicted in diagram (v). □

Some elements in an ordered set are more important than others and therefore require special consideration.

Definition 3.4 (maximal element, minimal element) Let $\langle P;\ \sqsubseteq \rangle$ be an ordered set and let $S \subseteq P$. Then $x \in S$ is a *maximal element* of S, if $x \sqsubseteq y \in S$ implies $x = y$; $x \in S$ is a *minimal element* of S, if $x \sqsupseteq y \in S$ implies $x = y$; □

Definition 3.5 (greatest element, least element) Let $\langle P;\ \sqsubseteq \rangle$ be an ordered set and let $S \subseteq P$. Then $x \in S$ is the *greatest element* of S, denoted $x = \max S$, if $x \sqsupseteq y, \forall y \in S$. $x \in S$ is the *least element* of S, denoted $x = \min S$, if $x \sqsubseteq y, \forall y \in S$ □

Note that an ordered set may have several maximal (minimal) elements without having a greatest (least) element, that S has a greatest (least) element, iff it has precisely one maximal (minimal) element, and that this element is unique (if it exists).

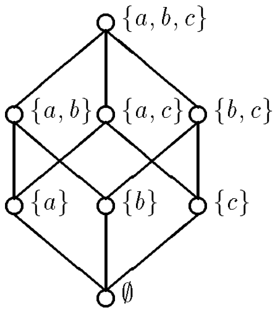


Figure 3.2: A lattice.

Definition 3.6 (top, bottom) Let $\langle P;\ \sqsubseteq\rangle$ be an ordered set. The greatest element of P, if it exists, is called the *top* element of P and is denoted $\top$. The least element of P, if such an element exists, is called the *bottom* element and it is denoted $\perp$. □

Example 3.4 (Top and bottom) In $\langle \wp(X);\ \subseteq\rangle$, $\top = X$ and $\perp = \emptyset$, which is shown in Figure 3.2. □

3.2 Lattices and Complete Lattices

Many important properties an ordered set $\langle P;\ \sqsubseteq\rangle$ can be expressed in terms of the existence of certain upper or lower bounds of subsets of P. Lattices and complete lattices are two classes of ordered sets characterized in this way.

Definition 3.7 (upper bound, lower bound) Let $\langle P;\ \sqsubseteq\rangle$ be an ordered set and let $S \subseteq P$. An element $x \in P$ is an *upper bound* of S, if $s \sqsubseteq x, \forall s \in S$. An element $x \in P$ is a *lower bound* of S, if $s \sqsupseteq x, \forall s \in S$. $S^u = \{x \in P \mid \forall s \in S : s \sqsubseteq x\}$ is the set of all upper bound of S and $S^l := \{x \in P \mid \forall s \in S : s \sqsupseteq x\}$ is the set of all lower bounds of S. □

The difference between maximal (minimal) elements and upper (lower) bounds of a set $S \subseteq P$ is that the former elements are members of S, while the latter elements are members of P, and hence not necessarily members of S.

Definition 3.8 (least upper bound, greatest lower bound) Let $\langle P;\ \sqsubseteq\rangle$ be an ordered set and let $S \subseteq P$. An element $x \in P$ is the *least upper bound* of S, denoted $\operatorname{lub} S$, if x is an upper bound of S, and $x \sqsubseteq y$ for all upper bounds y of S. An element $x \in P$ is the *greatest lower bound* of S, denoted $\operatorname{glb} S$, if x is an lower bound of S, and $x \sqsupseteq y$ for all lower bounds y of S. □

The least upper bound of S is also called the supremum of S and is denoted by $\sup S$; the greatest lower bound of S is sometimes called the infimum of S, denoted $\inf S$. As a notational convention, we define

$$\begin{array}{rcl} x \sqcup y & := & \operatorname{lub}\{x, y\} \\ x \sqcap y & := & \operatorname{glb}\{x, y\} \\ \bigsqcup S & := & \operatorname{lub} S \\ \bigsqcap S & := & \operatorname{glb} S \end{array}$$

when the objects referred to by the right-hand-side of the definitions exist.

Example 3.5 (Least upper bound and greatest lower bound) In $\langle \wp(X);\ \subseteq \rangle$, the least upper bound of a collection of subsets of X is their union ($\cup$) and the greatest lower bound is their intersection ($\cap$). See Figure 3.2. □

Definition 3.9 (lattice, complete lattice) Let $\langle P;\ \sqsubseteq \rangle$ be an ordered set, where P is non-empty. $\langle P;\ \sqsubseteq \rangle$ is a *lattice*, if $x \sqcup y$ and $x \sqcap y$ exist for all $x, y \in P$. $\langle P;\ \sqsubseteq \rangle$ is a *complete lattice*, if $\bigsqcup S$ and $\bigsqcap S$ exist for all $S \subseteq P$ □

$\langle P;\ \sqsubseteq \rangle$ is a lattice, if $x \sqcup y$ and $x \sqcap y$ exist in P for all incomparable pairs $x, y \in P$. It is also a fact that *every finite lattice is complete*, as both the least upper bound and the greatest lower bound then always can be found for every subset. The difference between a lattice and a complete lattice is that a lattice may have certain 'gaps' in it.

Example 3.6 (Incomplete lattices) Every chain is a lattice with $x \sqcup y := \max\{x, y\}$ and $x \sqcap y := \min\{x, y\}$. Thus, $\langle \mathbb{N}_0;\ < \rangle$ is a lattice. It is not complete, however, as it lacks a top element. □

Example 3.7 (Complete lattices) $\langle \wp(X);\ \sqsubseteq \rangle$ is a complete lattice, as the least upper bound ($\cup$) and the greatest lower bound ($\cap$) of every subset of $\wp(X)$ always exist. See Figure 3.2. □

The following theorem shows how to find the complete lattices among the ordered sets.

Theorem 3.1 (Characterization of complete lattices) *Let $\langle P;\ \sqsubseteq \rangle$ be an ordered set, where P is non-empty. Then the following statements are equivalent:*

(i) P is a complete lattice;

(ii) $\bigsqcap S$ exists in P for every subset S of P;

(iii) P has a top element, $\top$, and $\bigsqcap S$ exists in P for every non-empty subset S of P.

Proof. [DP90], 2.16. ■

Definition 3.10 (join-irreducible) Let $\langle L;\ \sqsubseteq \rangle$ be a complete lattice. An element $x \in L$ is *join-irreducible*, if $x \neq \perp$ and $x = a \sqcup b$ implies $x = a$ or $x = b$, for all $a, b \in L$. The set of join-irreducible elements of L is denoted $\mathcal{J}(L)$. □

Example 3.8 (Join-irreducible elements) In a chain every element except $\perp$ is join-irreducible. □

Definition 3.11 (distributive) A complete lattice $\langle L;\ \sqsubseteq \rangle$ is *distributive*, if $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$ for all $x, y, z \in L$. □

Example 3.9 (Distributive lattices) $\langle \wp(X);\ \sqsubseteq \rangle$ and ordinary chains are examples of distributive lattices. □

The concepts below are defined in order to capture the notions of size and finiteness in ordered sets and lattices.

Definition 3.12 (length, finite length, noetherian) Let $\langle P;\ \sqsubseteq \rangle$ be an ordered set.

(i) If $C = \{c_0, c_1, \ldots, c_n\}$ is a finite chain with $|C| = n + 1$, then the *length* of C is n.

(ii) P has *length* n, denoted $\ell(P) = n$, if the length of the longest chain in P is n.

(iii) P has *height* h, if the length of the longest chain starting from $\perp$ is h.

(iv) $\langle P; \sqsubseteq \rangle$ is of *finite length*, if it has length n for some $n \in \mathbb{N}_0$

(v) P is *noetherian*, if every chain in P is finite.

□

Note that there are lattices which are noetherian without being of finite length!

Definition 3.13 (ascending chain condition (ACC)) Let $\langle P; \sqsubseteq \rangle$ be an ordered set. P satisfies the *ascending chain condition (ACC)*, if given any sequence $x_1 \sqsubseteq x_2 \sqsubseteq \ldots \sqsubseteq x_n \sqsubseteq \ldots$ of elements of P, there exists $k \in N$ such that $x_k = x_{k+1} = \ldots$. □

Definition 3.14 (descending chain condition (DCC)) Let $\langle P; \sqsubseteq \rangle$ be an ordered set. P satisfies the *descending chain condition (DCC)*, if any sequence $x_1 \sqsupseteq x_2 \sqsupseteq \ldots \sqsupseteq x_n \sqsupseteq \ldots$ of elements of P, is such that there exists $k \in N$ such that $x_k = x_{k+1} = \ldots$. □

From the above definitions it is clear that a lattice of finite length is noetherian and hence satisfies both (ACC) and (DCC).

The theorems below characterize noetherian ordered sets.

Theorem 3.2 (Infinite chain condition.) *An ordered set is noetherian iff it satisfies both (ACC) and (DCC).*

Proof. [DP90], 2.27. ■

Theorem 3.3 (Completeness of noetherian ordered sets) *A lattice which is noetherian is complete.*

Proof. [DP90], 2.28. ■

3.3 Complete Partial Orders

It is sometimes important to realize an element in an ordered set as the join of a set of approximating elements below it. The concept of complete partial order is defined in order to guarantee that such an element can be consistently found.

Definition 3.15 (directed, consistent) Let S be a non-empty subset of an ordered set $\langle P; \sqsubseteq \rangle$. S is *directed*, if, for every finite subset F of S, there exists $z \in S$ such that $z \in F^u$. S is *consistent*, if for every finite subset F of S, there exists $z \in P$ such that $z \in F^u$. □

Non-consistency arises only in ordered sets without the top element. A directed set is, of course, consistent.

Example 3.10 (Directed sets) In an ordered set, every non-empty chain is directed and every subset with a greatest element is directed. The simplest example of a directed set is a chain. □

Definition 3.16 (complete partially ordered set (CPO), pre-CPO) An ordered set $\langle P; \sqsubseteq \rangle$ is a *complete partially ordered set* (*CPO*), if

(i) P has a bottom element, $\bot$, and

(ii) $\bigsqcup D$ exists for every directed subset D of P.

$\langle P;\ \sqsubseteq\rangle$ is a *pre-CPO*, if it only satisfies requirement (ii). □

Example 3.11 (CPO:s) Every complete lattice is a CPO. □

3.4 Fixpoints

In order to guarantee the existence of fixpoints certain properties of maps between ordered sets are needed.

Definition 3.17 (monotone, continuous, strict, additive, completely additive) Let $\langle P;\ \sqsubseteq_P\rangle$ and $\langle Q;\ \sqsubseteq_Q\rangle$ be ordered sets. A map $\phi : P \to Q$ is *monotone*, if $x \sqsubseteq_P y$ implies $\phi(x) \sqsubseteq_Q \phi(y)$. The map is *continuous*, if for every directed set D in P, $\phi(\bigsqcup D) = \bigsqcup \phi(D)$, where $\bigsqcup \phi(D) := \bigsqcup \{\phi(x) \mid x \in D\}$. The map is *strict*, if $\phi(\bot) = \bot$. The map is *additive*, if $\phi(x \sqcup y) = \phi(x) \sqcup \phi(y)$ for all $x, y \in P$. The map is *completely additive*, if $\phi(\bigsqcup S) = \bigsqcup \{\phi(x) \mid x \in S\}$, where $S \subseteq P$. □

A monotone map is sometimes called order-preserving or isotone. For complete and finite lattices a strict, additive map is also completely additive.

As monotonicity of maps is a desirable property, a definition of a monotonic version of an arbitrary map is sometimes needed.

Definition 3.18 (monotonic closure) Let $\langle P;\ \sqsubseteq_P\rangle$ be an ordered set and let $\phi : P \to Q$ be a map. The *monotonic closure* of ϕ is defined by

$$\phi \uparrow \mathrm{mon} := \bigsqcap \{\phi' \mid \phi \sqsubseteq \phi' \text{ and } \phi' \text{ is monotonic }\}$$

□

The intuition of monotone maps is that where the order represents e.g. 'is less defined than' or 'is a worse approximation than', a monotone map ϕ is one which is such that, the better the input x, the better the output $\phi(x)$.

The n-fold composite, Φ^n, of a map $\Phi : P \to P$ is defined as follows: Φ^n is the identity map, if $n = 0$; and $\Phi^n = \Phi \circ \Phi^{n-1}$ for $n > 0$. If Φ is monotone, so is Φ^n.

Theorem 3.4 *Every continuous map is monotone.*

Proof. [DP90]. Let $\phi : P \to Q$ be a continuous map, which implies that for every directed set D in P,

$$\bigsqcup \phi(D) \sqsubseteq_Q \phi(\bigsqcup D).$$

Now, choose $x, y \in P$ such that $x \sqsubseteq_P y$. Then $D := \{x, y\}$ is directed with $\bigsqcup D = y$, by definition. Hence,

$$\phi(x) \sqsubseteq_Q \phi(x) \sqcup \phi(y) = \bigsqcup \phi(D) \sqsubseteq_Q \phi(\bigsqcup D) = \phi(y)$$

whence $\phi : P \to Q$ is monotone. ■

Note, however, that there are monotone maps which are not continuous.

Definition 3.19 (fixpoint equation, fixpoint, least fixpoint) Let $\langle P; \sqsubseteq\rangle$ be an ordered set and let $\Phi : P \to P$ be a map. A *fixpoint equation* is an equation of the form $\Phi(x) = x$ in which $x \in P$ is a *fixpoint* of Φ. The set of all fixpoints of Φ is defined by $\mathsf{fix}(\Phi) := \{x \in P \mid \Phi(x) = x\}$. The *least fixpoint* of Φ is defined by $\mu(\Phi) :=$ min $\mathsf{fix}(\Phi)$, when min $\mathsf{fix}(\Phi)$ exists. □

Theorem 3.5 (CPO Fixpoint I) *Let $\langle P; \sqsubseteq\rangle$ be a CPO, let $\Phi : P \to P$ be a monotone map and define $\alpha := \bigsqcup_{n\geq 0} \Phi^n(\bot)$.*

(i) If $\alpha \in \mathsf{fix}(\Phi)$, then $\alpha = \mu(\Phi)$.

(ii) If Φ is continuous, then $\alpha = \mu(\Phi)$.

Proof. [DP90].

(i) Certainly, $\bot \sqsubseteq \Phi(\bot)$, whence $\Phi^n(\bot) \sqsubseteq \Phi^{n+1}(\bot)$ by monotonicity of Φ^n. Hence, there is a chain

$$\bot \sqsubseteq \Phi(\bot) \sqsubseteq \ldots \Phi^n(\bot) \sqsubseteq \Phi^{n+1}(\bot) \sqsubseteq \ldots$$

in P. Since P is a CPO, $\alpha := \bigsqcup_{n\geq 0} \Phi^n(\bot)$ exists. Let $\beta \in \mathsf{fix}(\Phi)$. By induction, $\Phi^n(\beta) = \beta$, for all n. As $\bot \sqsubseteq \beta$, $\Phi^n(\bot) \sqsubseteq \Phi^n(\beta) = \beta$, by monotonicity of Φ^n. The definition of α forces $\alpha \sqsubseteq \beta$, which means that if $\alpha \in \mathsf{fix}(\Phi)$, then $\alpha = \mu(\Phi)$.

(ii) It suffices to show that $\alpha \in \mathsf{fix}(\Phi)$.

$$\Phi(\alpha) = \Phi(\bigsqcup_{n\geq 0} \Phi^n(\bot)) = \bigsqcup_{n\geq 0} \Phi(\Phi^n(\bot))$$

since Φ is continuous. Then,

$$\bigsqcup_{n\geq 0} \Phi(\Phi^n(\bot)) = \bigsqcup_{n\geq 1} \Phi^n(\bot) = \bigsqcup_{n\geq 0} \Phi^n(\bot) = \alpha$$

since $\bot \sqsubseteq \Phi^n(\bot)$, for all n. ∎

The following theorems are important for the clues they provide to a systematic search for fixpoints.

Theorem 3.6 (Knaster-Tarski) *If $\langle L; \sqsubseteq\rangle$ is a complete lattice and $\Phi : L \to L$ is a monotone map, then $\bigsqcup\{x \in L \mid x \sqsubseteq \Phi(x)\} \in \mathsf{fix}(\Phi)$*

Proof. [DP90]. Put $G = \{x \in L \mid x \sqsubseteq \Phi(x)\}$ and $g = \bigsqcup G$. By definition, $x \sqsubseteq g, \forall x \in G$, so $x \sqsubseteq \Phi(x) \sqsubseteq \Phi(g)$, by monotonicity of Φ. Thus $\Phi(g) \in G^u$, whence $g \sqsubseteq \Phi(g)$.

Since Φ is monotone, $\Phi(g) \sqsubseteq \Phi(\Phi(g))$, hence $\Phi(g) \in G$, so $\Phi(g) \sqsubseteq g$. This, together with the fact that $g \sqsubseteq \Phi(g)$, says that $g = \Phi(g)$, so $g \in \mathsf{fix}(\Phi)$. ∎

Theorem 3.7 (Tarski) *If $\langle L; \sqsubseteq\rangle$ is a complete lattice and $\Phi : L \to L$ is a monotone map, then Φ has a least fixpoint $\mu(\Phi) = \bigsqcap\{x \mid \Phi(x) = x\} = \bigsqcap\{x \mid \Phi(x) \sqsubseteq x\}$.*

Proof. [LLo87]. Let $G = \{x \in L \mid \Phi(x) \sqsubseteq x\}$, $g = \bigsqcap G$, and, similarly, let $G' = \mathsf{fix}(\Phi) = \{x \in L \mid \Phi(x) = x\}$ and $g' = \bigsqcap G'$. As $\langle L; \sqsubseteq\rangle$ is a complete lattice, both g and g' exists. Three steps are necessary in order to show that $g = g'$ is the least fixpoint of Φ; $g \in G$, $g = \Phi(g)$ and $g = g'$.

