MATHEMATISCHES FORSCHUNGSINSTITUT OBERWOLFACH

T a g u n g s b e r i c h t 2/1979

Formale Methoden und mathematische Hilfsmittel für die Softwarekonstruktion

7.1. bis 13.1.1979

Die diesjährige Tagung über "Formale Methoden und mathematische Hilfsmittel für die Softwarekonstruktion" stand unter der Leitung von H. Langmaack (Kiel), E.J. Neuhold (Stuttgart) und M. Paul (München).

Die Tagung hat sich vor allem mit den Gebieten der Programmiersprachenübersetzung, der Betriebssysteme und der Datenbanksysteme beschäftigt. In den letzten Jahren wurden die vielen, rein praktischen Techniken mehr und mehr durch mathematische Überlegungen untermauert und des öfteren auch korrigiert und verbessert. Die Vollständigkeit dieser Betrachtungen läßt aber trotz allem noch sehr zu wünschen übrig, aber man darf doch auf Grund der bis jetzt gefundenen Ergebnisse mit großer Hoffnung in die Zukunft blicken. Diese Auffassung wurde von den Teilnehmern einhellig vertreten, und es ist deshalb beabsichtigt, dem sehr häufig geäußerten Wunsch zu folgen und im Abstand von etwa zwei Jahren wieder eine ähnliche Tagung durchzuführen.

Teilnehmer

K.R. Apt, Rotterdam
E. Bertsch, Hagen
D. Bjørner, Lyngby (Dänemark)
A. Blikle, Warschau
C. Boehm, Rom
W. Brauer, Hamburg
M. Broy, München
G. Cousineau, Paris
A.B. Cremers, Dortmund
W. Damm, Aachen
P. Deussen, Karlsruhe
M.S. Feather, Edinburgh
H. Ganzinger, München
G. Goos, Karlsruhe
K. Indermark, Aachen
H. Kääb, Konstanz
P. Kandzia, Kiel
U. Kastens, Karlsruhe
H.-J. Klein, Kiel
L. Kott, Paris
F. Kröger, München
H. Kröger, Kiel
K. Lagally, Stuttgart
H. Langmaack, Kiel
P. Lauer, Newcastle-upon-Tyne
J. Loeckx, Saarbrücken
R. Loos, Karlsruhe
O. Mayer, Kaiserslautern
E.J. Neuhold, Stuttgart
E.-R. Olderog, Kiel
Th. Olnhoff, Stuttgart
M. Paul, München
V. Penner, Aachen
P. Raulefs, Bonn
J.-L. Remy, Nancy
W.P. de Roever, Utrecht
L. Schmitz, Neubiberg
W. Schönfeld, Stuttgart
H.-E. Sengler, Hamburg
J.E. Stoy, Oxford
R. Valk, Hamburg
A. Wang, Oslo
M. Wirsing, München

Vortragsauszüge

W. P. de Roever: Progress on finite, communicating sequential processes

Denotational semantics are obtained for the flow-of-control in finite, communicating, sequential processes in Hoare's language CSP (CACM Aug. '78). The project derives its interest from occurrence of two different forms of nondeterminism, (a) local nondeterminism restricted to one process only, and (b) global nondeterminism, which can only be resolved after consultation between processes. Since only

finite executions of processes are focussed upon, and resulting local final states of each process should satisfy postconditions - e.g., a parallel sorting routine should produce a sorted outcome - we utilize a variant of Dijkstra's weakest precondition semantics wp [S] q, S describing a CSP command, q a postcondition. Integration of communication in this framework takes place by introducing (a) a-priori semantics for every process P_i, describing P_i's behaviour for every message communicable which contributes to termination of S inside q, and (b) a binding operation which "binds" the a-priori semantics fo all communicating processes by establishing matching pairs of input/output communications, preserving the relation order in which messages are communicated. A system of simple equations for constructs of CSP is obtained improving upon previous results by Francez, Hoare, Lehmann, de Roever. The rule for boolean guarded commands for non-communicating computation must be changed because of inter-process communication. Equations for do-loops are of the same order of complexity as for the sequential case.

P.E. Lauer: On the design and certification of asynchronous systems of processes

An abstract notation (called COSY (concurrend system) notation) for specifying the required behaviours of concurrent systems with distributed control was presented. Its associated formal theory of system adequacy (a strong form of absence of deadlock) was briefly sketched. The formal theory involves the definition of the semantics of COSY programs at differing levels of abstractions. The lowest level associates a program with its set of possible sequences of operation activations, i.e. totally ordered histories. This semantics is sufficient for questions of absence of deadlock, for instance, but does not permit explicit representation and subsequent analysis of concurrency in the system. An intermediate semantics associates a program with its set of possible practical orders of operation activations, i.e. partially ordered histories called

trees. This semantics permits explicit representation of concurrency but does not permit explicit representation of points of choice resolution in non-deterministic systems. Our standard semantics associates a program with a finite condition-event net (Petri net), i.e. a finite representation of a finite set of possibly infinite traces. This semantics permits explicit representation of points of arbitrary choice resolution. All three semantics are consistent relative to each other.

