ABSTRACTS OF THE TALKS AT THE SECOND INTERNATIONAL WORKSHOP ON THE SEMANTICS OF PROGRAMMING LANGUAGES IN BAD HONNEF MARCH 19-23, 1979

Edited by
K. Indermark (*) and J. Loeckx

A 79/1o

Fachbereich 1o
Universität des
Saarlandes
D-66oo Saarbrücken

(*) Rheinisch-Westfälische Technische Hochschule, Aachen

The present report contains the abstracts of the talks at the Second International Workshop on the Semantics of Programming Languages which took place in Bad Honnef on March 19-23, 1979, in the buildings of the Elly-Hölterhoff-Böcking Stiftung. The Workshop was sponsored by the European Association for Theoretical Computer Science and financially supported by the Deutsche Forschungsgemeinschaft and the Minister für Wissenschaft und Forschung des Landes Nord-Rhein-Westfalen.

A B S T R A C T S O F T H E T A L K S

(in the order of their presentation)

INFINITE PROOF RULES FOR WHILE PROGRAMS

Fred Kröger (München)

Many methods and formal proof rules for proving partial correctness of (while) loops by inductive assertions have been proposed in the literature. Well known keywords are for example computation induction, predicate transformers, structural induction and others which all focus - in one way or the other - on the fundamental notion of loop invariants, most concisely expressed in Hoare's original rule $P \wedge B[\Pi]P \vdash P[\text{while } B \text{ do } \Pi \text{ od}]P \wedge \neg B$. As also known, this rule bears a certain kind of incompleteness: we cannot really verify all correct while loops by it.

We introduce infinite proof rules (rules with infinitely many premises) for while loops in order to overcome this proof-theoretical incompleteness and to get some insight into it. In fact, partial correctness of while programs can be completely axiomatized using such a rule. However, these rules are also of practical interest: they put the invariant method into a more general framework and open the view to other kinds of inductive assertions carrying a proof through a loop. They naturally suggest to use "parameter representations" of invariants which might be more manageable in some applications. This being principally not different from the invariant method, the infinite rules give rise also to a proper extension of the latter method: the method of generalized invariants. A generalized invariant is an assertion which need not hold after each single run through the loop but recurs in possibly longer (not necessarily fixed) periods. There are programs for which an induction over such arbitrary "pillars" is adequate and much simpler than an induction with a classical invariant.

AN AXIOMATIC PROOF SYSTEM FOR WHILE-PROGRAMS

Jacques Loeckx (Saarbrücken)

The axiomatic system presented is an extension of the first-order predicate calculus. It is characterized by additional formulas and a few additional inference rules.

The additional formulas are of the form

ij:q

where i and j are cutpoints in a while-program and where q is a formula of the predicate calculus. Intuitively ij:q means that q holds for each path leading from cutpoint i to cutpoint j.

The additional inference rules correspond to classical proof methods: two inference rules may be viewed as an implementation of the inductive assertion method, two further inference rules implement the subgoal induction method and the fifth inference rule implements the well-founded sets method.

For proving the partial correctness or termination of a while-program it suffices to "translate" this program in a set of axioms and then to apply the inference rules.

While being completely formal, proofs are substantially shorter and more readable than proofs in the Hoare system; this essentially results from the simplicity of the formalism used. Some experience with the abstract data programming language Alphard suggests that this may be a definite advantage in practical work.

A formal description of the axiomatic system may be found in [1,2]; a consistency proof is in [2].

[1] J. Loeckx, I. Glasner, "A calculus for proving properties of programs"; presented at the Intern. Conf. on Math. Studies of Inf. Proc. in Kyoto, 1978 (the proceedings of this Conference are to be published as Springer Lecture Notes in Comp. Science)

[2] I. Glasner, "Formale Beweise über while-Programme : Ein Kalkül und sein Modell", Diplomarbeit, Fachbereich 1o, Universität des Saarlandes, Saarbrücken, 1978

RECURSIVE TYPE OPERATORS WHICH ARE MORE THAN TYPE SCHEMES

C.P. Wadsworth (Edinburgh)

To date, natural examples of recursive type operators have been relatively few and limited in the nature of their recursion. Lists, parameterized on the type of their elements, and binary trees, parameterized on the type of their tips, are two well-known examples; these are "schematic" in the sense that they are a schema for a single recursive type definition at each particular type instance of their type parameter (since, e.g., lists of elements of some type α have tails which are lists of elements of the same type α).