Now, $g \sqsubseteq x, \forall x \in G$, by definition of greatest lower bound; hence by monotonicity of Φ, $\Phi(g) \sqsubseteq \Phi(x), \forall x \in G$. Thus $\Phi(g) \sqsubseteq x, \forall x \in G$, so, by definition of glb, $\Phi(g) \sqsubseteq g$, which means that $g \in G$.

To show that $g \in \textsf{fix}(\Phi)$, it remains to be shown that $g \sqsubseteq \Phi(g)$, since it is known from above that $\Phi(g) \sqsubseteq g$. From $\Phi(g) \sqsubseteq g$ it follows by monotonicity of Φ that $\Phi(\Phi(g)) \sqsubseteq \Phi(g)$ which implies that $\Phi(g) \in G$. Hence, $g \sqsubseteq \Phi(g)$ so that $g \in \textsf{fix}(\Phi)$.

Since $g \in \textsf{fix}(\Phi)$, it is known that $g' \sqsubseteq g$. On the other hand, $G' \subseteq G$, so $g \sqsubseteq g'$. Thus, $g = g'$. ∎

Theorem 3.8 (CPO Fixpoint II) *If $\langle P; \sqsubseteq \rangle$ is a CPO and $\Phi : P \to P$ is a map such that $x \sqsubseteq \Phi(x)$ for all $x \in P$, then Φ has a fixpoint.*

Proof. [DP90], 4.14 (hard). ∎

Theorem 3.9 (CPO Fixpoint III) *If $\langle P; \sqsubseteq \rangle$ is a CPO and $\Phi : P \to P$ is a monotone map, then Φ has a* least *fixpoint.*

Proof. [DP90] 4.15. ∎

3.5 Summary

To summarize; the existence of a least fixpoint is guaranteed when the ordered set is a CPO and the map is continuous or monotone; or, when the ordered set is a complete lattice and the map is monotone.

Noetherian ordered sets and finite ordered sets are complete lattices and complete lattices are CPO:s.

Chapter 4

Graph Theory

Graph theory is very important in fixpoint computations, as relations between equations can be described by a certain kind of directed graphs called dependency graphs (Chapter 6). Hence, in this chapter, basic material from graph theory is presented. A modern textbook where a good treatment of graph theory can be found is [Big85].

4.1 Basic Concepts

Intuitively, a directed graph is just a generalization of an ordered set in the sense that the only requirement on the relation holding between objects in a directed graph is that it should have a direction, i.e. the members of the relation should be ordered pairs.

Definition 4.1 (directed graph, vertices, edges) Let $V = \{v_1, \ldots, v_n\}$ and let $E = \{\langle v_i, v_j\rangle \mid 1 \leq i \leq n, 1 \leq j \leq n\} \subseteq V \times V$. $G = \langle V, E\rangle$ is a *directed graph* on V, where V is the set of *vertices* and E is its set of *edges*. □

Directed graphs, vertices and edges are sometimes called digraphs, nodes and arcs, respectively. An edge between vertices v and w is sometimes denoted $v \rightarrow w$.

Definition 4.2 (path) A sequence of vertices $\{v_1, v_2, \ldots, v_{n-1}, v_n\}$ in a directed graph $G = \langle V, E\rangle$ is a *path* between vertices v_1 and v_n, if $\{\langle v_1, v_2\rangle, \ldots, \langle v_{n-1}, v_n\rangle\}$ is a set of edges. □

Note that an empty sequence of vertices is allowed by the definition.

Definition 4.3 (reachable) Let $G = \langle V, E\rangle$ be a directed graph. A vertex $w \in V$ is *reachable* from a vertex $v \in V$, if there is a path between v and w. □

Definition 4.4 (cycle) Let $G = \langle V, E\rangle$ be a directed graph. A path between vertices v_1 and v_n is a *cycle*, if $v_1 = v_n$. □

Example 4.1 (Paths, cycles and reachability) In Figure 4.1, there are two paths connecting vertices 1 and 4; $\{1, 4\}$ and $\{1, 2, 3, 4\}$, but there is no cycle. Every vertex is connected to itself by an empty path. Vertex 4 is reachable from every other vertex, whereas vertex 1 is unreachable. In Figure 4.4, there are two cycles: $\{\langle 1, 2\rangle, \langle 2, 3\rangle, \langle 3, 1\rangle\}$ and $\{\langle 1, 4\rangle, \langle 4, 3\rangle, \langle 3, 1\rangle\}$; moreover every vertex is reachable from every other vertex. □

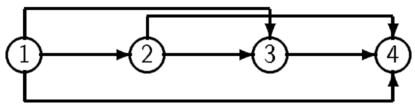


Figure 4.1: The directed graph $G = \langle \{1, 2, 3, 4\}, \{\langle i, j\rangle \mid i < j\}\rangle$.

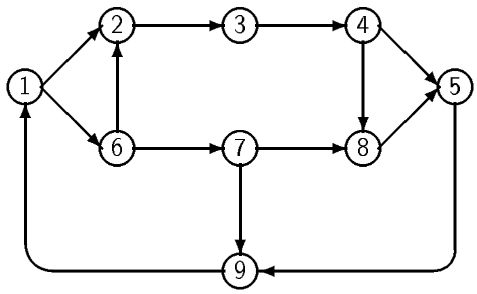


Figure 4.2: Chains in a directed graph.

A chain in a directed graph [SS95] consists of sequence of connected vertices, where each vertex has only one ingoing and one outgoing edge, with the exception of the starting vertex which may have several ingoing edges, and the end vertex which may have several outgoing edges. Chains in graph theory should not be confused with chains in lattice theory.

Definition 4.5 (chain) Let $G = \langle V, E\rangle$ be a directed graph. A path $\{v_1, v_2, \ldots, v_{n-1}, v_n\}$ between vertices v_1 and v_n is a *chain*, if $\forall v_i, 1 < i \leq n : (\exists! v : \langle v, v_i\rangle \in E) \wedge \forall v_i, 1 \leq i < n : (\exists! w : \langle v_i, w\rangle \in E)$. □

Example 4.2 (Chains) In Figure 4.2, the only chains are $\{9, 1\}$ and $\{2, 3, 4\}$. □

4.2 Reducible Graphs

The following concepts lead to the definition of reducible graphs, which are often used as flow-charts when modeling inter-procedural execution of languages without the "goto"-construct [ASU86].

Definition 4.6 (head, tail, initial vertex) Let $G = \langle V, E\rangle$ be a directed graph and let $e = \langle v, w\rangle$ be an edge in G. The *head* of e is v and the *tail* of e is w. A vertex $u \in V$ is the *initial vertex* of G, if $\exists! u \in V : \nexists u' : \langle u', u\rangle \in E$. □

Definition 4.7 (domination) Let $G = \langle V, E\rangle$ be a directed graph. A vertex v *dominates* vertex w, if every path between the initial vertex of G and w contains v. □

Definition 4.8 (forward edges, back edges, reducible) Let $G = \langle V, E\rangle$ be a directed graph. The set $F \subseteq E$ is the set of *forward edges* in G, if $G' = \langle V', F\rangle$ forms an acyclic subgraph of G in which every vertex can be reached from the initial vertex of

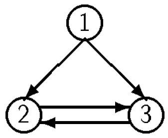


Figure 4.3: An irreducible, directed graph.

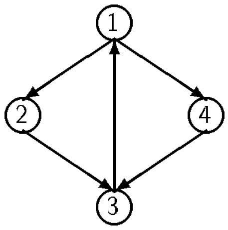


Figure 4.4: A directed graph which consists of a single, strongly connected component.

G. The set $B \subseteq E$ is the set of *back edges* in G, if every edge in B has a head which dominates its tail. G is *reducible*, if $E = F \cup B, F \cap B = \emptyset$. □

The most simple example of an irreducible directed graph is given in Figure 4.3.

4.3 Orders on Graphs

In the context of fixpoint computations, the ordering of the vertices of a directed graph is crucial, as such an order (when possible) provides the fixpoint solver with a very good order of computation.

Definition 4.9 (topological order) Let $G = \langle V, E \rangle$ be a directed graph. An linear order relation $\preceq$ on G is a *topological order*, if G is acyclic and $\langle v, w \rangle \in E \Rightarrow v \preceq w$. □

From the above definition, it is clear that a large number of directed graphs are impossible to order topologically; these are the graphs containing cycles.

Strongly connected components (SCC) of a directed graph are subgraphs where every vertex is reachable from any other vertex.

Definition 4.10 (path equivalence, strongly connected component (SCC)) Let $G = \langle V, E \rangle$ be a directed graph. Two vertices v and w are *path equivalent*, symbolically vRw, if there is a path from v to w and a path from w to v. The distinct equivalence classes formed by the path equivalence relation R are $V_i, 1 \leq i \leq n$. The subgraph $G_i = \langle V_i, E_i \rangle$ is a *strongly connected component* of G, if $E_i = \{\langle v, w \rangle \in E \mid v, w \in V_i\}$. □

As every vertex is reachable from the other vertices in Figure 4.4, this graph is strongly connected. If the edges are thought of as representing dependencies (Definition 6.1), a strongly connected component is a subgraph where a change in a vertex may affect the whole component.

Part II

Fixpoint Computations in Static Program Analysis

Chapter 5

Program Analysis

The purpose of this chapter is to establish the connection between (static) program analysis and the computation of fixpoints, and to introduce the analysis framework for AKL.[1] Good introductions to static program analysis and analysis of AKL are [MJ81] and [SS95], respectively.

5.1 Basic Concepts

Program analysis is a way of extracting semantic information from a program; *static program analysis* is a method for discovering (approximate) global properties of the run-time behaviour of a program on arbitrary input data without actually running it, whereas *dynamic program analysis* is a way of analyzing a program by running it on various sets of input data, collecting statistics and extrapolate the behaviour on other data sets from these facts.

The *syntax* of a programming language deals with the structure of expressions, whereas the *semantics* gives an interpretation by way of supplying a meaning to each expression in the language. An *operational semantics* is a description of how the expressions of a programming language behaves in an operational sense in terms of an abstract machine defined by its state, instructions and transition rules. A *denotational semantics*, on the other hand, is a set of functions mapping syntactic structures (representing expressions) in a programming language to the abstract values which they denote. The operational approach is thus to give a notion of what a program *does* in terms of an execution mechanism and the denotational approach is to provide an explanation of what the program *mean* independent of any execution mechanism.[2]

5.2 Static Program Analysis

Static program analysis is a tool for discovering properties of the run-time behaviour of program without actually running it, taking both control flow and data flow into account. Whereas the actual execution of a program produces one sequence of states the data and control pass through, static program analysis produces a *static semantics* defined as follows: for each control point in the program, a finite description of the set of

[1] Appendix A gives a brief introduction to the Agents Kernel Language (AKL).

[2] See [Sto77] for a textbook on semantics in general and denotational semantics in particular.

data states the program could be in when any conceivable execution passes through that point, is given. The goal is thus to produce descriptions of certain program properties which applies to all possible executions of the program for arbitrary input values and the idea is to use the information found when:

- compiling programs, giving smaller, faster, less memory consuming code.
- debugging programs, finding dead code and type incorrect calls.

Normally, analysis is *global*, which means that no part of the program being analyzed is investigated in isolation.

The descriptions of program properties must be approximate, since Rice's theorem [HU79] says that all interesting extensional properties of programs are undecidable.[3] Examples of interesting (and thus undecidable) program properties are "The program terminates" and "Variables x and y must be aliased at some point during the program execution". The descriptions must therefore be approximate in the sense that they cannot provide full information, not in the sense that they may be wrong! Other examples of program properties which may be of interest include: call/exit-patterns of procedures, reachable/non-reachable code and memory usage.

The main building blocks of static program analysis are:

- the *program properties* of interest
- the *program points* which probe the program structure
- the *program functions* modeling an approximate execution by relating program properties and program points

Typical program properties of interest include possible values of variables at run-time, whereas program points e.g. could be located between statements or at procedure entry/return in the program to be analyzed. The program functions can model concurrent execution of statements or ordinary procedure calls which introduce change in scope.

These intuitive notions are captured by concepts from lattice theory. Program properties are modeled by a complete lattice or a complete partial order L; thus an order relation holding between the program properties must exist. If the program property investigated is the possible values of variables, the top and bottom element of the lattice may represent that the variable in question can be bound to arbitrary values and that the corresponding program point is not reachable, respectively, whereas an intermediate value indicate that a variable may be bound to certain ranges or types. A set of variables x_i ranging over program properties is used to hold information about these properties at certain program points. Every program point i then has an associated function ϕ_i which computes the program property valid at this point. All this leads to an equation system

$$x_i = \phi_i(x_1, \ldots, x_n), x_i \in L, 1 \leq i \leq n$$

which must be solved either symbolically or iteratively. The engine for solving this equation system is thus the heart of static program analysis, the vehicle which performs the approximate execution. In this thesis, the method used by the engine is based on finding the fixpoint of the equation system by means of iteration.

[3] An *extensional* program property is a property which depends only the input/output behaviour of a program. An *interesting* property is any property except the two trivial properties: **true** which holds for every program and **false** which holds for no program.

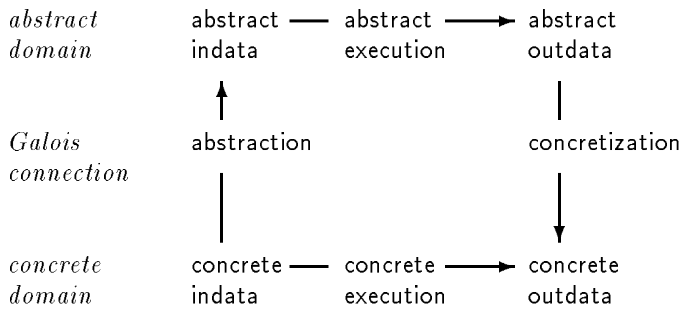


Figure 5.1: Abstract interpretation and the Galois connection.

5.3 Abstract Interpretation

The theory of abstract interpretation [AH87, CC92a, CC92b] facilitates a systematic approach to the problem of extracting properties from programs in order to provide provably correct answers to questions about their run-time behaviour. These properties are properties of some semantic function which characterizes certain aspects of the program. The domain of the semantic function consists of the program properties of interest.

There are two ways of understanding abstract interpretation; (i) to design hierarchies of interrelated semantics specifying the behaviour of programs in various levels of detail, hence, to understand the program in certain specific ways. (ii) to design automatic, static program analysis tools for determining dynamic properties of programs, hence to execute the program on a non-standard domain of values and operators. The first way interprets (!) interpretation as *understanding*, while the second emphasizes *execution*.

The first aspect of abstract interpretation can thus be used in formalizing the notion of approximate program properties in static program analysis by relating the concrete semantics to an abstract semantics modeling certain interesting properties of the program and its execution. This relationship between concrete and abstract is provided by a *Galois connection* (see Figure 5.1), which guarantees that the approximate program properties are correct.

In short, the main idea of static program analysis is to project the infinite concrete domain of the original program onto a finite abstract domain, where it is possible to execute the program completely; the guarantee that this approximation is correct is then given by abstract interpretation based on the Galois connection.

5.4 Analysis of AKL

The purpose of analyzing AKL [SS95] is to provide the compiler with information which facilitates optimization of guard computations, leading to faster program execution and less memory consumption.[4] The analyzer provides the compiler with:

- *Alias information:* two variables are definitely not aliased at program point i
- *Type information:* a variable has type T at program point i

[4]Exactly how the AKL compiler can be enhanced by using global, static program analysis is described in [Bra94].

The interesting program properties, alias and type, are modeled by a complete lattice called the AT-domain. The value of a program property is of the form:

$$\langle \mathit{Aliased}, \mathit{Types} \rangle$$

where *Aliased* is a set of variables which may be aliased, i.e. refer to the same value, and *Types* is a set consisting of descriptions of the structures (name, arity, arguments) the variables in *Aliased* may be bound to. The bottom value of the lattice, $\perp$, is a dead code indicator and other values facilitates the decision whether two variables are unaliased at a certain program point in every possible execution and that a variable is always bound to e.g. a list, hence providing type information.

The two sets *Aliased* and *Types* are implemented by bit vectors, represented as large integers (BigNums), and the set operations are thus implemented as boolean functions on bit vectors. The use of bit vectors instead of lists for implementing sets speeds up execution.

Each construct in a normalized AKL program has a corresponding program point which consists of a pair $\langle pre, part \rangle$, where the intuition is that a value of a *pre*-point depends on all the program points leading to this *pre*-point and that the value of a *part*-point reflects a state of partial computation, e.g. suspended processes and parallel execution of non-atomic processes.

The result of possibly concurrent processes is modeled by the function **conc** and alternative computations are modeled by the function **lub**. These are the two most important program functions in the AKL analysis framework. Change in scope during execution is modeled by the functions **call** and **return**; built-in AKL-functions are handled by **builtin**.

One of the merits of the analyzer framework, written in AKL, is that it is modular; it separates

- the setting up of the equation system
- the implementation of program functions
- the fixpoint computation procedure

facilitating experimentation with and optimization of each module separately. This fact is used in Chapter 7, where different fixpoint solvers are compared.

Chapter 6

A Survey of Fixpoint Computations

The most time-consuming part in static program analysis is spent in the fixpoint solver. Sadly enough, this area has been much neglected in the past, but if static program analysis is to be a usable tool in software development, the execution times for analyzing programs must be considerably shorter than they are today. In recent years, however, more effort has been put into research in the area and this chapter features a survey of fixpoint computations in static program analysis. Chapter six in Nilsson's thesis [Nil92] and the paper by Vergauwen et al. [VWL94] are two highly readable accounts of fixpoint computations.