On the basis of these semantics we have developed a number of novel proof techniques for proving adequacy of systems specified in the COSY notation. The notation corresponds to so-called regular trace languages (Mazurkiewicz). The proof techniques fall into a number of categories:

(1) For certain types of programs there exist theorems which give sufficient and necessary structural (static) criteria for adequacy. That is, a proof of adequacy consists in showing that a given description of a system, i.e. a program, has a certain syntactic structure which implies the impossibility of deadlock in the system.

(2) For a larger class of programs there exist rules for eliminating parts of a given program under preservation of adequacy properties. For a large class of adequate programs an application of these excision rules leads either to the empty program, in which case the original program was adequate, or the program decomposes into disjoint subcomponents each of which allow the application of theorems of type (1) to prove their adequacy.

(3) For another large class of programs we have rules for starting with adequate programs for which theorems of type (1) are applicable and using a set of substitution rules to obtain programs for which type (1) theorems

do not hold but which are still adequate since the substitution rules preclude the introduction of structures responsible for deadlocks.

(4) Another class of programs can be obtained by means of rules for combining adequate subsystems to obtain larger adequate systems. These rules are based on formal notions such as extensions, joins and linkages one program by another.

The interrelationships of the larger classes of adequate programs obtained by (2) - (4) have not yet been exhaustively studied but it seems that the classes are not identical. But we have found no published adequate system strategy whose equivalent formulation in COSY was not possible and whose adequacy was not provable by using (1) - (4). The author would welcome requests for literature concerning COSY from those interested.

A.B. Cremers: Arbitration and queueing under limited shared storage requirements

An algorithm is presented which implements mutual exclusion for a system of n processes by means of protocol-controlled communication on an (n+const.)-valued shared buffer. The algorithm uses a generalized test-and-set instruction, and schedules processes into their critical sections on a first-come, first-serve basis. The method can be extended to accommodate any queueing discipline defined as a function of the system history between consecutive idle periods. (The results presented here have been obtained jointly with T.N. Hibbard.)

R. Valk: Über Verifikation und Reduktion paralleler Programme

Die bisher bekannten Methoden zur Verifikation von parallelen Programmen führen schon bei relativ einfachen Beispielen zu

Gefördert durch
DFG Deutsche Forschungsgemeinschaft

langen und unübersichtlichen Beweisen. Es wird gezeigt, wie die zuerst von Lipton eingeführte Methode der Reduktion eingesetzt werden kann, um solche Beweise stark zu verkürzen, das Finden von Invarianten zu erleichtern und informelle Argumentation durch ein exaktes Kalkül zu ersetzen. Zur Demonstration der Vorteile dieser Methode wird abschließend gezeigt, wie ein Korrektheitsbeweis von Owicki beträchtlich verkürzt werden kann.

K.R. Apt: Completeness issues in Hoare's logic

COOK (75) succeeded in finding a notion of completeness such that, among others, Hoare's usual proof system for while-programs is complete. GORELICK (75) showed how to extend this result to include recursive procedures. Since then several other completeness (and incompleteness) results have been proved.

These completeness proofs are of relevance in practical situations:

1. Since they are, in a way, constructive, they indicate how to proceed in concrete cases.
2. They should help in finding the strategies for interactive program verifiers and indicating where user interaction is needed.
3. They explain where, whether, and how to use auxiliary variables.
4. They show what kind of assertions are needed for correctness proofs (e.g. assertions with bounded quantifiers only are not sufficient).

E.-R. Olderog: Korrektheits- und Vollständigkeitsaussagen über Hoaresche Ableitungskalküle

Es wird eine Programmiersprache mit Prozeduren, die sowohl Prozeduren als auch Variablen für elementare Daten als Parameter besitzen dürfen, betrachtet.

Die Semantik dieser Sprache wird durch Kopierregeln auf die Semantik des Sprachanteils ohne Prozeduren zurückgeführt. Neben dieser vollen Semantik wird eine approximative Semantik betrachtet.

Für die eingeführte Sprache wird ein Hoarescher Ableitungskalkül angegeben, dessen Korrektheitsbeweis mit Hilfe der approximativen Semantik einfach durchzuführen ist.