This talk presents programming examples of non-schematic recursive type operators, arising from data structures, called tries, for

fast searching of a collection of records indexed by keys (as in, e.g., Knuth, Vol. 3). Tries, parameterized on the type of the records indexed, become nested recursive type operators when the keys are terms of more general languages than character strings. This is illustrated for a simple language of terms which includes combinations $(t_1\ t_2)$, where t_1 and t_2 are terms called the rator and rand, respectively. The genesis of the nested recursion of types is that the (sub-)collection of records indexed by combinations is first partitioned into terms indexed by the terms occurring as rators, the item indexed by each particular rator t_1 being the collection of records indexed by the rands of all combinations whose rator is t_1; then the partitioning is continued recursively for rators and rands which are themselves combinations.

Some implications for typechecking are mentioned briefly, observing that the examples lead to recursively defined, polymorphic functions whose inner and outer calls are at different instances of their "generic", polymorphic type.

The generalized tries are currently used for programming the term matching part of the simplification package in the Edinburgh LCF system, to determine "simultaneously" whether a subject term matches any one of a set of pattern terms occurring as the left hand side of simplification rules.

ABOUT ALGEBRAIC SEMANTICS

Irène Guessarian (Paris)

We study two usual properties of program schemes: termination and a simplification property. To this end, we introduce the notion of useless occurrence which enables us to consider these two problems in a unifying setting /1/; algebraic semantics provides us with a quite flexible framework for this study in that we can introduce the exactly desired amount of semantic information and derive the corresponding results.

We begin with a purely syntactic stage, and, working in the free interpretation, obtain easily at this stage quite worthwhile simplifications. Then we introduce some amount of semantic information, and, working in a class of interpretations defined by algebraic conditions, show how we can delete some of the useless occurrences and non-terminating procedures with respect to that class of interpretations; this is achieved by using only the algebraic specifications of the program scheme and of the class of interpretations. Finally, we show how one can, making more and more precise the specifications of the interpretations, derive

in the same way more and more refined simplifications of the program scheme /2/.
/1/ I. GUESSARIAN, Program transformations and algebraic semantics, to appear in TCS, 1979
/2/ F. ERMINE & I. GUESSARIAN, About termination and simplification of program schemes, in preparation

INFINITE TREES IN NORMAL FORM AND RECURSIVE EQUATIONS HAVING A UNIQUE SOLUTION

Bruno Courcelle (Bordeaux)

A system of recursive equations is C-univocal if it has a unique solution modulo the equivalence associated with a class C of interpretations. This concept yields simplified proofs of equivalence of recursive program schemes and correctness criteria for the validity of certain program transformations, provided one has syntactical easily testable conditions for C-univocality. Such conditions are given for equational classes of interpretations.
They rest upon another concept: the normal form of an infinite tree with respect to a tree rewriting system. This concept yields a simplified construction of the Herbrand interpretation of certain equational classes of interpretations.

The equivalence of two program schemes is a very strong requirement since they must compute the same function in every interpretation. We define the C-equivalence (denoted by $\equiv_C$) similarly, with respect to a given class C of interpretations instead of the full class of all interpretations. This flexibility is necessary for practical applications. In particular, the library of Darlington and Burstall [11] is in fact a set of triples $\langle \phi, \psi, A \rangle$ of two schemes ϕ and ψ and a set A of conditions such that Φ is C_A - equivalent to ψ (where C_A is the class of all interpretations which meet A).

It has been remarked by Courcelle [4], and by Elgot et al. [12]in a restricted case, that a system Σ has a unique solution in $M_\Omega^\infty(F,V)$, under some not very stringent syntactical conditions. Let us say that such a system is univocal. More generally, a system is C-univocal if it has a unique solution in $M_\Omega^\infty(F,V)/\equiv_C$. We raise the following problem:

Problem 1: Which systems are C-univocal ?