6.1 The Problem

In static program analysis, the program properties, program points and program functions add up to give an equation system

$$x = \phi(x)$$

which can be written as

$$x_i = \phi_i(x_1, x_2, \ldots, x_n), 1 \leq i \leq n$$

using the notation

$$\begin{aligned} x &:= \langle x_1, x_2, \ldots, x_n \rangle \\ \phi(x) &:= \langle \phi_1(x), \phi_2(x), \ldots, \phi_n(x) \rangle \end{aligned}$$

where each x_i belong to a CPO or complete lattice $\langle P; \sqsubseteq \rangle$, and the functions are monotone or continuous. These last requirements are needed in order to guarantee that the system has a solution.

The problem is to find a solution to this equation system as efficiently as possible. This is done by finding the (least) fixpoint of the system. Hence, the goal is to find an algorithm for computing the desired fixpoint which minimizes the total execution time. The total execution time is minimized by decreasing

- the number of function evaluations

- the cost of comparisons
- the number of comparisons needed to detect stability
- the cost of computing a new value of a function

6.2 Basic Concepts

At this point, some terminology must be introduced. A fixpoint computation is *local*, if the arguments to a fixpoint found are restricted to some subspace of the argument space of the functional for which it is sought; *global*, if no restriction is placed on the argument space of the fixpoint of a functional. An equation is *stable*, if an evaluation of the function does not lead to a new value.

Granularity

Le Charlier and Van Hentenryck introduce the following terminology concerning *granularity* for comparing static program analysis frameworks. The granularity is *finer*, if more program points are considered; *coarser* otherwise [LCVH92].

By analogy with partial evaluation, the granularity is *polyvariant*, if the analysis deals with multiple abstract descriptions for each object in the program depending on the context in which these objects are used and *monovariant*, when only one abstract description for each object in the program is considered. An example of polyvariance is analysis of logic programs where the resulting fixpoint is a map of predicate inputs to predicate outputs.

Finer, polyvariant granularity is likely to giver better results at the price of a higher computation cost compared to coarser, monovariant granularity.

Dependencies

How are the equations in the equation system related to each other? There are in fact certain *dependencies* [O'K87] between the equations; say that if function i needs the value of function j (j is one of its arguments), then function i depends on function j and function j should perhaps be evaluated before function i. The answer is thus to use these dependencies in the following way: a function which depends on values of other functions should be reevaluated only when the values the functions on which it depends have changed. The formal definition of these concepts is this:

Definition 6.1 (dependencies, dependency graph) Let

$$x_i = \phi_i(x_1, x_2, \ldots, x_n), 1 \leq i \leq n.$$

be an equation system. An equation j *depends on* equation i, denoted $\langle i, j \rangle$, if $x_j = \phi_j(\ldots, x_i, \ldots)$. A directed graph is a *dependency graph*, if its vertices represent the dependencies of an equation system. □

A dependency graph is *dynamic*, if the dependencies are changing with time; and *static* otherwise. As higher-order languages do not have static flow graphs (since higher-order programs treat procedures and functions as values leading to an intertwined control- and data flow), analysis of such languages leads dynamic dependency graphs, as do polyvariant granularity.

When speaking of dependency graphs (where each vertex has an associated equation), to visit to a vertex means to evaluate the function in the equation, and, to traverse the graph in a certain order means to evaluate the functions in this order.

In order to formulate computation strategies, a *stabilization operator*, denoted $[\cdot]^*$, is sometimes needed. Applied to a sequence of vertices, it works by computing the equations represented by the vertices in the order given, repeating this procedure until every equation in the sequence is stable.

Optimizations

In order to classify various optimizations Sahlin and Sjöland [SS93] have introduced the following terminology. Optimizations which guarantee an improvement are *algorithmic*; those giving an improvement in practice are *heuristic*. The dependency on a certain domain or on specific properties of functions in the equation system is expressed by saying that the optimizations are either *domain (function) dependent* or *domain (function) independent*.

6.3 A Declarative View

In order to illustrate the above concepts, this section follows [VWL94] in giving a declarative view on how to solve the problem. The equation system is defined by:

$$x_i = \phi_i, 1 \leq i \leq n$$

where

$$\begin{aligned} \phi_i &: \Theta \to \mathcal{V} \\ \Theta &: X \to \mathcal{V} \\ X &= \{x_1, \ldots, x_n\} \end{aligned}$$

An element $\theta \in \Theta$ is a solution of the above equation system, iff $\theta(x_i) = \phi_i(\theta), \forall x_i \in X$.

A data structure $val : X \to \mathcal{V}$ is used in the algorithm to store the current values of the variables. In the naive scheme, in each iteration a variable $x_i \in X$ is chosen, and $val(x_i)$ is updated by a call to the function $EvalRHS(x_i, val)$ which returns $\phi_i(val)$.

In order to minimize the number of evaluations, a computation ordering which avoids selecting equations that are currently stable should be used. If a workset $ws \subseteq X$ is known to contain only variables which are potentially unstable, then only variables in ws should be selected for re-evaluation. An invariant expressing the condition on ws is

$$(I_{ws}) \qquad \forall x_i \notin ws : \phi_i(val) = val(x_i)$$

i.e. all variables outside ws are stable. Now, (i) I_{ws} should be invariant and (ii) the size of ws as small as possible. This is achieved by a call to the procedure $Restore(x_i, ws)$ whenever the value of variable x_i changes.

The simplest implementation of $Restore(x_i, ws)$ is to schedule all variables for re-evaluation, i.e. $ws := X$, but this breaks requirement (ii). The solution to this is to keep track of dependencies between variables and right hand sides of equations.

Monovariance. Define $Arg(\phi_i)$ as the set of argument variables needed in order to evaluate ϕ_i. When modifying $val(x_i)$, only variables that depend on x_i are affected and hence may become unstable. Therefore

$$Restore(x_i, ws) := [ws \leftarrow ws \cup mathit Dep(x_i)]$$

where

$$Dep : X \rightarrow \wp(X)$$

$$Dep(x_i) := \{x_j \in X \mid x_i \in Arg(\phi_j)\}$$

Polyvariance. Take into account that the context val is changing and that this may affect dependencies. Define $arg(x_i)$ as the set of variables that x_i *currently* depends on. Introduce the predicate $\mathit{Suff}(\phi, \theta, A)$ which holds iff the variables in A are sufficient to evaluate ϕ in context θ and the new invariant

$$(I_{dep}) \qquad \forall x_i \notin ws : \mathit{Suff}(\phi_i, val, arg(x_i))$$

The function $EvalRHS(x_i, val)$ is extended so as to return an additional set A consisting of variables needed in evaluating $\phi_i(val)$, i.e. $A \subseteq Arg(f_i)$ and $\mathit{Suff}(f_i, val, A)$

Assuming invariance of I_{dep}

$$Restore(x_i, ws) := [ws \leftarrow ws \cup dep(x_i)]$$

where

$$dep : X \rightarrow \wp(X)$$

$$dep(x_i) = \{x_j \notin ws \mid x_i \in arg(\phi_j)\}$$

Dynamic dependencies are more interesting when the right hand sides of the equations are sparse. A function ϕ is *sparse*, if for most contexts θ only a small part of θ is needed in order to evaluate $\phi(\theta)$, as for example in boolean contexts.

Local. Local algorithms try to compute a certain part of $\mu(\phi)$ by investigating only a small fragment of the equation system. Instead of stabilizing all variables as in the global algorithm, the local algorithm tries to construct S, a stable subset of variables, called the *search space*. In addition to being stable, this set of variables should also be complete, which means that variables in S do not depend on values of variables outside S. The goal is thus to construct a search space $S \supseteq \{x\}$ such that

$$Stable(S, val) \wedge Complete(S, val)$$

where

$$Stable(S, val) := [\forall x_i \in S : val(x_i) = \phi_i(val)]$$

and

$$Complete(S, val) := [\forall x_i \in S : \mathit{Suff}(\phi_i, val, S)]$$

which means that $\forall x_i \in S : \mu(\phi)(x_i) = val(x_i)$. To achieve this, the procedures *Update* and *Expand* are defined. *Update*: Variables of S that are in ws are potentially unstable and must be reevaluated and updated (by *Restore*) just as in the global algorithm. *Expand*: S is extended when there are stable variables in S which depend on variables outside S.

The search space should be kept small, as variables which were added to S may cease to be needed, but can still cause further, useless expansions of the search space. Such variables (or, in some cases, calls) are called *spurious* [Ros93]. To keep the search space small, the procedure *Update* should be given priority over *Expand*. Another search space reduction technique is to remove all variables in S that are no longer directly or indirectly reachable from x.

6.4 Iteration Strategies

An *iteration strategy* is a heuristic for choosing the order in which the functions in the system of equations should be evaluated. This approach is a way of minimizing the number of evaluations necessary to find the least fixpoint of the system. As the number of equations typically is very large (≥ 100), this idea is crucial.

The *Jacobi iteration strategy* has its origin in numerical analysis where it is used in finding a solution to a set of linear equations. If x^k denotes the value of x in iteration step k, then the Jacobi iteration strategy amounts to the following sequence of successive approximations:

$$\begin{cases} x_1^{k+1} &= \phi_1(x_1^k, x_2^k, \ldots, x_{n-1}^k, x_n^k), \\ x_2^{k+1} &= \phi_2(x_1^k, x_2^k, \ldots, x_{n-1}^k, x_n^k), \\ & \vdots \\ x_n^{k+1} &= \phi_n(x_1^k, x_2^k, \ldots, x_{n-1}^k, x_n^k). \end{cases}$$

The *Gauss-Seidel iteration strategy* is an improvement to the previous scheme which uses those values already computed in iteration $k+1$ when computing the new values x_i^{k+1};

$$\begin{cases} x_1^{k+1} &= \phi_1(x_1^k, x_2^k, \ldots, x_{n-1}^k, x_n^k), \\ x_2^{k+1} &= \phi_2(x_1^{k+1}, x_2^k, \ldots, x_{n-1}^k, x_n^k), \\ & \vdots \\ x_n^{k+1} &= \phi_n(x_1^{k+1}, x_2^{k+1}, \ldots, x_{n-1}^{k+1}, x_n^k). \end{cases}$$

The above two iteration strategies are special cases of the *chaotic iteration strategy* [Cou81, CC92a], which intuitively states that in each step in the sequence of iterations, an arbitrary equation i may be chosen, as long as no equation is forgotten indefinitely (fairness). To capture this notion, the following two concepts are needed:

Definition 6.2 (weak fairness) Let $J \in \wp(\{1, 2, \ldots, n\})$. A sequence of such subsets, $J_k, k \geq 0$, is *weakly fair*, if $\forall k \geq 0 : \forall i \in \{1, 2, \ldots, n\} : \exists l \geq 0 : i \in J_{k+l}$. □

Definition 6.3 (ascending sequence of chaotic iterations) Let ϕ_J be the map defined by

$$\phi_J(x_1, x_2, \ldots, x_n) := \langle y_1, y_2, \ldots, y_n \rangle$$

where, for $i \in \{1, 2, \ldots, n\}$

$$y_i = \begin{cases} \phi_i(x_1, x_2, \ldots, x_n) & \text{if } i \in J \\ x_i & \text{if } i \notin J. \end{cases}$$

Let $\langle P^n; \sqsubseteq \rangle$ be an ordered set. A sequence x^k of vectors in P^n is an *ascending sequence of chaotic iterations* for F, if

$$x^k = \begin{cases} \langle \perp, \ldots, \perp \rangle & \text{if } k = 0. \\ \phi_{J_{k-1}}(x^{k-1}) & \text{if } k \geq 1. \end{cases}$$

where the sequence J_k is weakly fair. □

Different chaotic iteration strategies can thus be obtained by varying the specification of the choice sequence J_k, which "guides" the computation.

Example 6.1 (Jacobi and Gauss-Seidel as chaotic iteration strategies) First, note that $\phi_{\{i\}} = \phi_i$ and $\phi_{\{1,2,\ldots,n\}} = \phi$ by Definition 6.3. The Jacobi iteration strategy is obtained by choosing $J_k = \{1, 2, \ldots, n\}$, whereas the choice $J_k = \{(k \bmod n) + 1\}$ corresponds to the Gauss-Seidel iteration strategy. □

If it is known that an iteration strategy results in an ascending sequence of chaotic iterations, this fixpoint computation is guaranteed to converge in a finite number of steps.

Theorem 6.1 (Convergence of ascending sequences of chaotic iterations) *Let $\langle P^n;\ \sqsubseteq\rangle$ be a CPO or a complete lattice. If x^k is an ascending sequence of chaotic iterations for the map $\phi : P^n \to P^n$, then $\mu(\phi) = \bigsqcup_{k\geq 0} x^k$.*

Proof. [CC92a]. The proof consists of four steps:

(i) Let $x \sqsubseteq \phi(x) \sqsubseteq \mu(\phi)$. Now, $x_i \sqsubseteq \phi_i(x), 1 \leq i \leq n$; therefore, if $i \in J$, then $x_i \sqsubseteq \phi_i(x) = \phi_J(x)[i]$; otherwise $x_i \sqsubseteq \phi_J(x)[i] \sqsubseteq \phi_i(x)$, where $\phi_i(x) := \phi(x)[i]$. In both cases, $x_i \sqsubseteq \phi_J(x)[i]$; hence

$$x \sqsubseteq \phi(x) \sqsubseteq \mu(\phi) \Rightarrow \forall J \subseteq \{1, \ldots, n\} : x \sqsubseteq \phi_J(x) \sqsubseteq \phi(x) \sqsubseteq \mu(\phi). \tag{6.1}$$

(ii) Prove that

$$\forall k \geq 0 : x^k \sqsubseteq x^{k+1} = \phi_{J_k}(x^k) \sqsubseteq \phi(x^k) \sqsubseteq \mu(\phi) \tag{6.2}$$

Induction; $k = 0$: for $x^0 \sqsubseteq \mu(\phi), x^0 \sqsubseteq \phi(x^0) \sqsubseteq \phi(\mu(\phi)) = \mu(\phi)$ by monotonicity of ϕ and the definition of fixpoints; hence $x^0 \sqsubseteq x^1 = \phi_{J_0}(x^0) \sqsubseteq \phi(x^0) \sqsubseteq \mu(\phi)$ by Equation 6.1.

For the induction step, $k > 0$, assume that $x^{k-1} \sqsubseteq x^k = \phi_{J_{k-1}}(x^{k-1}) \sqsubseteq \phi(x^{k-1}) \sqsubseteq \mu(\phi)$. If $i \in J_{k-1}$, then $x_i^{k-1} = \phi_i(x^{k-1}) \sqsubseteq \phi_i(x^k) \sqsubseteq \mu(\phi_i)$ since $x^{k-1} \sqsubseteq x^k \sqsubseteq \mu(\phi)$ and ϕ_i is monotone. Otherwise, $i \notin J_{k-1}$ and $x_i^{k-1} \sqsubseteq x_i^k \sqsubseteq \phi_i(x^{k-1}) \sqsubseteq \mu(\phi_i)$ by the induction hypothesis, so that $x_i^k = \phi_i(x^{k-1}) \sqsubseteq \phi_i(x^k) \sqsubseteq \mu(\phi_i)$ by monotonicity of ϕ_i. In both cases, $x_i^k = \phi_i(x^k) \sqsubseteq \mu(\phi_i)$ for $1 \leq i \leq n$; therefore, $x^k \sqsubseteq \phi(x^k) \sqsubseteq \mu(\phi)$ which proves that $x^k \sqsubseteq x^{k+1} = \phi_{J_k}(x^k) \sqsubseteq \phi(x^k) \sqsubseteq \mu(\phi)$.

(iii) If $1 \leq i \leq n$ and $k \geq 0$, then, by weak fairness (Definition 6.2), $\exists \ell(i) : i \in J_{k+\ell(i)}$. This implies that $x_i^{k+\ell(i)+1} = \phi_i(x^{k+\ell(i)})$. Now, by induction using (6.2) above, $x^k \sqsubseteq x^{k+\ell(i)}$ so that by monotonicity, $\phi_i(x^k) \sqsubseteq x_i^{k+\ell(i)+1} \sqsubseteq x_i^m$ where m is the maximum of $\ell(i)$ for $1 \leq i \leq n$. Whence

$$\forall k \geq 0 : \exists m \geq k : \phi(x^k) \sqsubseteq x^m \tag{6.3}$$

(iv) Let $x^\omega := \bigsqcup_{k\geq 0} x^k$. By (6.2), definition of least upper bounds and continuity, $x^\omega \sqsubseteq \bigsqcup_{k\geq 0} \phi(x^k) = \phi(x^\omega)$. By (6.3), $\forall k \geq 0 : \phi(x^k) \sqsubseteq \bigsqcup_{m\geq 0} x^m$, which means that $\phi(x^\omega) = \bigsqcup_{k\geq 0} \phi(x^k) \sqsubseteq x^\omega$. Hence $x^\omega \in \mathsf{fix}(\phi)$, and by (6.2), $x^\omega \sqsubseteq \mu(\phi)$, whence $x^\omega = \mu(\phi)$, by definition of fixpoints. ■

Theorem 6.2 (Termination of ascending sequences of chaotic iterations) *Let $\langle P^n;\ \sqsubseteq\rangle$ be a CPO or a complete lattice, with $\ell(P^n) = l$ and let x^k be an ascending sequence of chaotic iterations for the map $\phi : P^n \to P^n$. If the number of steps necessary for every component to evolve in chaotic iteration x^k is bounded by m, then $\mu(\phi) = x^{l \cdot m}$.*

Proof. [CC92a]. From (6.1) and (6.3) in the proof of convergence of an ascending sequence of chaotic iterations it is known that $\forall k \geq 0 : x^k \sqsubseteq \phi(x^k) \sqsubseteq x^{k+m}$. The case $\forall k \geq 0 : x^k \sqsubset x^{k+m}$ is impossible, since it contradicts the fact $\ell(P^n) = l$. Hence, pick the least $k \geq 0$ such that $x^k = x^{k+m}$, which means that $x^k = \mu(\phi)$ and $x^{i+m}, 0 \leq i \leq k$, is a strictly increasing chain such that $k \leq l$. When a fixpoint is reached, $\mu(\phi) = x^{l \cdot m}$. ∎

6.5 Static Dependencies

The idea is that the structure given to the equation system by finding the static dependencies facilitates a good iteration strategy.