Ferner wird ein allgemeiner Vollständigkeitssatz über diesen Kalkül vorgestellt, aus dem sich mehrere Vollständigkeitsresultate als Korollare ergeben: z.B. die Vollständigkeit für den Fall, daß Prozeduren keine Prozeduren als Parameter besitzen.

F. Kröger: Some exercises in constructing easily verifiable programs

As taught by the protagonists of "structured programming", information for the correctness proof of a program should already be collected during the process of constructing the program. One basic information for verifying a loop construct is an appropriate invariant which can be used in Hoare's invariant rule.

We investigate some modified proof rules for (while-) loops which require some other kind of assertions to be associated with the loop. These formulas may be connected with the problem and the program construction in a more natural and obvious way than invariants.

The method involved by these rules extends also immediately to (classes of) parallel loop programs. As in the sequential case, it is shown by examples, how the formulas carrying the correctness proof can follow directly from the construction of the respective program.

W. Schönfeld: Ein Beweissystem für die Allgemeingültigkeit von Relationengleichungen

Relationengleichungen sind Gleichungen zwischen Relationentermen, d.h. formalen Ausdrücken zur Beschreibung von binären Relationen, die aus Variablen für Relationen mit Hilfe von gewissen natürlichen Relationenoperationen in der üblichen Weise aufgebaut sind. Sie bilden eine deskriptive Sprache, die zur Beschreibung und Definition von Datenstrukturen geeignet ist. Mit Hilfe von Netzen formulieren wir ein Regelsystem, in dem für jede beliebige Gleichung die Allgemeingültigkeit bzw. Widersprüchlichkeit bewiesen oder Gegenbeispiele bzw. Lösungen erzeugt werden können. Dann geben wir ein Verfahren an, das diese Regeln systematisch anwendet. Es ist als korrekter Übersetzer für die deskriptive Sprache anzusehen.

A. Blikle: Assertion programming

The subject of the talk is a method of the systematic development of totally correct programs. The described method has two characteristic features: 1) programs are developed together with their specifications, 2) the specifications consist of a precondition, a postcondition and a set of assertions which represent the correctness proof of the program. Due to the latter property our specifications may be useful in the documentation, maintainance and testing of the program. The paper presents an experimental version of an abstract programming languages oriented towards the proposed method and contains an example of the development of a bubblesort program.

J. Loeckx: Programmierung und Korrektheitsbeweis in der Programmiersprache ALPHARD

ALPHARD ist eine Programmiersprache mit abstrakten Datentypen,

die u.a. darauf zielt, Korrektheitsbeweise praktisch durchführbar zu machen [1].

Der Vortrag bringt eine kurze informale Beschreibung von ALPHARD und bespricht die Regeln für Korrektheitsbeweise. Außerdem wird über einige Erfahrungen beim Programmieren und Beweisen in dieser Sprache berichtet.

[1] W.A. Wulf et al.: "An Informal Definition of ALPHARD" Report CMU-CS-78-105, Carnegie-Mellon-University, Feb. 1978

J.E. Stoy: Correctness of implementation

The problem is discussed of showing the correctness of an implementation of a programming language with respect to the denotational semantic definition of the language. A simple example language is used to demonstrate the advantage of proceeding in several stages, perhaps involving several alternative denotational definitions each embodying successively more implementation decisions, and perhaps including the definition of a compiler and the semantic definition of a machine code. The stages in the application of this technique to realistic languages are briefly discussed.

Another example is used to illustrate the particular problem of demonstrating the congruence of denotational and operational semantics for a language whose denotational semantics uses reflexive domains: problems caused by the lack of correlations between the orderings used in the various inductions. Thus, analysis of the operational definition by fixed point induction can show only that it approximates to the standard set by the denotational one;

conversely, analysis of the denotational definition by structural induction shows only the reverse approximation, since the existence of some of the predicates used in the analysis must be demonstrated by an induction based on the functional complexity ordering.

M. S. Feather: A system for developing programs by transformation

Why is programming hard? A major cause is the clash between efficiency and other programming goals. Program transformation avoids this clash be separating out efficiency as a distinct step in program design.

In order that transformation be a practical method, it must be both reliable and easy to do. A machine-based transformation system is essential.

The concepts underlying an implemented transformation system are presented and illustrated on small examples. The application of the system to non-trivial problems is discussed.

L. Kott: Program transformations

In the area of program transformations we are interested in the system elaborated by R. Burstall and I. Darlington dealing with recursive programs (written in an equational style).

A crucial problem, raised by the authors themselves, is the correctness of the transformation. In other words, if P' is the program obtained from P by transformation does the equivalence of P and P' hold?