Clearly the answer will depend on the way the class C is specified. We shall only answer to the question by giving sufficient conditions, in the case where C is defined by equations. Before describing these conditions we give two applications to program proofs and program transformations of the concept of C -univocality.

Application 1: Simplified program proofs.

Let Σ be a system of the form $< \phi_1 (v_1,\ldots,v_{k_1}) = \tau_1,\ldots,$
$\phi_n (v_1,\ldots,v_{k_n}) = \tau_n >$;
let $\alpha_1,\ldots,\alpha_n$ be functions defined by another system of equations Σ'.
Assume that $< \alpha_1,\ldots,\alpha_n >$ satisfies the system Σ modulo $\equiv_C$ i.e. that :

(1.3.1) $\alpha_i \equiv_C \tau_i\{\alpha_1/\phi_1,\ldots,\alpha_n/\phi_n\}$ for all $1\leq i\leq n$.

(The right hand side of each equation is the substitution of α_j for each occurrence of ϕ_j in τ_i). This means that for all interpretation I in C, $<\alpha_{1_I},\ldots,\alpha_{n_I}>$ is a particular solution of Σ hence that $\phi_{i_I} \leq \alpha_{i_I}$ since $<\phi_{1_I},\ldots,\phi_{n_I}>$ is the least solution of Σ in I. Since this holds for all $I \in C$ we write this:

(1.3.2) $\phi_i \leq_C \alpha_i$ for all $1\leq i\leq n$.

This is essentially the Recursion Induction of McCarthy [23] or the Fixpoint Induction of Park.
If we also know that Σ is C-univocal, we can infer directly (1.3.3.) from (1.3.1.);

(1.3.3) $\phi_i \equiv_C \alpha_i$ for all $1\leq i\leq n$.

Hence we obtain a powerful method, provided the conditions insuring the C-univocality of Σ are not too difficult to test.

Application 2: Validation of certain transformations of recursive programs.

Let us recall the transformation method of Burstall and Darlington [3] using the "unfold-fold-redefine" technique. Given a system Σ reduced to a single equation: $\phi(v_1,\ldots,v_k) = \tau$ (this is only to simplifiy this informal presentation but the extension to general systems is straightforward) and a fixed class of interpretations C, one wants to find Σ' : $\phi'(v_1,\ldots,v_k) = \tau'$ such that $\phi' \equiv_C \phi$ and Σ' is simpler than Σ in some sense.
A possible way to find τ' is to build a sequence $\tau_0,\tau_1,\ldots,\tau_m$ such

that $\tau_0 = \tau$ and for all $0 \leq i < m$:

$\tau_i \xrightarrow[\Sigma]{} \tau_{i+1}$ i.e. some ϕ in τ_i has been rewritten as τ ("unfolding" of[3]) or

$\tau_{i+1} \xrightarrow[\Sigma]{} \tau_i$ ("folding" of [3]) or

$\tau_i \equiv_C \tau_{i+1}$ by some property of the base functions (such that associativity of a binary function) always satisfied in C("using laws about primitives" of [3]).

Assuming that τ_m is "simpler" than τ we would like to take $\tau' = \tau_m \{\phi'/\phi\}$. A sufficient condition is that the equation $\Sigma' : \phi'(v_1,\ldots,v_k) = \tau'$ is C-univocal. It is clear that $\tau_i \equiv_C \tau_{i+1}$ for all i hence that $\phi \equiv_C \tau_m$. This means that ϕ (defined by Σ) is a particular solution of Σ'. Hence $\phi \equiv_C \phi'$ if Σ' is C-univocal. (Note that $\phi' \leq_C \phi$ always holds; an example such that $\phi' \not\equiv_C \phi$ is given in (5.22)).

RATIONAL OPERATIONS ON TREES WITH APPLICATION TO CONTROL STRUCTURES

Guy Cousineau (Paris)

Regular trees can be defined as solutions of regular systems of equations or equivalently as those trees that have only a finite number of distinct subtrees. Here we characterize them as the tree that can obtained from finite ones by means of a finite number of applications of so-called rational operations including some concatenation and star operation on trees. We should like to emphasize the fact that giving such a characterization was not merely for us a tree-theoretic task (if it had been the case, other notions of rationality could have been chosen) but we wanted our rational operations to be meaningful as program constructs.