6.5.1 Worklists

In strategies based on worklists, the function to be evaluated is chosen from a *worklist* containing functions which should be evaluated. This worklist initially contains all functions and if the value of function i, picked (and removed) from the worklist, changes, then all those functions depending on function i are added to the worklist. A fixpoint is found when the worklist is empty.

Workstacks

One way of choosing functions from a worklist is to see it as a stack. Thus, in a *workstack* approach, the functions are added to and removed from the worklist in a LIFO-manner, i.e. pushed and popped, until the stack is empty [O'K87].

Priority-Queues

According to the *priority-queue* method [HDT87], the vertices in the worklist should be ordered in reverse postorder.

6.5.2 Topological Sorting

When the static dependency graph is acyclic, the problem of finding an optimal chaotic iteration strategy is easy; just sort the graph topologically and evaluated the functions at vertices in the order given. Unfortunately, global static program analysis normally generates equation systems with cyclic dependency graphs, hence this method is not immediately applicable.

Strongly Connected Components

One way of overcoming the problem of cyclic dependency graphs is to topologically sort the strongly connected components (found by Tarjan's algorithm) in the graph and then visit the SCCs in this order, employing various function and domain independent optimizations *inside each SCC*, e.g.

- *Following chains and worklist.* Identify the chains of the dependency graph prior to the fixpoint computation. Then use a workstack, except when the value of the first vertex in a chain has changed; then all the vertices in the chain should be recomputed. This is an algorithmic optimization used in the AKL analysis framework [SS95].

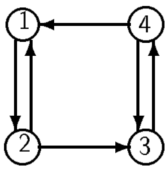


Figure 6.1: Illustrating the youngest vertex first heuristic $(1, 2, 3, 4)$.

- *Reverse postorder.* Perform a reverse postorder traversal [HDT87].
- *Reverse postorder and worklist.* The vertices are picked in reverse postorder from the worklist [HDT87].

The following optimizations are all based on a workstack, adding various heuristics.

- *Youngest vertex first.* When choosing between two vertices, the youngest (the one which has been evaluated a smaller number of times of the two) should be evaluated first. This captures the requirement of "fairness" in chaotic iteration strategies; that no equation should be forgotten indefinitely [SS93]. In Figure 6.1 this means that the order of evaluation should be $1, 2, 3, 4$.
- *Depth first.* Perform a depth first traversal of the spanning tree of the SCC [SS93]. The vertices in Figure 6.2 should then be visited in the order $1, 2, 4, 5, 3$.
- *Direct dependency.* If vertices A and B are put on the workstack for reevaluation and B depends on A, then A should be visited first [SS93]. In Figure 6.2 the vertices should be visited in the order: $1, 2, \mathit{5}, 4, 3$, whereas in Figure 6.3 the visiting sequence should be $1, \mathit{2}, 3, 4, 5$.
- *Indirect dependency.* If vertices A and B need to be evaluated and the paths from A to B do not contain vertices in need of evaluation, then vertex A should be visited first [SS93].
- *Prefer global effects.* The vertex with the largest number of edges should be choosen first [SS93].

Weak Topological Orders

The concept of a *weak topological order* is a generalization of a topological order of a directed graph possibly containing cycles. Its major advantage is that it both subsumes a topological sort of the strongly connected components of a directed graph *and* automatically, i.e. without using dynamic heuristics (cf. above), orders the the vertices inside each SCC in an ingenious way [Bou92b, Bou93].

Bourdoncle's algorithm for finding a weak topological order makes a hierarchical decomposition of a directed graph into strongly connected components and subcomponents, by recursively applying Tarjan's algorithm to each non-trivial component after having removed w (the head of the component) and every back-edge $v \rightarrow w$. For an experimental comparison of weak topological orders and strongly connected components in fixpoint computations, see Chapter 7.

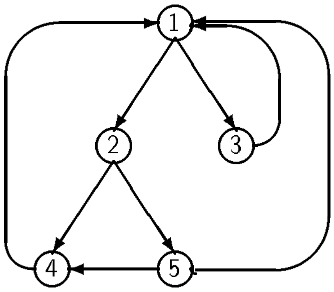


Figure 6.2: Illustrating the depth first heuristic $(1, 2, 4, 5, 3)$ and the direct dependency heuristic $(1, 2, \mathit{5}, 4, 3)$.

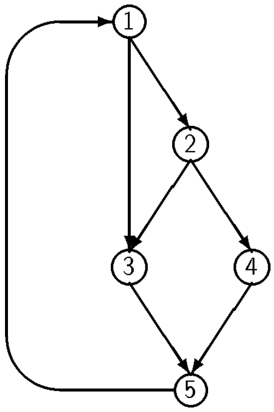


Figure 6.3: Illustrating the direct dependency heuristic $(1, \mathit{2}, 3, 4, 5)$.

6.6 Dynamic Dependencies

Dynamic dependencies arise when analyzing higher-order programs or when the analysis is polyvariant. In these two cases the dependency graph is not known before the analysis starts and it changes dynamically during analysis. The solving of the corresponding equations system is different in that the goal is to find the fixpoint of a monotone functional instead of the fixpoint of a function.

The main idea is to use a dependency relation in the style of [O'K87] and to transfer it from an ordinary system of equations to the more expressive system of functional equations.

$$f_i(x_1, \ldots, x_k) = E_i(f_1, \ldots, f_n, x_1, \ldots, x_k), 1 \leq j \leq n$$

6.6.1 The Operational Approach

Le Charlier and Van Hentenryck [LCVH92] describe a universal algorithm which finds the local least fixpoint of a monotone functional on a noetherian CPO. The actual computation is *top-down* in the sense that it is viewed as a generalization of computing the values of a function from its recursive definition.

The starting point is a monotone functional

$$\phi : (A \rightarrow A') \rightarrow (A \rightarrow A')$$

and its high-level procedural representation

$$phi(f : A \rightarrow A', \alpha : A) : A'$$

where A, A' are noetherian CPOs.

Given ϕ and $\alpha \in A$, the algorithm computes a local version of the least fixpoint $\mu(\phi)$, thus avoiding to compute the entire table of $\mu(\phi)$. The result is partial function f (represented as a partial table, pt) such that

$$f(\alpha) = \mu(\phi)\alpha.$$

Simulate the execution of (a procedure representing ϕ) on argument α; when f is applied to another element β, suspend the simulation of f on α, and start a sub-simulation of f on β. The sub-simulation provides new values for the partial table (pt) of f, and these values are used when the initial simulation is resumed. The sub-simulation proceeds in much the same way as the initial simulation, the only difference being that when f is simulated on α, no new sub-simulation is started; instead the current value of α, as found in the partial table, is used. Note that this means that β depends on α, and that whenever the value of α is refined, the sub-simulation on β has to be redone.

To avoid unnecessary computations, the algorithm uses a dynamic dependency graph. Intuitively, the set $dg(\alpha)$ represents the elements on which α depend and $dg^+(\alpha)$ represents the reflexive and transitive closure of dg, i.e. all the elements that α directly or indirectly depend on. If any of the elements in the set $dg(\alpha)$ is refined, then α has to be refined.

If α depends on a set of elements, which have not been refined, no further refinement of α is possible at this moment. This is also the case when a sub-computation for α already has been started. In all other cases, α is a new sub-computation, and a local fixpoint for α is computed in the following way: repeatedly adjust the partial table by calling the procedure *phi*. When calling *phi*, it may happen that the dependency graph of α is extended (i.e. α depends on some values) and if the new value, β, computed by *phi*, leads to an adjustment of the partial table, all dependency graphs with elements which directly or indirectly depend on elements which have been modified, are removed. The repetition terminates and a local fixpoint is found when α depends on some elements which have not been refined (i.e. there is a dependency graph left after removing those with modified elements, hence the elements on which α depend have not been refined). Note that *phi* may trigger sub-simulations and extend dependency graphs.

In [LCDMVH93] an instance of the above universal top-down algorithm for Prolog along with caching of operations is described.

Optimizations

Chen [Che94] expresses an analysis problem as the least fixpoint of an abstract semantic functional over abstract domains. The monotone function (the abstract interpreter) is

$$I : \sigma \times P \rightarrow P'$$

where σ is the set of expressions in a program, P and P' are sets of prestates and poststates of expressions, respectively.

The functional is

$$F : (\sigma \times P \rightarrow P') \rightarrow (\sigma \times P \rightarrow P')$$

Next, a relationship called *entailment* is introduced.

Definition 6.4 (entailment) Let $F : X \rightarrow Y \rightarrow X \rightarrow Y$ be a map and let $x, x' \in X$. x *entails* x' under F and f, denoted $x \overset{F,f}{\rightsquigarrow} x'$, if $\exists y \sqsupset f(x') : F(f[x' \mapsto y])x \sqsupset Ffx$. □

The intuition of this definition is that $F(f)x$ somehow depends on the value of $f(x')$; hence x depends on x'. On the other hand, if x disentails (does not entail) x', then

$$F(f[x' \mapsto y])x = Ffx$$

The basic fixpoint computation uses the following four equations

(E1) $f_0 = \lambda x.\bot \mid_\emptyset$

(E2) $X_0 = \{x_0\}$

(E3) $f_{i+1} = \{\langle x, F(f_i \mid_{X_i})x\rangle \mid x \in X_i\}$

(E4) $X_{i+1} = X_i \cup \{x' \mid x \overset{F,f_i\mid_{X_i}}{\rightsquigarrow} x', \forall x \in X_i\}$

(E3) says that a new approximation of the least fixpoint is found by applying F to the current approximation restricted to those elements that are currently entailed (X_i). (E4) makes sure that the vertices entailed under F and $f_i \mid_{X_i}$ are added to the set of entailed vertices.

Note that f_{i+1} does not differ from f_i on all elements in X_i. The differences can be divided into two subsets: (1) Vertices that are newly entailed. (2) Vertices entailing "old" vertices that are improved; i.e. vertices that depend on other vertices, which have been refined. We thus get

(A1) $f_0 = \lambda x.\bot \mid_\emptyset$

(A2) $X_0 = \{x_0\}, W_0 = \{x_0\}$

(A3) $f_{i+1} = \{\langle x, F(f_i\mid_{X_i})x\rangle \mid x \in W_i\}$

(A4)

$$\begin{aligned} W_{i+1} &= \{x' \mid x \overset{F,f_i|_{X_i}}{\rightsquigarrow} x', x \in W_i\} \cup \\ &\quad \{x \mid x \rightsquigarrow x', F(f_i\mid_{X_i})x' \sqsupset (f_i\mid_{X_i})x', x \in X_i, x' \in W_i\} \end{aligned}$$

The next concept introduced is the *entailment graph* G; if x entails x' there is a directed edge $\langle x, x'\rangle$ in the entailment graph. This graph will be constructed dynamically during the fixpoint computation.

It is generally impossible to compute the entailment relation precisely, and therefore a safe approximation is computed by adding an edge $\langle a, b\rangle$ to G if the value $f(b)$ is used during the computation of $F(f)a$, as a then depends on b. It is easy to show that superfluous edges do not affect the correctness of the fixpoint.

Basic Entailment Algorithm. The algorithm uses two tables $T_P : \sigma \longrightarrow P$ and $T_{P'} : \sigma \longrightarrow P'$ for keeping track of the prestate and poststate of each expression in the program. Figuratively speaking, prestates flow forward in the entailment graph and poststates flow backwards. The main iteration retrieves vertices from a workstack W, finishing when W is empty. If the evaluation of a vertex w leads to an improved value, all the vertices that entail w are put on the workstack (i.e. those that depend on w). In the evaluation procedure, if a prestate value has been refined, the vertex w' that w entails is put onto the workstack. Regardless of the result of the evaluation, the evaluation procedure adds the edge $\langle w, w'\rangle$ to the entailment graph.

Guided Entailment Algorithm. Local knowledge of the entailment graph at each vertex is used to dynamically order the evaluations necessary for finding the least fixpoint. The strategies are:

- *Waiting for all successors.* There is no need to compute a new poststate value unless the values of every successor vertex is known.
- *Leading vertex first.* Compute a new poststate of every vertex that has an improved prestate (a *leading vertex*) before considering *retreating vertices*, ie. those vertices added to the workstack because one of its successor vertices has been improved.
- *Suspended evaluation.* Perform only a minimum amount of evaluation of a vertex v in order to supply its successor vertices with sufficient prestate information, then put the successor vertices on the workstack, suspend evaluation of vertex v, evaluate the successor vertices, and finally resume execution of vertex v. This may, however, lead to problems when there is an internal entailment between the successor vertices such that the entailment graph is cyclic.
- *Back edge first.* This strategy deals with the above problem. If a vertex entails one of its ancestors, there is a back edge in the dependency graph (a loop) and new poststates should always propagate along a back edge first (if there is one) until the loop is stable.

In the algorithm, vertices are seen as objects receiving messages. The messages are *activate* which computes a new value of the vertex by activating its successor vertices according to the above strategies. The computation is thus driven forward along the entailment edges. At the end of an activate session, *propagate* is used to pass information backwards to predecessor vertices. The method *propagate* also deals with propagating new values along back edges until the loop is stable.

Context Projections. A large context (the current state of the computation) is often propagated during the fixpoint computation. A solution is to use a projected context, i.e. the part of a context which is needed for evaluation and which makes it possible to say whether the vertex should be reevaluated or not. The prestate of a vertex v is split into a projected part, needed in the evaluation of the vertex, and the part unnecessary at this point, i.e.

$$\rho = \rho_1 \cup \rho_2$$

where $\rho_1 = Project(\rho, n)$

If the projected context has changed since the last time the vertex was evaluated the vertex must be reevaluated in this new context. The goal is now to find a function $Project$ such that $Project(\rho)$ is as small as possible.

6.6.2 The Denotational Approach

The papers by Rosendahl[Ros93] and Jørgensen[Jør94] approach the problem of finding fixpoints of functionals from a denotational semantics perspective. Starting with a domain D and a language of expressions L, a corresponding evaluation function is defined:

$$\mathbf{E} : L \rightarrow (D^k \rightarrow D) \rightarrow D^k \rightarrow D$$

The syntactic equation $f(x_i, \ldots, x_k) = E$ now induces a semantic functional $\mathbf{E}[E]$ and its least fixpoint $\mu(\mathbf{E})[E]$.

The dependency (or sometimes neededness) relation is expressed by the neededness function $\mathbf{N}$, which can be used to infer that certain applications of the induced functional $\mathbf{E}[E]$ may safely be omitted as these applications will not change the current fixpoint approximation. The neededness function is defined by

$$\mathbf{N} : L \rightarrow (D^k \rightarrow D) \rightarrow D^k \rightarrow \wp(D^k)$$

$$\mathbf{N}[E]\phi\Delta = \bigcup_{f(\ldots,e_i,\ldots)\in R_E} \{\langle \ldots, \mathbf{E}[e_i]\phi\Delta, \ldots\rangle\}$$

where R_E is the set of recursive calls to f in E

$$R_E = \{f(\ldots, e_i, \ldots) \mid f(\ldots, e_i, \ldots) \text{ is a sub-expression of } E\}$$

The intuition is that the neededness set consists of the argument tuples Δ_{need} needed in the computation of the functional relative to the current approximation ϕ. This means that the information is dynamic in nature, and hence cannot be computed prior to the fixpoint iteration.

The algorithm uses the functions ϕ and N (think of them as tables being gradually extended) mapping $\Delta \in D^k$ to its currently approximated fixpoint, and to its neededness set, respectively. In addition to this, there is a workset $W \subseteq D^k$ which contains all those tuples Δ that must be processed in some subsequent iteration. The algorithm is global in the sense that the initial workset W^0 contains the whole of D^k which means that every tuple is processed at least once. The initial fixpoint approximation ϕ^0 maps all tuples to $\perp$ and the function N^0 maps every tuple to the empty set.

In each iteration step $\langle \phi^n, N^n, W^n \rangle$ is used to compute $\langle \phi^{n+1}, N^{n+1}, W^{n+1} \rangle$, finishing with the result $\phi = \phi^s$ when the workset W is empty. The argument tuple Δ^{n+1} is arbitrarily (!) chosen from the workset, and is used to extend the approximate functional of the previous iteration

$$\phi^{n+1} = \phi^n[\Delta^{n+1} \mapsto \mathbf{E}[E]\phi^n\Delta^{n+1}] \uparrow \text{mon}$$

The monotonic closure operation speeds up convergence, as some tuples are assigned larger values without being processed.

When for some argument tuple Δ there is no need to re-compute the fixpoint approximation, there is no need to update the neededness information either. The new neededness set is given by

$$N^{n+1} = N^n[\Delta^{n+1} \mapsto \mathbf{N}[E]\phi^n\Delta^{n+1}]$$

Note that the set of tuples $\mathbf{N}[E]\phi^n\Delta^{n+1}$ is already computed during the evaluation of $\mathbf{E}[E]\phi^n\Delta^{n+1}$ and that no other neededness sets are modified. The new workset is

$$W^{n+1} = (W^n \setminus \{\Delta^{n+1}\}) \cup A^{n+1}$$

where

$$A^{n+1} = \{\Delta \in D^k \mid \exists \Delta_{need} \in (N^{n+1}\Delta) : \phi^n\Delta_{need} \sqsubset \phi^{n+1}\Delta_{need}\}$$

Hence, the workset is extended with all those tuples with neededness sets having members Δ_{need} which are mapped to strictly larger values under the new approximation ϕ^{n+1}.