Our results are:
1) In general equivalence does not hold.

2) P' computes a function less defined than the one computed by P.
3) We give an effective and sufficient condition for the equivalence.
4) We study cases where this condition became necessary.

P. Deussen: Zerteilerprogramme als Verfeinerungen eines einzigen abstrakten Akzeptionsalgorithmus

Ausgehend von akzeptierenden Semi-Thue-Systemen Π_Q wird ein iterativer Akzeptionsalgorithmus $\mathcal{A}$ formuliert, der die anzuwendenden Produktionen von Π_Q gemäß einem Prädikat $\mathcal{P}$ (nichtdeterministisch) auswählt.

Ist Π_Q vom Typ "top-down" oder "bottom-up", so ist es gerade die Determinismusforderung an $\mathcal{A}$, die bei geeigneter Wahl von $\mathcal{P}$ in natürlicher Weise zu LR(k)-, SLR(k)-, LALR(k)-, LL(k)-, simple LL(k)- und LC(k)-Grammatiken führen. In diesen Fällen ergeben sich aus $\mathcal{P}$ in deduzierbarer Weise die bekannten Parser-Tabellen.

Wird $\mathcal{A}$ rekursiv formuliert und leicht abgeändert, so erhält man die allgemeine Form des rücksetzenden Algorithmus ARIADNE.

ARIADNE ist eo ipso deterministisch und unter wesentlich geringeren Forderungen als für $\mathcal{A}$ bereits partiell korrekt. Ist überdies das Prädikat wieder so gewählt, daß $\mathcal{A}$ deterministisch wäre, so braucht ARIADNE nicht mehr zurückzusetzen und man gewinnt damit den Ursprung des Verfahrens des 'recursive descend'.

U. Kastens: Attributierte Grammatiken im Übersetzerbau

Attributierte Grammatiken eignen sich gut zur formalen Definition von Programmiersprachen und deren Übersetzern. Die Überprüfung

der Wohldefiniertheit einer attributierten Grammatik nach Knuth ist ein exponentielles Problem. Wir haben geordnete attributierte Grammatiken als Unterklassen der wohldefinierten attributierten Grammatiken definiert. Die Klassenzugehörigkeit basiert auf einer Ordnungsrelation über den Attributen. Sie ist mit polynomialem Aufwand entscheidbar. Geordnete attributierte Grammatiken sind hinreichend mächtig für die Definition von Programmiersprachen. Sie umfassen insbesondere alle Klassen von attributierten Grammatiken, die auf bestimmten Strategien zur Attribut-Auswertung basieren. Für geordnete attributierte Grammatiken können automatisch Übersetzer mit tabellengesteuerter Attributauswertung erzeugt werden.

H. Ganzinger: Strukturelle Zusammenhänge zwischen funktionalen Sprachbeschreibungen und Übersetzerbeschreibungen

Die mathematisch-funktionale Methode von Scott-Strachey-Milne ist als ein universelles, aber dennoch hinreichend einfaches Konzept zur abstrakten Sprachbeschreibung bekannt. Es ist daher ein alter Wunsch, spezielle Implementierungen einer Sprache, z.B. in Form eines Übersetzers, möglichst automatisch aus solchen abstrakten Definitionen zu gewinnen. Die heutigen übersetzererzeugenden Systeme basieren jedoch auf den attributierten Grammatiken von Knuth, d.h. sie erzeugen Übersetzer aus Beschreibungen in Form solcher attributierten Grammatiken. Somit stellt sich immer noch das Problem, solche Übersetzerbeschreibungen aus vorgegebenen Sprachbeschreibungen zu entwickeln oder zumindest Konsistenzbeweise zu erbringen.

In diesem Vortrag wird nun versucht, eine neue Strategie zur Entwicklung von Übersetzer- und Sprachbeschreibungen zu skizzieren. Ziel ist es, wesentliche Entwicklungsprinzipien auf formal möglichst einfache Zusammenhänge zwischen verschiedenen Beschreibungsniveaus zurückzuführen. Wesentlich wird dabei sein, daß die attributierten Grammatiken von einem erweiterten, nicht-klassischen Standpunkt betrachten werden. Dies

wird es ermöglichen, alle Zwischenstufen bei der Entwicklung eines Übersetzers aus der Sprachbeschreibung im Rahmen der (erweiterten) attributierten Grammatiken zu formulieren. Konsistenzbeweise werden daher nicht durch metasprachliche Aspekte unnötig verkompliziert, sondern basieren auf bekannten Induktionsprinzipien.