The tree we manipulate are elements of $M^\infty(F, \mathbb{N})$ in the notations of Nivat when F is a set of function symbols of arity ≥ 1 and $\mathbb{N}$ the set of integers. Concatenation is defined by $a_1 \cdot a_2 = a_1[o \backslash a_2]$ and star by $a^* = \downarrow(a^\infty)$ where a^∞ is the limit of the sequence a^n and $\downarrow$ an operation that decreases by one all the integers that can be found in a tree.

Let us point out some results that can be obtained using this formalism:

1. Semantics of programs containing loops can be very naturally defined and properties of rational expressions induce properties of programs. For example the fact that you can solve a regular system into a rational expression explains why flowchart programs can be structured using repeat - exit and hierarchy results on rational expressions induce the Kosaraju hierarchy on repeat - exit schemes.

2. The rule used to solve regular equations can be used to design a Salomaa-like system for proving identities of rational expressions. From that system, one can derive a sound and complete transformation system to deal with syntactical transformations of programs.

3. The formalism extends very easily to non-regular cases and it gives a way to measure the expressive power of different classes of programs by means of the associated class of trees.

SYNTAX-DIRECTED, SEMANTICS-SUPPORTED PROGRAM SYNTHESIS

Wolfgang Bibel (München)

The semantics of programs simplifies substantially if algorithms are represented by their logic and their control separately. Therefore the question arises whether it is feasible to synthesize the logic and the control of algorithms in separate steps. In this talk a number of strategies for the synthesis of the logic part of algorithms from a given input-output specification of a problem are presented which are centered around a few basic principles. It has been verified for more than ten different algorithms that their uniform application in all cases results in a successfull deductive synthesis, among which are a spanning-tree algorithm, a graph-circuits algorithm, a finding-the-ith-smallest-element algorithm, and a linear (string) pattern matching algorithm. In all cases the subsequent determination of the control is a simple, mechanizable task. Also the correctness of the resulting algorithm is guaranteed by the generated deduction.

This provides some support to the thesis that such a "logic programming" approach is not only desirable but also feasible. It also reveals arguments in favor of a classification of algorithms according to relevant syntactic properties of the logic of algorithms as opposed to one according to program (control) structures such as program schemes.

FUNCTIONAL SEMANTICS OF A RELATIONAL DATABASE-QUERY-LANGUAGE WITH SOME RESPECT TO A HOSTLANGUAGE

Thomas Olnhoff (Stuttgart)

Database-sublanguages become embedded into major programming-languages (e.g. PASCAL by J.W. Schmidt). The semantic-description should not fall short of these sublanguages. A program-state can be split into a non-db-component and a db-component. This talk is mainly concerned with describing a query-sublanguage's semantics as a state-mapping of the db-component.

A database description mostly distinguishes three kinds of models: the external, conceptual and internal model. Several external models are mapped into a conceptual one which is mapped into the internal one. A query is formulated by means of external objects. Its final interpretation is done by internal objects.

The query-language studied has the power of the relational algebra confined to the operators select, join, project. The mappings we allow from one model to the other are noted as well - formed sentences of the query-language (QL). Another language (OQL) is used to express the query on the internal model. Thereby we hope to be able to give the semantics of a query as a state-function being the composition of "elementary" functions that can be thought of as abstractions of modules interpreting the query in an actual implementation.

How equivalence of sentences q and oq (q of QL, oq of OQL) can be decided is sketched.

The Vienna-Definition-Method is used as the metalanguage for the presentation of this talk.

AN EXERCISE IN BEHAVIOUR ALGEBRA; PROVING THAT A QUEUE WORKS PROPERLY.