The main difference compared to [LCVH92] is that the neededness analysis of Jørgensen's algorithm is more *shallow*, i.e. it cares only about direct needs, whereas the algorithm in [LCVH92] also deals with indirect needs by performing a transitive closure operation. Jørgensen's algorithm is also more iterative as each iteration corresponds to the processing of a single element $\Delta \in D^k$, during which all needs Δ_{need} are treated by look-ups in the table holding the previous fixpoint approximation, whereas the "depth-first" strategy of [LCVH92] sometimes suspends the processing of an element in order to improve the approximating value for a need.

This can be generalized to a system of m functional equations, where two two-dimensional tables are used: $\Phi = \langle \phi_1, \ldots, \phi_m \rangle$ with $\phi : D^k \to D$ for the fixpoint approximation, and $N = \langle N_1, \ldots, N_m \rangle$ for the neededness information, where $N_i \in \wp(\{f_1, \ldots, f_m\} \times D^k)$, i.e. the first argument is the function of the subexpression that gives rise to the need in question. Now, in iteration step $n+1$, choose a pair, $\langle j, \Delta^{n+1} \rangle$ from the workset W^n, which means that Δ^{n+1} should be given as input to equation j. If equation j contains a call to the r'th function, f_r, then a pair $\langle f_r, \Delta_{need} \rangle$ will be placed in the neededness set $N_j^{n+1}(\Delta)$. As in the single equation case, every tuple with a neededness set which contains tuples that lead to better functional approximations ϕ^{n+1} is included in the new workset.

Instead of trying to find a global fixpoint, it should be less expensive to find a local fixpoint, i.e. a functional fixpoint only defined for a subset of the domain

$$\forall \Delta \in S^0 : \phi\Delta = \mu(\mathbf{E}[E])\Delta$$

where $S^0 \in D^k$ is the set of tuples for which fixpoint values are sought (cf. [LCVH92]). The idea is to select tuples from the workset according to the following scheme

$$\Delta^{n+1} \in W^n \cap Seen^n$$

where

$$\begin{aligned} Seen^0 &= S^0 \\ Seen^{n+1} &= Seen^n \cup \mathbf{N}[E]\phi^n \Delta^{n+1} \end{aligned}$$

Seen thus accumulates neededness information.

Example 6.2 An instantiation of the above scheme can be found in [Jør93] which maintains a distinction between the different ways in which a Prolog predicate can be called (the running example is groundness analysis). The dynamics of algorithm is due to the following definition. Given a tuple $\langle C, \Delta \rangle$ where C is a Prolog clause and Δ is a specific call pattern of C, this tuple depends on Δ', if Δ' has been computed as a call pattern for an atom in the body of C during the most recent evaluation of Δ against C. This means that the dependency relation is dynamic, i.e. changing from one iteration to the next. □

6.7 Optimizations Based on Properties of Functions

To minimize the cost of computing a new value of a function, a number of algorithmic, function dependent optimizations have been proposed.

- *Simplifying argument lists.* For an associative, commutative and idempotent operator, all duplicate arguments may be removed, leading to a shorter evaluation time [SS93].

- *Simplifying single equations.* The lub-function has certain nice properties, facilitating simplification [SS93]. The n argument equation

  $$x_i = lub(x_1, x_2, \ldots, x_i, \ldots, x_n)$$

  can be replaced by the $n-1$ argument equation

  $$x_i = lub(x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$$

  as

  $$\min x_i = lub(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$$

- *Strongly connected subcomponents.* The type T of a vertex is determined by its associated function, and a *strongly connected subcomponent* is a subgraph where all vertices have the same type. Now, the vertices in a strongly connected subgraph of type *lub* can be collapsed into a single vertex, as they will all get the same value [SS93].

- *Incremental computation on totally ordered domains.* The method in [Sah90] essentially uses the old value of a vertex when computing the new value, instead of computing every new value from scratch. This is possible when the domain is a chain and the functions are monotone. The question is whether this idea could be extended to domains which are ordered sets.

framework	X_w	k
total or monotone	A	$\|A\| \cdot \ell(B)$
completely additive	$\{x \in A \mid x \in \mathcal{J}(A)\}$	$\|\mathcal{J}(A)\| \cdot \ell(B)$
iterative scheme	$\{w\}$	$\ell(A)$

Table 6.1: Iteration bounds for fixpoints of functionals

6.8 Characterization of Frameworks

In [NN92] Nielson and Nielson conducts a theoretical study of various frameworks for fixpoint computations. To each program the analysis associates an element d of a complete lattice $\langle D; \sqsubseteq \rangle$ of abstract values. The value d is determined as the least fixpoint, $\mu(H)$, of a continuous functional $H : D \to D$, possibly parameterized by the argument w which the resulting fixpoint is to be evaluated on. Furthermore, it is assumed that the lattice D has the form $A \to B$, where A and B are *finite* complete lattices, which guarantees termination of the standard iteration procedure for finding the least fixpoint. The goal is to give bounds to size of the set X_w in

$$\text{if } H^k \bot v = H^{k+1}, \forall v \in X_w \text{ then } \textbf{fix}(H)w = H^k \bot w$$

and the number of iterations k in

$$\textbf{fix}(H)w = H^k \bot w$$

The three frameworks investigated are: the *general framework*, where the functions of $A \to B$ are only required to be total; the *monotone framework*, where the functions $A \to B$ must be monotone, the *completely additive framework*, where the functions of $A \to B$ must be strict and additive (or distributive), and, finally, the case where H is in iterative form H is on *iterative form*, if it is of the form $H\ h = f \sqcup (h \circ g)$. The bounds on the number of iterations k needed to compute the fixpoint of the functional H are presented in Table 6.1. If, furthermore, f and g are strict and additive it is shown that the dependence of k on p is at most *linear* and that the fixpoint can be computed pointwise.

In the setting where $A = B = C^p, p \geq 1$ (which incorporates imperative programs with roughly p variables) the above results yield that the dependence of k on p is at most exponential in the general and monotone frameworks; quadratic in the completely additive framework.

Part III

Towards Efficient Fixpoint Computations

Chapter 7

Practical Iteration Strategies

This chapter describes and compares two methods for solving to the equation systems that arise in the framework for analyzing AKL-programs. As the dependency graphs are static in this particular case, both methods work by basing their respective chaotic iteration strategies on a topological sort of the already given graph.

7.1 Strongly Connected Components

The current AKL analysis framework sets up a system of semantic functions whose dependencies are modeled by a directed graph containing cycles. Because of the cycles, an ordinary topological sort of the graph is impossible. Instead, the strongly connected components of the graph are topologically sorted (Section 4.3). Although this method imposes a certain structure on the graph, it is by no means enough; inside each strongly connected component there is still no order guiding the fixpoint computation and various optimizations, apart from the workstack-approach must be used.

One optimization is based on chains in graphs (Definition 4.5). This algorithmic optimization works by reevaluating every member of a chain as soon as the first vertex in this chain has been reevaluated. As a chain in a graph is isolated from the rest of the graph, no other vertices are affected by this procedure.

In order to find the strongly connected components of the graph, *Tarjan's algorithm*[1] is used. This algorithm, described in Figure 7.1[2], is basically a depth first traversal of the dependency graph where each visited vertex is given a depth-first number (*dfn*) and is stored on a stack. The time complexity of Tarjan's algorithm is linear in the number of edges.

7.2 Weak Topological Orders

When surveying the field of fixpoint computations, the method based on a weak topological order (WTO) [Bou92b, Bou93] of the static dependency graph seemed to be the best way of improving the execution speed of the fixpoint solver in the AKL analysis

[1] The original idea was first published by Tarjan in [Tar72]; the papers [NSS94] and [Jia93] are recent enhancements.

[2] The pseudo code loosely resembles Pascal, the modifications being that arguments to functions may be labeled **in**, **out** or **inout** signifying how values of the arguments are passed through the function interface. The symbol :: denotes list concatenation

```
func Partition(in G)
var vertex, partition, num
begin
  foreach vertex ∈ vertices_G do
    dfn[vertex] ← 0
  num ← 0
  partition ← nil
  Visit(root_G, partition)
  return partition
end

func Visit(in vertex, inout partition)
var head, min, succ, element, component
begin
  push(vertex)
  head ← dfn[vertex] ← num ← num + 1
  foreach succ ∈ succ_G[vertex] do
    if dfn[succ] = 0 then
      min ← Visit(succ, partition)
    else
      min ← dfn[succ]
    if min < head then
      head ← min
  if head = dfn[vertex] then
    component ← nil
    repeat
      pop(element)
      dfn(element) ← +∞
      component ← element :: component
    until element = vertex
    partition ← component :: partition
  return head
end
```

Figure 7.1: Tarjan's algorithm for finding the SCC's of a directed graph.

framework, as it is a natural generalization of the current approach based on strongly connected components. Another reason for this choice is that the weak topological order approach is relatively new (1992) and that it would be interesting to compare it to other, related methods. To the author's knowledge, this is the first major experimental evaluation of WTOs in fixpoint computations.

The main idea is to find the structure *inside* the strongly connected components and the approach is to generalize the concept of topological order to work with directed graphs containing cycles. This generalized order-relation between equations (vertices) then facilitates at least two efficient chaotic iteration strategies.

7.2.1 Basic Concepts

In contrast to the approach using work stacks where checking for stability is a constant operation ("is the stack empty?"), more work is required in approaches based on following a directed graph containing cycles. In order to detect stability of a fixpoint computation every vertex (representing a function in the system of equations) of the dependency graph must be examined in the worst case. When the dependency graph is cyclic, however, there exist smaller sets of vertices which it is sufficient to check. The definition follows:

Definition 7.1 (feedback vertex set) A set of vertices V in a directed graph $G = \langle V, E\rangle$ is a *feedback vertex set*, if every cycle in G contains at least one element of V. □

Thus, it is of great interest to find the *minimal* feedback vertex set as this will reduce the time spent checking whether the equation system is stable or not. Finding this set is a classical problem which unfortunately is NP-complete [GJ79] and the time complexity of the naive algorithm is quadratic in the number of vertices.

Bourdoncle is the first author to investigate the possibility of combining the order of computation and the checking for stability. The answer proposed is based on a weaker form of topological order of a directed graph and an algorithm (generalizing Tarjan's algorithm) for sorting the graph according to this order.

Definition 7.2 (hierarchical order, component, component-head) A *hierarchical order* on a set A is a well-parenthesized permutation of A without two consecutive "(". A set $C = \{a_1, \ldots, a_k\} \subseteq A$ is a *component* where a_1 is its *component-head*, if the elements of C are placed between two matching parentheses. □

The sets of component-heads of all nested components containing $a \in A$ is denoted $\omega(a)$ and the set of all component-heads is denoted H. The *depth* of a is defined by $\delta(a) = |\omega(a)|$.

Theorem 7.1 *A hierarchical order on the set A defines a chain $\langle A; \preceq\rangle$.*

The concept of a weak topological order is based on feedback edges; a *feedback edge* is an edge $\langle v, w\rangle$ such that $w \preceq v$ in the chain given by a hierarchical order on the graph. Intuitively, in a weak topological order, the head w of a feedback edge $\langle v, w\rangle$ is also the component-head of the component containing its tail v. More formally:

Definition 7.3 (weak topological order (WTO)) Let $G = \langle V, E\rangle$ be a directed graph. A hierarchical order $W = \langle V; \preceq\rangle$ is a *weak topological order*, if for every edge $\langle v, w\rangle \in E$:$(v \prec w \wedge w \notin \omega(v)) \vee (w \prec v \wedge w \in \omega(v))$ □

7.2.2 Bourdoncle's Algorithm

A generalization of Tarjan's algorithm for finding the strongly connected components of a directed graph, *Bourdoncle's algorithm* (Figure 7.2) finds a weak topological order of a directed graph which is possibly irreducible and contains cycles. The basic idea is to apply Tarjan's algorithm recursively to each component found, after having removed the component-head, h, and every feedback edge to h.

The various interrelationships between strongly connected components, weak topological order, Tarjan's algorithm and Bourdoncle's algorithm are depicted in Figure 7.3, where the arrows indicate logical dependency.

Bourdoncle's algorithm is correct and its time complexity is $O(\delta \cdot E)$, where δ is the maximum depth of the graph and E is the number of edges ([Bou92b], Section 3.6.3).

Example 7.1 (Weak topological ordering by Bourdoncle's algorithm) The weak topological order of the directed graph in Figure 7.4 as given by Bourdoncle's algorithm is:

$$1\ 2\ (3\ 4\ (5\ 6)\ 7)\ 8$$

The strongly connected components as found by Tarjan's algorithm are:

$$(1)\ (2)\ (3\ 4\ 5\ 6\ 7)\ (8)$$

□

7.2.3 Iteration Strategies

From every weak topological order the following two chaotic iteration strategies can be derived: the *recursive strategy* and the *iterative strategy*. Both these strategies are based on the stabilization operator which uses the component-heads of the weak topological order to check for stability.

In the recursive strategy the stabilization operator is applied recursively to the nested sequence of components and subcomponents which make up the weak topological order. When a subcomponent C' is found inside a component, the stabilization operator is applied to C before continuing with the remaining part of component C. The stability of a component is detected by examining its component-head.

Lemma 7.2 *A set of consecutive vertices, V_0, of depth 0 in a WTO requires only one iteration.*

Proof. [Bou93]. Assume that $\forall v \in V_0 : \delta(v) = 0$. Now, by definition of a WTO, $\langle v, w\rangle \Rightarrow v \prec w$; hence,

$$\forall v : \langle v, w\rangle \wedge stable(v) \Rightarrow stable(w) \tag{7.1}$$

As there are no feedback-edges,

$$\text{The first vertex in the ordering is stable after one evaluation.} \tag{7.2}$$

From these two facts, it follows that after one iteration where every vertex is evaluated once, every $v \in V_0$ is stable. ■

Theorem 7.3 (Detection of stability, recursive strategy) *In the recursive strategy, a component of a weak topological order is stable, if its component-head is stable.*

```
func Partition(in G)
var vertex, partition, num
begin
   foreach vertex ∈ vertices_G do
      dfn[vertex] ← 0
   num ← 0
   partition ← nil
   Visit(root_G, partition)
   return partition
end

func Component(in vertex)
var succ, partition
begin
   partition ← nil
   foreach succ ∈ succ_G(vertex) do
      if dfn[succ] = 0 then
         Visit(succ, partition)
   return (vertex :: partition)
end

func Visit(in vertex, inout partition)
var head, min, succ, element, loop
begin
   push(vertex)
   head ← dfn[vertex] ← num ← num + 1
   loop ← false
   foreach succ ∈ succ_G[vertex] do
      if dfn[succ] = 0 then
         min ← Visit(succ, partition)
      else
         min ← dfn[succ]
      if min ≤ head then
         head ← min
         loop ← true
   if head = dfn[vertex] then
      dfn[vertex] ← +∞
      pop(element)
      if loop then
         while element ≠ vertex do
               dfn[element] ← 0
               pop(element)
         partition ← Component(vertex) :: partition
      else
         partition ← vertex :: partition
   return head
end
```

Figure 7.2: Bourdoncle's algorithm for finding a WTO of a directed graph

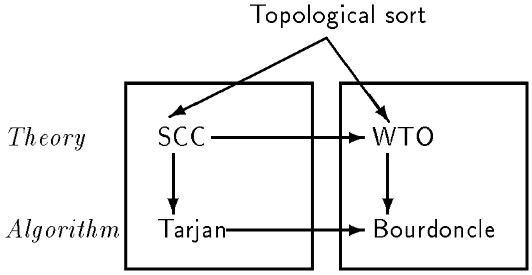


Figure 7.3: Relationships SCC, WTO, Tarjan's algorithm and Bourdoncle's algorithm.

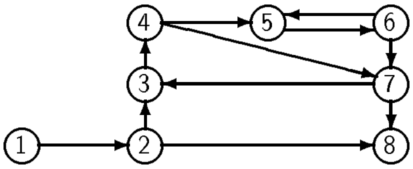


Figure 7.4: Static dependency graph.

Proof. [Bou93]. Without loss of generality, suppose that the WTO consists of a single outermost component C, i.e. a SCC. Furthermore, assume that after applying all the equations in C and stabilizing each sub-component recursively, the value of the component-head of C remains unchanged. Equation 7.1 in the proof of Lemma 7.2 then shows that evaluation of the vertices once more is unnecessary, as no vertex inside the component will change. Hence, the stability of a component-head imply the stability of the entire component. ∎

The iterative strategy works by applying the stabilization operator to every outermost component of a weak topological order (aka strongly connected component), thereby removing the subcomponent structure without loosing the order structure. Once again the stability is checked by investigating component-heads.

Theorem 7.4 (Detection of stability, iterative strategy) *In the iterative strategy, an outermost component C of a weak topological order is stable, if all component-heads of C and its subcomponents are stable.*

Proof. [Bou93]. Without loss of generality, suppose that the WTO consists of a single outermost component C, i.e. a SCC. The idea is to prove that after applying the equations in the sequence given, either at least one of the component-heads of C and its subcomponents have changed or that none of the values associated with vertices in W have changed.