V. Penner: Eine Eingabesprache in ein Compiler-erzeugendes System und Aspekte ihrer Implementierung

Die Eingabesprache in ein Compiler-erzeugendes System auf der Basis attributierter Grammatiken muß im wesentlichen Möglichkeiten zur Angabe regulärer Ausdrücke, kontextfreier Produktionen und von Auswertungsfunktionen für Attribute zur Beschreibung der Microsyntax, Makrosyntax und Semantik von Programmiersprachen bieten.

Aufgabe eines Compiler-erzeugenden Systems ist nun, der Microsyntax einen Scanner, der Macrosyntax einen Parser und der Attributierung einen Algorithmus zur Auswertung von Attributen zu entnehmen.

Bei der Implementierung einer Eingabesprache stellt sich zunächst das Problem, welche Interndarstellung als Schnittstelle für den Scannergenerator etc. gewählt werden soll.

Im Vortrag wird eine Darstellung für Programme der Sprache LDL (Language Description Language) vorgestellt, die zum Aufbau von Scannern basierend auf der "First-Mengen"-Methode, von LR-Parsern und zur Anwendung des Verfahrens von Kennedy/Warren zur Erzeugung von Auswertungsalgorithmen für Attribute günstig ist.

G. Cousineau: Syntactic trees: some of their uses

We call a syntactic tree the (usually infinite) tree that we can associate with a program by splitting all its junctions

Gefördert durch
DFG Deutsche Forschungsgemeinschaft

and unrolling all its loops. These trees can be formally defined as solutions of systems of equations (corresponding to goto programs or recursively defined programs) or described by formal expressions (corresponding to structured programs). We show here that syntactic trees provide a semantics of programs which is well adapted to applications such as:
- comparison of the power of different control structures
- design of a complete axiom system for some classes of program transformations
- studying the power of Hoare-like systems.

A. Wang: Denotational Semantics for non-deterministic goto-programs

A program execution is considered as an iteration of execution steps through internal (non-final) states, until it, eventually, terminates in an external (final) state. Executions that never terminate are sometimes considered as having entered a special external state $\perp$. Two kinds of state transformers and their relationship are investigated: The first kind takes a state as input and yield as output the set of possible next states after one execution step. The other kind takes a set of states as input and yield as output the largest set of internal states which are guaranteed to lead into the original set in one execution step, (cf. Dijkstra's "weakest preconditions"). The state transformers are also investigated w.r.t iteration through several execution steps, and (program-)composition. In the end a rudimentary programming language is proposed to illustrate some of the results.
Source: Wang, A.: Denotational and Axiomatic Semantics for Non-Deterministic Goto-Programs, Part I, Inst. of Inform., Univ. of Oslo, 1978 (Draft)

H. Kröger: Abschätzungsverfahren in der Fixpunktsemantik

Manna und Shamir haben für Funktionale $\tau: \mathcal{F}_n \to \mathcal{F}_n$ den Begriff des optimalen Fixpunktes [1] neben den des minimalen zur Diskussion bestellt und in [2] Verfahren angegeben, wie man von einer beliebigen Startfunktion $f \in \mathcal{F}_n$ zu weiteren Fixpunkten von τ gelangt. Analog zur 'ascending access method' kann man Kontraktionsverfahren konstruieren, mit deren Hilfe die Gesamtheit $Fxp(\tau)$ aller Fixpunkte von τ sowie Teilmengen von $Fxp(\tau)$ abgeschätzt werden können. Dabei werden nur Abbildungs- und mengentheoretische Eigenschaften, insbesondere die Definition des Fixpunktes, benutzt. Die praktische Durchführung wird durch eine Hüllenbildung wesentlich erleichtert. Modifizierte Kontraktionsverfahren kann man in Anlehnung an die bekannten Einzelschritt- und Gesamtschrittverfahren für lineare Gleichungssysteme entwickeln. In dualer Weise kann man zu der 'descending access method' eine Serie von Expansionsverfahren angeben.

[1] Z. Manna, A. Shamir: "The optimal approach to recursive programs", Comm. ACM 20 (1977), 824-831

[2] Z. Manna, A. Shamir: "The convergence of functions to fixedpoints of recursive definitions", Theoretical Comp. Science 6 (1978), 109-141

[3] H. Kröger: "Fixpunkt-Abschätzungen", Bericht Nr. 01/78, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1978

W. Damm: An algebraic analysis of some aspects of the ALGOL 68-procedure concept

We investigate three problems regarding Algol 68-procedures with finite mode:

(1) Macro-Property: is there an input independent bound on the depth of recursive procedure calls during execution of a given program? (Langmaack)

(2) Formal-Termination-Property: is there a finite path in the "execution-tree" of a given program? (Langmaack)

(3) Is any procedure with mode depth n+1 "structurally equivalent" to a procedure with mode depth n? (Indermark)

These problems are lifted to problems on a family of formal languages called "level-n languages" and solved using algebraic techniques. In particular, (1) and (2) reduce to the emptiness problem while (3) leads to the question of strictness of the hierarchy of level-n languages.