Robin Milner (Edinburgh)

A queue system built from active cells was considered. Each cell represents a place in the queue, and it may be in one of three states. It may contain a member, in which case it can (if allowed)

pass this member forward to the next higher place, and become empty itself. An empty place has two capabilities - it may receive a member passed from the next lower place, or it may receive a signal from the next lower place that it should become the end place. An end place has two capabilities; it may signal to the next higher place (if empty) to become the end place, or it may receive a new member from outside and then divide into two places -an occupied place and an end place. Any communication requires both participants to perform it simultaneously.

In Behaviour Algebra, the behaviours of the three kinds of place may be defined by mutual recursion

An occupied place:	Place (a)	=	$\overline{\alpha}_1$ a. Empty
An empty place :	Empty	=	β_1 x. Place $(x)+\beta_2$.End
An end place :	End	=	$\overline{\alpha}_2$.NIL+ γx.(Place(x)$\subset$End)

The five ports $\overline{\alpha}_1, \overline{\alpha}_2, \beta_1, \beta_2, \gamma$ are for the following purposes

$\overline{\alpha}_1$	:	Pass a member (up)
$\overline{\alpha}_2$	:	Signal (up) to become end place
β_1	:	Receive member (from below)
β_2	:	Receive signal (from below) to become end place
γ	:	Receive member (from outside)

The chaining combinator $\subset$ is used to join ports of places ($\overline{\alpha}_1$ to β_1, $\overline{\alpha}_2$ to β_2); e.g. the queue with two members (a_1 and a_2) and no empty places is represented by

$$\text{Place } (a_1) \subset \text{Place } (a_2) \subset \text{End}.$$

The exercise consisted in defining, by recursive equations, the behaviour that a queue should have (that is giving its specification) and then proving that this is identical with the behaviour of the composite queue system.

For this purpose, the Laws of Behaviour (presented in "Synthesis of Communicating Behaviours" by the author, in Proc. 7th MFCS Colloquium, Zakopane, Poland, 1978 (Springer - Verlag)) were used, together with three further laws which are part of an equational system which we hope to prove complete (that is, behaviours will be provably identical iff they are observationally equivalent). The proof used a simple case of induction upon the iterates of recursively defined behaviours.

SEMANTICS FOR CONCURRENTLY COMMUNICATING FINITE SEQUENTIAL PROCESSES, BASED ON PREDICATE TRANSFORMERS.

W.P. de Roever (Utrecht)

Denotational semantics are obtained for the flow-of-control of 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 denoting a CSP command, q a postcondition on tuples (s,T) of local states s, and communication futures T. 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 operator which "binds" the a-priori semantics of all communicating processes by establishing matching pairs of input/output communications, preserving the relative order in which messages are communicated. A system of simple equations for constructs of CSP is obtained improving upon previous result 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.