Suppose that the contrary holds and that the value associated with a vertex $w \notin W$ has changed. By definition of a WTO, every edge $\langle v, w \rangle$, where $w \notin W$ is such that $v \prec w$. Hence, there is at least one vertex v whose value has changed, inducing a change in w. Inductive application of this argument shows that the component-head has changed, which contradicts the assumptions. Hence, either at least one of the component-heads of C and its subcomponents have changed or that none of the values associated with vertices in W have changed. This is equivalent to the fact that if all component-heads of C and its subcomponents are stable, then C is stable. ∎

Theorem 7.5 (Complexities) *If L is a lattice with $\ell(L) = l$, then the time complexity of the recursive strategy for a strongly connected graph $G = \langle V, E \rangle$ is:*

$$O(l \cdot \sum_{v \in V} \delta(v))$$

and the time complexity of the iterative strategy is:

$$O(l \cdot |V| \cdot |H|),$$

where H is the set of component-heads

Proof. [Bou92b], Section 3.7.1. ∎

The theoretical upper bound of the recursive strategy is always better than that of the iterative strategy, since $\delta(v) \leq |H|, \forall v \in V$.

Note, however, that the above time complexity results are based on the number of function applications. The assumption is that every function application, regardless of the number of arguments and the size of the arguments, executes for a certain, constant time. In practice, this is not the case.

Example 7.2 (Strategies based on weak topological orders) For the directed graph in Figure 7.4 the iterative strategy is:

$$1\ 2\ [3\ 4\ 5\ 6\ 7]^*\ 8$$

whereas the recursive strategy gives the following order of evaluation

$$1\ 2\ [3\ 4\ [5\ 6]^*\ 7]^*\ 8$$

where $[\cdot]^*$ is a stabilization operator (Section 6.2). □

7.3 Implementation of the Algorithms

The actual implementation is done in AKL (Appendix A). The algorithms are coded in a declarative way and deterministic execution is guaranteed. A store is used to represent the equation of every program point including left hand side variable, current value, right hand side function, arguments, and the equations which depend on the equation in question. First, the dependencies of the equation system are turned into a dependency graph. Secondly, Bourdoncle's algorithm is used to find the weak topological order of the dependency graph. The equations are then linked together according to the weak topological order before the fixpoint computation starts. This link information is also placed in the store.

In the iterative chaotic iteration strategy the vertices representing equations in a strongly connected component are traversed until each component head is stable. During the traversal only those equations with arguments that have changed are evaluated. This procedure is repeated for every strongly connected component.

Strategies	*number of function applications*	*execution time*
old/recursive	2.04	2.49
old/iterative	2.65	3.41
recursive/iterative	1.19	1.22

Table 7.1: How the different chaotic iteration strategies compare.

In the recursive chaotic iteration strategy each component head is linked to two vertices: (1) the vertex following the component head inside the component and (2) the vertex to be considered when the current component is stable. The last vertex in a component is linked back to the head of the component, which, after a traversal of the component, is checked for stability. If unstable, route (1) is taken whereas route (2) is choosen if the component head is stable.

7.4 Experimental Results

The different performances, i.e. execution time and number of function applications, of three different fixpoint solvers in the analysis framework of AKL are presented and compared. The solvers used the iteration strategies described above in analyzing 35 programs from the AKL demo directory.[3]

Below when speaking about strategies, *recursive* and *iterative* refers to the recursive and iterative chaotic iteration strategy, respectively, where both strategies are based on weak topological orders; *old* refers to the chaotic iteration strategy based on a topological sort of strongly connected components and the chain optimization inside strongly connected components (Section 6.5.2).

7.4.1 Presentation of Results

In Table 7.1, the three strategies are compared. The results shown are the averages of comparing the number of applications or the fixpoint execution time, i.e.

$$strategy_A/strategy_B = \overline{strategy_{A_i}/strategy_{B_i}} = \frac{1}{n}\sum_{i=1}^{n}\frac{strategy_{A_i}}{strategy_{B_i}}$$

These results show that the iterative strategy is best on average. It is 3.4 times faster than the old strategy and 1.2 times faster than the recursive strategy.

Comparisons of each program with respect to the different strategies are given in Figures 7.5, 7.6 and 7.7. Figures 7.8, 7.9 and 7.10 show how execution time and number of applications vary with number of equations, number of heads and maximum depth.[4]

7.4.2 Analysis of Results

The reason why the approach based on a weak topological order of the dependency graph is better than the approach based on strongly connected components and chains

[3] For more information about the experimental setup, see Appendix B.

[4] I am indebted to Mats Carlsson, Dan Sahlin and Thomas Sjöland (SICS) for suggestions on how to present the results graphically.

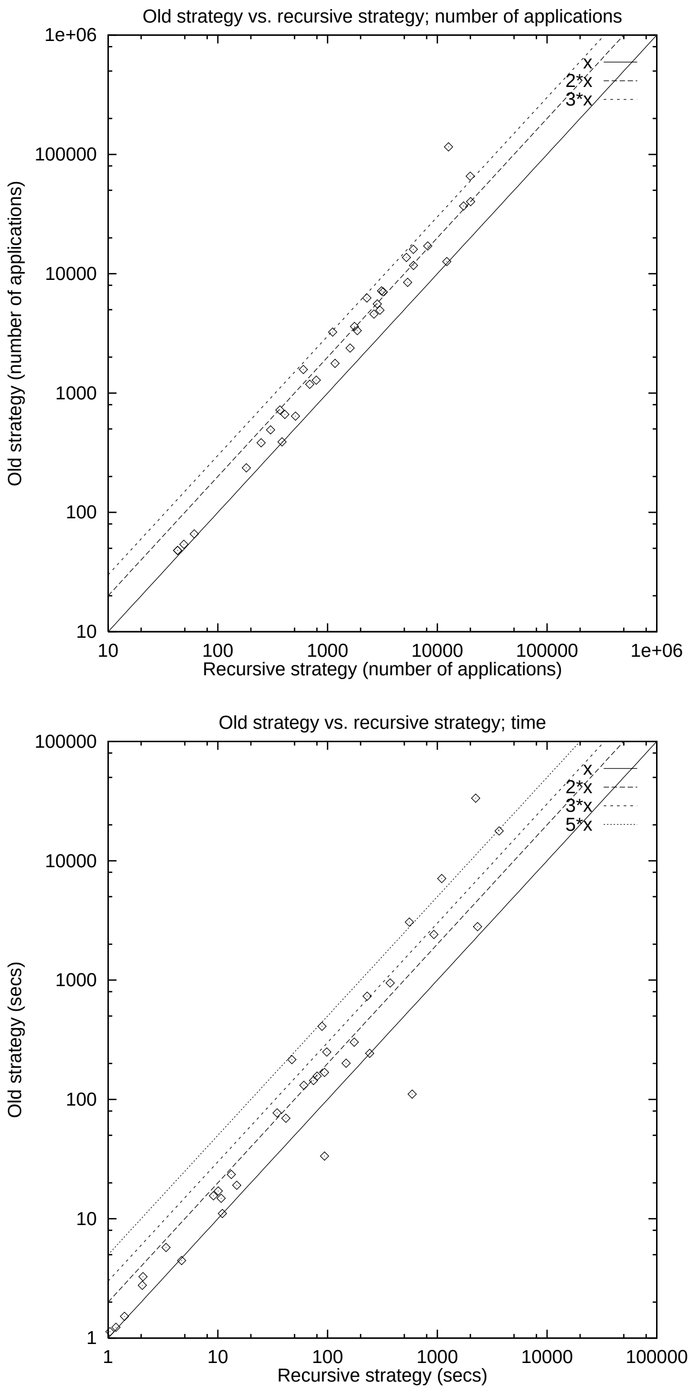
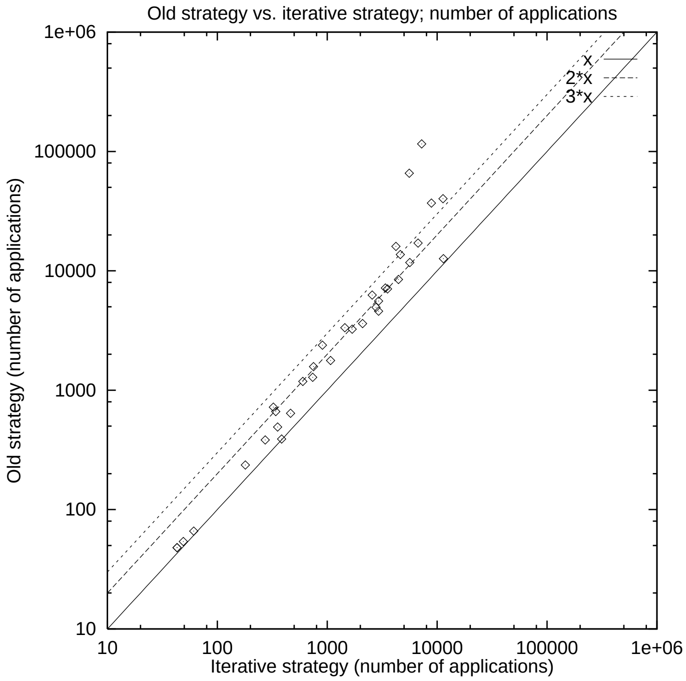


Figure 7.5: Comparison: *recursive* – *old* strategy.



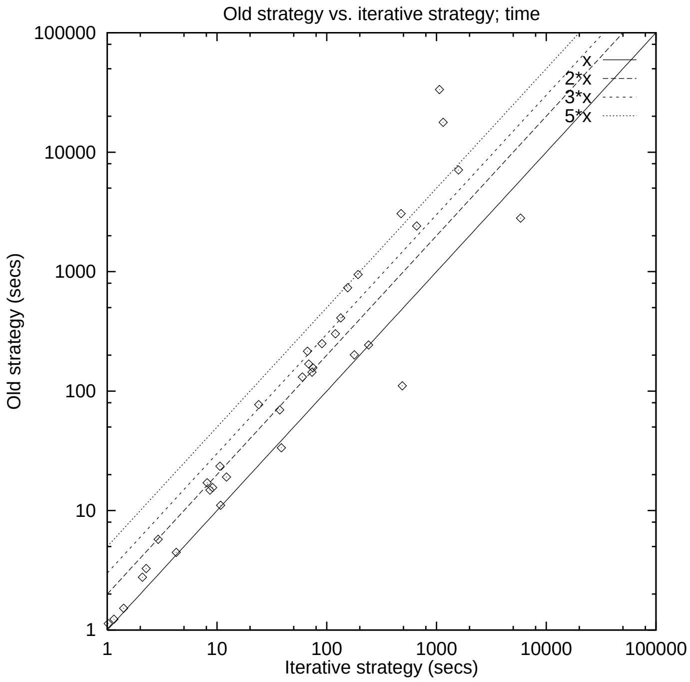
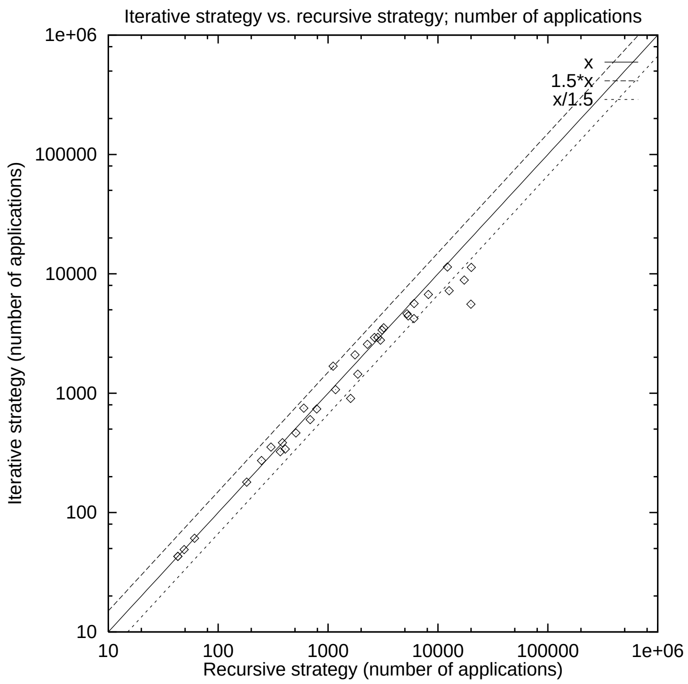


Figure 7.6: Comparison: *iterative* – *old* strategy.



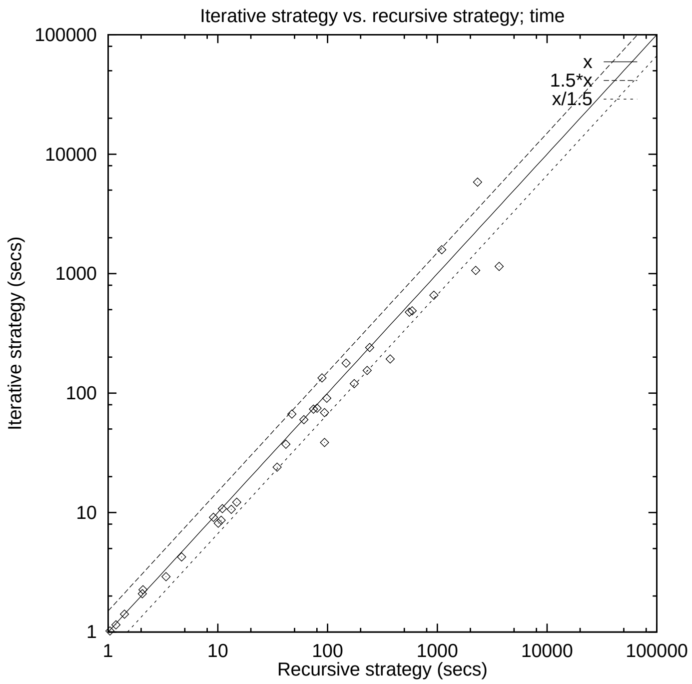


Figure 7.7: Comparison: *recursive* – *iterative* strategy.

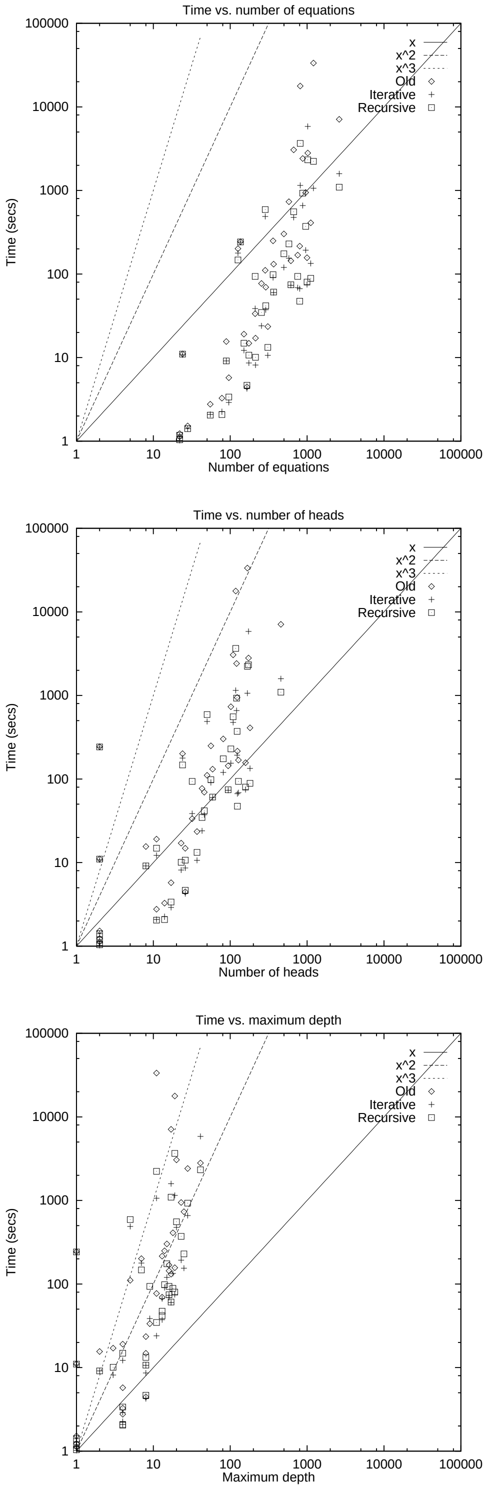


Figure 7.8: Comparison: execution time vs number of equations, number of heads and maximum depth, respectively.