D. Bjørner: From formal models of data base models to concrete realizations of data base management systems

Formal, denotational semantics models are given of relational, hierarchical and network data base models. From the last two models are then systematically derived, in carefully controlled steps of formal development, increasingly concrete models of the realizations. Emphasis is put on representational abstraction, and hence on the refinement of abstract data types into machine-oriented data structures. The data base model abstractions represent substantial abstractions 'over-and-above' the data model notions usually presented. It is therefore also the aim of the talk to advocate this approach as a means to come to grips with the real complexities of present-day systems.

It is emphasized that the idea of denotational semantics, as here used, is to make the various data base models mathematically tractable, and likewise their realizations.

P. Kandzia: Zur Äquivalenz relationaler Datenbanken im Zusammenhang mit Normalisierungen

Eine relationale Datenbank läßt sich als abstrakter Automat ansehen, bei dem die Zustände Tupel von Relationen sind und bei

dem die Übergangsfunktionen durch die Grundoperationen Einsetzen, Löschen bzw. Ändern eines Relationen-Elementes erzeugt werden. Mit Hilfe dieser Betrachtungsweise läßt sich ein Äquivalenzbegriff für relationale Datenbanken definieren (in Zeichen: $\sim$), durch den einige der in der Literatur angegebenen Äquivalenzbegriffe eine exakte Grundlage erhalten.

Es wird gezeigt, daß bei dem von R. Fagin angegebenen Zerlegungsprozeß, der zu einer gegebenen Datenbank $\mathfrak{d}$ eine Datenbank $\mathfrak{d}'$ in Boyce-Codd-Normalform (BCNF) liefert, $\mathfrak{d}$ und $\mathfrak{d}'$ nicht notwendig im obigen Sinn äquivalent sind. Jede Datenbank $\mathfrak{d}$ kann aber so zerlegt werden, daß die entstehende Datenbank $\mathfrak{d}'$ in BCNF ist und auch $\mathfrak{d} \sim \mathfrak{d}'$ gilt. Der zu einer derartigen Zerlegung gehörende Algorithmus läßt sich durch Hinzunahme von Zusatzattributen so abändern, daß er auf Datenbanken mit beliebigen, die Zustände betreffenden Integritätsbedingungen angewandt werden kann. Er liefert in dieser Form unter Beachtung der obigen Äquivalenz eine Datenbank, die eine auf beliebige Integritätsbedingungen verallgemeinerte BCNF besitzt.

Th. Olnhoff: Semantics of a relational database-query-language in Logic for Computable Functions

The query-language (QL) is essentially the relational algebra (Codd) with the operators confined to join, select, project. A query gets evaluated after some modification (optimization). A language (OQL) is proposed to define the output of such an optimization. LCF which essentially is a typed λ-calculus enhanced by a fixpoint-operator is used to give the semantics of OQL as a set of functions that map a database into itself. Only the type "list" is used. The database is a list with the relations as sublists including their description. A LCF function "inverse optimization" produces the equivalent QL-representation of the optimized query representation. This enables to check correctness corresponding to

Gefördert durch
DFG Deutsche Forschungsgemeinschaft

an optimization and/or to some (differently) defined search procedures. To do so LCF is of help. -- The use of the proposed semantic description for a query-hostlanguage is sketched.

M. Wirsing: Existential quantifiers in abstract data types

Abstract data types are an important tool for the formal specification of problems, and for the reliability and the verification of programs. If one restricts the form of the axioms of a type there exists a "greatest" model of the type - the initial algebra - and a "least" model - the terminal algebra. The conditions for the existence of initial algebras were investigated by Thatcher et al. in 1976. We show that terminal algebras exist if the axioms are positive formulas, i.e. if they have the form $Q_1x_1 \ldots Q_kx_kA$ where $Q_1,\ldots,Q_k$ are universal or existential quantifiers and A is built from disjunctions and conjunctions of equalities. In the (only nontrivial) case where an abstract data type is based on primitive types a supplementary but necessary condition is required: the type must be t-complete. Routhly speaking, each sufficiently complete type is t-complete, too. Therefore this approach provides great freedom to define formal specifications and assures simultaneously the existence of a standard model: the terminal algebra.