THE USE OF A FORMAL SEMANTICS TO PRODUCE AND PROVE COMPILERS

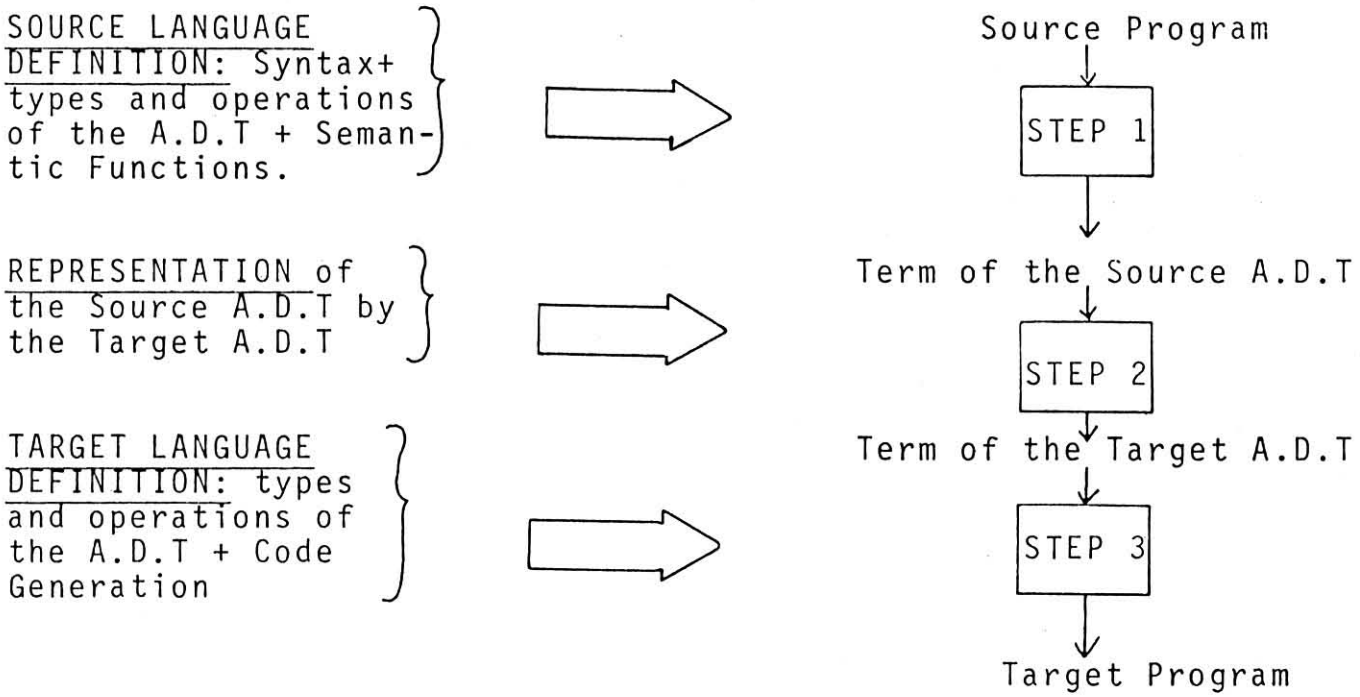
M.C. Gaudel (Iria, Rocquencourt)
C. Pair (CRIN, Nancy)

We present a formal semantics which is used to specify, produce and prove compilers. The basic idea is to use some results on the representation of algebraic abstract data types and the validation of such representations.

In the system currently developed , a definition of the Source

Language and of the Target Language is given. The same formalism is used to describe the semantics of these two languages. Then the translation process can be specified in a formal way, and proved. To each language is associated an algebraic abstract data type, i.e. a list of names of type; a list of names of operations with their domains and co-domain; a list of axioms to define the relationship between the operations. The semantic value of a program is a term of the abstract data type. The translation of the semantic values of Source programs to semantic values of Target programs can then be specified and proved as the representation of an abstract data type by another one.

The general scheme of the system is:



The translators obtained work in three steps. The axioms are used to perform proofs of the second step of translation.

The specific goal of this paper is to show, by the way of an example, what the semantics of a realistic programming language looks like in this formalism. We consider a usual Algol-like programming language which allows to declare and use simple variables, arrays and procedures. Simple and mutual recursivities can occur. Procedure parameters are called by value or called by variable.

The characteristic axioms of this language are stated and the semantic functions are given.

AN ABSTRACT NOTION OF ONE-DIMENSIONAL ARRAY WITH SOME APPLICATIONS ON FORESTS AND TREES.

C. Boehm (Roma)

The non adequacy of Church's tuple concept in λ-calculus as an abstract model for one-dimensional arrays is first illustrated. In fact there is no operation which independently of the 'nature' of the components, can compute the length of each array, or even select its first element; there is moreover no predicate distinguishing the empty array from the non-empty ones. It is probably true that because of such difficulties J.Mc Carthy decided twenty years ago to deviate from the pure λ-calculus by designing the programming language LISP.

Strangely enough, the preceding 'impossibilities' can all be eliminated by a different basic approach to the notion of one dimensional array: this choice enables one also to describe two important operations on arrays, the 'append' operation and the 'rotation to the right' by two non recursive operators.

Operators for handling errors may also be introduced. As a further application of the preceding notion to the case where atoms (nodes) are restricted to be non negative integers, it is possible to develop a consistent notion of ordered forest and tree, giving a semantic counterpart to LISP data structures.

PURE LISP IN LCF, AGAIN !

Malcolm C. Newey (Canberra)

Six years ago the author pushed the Stanford LCF system to its limits with experiments in the verification of some LISP programs. At the time of the workshop, one of these experiments was being repeated to gauge the effectiveness of improvements to LCF. This was the proof of correctness of some Pure LISP functions, notably the EVAL function given by McCarthy.