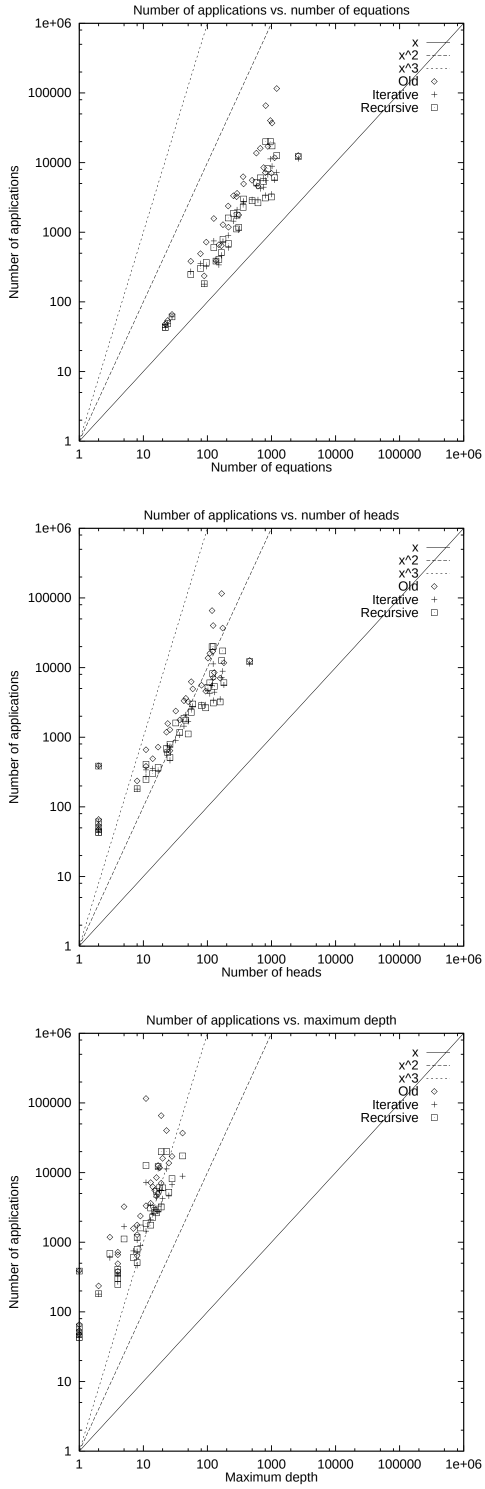
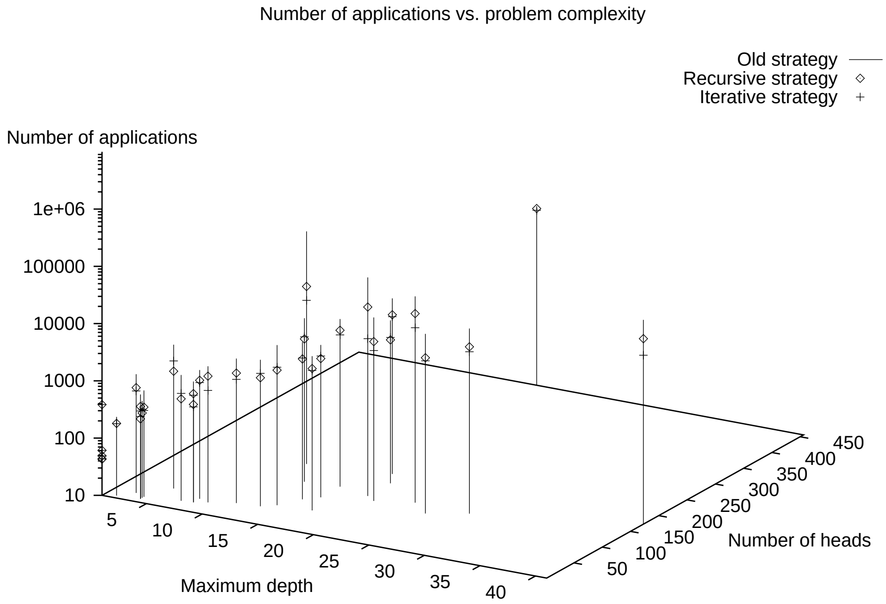


Figure 7.9: Comparison: number of applications vs number of equations, number of heads and maximum depth, respectively.



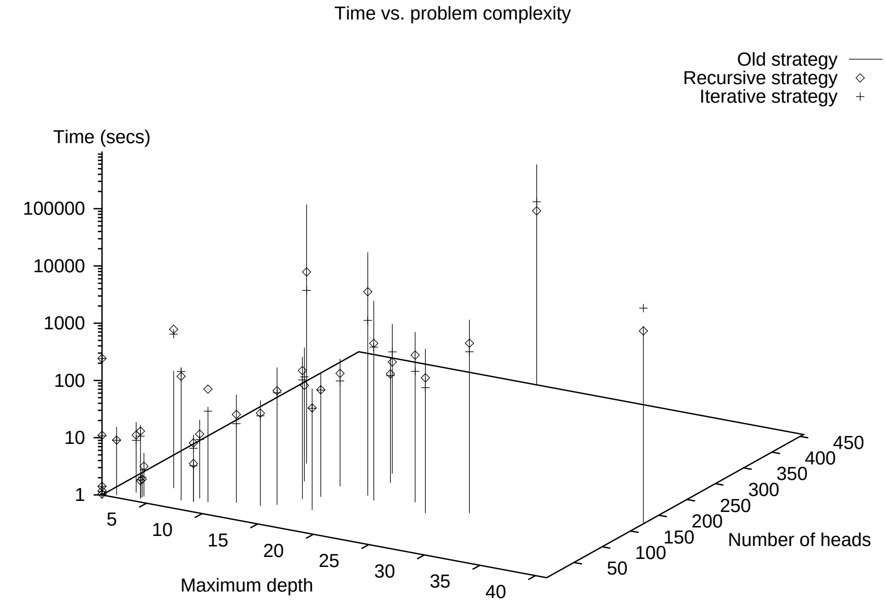


Figure 7.10: Number of applications and execution time vs the intrinsic complexity of the dependency graph.

should be fairly obvious, as it probes the dependency graph with finer granularity; a specific order is imposed on the strongly connected components. What makes it so much faster is that the heuristic optimizations *strongly connected components, following chains and worklist, youngest vertex first, depth first, direct dependency and indirect dependency* (Section 6.5.2) are incorporated automatically into the weak topological order; hence, the optimizations are taken care of *before* the actual fixpoint computation starts.

Contrary both to Bourdoncle's claim that the recursive strategy is almost always better in practice ([Bou93], p. 133) and the theoretical worst-case result, the results presented here show that this is heavily application dependent; in analysis of AKL, the iterative strategy is approximately 20% faster. The maximum depth of the dependency graph might be a deciding factor in choosing which iteration strategy to use in a specific analysis framework, as a large maximum depth can introduce a great deal of overhead in unfavorable cases when the iterative strategy is used. However, the plots indicate that the speed-up when comparing the old and the recursive strategy is increasing with intrinsic program complexity, which is a nice phenomenon, probably due to the explicit computation order given to each SCC.

An interesting fact is that the spread in the measurements is always greater when the comparison is based on the execution time instead of the number of function applications. This indicates that a measure of the cost of each function application and/or the number of comparisons needed in order to detect stability of a (sub)component could be of use.

In the cases where the old strategy outperforms the new strategies in terms of execution time, the number of function applications is still larger for the old strategy (Figure 7.10). The explanation of this phenomenon is that the cost of function applications varies between different equations *and* even for the same function with different arguments; hence fewer, more costly function applications may take longer time to execute than a larger number of cheap function applications. What should be minimized is not the number of function applications, but the total execution time.

7.5 Related Work

One way of sorting a directed graph is to partition the graph into disjoint *intervals* and then collapse the intervals into a *limit flow graph* [ASU86]. A limit flow graph can only be found if the graph is reducible. A method for overcoming this difficulty is using *node splitting*, but this is rather ad-hoc. There are two reasons why the weak topological order approach is better; first, it deals with non-reducible graphs automatically and secondly, it gives an easy way of checking for stability. Handling non-reducible graphs is very important, as almost all global analyses of programming languages supporting recursion give rise to this kind of dependency graphs.

Chapter 8

Properties of Functions

The repetitive approach in fixpoint computations leads to many evaluations of each function in the equations system, hence the cost of a function evaluation is one of the parameters to minimize. It is also the case that the functions in static program analysis frameworks tend to have many arguments. Now, if it is assumed that the cost of evaluating a function is proportional to the number of its arguments, it is a natural step to investigate the functions which don't need to be evaluated with respect to all their arguments in every evaluation.

The following theorem guarantees the existence of certain "transformed" functions that do not need all of their arguments.

Theorem 8.1 (Kleene) *If*

$$\lambda y_1, \ldots, y_m, x_1, \ldots, x_n . \phi(y_1, \ldots, y_m, x_1, \ldots, x_n) := E$$

where E is a recursive definition, then for fixed arguments $y_1, \ldots, y_m$ there exists a function S_n^m such that

$$\lambda x_1, \ldots, x_n . \phi(y_1, \ldots, y_m, x_1, \ldots, x_n) := S_n^m(E, y_1, \ldots, y_m)$$

Proof. [Kle52], Theorem XXIII (Kleene's $s - m - n$ theorem). Recursive definitions are defined in [Kle52], §43. ∎

Starting from this theorem the goal is to investigate the functions $S_n^m(E, y_1, \ldots, y_m)$ which are computationally inexpensive.

8.1 Argument-change dependence

An idea of Sahlin [Sah90] is to compute a function incrementally, which means that the old value of a function is used when computing a new function value. A slightly different viewpoint is to see the new function value as depending on the changes in its arguments since the last evaluation. This could be interesting from a memory perspective as values of arguments need not be saved; what is important is the current value of the function and the changes in its arguments and the function thus depends on the changes in its arguments.

Definition 8.1 (argument-change dependence) Let $\phi : L^n \to L$ be a function. ϕ is *argument-change dependent*, if there exists a function $\mathcal{F}$ and an operation $\circ : L^2 \to L$ such that

$$\phi(x'_1, \ldots, x'_n) = \mathcal{F}_{\{c_1,\ldots,c_k\}}(\phi(x_1, \ldots, x_n), \Delta_{c_1}, \ldots, \Delta_{c_k})$$

where

$$x'_i = \begin{cases} x_i \circ \Delta_i & \text{if } i \in \{c_1, \ldots, c_k\} \\ x_i & \text{otherwise.} \end{cases}$$

□

Thus, instead of performing the operator twice, first updating the arguments in question and then updating the function with its new arguments, the operator is just used once for updating the previous value with respect to "changes".

Example 8.1 (Argument-change dependent functions) An example of a function with this property is addition on $\mathbb{N}_0$: given the previous function value and the change in (some of) its arguments, a new function value is computed by simply adding the changes to the previous function value. Thus, define

$$\phi(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i$$

$$\mathcal{F}_{\{c_1,\ldots,c_k\}}(\phi(x_1, \ldots, x_n), \Delta_{c_1}, \ldots, \Delta_{c_k}) = \phi(x_1, \ldots, x_n) + \sum_{i=1}^{k} \Delta_{c_i}$$

which gives

$$\sum_{i=1}^{n} x'_i = \sum_{i=1}^{n} x_i + \sum_{i=1}^{k} \Delta_{c_i}$$

□

8.2 Partial argument-independence

The assumption that the cost of evaluating a function is proportional to the number of its arguments and the fact that functions in static program analysis tend to have many arguments point to the class of functions which do not need all of their arguments in every evaluation as a particularly interesting one. Alas, the basic intuition is that of a function ϕ which in an iterative computation only needs to be evaluated with respect to those arguments which have changed since its latest evaluation and the previous value of the function. The function is thus independent of some of its arguments. A more formal definition is the following:

Definition 8.2 (partial argument-independence) Let $\phi : L^n \to L$ be a function. ϕ is *partially argument-independent*, if there exists a function $\mathcal{F}$ such that

$$\phi(x'_1, \ldots, x'_n) = \mathcal{F}_{\{c_1,\ldots,c_k\}}(\phi(x_1, \ldots, x_n), x'_{c_1}, \ldots, x'_{c_k})$$

where $\forall i \in \{1, \ldots, n\} : i \in \{c_1, \ldots, c_k\} \Leftrightarrow x'_i \neq x_i$. □

Example 8.2 (Partially argument-independent functions) Constant functions are trivial examples of partial argument-independence. On a lattice, the lub-function is non-trivially partially argument-independent; to specialize further, the max-function on $\mathbb{R}$ is also partially argument-independent. □

Theorem 8.2 (lub is partially argument-independent)

$$lub(x'_1, \ldots, x'_n) = lub(lub(x_1, \ldots, x_n), x'_{c_1}, \ldots, x'_{c_k})$$

where $\forall i \in \{1, \ldots, n\} : i \in \{c_1, \ldots, c_k\} \Leftrightarrow x'_i \neq x_i$.

Proof. Follows from the definition with $\mathcal{F}_{\{c_1,\ldots,c_k\}} = lub$. ∎

8.3 Application

All this suggests that argument-change dependent functions as well as partially argument-independent functions should be identified and used more in static program analysis, of course bearing in mind that some overhead is introduced by the book-keeping necessary for selecting the appropriate arguments in every function evaluation.

In the current framework for analyzing AKL, the **lub** functions account for approximately 20% of the total number of functions in the equations systems generated. The **conc** functions are similar to **lub** functions and these two types can therefore be merged. If this merger is done, functions of the lub type account for more than 50% of the total number of functions.

Chapter 9

Conclusion

This final chapter summarizes what has been said and tells what lies ahead.

9.1 Summary

In this thesis, a self-contained presentation of fixpoint computations in static program analysis, from theory to application, has been presented, the most important parts of which are: an introduction to lattice theory and fixpoints, a survey of methods used in fixpoint computations in static program analysis, the first major experimental evaluation of iteration strategies based on weak topological orders found by Bourdoncle's algorithm, an implementation giving a speed-up of, on average 300% and in some cases up to 3000% in the analysis framework for AKL, and, finally, the concept of *partially argument-independent functions* which decreases the cost of function applications.

9.2 Future Work

In the spirit of topological sorting of *weighted* directed graphs, it would be very interesting to try to generalize the concept of a weak topological order to account for weighted edges, each weight corresponding to the computational cost of the corresponding functions. The experimental results, where a small number of applications yet leads to large execution times in some of the examples, indicates that such a sorting method could lead to further speed-ups.

Further analyses of the merits of the iterative and recursive WTO-strategies in relation to each other and to the old strategy based on SCCs and chains, would have to take not only the number of function applications and the execution time into account, but also *the number of comparisons* needed to detect stability.

Dependency graphs are often dynamic in more general frameworks. A natural question is whether it would be computationally feasible to use Bourdoncle's algorithm to dynamically sort those parts of the dependency graph which are affected by the dynamic changes (new edges etc).

With respect to parallel execution, it would be interesting to see if Bourdoncle's algorithm can be parallelized. This could be quite hard, as the algorithm is based on Tarjan's algorithm which deals with the problem of finding the strongly connected components of a directed graph; a problem which has been known for some time to be very difficult to solve in parallel. On the other hand, the complexity of Tarjan's algorithm is already linear, whereas the complexity of Bourdoncle's algorithm is a function of the complexity of the graph, which should leave room for improvement in the case of a parallel algorithm. In this context, a partial order on SCCs facilitating parallel evaluation of these components could be of interest.

In order to see if there are any practical advantages of using argument-change dependent functions and partially argument-independent functions an experimental test of partially argument-independent functions, e.g. lub, should be performed. This test will show if the reduction in time for function applications outweighs the loss introduced by the book-keeping overhead. If this turns out to be the case, **conc** functions should be merged with the **lub** functions in the AKL-analysis framework, leading to less redundant work in the fixpoint solver.

A spin-off of weak topological orderings in static program analysis is the following. Given a static program analysis and the resulting equation system, the weak topological order of the dependency graph can be used as a way of automatically measuring the complexity (or, perhaps better, the degree of intricacy) of a program statically [Zus91]. This measure should then be based on the intrinsic complexity of the graph, as given by the weak topological order, e.g. as $(\max depth) \cdot edges$, or, as the sum of the mean values of $depth \cdot edges$ in every strongly connected component.

Finally, the implementation of the current fixpoint solver in the AKL analysis framework is written in AKL itself. What would the speed up be if the solver was written in an imperative language like C?

Bibliography

[AH87] Samson Abramsky and Chris Hankin. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, Chichester, 1987.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[BBP93] Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors. *Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, Academgorodok, Novosibirsk, Russia, June/July 1993. Springer-Verlag.

[Big85] Norman L. Biggs. *Discrete Mathematics.* Oxford University Press, Oxford, 1985.

[Bou92a] François Bourdoncle. Abstract interpretation by dynamic partitioning. Technical Report 18, DIGITAL Paris Research Laboratory, March 1992.

[Bou92b] François Bourdoncle. *Sémantiques des langages impératifs d'ordre supérieur et interprétation abstraite.* PhD thesis, École Polytechnique, 1992.

[Bou93] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In Bjørner et al. [BBP93], pages 128–141.

[BP93] Maurice Bruynooghe and Jaan Penjam, editors. *5th International Symposium on Programming Language Implementation and Logic Programming, PLILP '93*, volume 714 of *Lecture Notes in Computer Science*, Tallinn, Estonia, August 1993. Springer-Verlag.

[Bra94] Per Brand. Enhancing the AKL compiler using global analysis. Deliverable number D.WP.2.1.3.M2 of the ESPRIT project ParForce 6707, `ftp://ftp.sics.se/www/ps/parforce/deliverables/D.WP.2.1.3.M2.ps.Z`, September 1994.

[CAA+94] Mats Carlsson, Johan Andersson, Stefan Andersson, Kent Boortz, Hans Nilsson, Thomas Sjöland, and Johan Widén. SICStus Prolog User's Manual Version 2.1 #9. SICS Research Report R93:01, April 1994.

[CC77] Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. *SIGPLAN Notices*, 12(8):1–12, 1977.

[CC92a] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2 & 3):103–179, July 1992.

[CC92b] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[CFFR93] Patrick Cousot, Moreno Falaschi, Gilberto Filè, and Antoine Rauzy, editors. *Third International Workshop on Static Analysis, WSA '93*, volume 724 of *Lecture Notes in Computer Science*, Padova, Italy, September 1993. Springer-Verlag.

[Che94] Li-Ling Chen. *Efficient Computation of Fixpoints that Arise in Abstract Interpretation*. PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[Cou81] Patrick Cousot. *Semantic Foundations of Program Analysis*, chapter 10. In [MJ81], 1981.

[DM90] Pierre Deransart and Jan Małuszyński, editors. *2nd International Symposium on Programming Language Implementation and Logic Programming, PLILP '90*, volume 456 of *Lecture Notes in Computer Science*, Linköping, August 1990. Springer-Verlag.

[DP90] B.A. Davey and H.A. Priesley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, 1990.

[Fri94] Peter A. Fritzson, editor. *5th International Conference on Compiler Construction, CC '94*, volume 786 of *Lecture Notes in Computer Science*, Edinburgh, U.K., April 1994. Springer-Verlag.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman, San Francisco, 1979.

[Grä78] George Grätzer. *General Lattice Theory*. Birkhäuser Verlag, Basel, 1978.