J.L. Remy: Systematic construction, evaluation and improvement of recursive definitions

A very simple formalism, abstract data tuples using equations and preconditions, is proposed in order to describe the properties of the objects and the operations of a data structure. Two examples, sets and dictionaries (or binary search trees) are given. A small number of rules and heuristics is developed

to systematically construct and also to evaluate and improve some algorithms of search and insertion in a dictionnary. We are thus led to transform the trivial algorithm of insertion so that the heights of the trees are kept constant as long as possible. Some operations allowing to rebalance a tree (e.g. simple and double rotations) and a property of quasi-balanced trees (AVL) are almost automatically reinvented. A lot of "eureka" is indeed necessary and each is pointed out in the paper.

The final algorithm is not new but is sufficiently difficult for justifying the interest of this type of research which is at the join of two preoccupations: abstract specifications of data structures and evaluation of algorithms.

M. Broy: Correctness of program transformation rules

A program transformation rule is a partial mapping between sets of programs such that certain relations between the given and the generated programs are valid, e.g. both programs are semantically equivalent. Then the correctness of a transformation rule is strongly connected to the particular semantics of the underlying languages. So transformation rules have to be verified with respect to the particular definition of the semantics. On the other hand transformation rules can be given as axioms, e.g. as algebraic laws of an abstract data type. Even (parts of) the semantics of languages can be determined by "definitional transformations". Thus an equivalence relation between programs is established, such that each definition of semantics for a representative kernel of the language can be uniquely extended to the whole language. Then complex transformation rules can be proved by combining definitional transformations, induction proof rules, or model theoretic proof rules of the particular semantics.

H.-J. Klein: Zur Modularitätseigenschaft von Programmen.

Programme ALGOL-ähnlicher Programmiersprachen, in denen geschachtelte Prozeduren vorkommen, können i.a. nicht in äquivalente Programme ohne Prozedurschachtelung umgeformt werden. Es kann gezeigt werden, daß diese Umformung dann möglich ist, wenn Programme die "most-recent"-Eigenschaft haben oder wenn anstatt der ALGOL-Kopierregel die "naive" Kopierregel angewandt wird. Mit Hilfe der "most-recent"-Eigenschaft können wir eine notwendige Bedingung für das Erfülltsein der Modularitätseigenschaft angeben und erhalten einen einfachen Beweis dafür, daß PASCAL-Programme i.a. nicht die Modularitätseigenschaft haben.

C. Boehm: Data structure driven strategies in graph isomorphism algorithms

We first searched for a class of graphs where the isomorphism problem can be decided in a very small number of steps (i.e. polynomial in the number of vertices). Our research group found such a class - the set $\mathcal{L}_n$, formed from $K_n^{(2)}$ (the complete multi-graph of order n and multiplicity 2). By adjoining a loop to each vertex, and a direction and colour to each edge, in such a way that n different coloured edges start from, and arrive at, each vertex. $\mathcal{L}_n$ can be identified with the set of Latin quares of order n.

We then turned to the isomorphism problem of Latin squares, for which a $n^{\log_2 n}$ algorithm is known. A strategy is shown which on average is better than the brute force application of the preceeding algorithm.

The key to both results lies in assuming that the sets of permutations associated with each Latin square are the "driving" data structures.

L. Schmitz: Observations made when synthesizing transitive closure algorithms

Starting from an implicite (non-recursive) problem specification recursive equations are derived that describe Warshall's algorithm. In the same way a family of transitive-closure-algorithms have been obtained using Burstall & Darlington's unfolding-and-folding technique.

It is argued that in the program development process the introduction of programming language constructs (including " $\Leftarrow$ ") should be delayed as long as possible. A systematic method for introducing recursion parameters is presented.

H. Langmaack: On termination of programs

The formal termination problem for a high level programming language is of considerable importance towards compilation and generating efficient object code. In order to study this problem for ALGOL 60 it suffices to investigate the actual termination problem for a dialect A60D. The dialect has only procedure declarations and no other declarations, it has only dummy and procedure statements as basic statements, it has the usual composition of statements, and non-deterministic alternative statements instead of conditional statements.

K. Winkelmann has given an effective method how to transform a program π with finite modes in the sense of ALGOL 68 into a formally equivalent one π' which is strictly monadic, i.e. if $\alpha_o(\alpha_1,\ldots,\alpha_n)$ is a procedure statement in π' and α_i is a formal identifier then $i=0$. So the termination problem for finite mode programs can be reduced to strictly monadic programs which form a more powerful sublanguage of ALGOL 60 than finite mode programs because we can write procedure statements f(...,f,...) with self applications.