[HDT87] Susan Horowitz, Alan Demers, and Tim Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, (24):679–694, 1987.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[Jan94] Sverker Jansson. AKL a Multiparadigm Programming Language. Uppsala Theses in Computing Science 19, SICS Dissertation Series 14, 1994.

[Jia93] Bin Jiang. I/O- and CPU-optimal recognition of strongly connected components. *Information Processing Letters*, 45:111–115, 1993.

[Jør93] Niels Jørgensen. Chaotic fixpoint iteration guided by dynamic dependency. In Cousot et al. [CFFR93], pages 27–44.

[Jør94] Niels Jørgensen. Finding fixpoints in finite function spaces using neededness analysis and chaotic iteration. In Le Charlier [LC94], pages 329–345.

[JS94] Gerda Janssens and Wim Simoens. On the implementation of A.I systems for (constraint) logic programs. In Fritzson [Fri94], pages 27–44.

[Kle52] Stephen C. Kleene. *Introduction to Metamathematics.* D. van Nostrand, Princeton, New Jersey, 1952.

[Las87] Jean-Louis Lassez, editor. *Fourth International Conference of Logic Programming, ICLP '87*, Melbourne, May 1987. MIT Press.

[LC94] Baudouin Le Charlier, editor. *First International Static Analysis Symposium, SAS '94*, volume 864 of *Lecture Notes in Computer Science*, Namur, Belgium, September 1994. Springer-Verlag.

[LCDMVH93] Baudouin Le Charlier, Olivier Degimbe, Laurent Michel, and Pascal Van Hentenryck. Optimization techniques for general purpose fixpoint algorithms. Practical efficiency for the abstract interpretation of Prolog. In Cousot et al. [CFFR93], pages 15–26.

[LCVH92] Baudouin Le Charlier and Pascal Van Hentenryck. A universal top-down fixpoint algorithm. Technical Report 25, Brown University Computer Science, May 1992.

[LLo87] John W. LLoyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second, extended edition edition, 1987.

[MJ81] Steven S. Muchnik and Neil D. Jones. *Program Flow Analysis: Theory and Applications.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Nil92] Ulf Nilsson. *Abstract Interpretations & Abstract Machines.* PhD thesis, Linköping University, 1992.

[NN92] Hanne Riis Nielson and Flemming Nielson. Bounded fixed-point iteration. *Journal of Logic and Computation*, 2(4):441–464, August 1992.

[NSS94] Esko Nuutila and Eljas Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1):9–14, 1994.

[O'K87] Richard A. O'Keefe. Finite fixed-point problems. In Lassez [Las87], pages 729–743.

[Ros93] Mads Rosendahl. Higher-order chaotic iteration sequences. In Bruynooghe and Penjam [BP93], pages 332–345.

[Sah90] Dan Sahlin. Finding the least fixed point using wait-declarations in Prolog. In Deransart and Małuszyński [DM90], pages 332–345.

[Sar93] Vijay A. Saraswat. *Concurrent Constraint Programming.* Logic Programming and Doctoral Dissertation Award. MIT Press, Cambridge, Massachusetts, 1993.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Massachusetts, 1986.

[SS93] Dan Sahlin and Thomas Sjöland. Towards an analysis tool for AKL. Deliverable number D.WP.1.6.1.M1 of the ESPRIT project ParForce 6707, `ftp://ftp.sics.se/www/ps/parforce/deliverables/D.WP.1.6.1.M1.ps.Z`, July 1993.

[SS95] Thomas Sjöland and Dan Sahlin. Fixpoint analysis of type and alias in AKL programs. SICS Research Report R94:13b, `ftp://ftp.sics.se/pub/SICS-reports/Reports/SICS-R--94-13b--SE.ps.Z`, January 1995.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, Massachusetts, 1977.

[Tar72] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

[VWL94] B. Vergauwen, J. Wauman, and J. Lewi. Efficient fixpoint computation. In Le Charlier [LC94], pages 314–328.

[Zus91] Horst Zuse. *Software Complexity: Measures and Methods.* de Gruyter, Berlin, 1991.

Appendix A

AKL

A.1 Overview

AKL (Agents Kernel Language, also known as Agents) [Jan94] is a multiparadigm programming language based on a concurrent constraint framework (CCP) [Sar93]. Paradigms supported include: *imperative* (ability to mutate data-objects, e.g. Basic), *procedural* (operations associated with certain data-types, e.g. Algol), *object-oriented* (types of data-objects with associated operations and the ability to define new types based on existing ones, e.g. Smalltalk), *functional* (evaluations of recursive expressions, e.g. Lisp), *logical* (deductions from facts and relations, e.g. Prolog) and *process-based* (processes which communicate by message-passing, e.g. Erlang).

The unifier of these paradigms is the notion of a *constraint system*, which is a collection of *constraints*, i.e. pieces of partial information about the values that variables can take and an *entailment relation*, which specifies what can be deduced from a set of constraints. Instead of equating a variable with a value in a store, a variable is specified (by various constraints) to be in a certain range, e.g. $X > 17, X < 42$. The execution is then driven by constraints; information about a variable can be added to the constraint store and choices can be made based on what is known (or follows logically from what is known) about certain variables.[1]

The language incorporates ideas of predicates as processes (concurrency) which synchronize through guard operators and conditions on variables (constraints) without sacrificing the nondeterministic search capabilities of Prolog.[2] It can be seen as an effort to generalize Prolog in a number of directions: the constraint solving for equality on Herbrand terms is generalized to arbitrary constraint domains; the cut-operator is abandoned in favour of other guard operators with more well-defined semantics, the introduction of ports which enhances I/O-capabilities and facilitates object-oriented programming techniques, the notions of encapsulation and stability for taming the computationally expensive non-determinism to name but a few.

[1] It is reassuring to know that a constraint system is nothing but a complete lattice of simple constraints where the entailment relation $c \vdash d$ ("d follows logically from c") imposes an order on the constraints.

[2] See [SS86] for an advanced textbook on Prolog and [LLo87] for a rigorous treatment of logic programming in general.

P	$::=$	A	*agent*
	$\mid$	D . P	
D	$::= \{$	$p(\bar{x})$ **:=** A $\}$	*definition*
A	$::=$	$c(\bar{x})$	*constraint*
	$\mid$	$p(\bar{x})$	*call*
	$\mid$	A , A	*composition*
	$\mid$	$\bar{x}$: A	*hiding*
	$\mid$	C	*choice*
	$\mid$	**bagof**(x, A, y)	*aggregate*
C	$::=$	($\bar{x}$: A % A $\{$; $\bar{x}$: A % A $\}$)	
%	$::=$	$\rightarrow$	*conditional guard*
	$\mid$	$\mid$	*commit guard*
	$\mid$	?	*wait guard*

Table A.1: Statement Syntax

A.2 Syntax

Programs in AKL can be written according to two different syntaxes; either in the clausal style of Prolog, where a number of clauses, consisting of a head followed by statements, make up the definition of a predicate, or in a more statement-like style, where a predicate is defined as a sequence of statements. In fact, the clausal syntax is just syntactic sugar[3] for the statement syntax and set of simple transformation rules facilitates a translation between the two syntaxes of AKL.

The syntax presented in Tables A.1 and A.2 uses the following conventions. Assuming a set of variables and names given, a *constraint atom* is an expression of the form $c(\bar{x})$ expressing a constraint on the variables $\bar{x}$, where c is a name; a *program atom* is an expression of the form $p(\bar{x})$, where p is a name.

Example A.1 (Append) The classical append predicate could be written in the following way using the statement syntax:

```
append(X, Y, Z) :=
           (X = [] -> Z = Y
             ; X = [Head|Tail], append(Tail, Y, Z1), Z = [Head|Z1]).
```

[3]SAn alternative way of coding which is more succinct or more like some familiar notation. It does not affect the expressiveness of the formalism.

P	$::=$	A	*agent*
	$\mid$	$D\ P$	
D	$::=$	$\{\ p(\bar{x})\ \textbf{:-}\ A\ \%\ A\ \textbf{.}\ \}$	*clausal definition*
A	$::=$	$c(\bar{x})$	*constraint*
	$\mid$	$p(\bar{x})$	*call*
	$\mid$	$A\ \textbf{,}\ A$	*composition*
	$\mid$	$\bar{x}\ \textbf{:}\ A$	*hiding*
	$\mid$	$\textbf{bagof}(x, A, y)$	*aggregate*
$\%$	$::=$	$\rightarrow$	*conditional guard*
	$\mid$	$\mid$	*commit guard*
	$\mid$	$?$	*wait guard*

Table A.2: Clausal Syntax

When using clausal syntax, it takes the form

```
append([], Y, Z) :-
     ->    Y = Z.
append(X, Y, Z) :-
     ->    X = [Head|Tail],
           Z = [Head|Z1],
           append(Tail, Y, Z1).
```

□

A.3 Semantics

The defining semantics of AKL is given as an operational semantics. It is formalized by the *computation model* which is a labeled transition system on configurations with a set of rewrite rules defining the transition relation. Each of the computation rules correspond to an actual computation step and an AKL execution is a (possibly non-deterministic and possibly infinite) sequence of configurations.

An actual computation proceeds in a hierarchical system of state objects (*and*-, *choice*- and *or*-boxes) where each object has its local store of constraints. A constraint solver used together with the rewrite rules determines whether a constraint or a set of constraints are entailed or not with respect to the local constraint store and all parent constraint stores.

The different guard operators in the language provide three different kinds of choice: deterministic (always the same choice under the same circumstances), don't care non-deterministic (possibly different choices under the same circumstances) and don't know nondeterministic (different time-separated choices possible). As seen in the syntax definition a choice statement has the form:

$$\begin{array}{l} (\ G_1\ \%\ B_1 \\ \ \ ;\ldots \\ \ \ G_n\ \%\ B_n) \end{array}$$

where G_i % B_i is a *guarded clause* with G_i as *guard* and B_i as *body*. The *guard operator* $\% \in \{\rightarrow, |, ?\}$ must be the same in every guarded clause.

The conditional guard ('$\rightarrow$') is a way of ensuring a deterministic conditional branching as the guards are evaluated in "textual order" and *the first* body which has all its guard goals solved and the resulting constraints are entailed by the surrounding computation is choosen.

The commit guard ('$|$') facilitates don't care nondeterminism as *any* body which all its guard goals solved and the resulting constraints entailed by the surrounding computation can be chosen.

The wait guard ('?') provides don't know nondeterminism; execution proceeds in *all* bodies with all its guard goals solved. Note that there is no requirement that the resulting constraints must be entailed by the surrounding environment; instead the (possibly concurrent) execution of these bodies is delayed "as far as possible".

An interesting aspect of AKL is that it has *deep guards*, which means that any agent expression may appear before a guard operator, giving the possibility of *local* computations in nested guards. Usually, guards are required to contain only constraint expressions and/or built-in predicates.

Composition expresses concurrency, i.e. that the two components can proceed independently. Hiding is a way of introducing local variables and aggregates are used as a means of interaction between determinate and nondeterminate code. Nondeterminism inside an aggregate is not propagated outside of it; hence the term *encapsulation*.

Appendix B

Benchmarks

35 programs from the AKL demo directory were analyzed by the AKL analysis framework (compiled by the sequential AGENTS 1.0 system as of 25 January 1995) running on a Sun SPARCstation 5 (70 MHz) with 256 MB of main memory.

A very brief description of each program is given in Table B.1. The results are presented in Table B.2 sorted by number of equations, where *filename* is the name of the AKL source program, *size* its size in bytes, *eqs* is the number of equations (or program points) in the equation system provided by the AKL analysis framework, *scc* is the total number of strongly connected components (including strongly connected components consisting of a single vertex) of the dependency graph of the equation system, *dp* and *hd* is the maximum depth, i.e. $\max \delta(v)$, and the total number of heads given by the weak topological order of the dependency graph, respectively. *Rec (WTO)* and *Iter (WTO)* denotes the recursive and iterative strategy, respectively, where both strategies are based on a weak topological order of the dependency graph; *Old (SCC)* denotes the old iteration strategy based on strongly connected components and chains; finally, the sub-columns show the total number of applications (*appls*) and the total execution time in seconds of the fixpoint solver (*time*) for the three different iteration strategies compared.

A rough estimation of the execution time of AGENTS compared to Prolog and C is given by the following argument. For a small program like *naive reverse* SICStus Prolog 2.1 [CAA+94] running native code is about 400% faster than AGENTS 1.0; for emulated code it is approximately 25% faster.[1] The execution speed of C is reported to be about 10 times faster than that of Prolog due to the semantic gap between C and Prolog [JS94]. This indicates that programs compiled by the AGENTS system is in the order of 10 to 100 times slower than comparable programs compiled by a standard C compiler. Note, however, that it is notoriously difficult to compare the implementation of a declarative language to an implementation of an imperative language, as the comparison pertains to both language implementation *and* the implementation of the programs on which the comparison is based.

[1] Johan Montelius (SICS), personal communication.

Filename	*description*
fd-eq10	Solves an equations system with 10 unknowns by finite domain constraints.
fd-eq20	Solves an equations system with 20 unknowns by finite domain constraints.
fd-alpha	Finite domain constructs are used to solve the alpha cipher.
fd-sendmm	Finite domain constructs for solving $SEND + MORE = MONEY$.
merge	The standard merge predicate
proc	An example for monitoring processes.
fd-five	Solves the five-puzzle by using finite domain constraints.
sublist	Illustrates how the sublist predicate can be written efficiently.
orp	An or-parallel Prolog interpreter; originally for Concurrent Prolog.
fd-cars	A finite domain constraint version of the car puzzle.
mwmerge	An illustration of the unordered bagof by means of merging of streams.
ancestors	Simple facts-relations example.
fd-magic	Finds magic series by using finite domain constraints.
prove	A meta-interpreter for AKL.
lookup	Modeling Prolog's lookup predicate in AKL showing quit/noisy guards.
kkqueen	Solves the N-Queens puzzle using the candidates/non-candidates.
fd-suudoku	A finite domain constraint version of the Japanese game.
lsqueens	Solves the N-Queens puzzle using layered streams.
fd-queens	The N queens puzzle using finite domain constraints.
cube	Tick's Instant insanity program.
pascal	Computes Pascal's Triangle.
klife	Conway's "Game of Life".
mastermind	Naive generate and test implementation of the classical game.
life	Conway's "Game of Life".
turtles	Tick's turtle puzzle.
ghcexamples	Six GHC sample programs.
zebra	A version of the houses (zebra) program by Van Hentenryck.
puzzle	The puzzle program by Tick.
waves	The waves program by Foster.
knights	Solves the knights tour puzzle.
queens	A program for solving the N-queens puzzle.
semigroup	A filter for unbalanced binary trees.
money	The classical cryptarithmetic puzzle in standard AKL.
triangle	A hand-optimized version of source code written by Okimura.
cipher	Decoding simple substitution ciphers.

Table B.1: Program descriptions.

Filename	*size*	*eqs*	*scc*	*dp*	*hd*	*Rec (WTO)*		*Iter (WTO)*		*Old (SCC)*	
						appls	*time*	*appls*	*time*	*appls*	*time*
fd-eq10	1289	22	10	1	2	43	1.04	43	1.02	48	1.13
fd-eq20	2344	22	10	1	2	43	1.18	43	1.15	48	1.23
fd-alpha	2152	24	10	1	2	49	11.0	49	10.8	54	11.1
fd-sendmm	358	28	10	1	2	61	1.4	61	1.4	66	1.5
merge	182	55	4	4	11	248	2.1	273	2.1	383	2.8
proc	396	78	10	4	14	302	2.1	354	2.3	492	3.3
fd-five	1368	89	25	2	8	182	9.1	180	9.1	236	15.6
sublist	717	96	8	4	17	368	3.4	323	2.9	721	5.8
orp	839	126	4	7	24	602	147	750	178	1576	201
fd-cars	4440	136	10	1	2	385	242	385	241	390	243
mwmerge	630	151	11	4	11	408	15	340	12	662	19
ancestors	781	165	25	8	26	510	4.7	464	4.3	641	4.5
fd-magic	3619	175	10	8	26	788	11	738	9	1283	15
prove	930	211	4	9	32	1602	94	904	39	2381	34
lookup	992	213	39	3	23	688	10	600	8	1183	17
kkqueen	1317	255	16	11	43	1864	35	1445	24	3336	77
fd-suudoku	3060	285	10	5	50	1113	590	1686	488	3243	111
lsqueen	1539	290	16	13	46	1757	42	2095	37	3615	70
fd-queens	2489	308	69	8	37	1171	13	1072	11	1772	24
cube	2947	361	13	14	56	2278	98	2565	91	6262	249
pascal	1766	367	16	17	59	2995	61	2783	60	4942	131
klife	6745	498	19	15	81	2835	175	2934	120	5581	302
mastermind	3337	580	16	25	102	5227	230	4625	155	13687	734
life	4321	615	38	16	94	2646	74	2929	73	4590	144
turtles	3426	669	13	20	109	6040	556	4219	476	16016	3065
ghcexamples	3796	754	51	16	128	5365	94	4442	69	8476	168
zebra	6025	802	35	13	124	3104	47	3380	67	7179	216
puzzle	9638	813	13	19	118	19965	3658	5574	1150	65749	17749
waves	7420	877	16	28	121	8167	929	6710	660	17090	2411
knights	5716	959	49	23	123	20129	372	11313	193	40171	945
queens	3988	996	28	19	158	3210	80	3532	75	7045	157
semigroup	5270	1021	10	41	173	17329	2327	8873	5837	36953	2808
money	6692	1116	34	18	181	6068	89	5636	134	11721	410
triangle	6865	1208	22	11	167	12644	2234	7227	1063	115583	33437
cipher	7820	2615	35	17	456	12202	1096	11393	1584	12630	7094

Table B.2: Benchmarks