The run time stack of a non-critical strictly monadic program $\pi \in$ A60D behaves like a nested stack system. The termination problem for such systems is algorithmically solvable due to A. Aho.

In case of a critical strictly monadic program π we see that the lengths of critical chains in the run time stack are bounded by $t_{max} = 2^{\cdot}\lambda_{max} + 1$ where λ_{max} is the maximal procedure nesting level in π. With the help of the notion of standard procedures of certain order i a critical strictly monadic program π can be rewritten as a non-critical program π' which is not necessarily formally equivalent but has an invariant termination property.

So the formal termination problem for strictly monadic ALGOL 60 - programs is algorithmically solvable. We hope to extend our techniques to generally monadic programs defined as follows: If $\alpha_o(\alpha_1,\ldots,\alpha_n)$ is a procedure statement and α_i,α_j are formal identifiers then their static levels are equal: $\lambda(\alpha_i) = \lambda(\alpha_j)$. We mention that our proof techniques can be formulated purely in phrases well known to (system) programmers and compiler constructors.

P. Raulefs: Semantics of concurrent actor systems

A model of computation based on the folowing assumptions is developed:

1) Computation is done by a society of independent, concurrent sequentially working machines we call actors.
2) Actors may communicate by sending messages to each other. Messages are syntactic entities decoded by recipient actors.
3) An actor may only transmit a message to an actor it is acquainted with. The relation "is acquainted with" is reflexive, but not necessarily transitive nor symmetric. Acquaintances may be transmitted within messages. An

Gefördert durch
DFG Deutsche Forschungsgemeinschaft

acquaintance provides access to an object, and enforces a particular interpretation of the object.

4) Transmitting messages takes a positive, but indefinite amount of time. Computation done by actors takes no time at all.

This model is specified in terms of the programming language CSSA. We extend Lehmann's notion of powerdomain categories to power refinement categories to develop a denotational semantics of CSSA. We show that Petri nets and systems with synchronized communication between concurrent sequential processes (Milner/Milne, Hoare) are isomorphic to subdomains of the domain of static concurrent actor systems (static = no actors created/ no acquaintances transmitted in computation). A formal system AL (Actor Logic) is indicated that provides a proof theory for showing fairness, total/partial correctness and determinacy of static concurrent actor systems.

Open problems are: The semantics is not yet shown to be fully abstract; there are completeness problems for AL; AL should be extended to non-static concurrent actor systems.

E. Bertsch: Representation of precedence tables

We give some results concerning advantageous representations of precedence relations.

In particular, precedence functions in the sense of Floyd et al. appear as special cases of more general notions. Some constructive size estimates are developed, ranging from relative complexities in terms of grammatical families to arc numbers in terms of Bell's algorithm. Some concrete examples (Aho and Ullman's book) demonstrate the usefulness of these concepts.

H.E. Sengler: Language design for comprehensible programs

A basic problem of the use of computers is the complexity of the programs driving them. Programming languages influence the understandability of the programs written using them. Five topics of language design are discussed: The basic concept of a language, the behaviour of the program structure during execution, the visibility of the program structure, abstraction mechanisms as well as the elementary objects of a language. A language then is proposed that uses only four basic elements (store, processor, data-transport, control-transport). The graphical representation of these elements provides a simple parameter mechanism and solves the problem of protecting program elements against unintended use.

G. Goos: Einige Probleme beim Entwurf der Sprache GREEN

GREEN, entwickelt von J. Ichbiah et al., ist eine der beiden Sprachen, die gegenwärtig zur Erfüllung der vom amerikanischen DoD aufgestellten STEELMAN requirements definiert werden. Bei der Umsetzung theoretischer Prinzipien in Spracheigenschaften sind bereits bei sehr einfachen Aufgaben unerwartete Schwierigkeiten aufgetreten. Der Vortrag erörtert 3 Probleme im Zusammenhang mit Datentypen. Zuerst wird gezeigt, daß es für Zahlkonstanten üblicher Schreibweise nicht möglich ist, den Datentyp zu bestimmen, ohne den Kontext zu berücksichtigen. Daraus folgt, daß abstrakte Datentypen in elementaren Fällen nicht den Notwendigkeiten praktischer Programmierung entsprechen. Zweitens wird gezeigt, daß beim Identifizieren von Operatoren und Funktionen die Resultattypen und nicht nur die Operandentypen berücksichtigt werden müssen. Ein praktikabler Fixpunktsatz für diese Aufgabe ist noch nicht bekannt. Schließlich werden verschiedene Methoden der Typidentifizierung erörtert.

E.J. Neuhold, Universität Stuttgart