With an interpretive semantics completed and about 5o useful theorems about PureLISP proved (in the new system) it is concluded that the formal description of this language was more elegant and that deduction had proceeded at about three to four times the pace.

The most significant improvement in expressive power was found to be the much richer type structure - one that brings LCF users more

into line with other people operating on Scott foundations. In particular, the use of the new type building operators made the axiomatisation of the domain of S-expressions much cleaner. The talk uses this domain to illustrate a technique of domain definition using type equations that gives greater confidence in the consistency of the theory being developed.

Another advance in LCF which had a significant effect in this experiment was that of polymorphic types. A general theory of equality had already been developed for arbitrary domains and so the usual theorems about equality were available without cost. Furthermore, this and other uses of the theories feature impart much needed structure and modularity to the enterprise.

In developing the theory of Pure LISP, most time was spent proving theorems and so the most valuable advance , that Edinburgh LCF has contributed to the technology of machine checked proof, is the use of a special purpose programming language. The talk illustrates how, in using ML to code proof recipes, enormous amounts of human effort can be saved. So far, some proofs have been shortened by a factor of five (in terms of keystrokes). However, the theorems proved so far have been quite simple; even larger savings are expected on more complex proofs.

NON DETERMINISTIC COMPUTATIONS IN METRIC SPACES

A. Arnold & M. Nivat (Poitiers, Paris)

In the now standard theory of computation in an ordered domain one proves the equivalence between the definition of the computed function as the smallest fixed point of certain functional and the definition of the same function by means of terminating computation sequences of the program at a given point. This equivalence holds when the computation domain is a flat, or discrete, domain in which different defined values are incomparable : the only converging sequences whose terms are all equal, for sufficiently large n, to the limit of the sequence. In such a domain it is clear that any computed value is the result of some terminating computation sequence.

The situation is entirely different if, following D. Scott, one starts computing in a partially ordered domain which contains infinite ascending chains. A computed value may then be the lub of such a chain and as such can well be the result of no finite computation : a typical example is the domain of real numbers, if basic functions are the four arithmetic operations and the initial values are rational numbers, after any finite amount of time one will have computed only a rational number when the result may well be irrational.

We propose in this situation to give a meaning to successful infinite computation sequences which will be said to produce a result and to define the computed function by stating that its value at a given point is the set of results of both finite terminating and infinite successful computation sequences at that point. Obviously doing so one accepts the idea that a computed function is many valued since there is absolutely no reason why all computation sequences would lead to the same result. But indeed many-valued functions were already considered as the normal output of non deterministic programs.

Our point of view thus amounts to consider deterministic programs as special cases of non deterministic programs with the advantage that our result will hold in the general case of non deterministic programs (this was in fact the original motivation of the whole study).

In order to give a meaning to successful computation sequences we found extremely convenient to replace the order structure on the computation domain by a complete metric topology. (This is not at all to say that one cannot use the structure of a cpo to build a theory in many respects analogous to ours and indeed it has been done).

The results we get to are mainly conditions for the equivalence of this definition of the computed function and a mathematical definition by means of fixed point : it happens that in a very natural way one is lead to consider greatest fixed points rather than smallest. Intuitively this corresponds to the idea that, at the beginning of the computation we only know that the value of the computed function lies in a certain range, a priori the whole computation domain and in the course of the computation this range is reduced (may be to just one value but usually to a set of values). This is dual of the point of view expressed by Dana Scott that an a priori undefined initial value gets more and more defined in the course of the computation. We have borrowed for a large part this idea of decreasing range to L. Nolin (in a uncountable number of discussions).

In the course of our study we consider infinite trees for the following reason : algebraic infinite trees which can be generated by a recursive program scheme are at the basis of the theory called "algebraic semantics" of recursive programs.

The algebraic tree thus attached to a program scheme incorporates the whole semantics of the program in the sense that an interpretation being defined as a morphism, the function computed by the program resulting of the interpretation of the scheme is the morphic image of this algebraic infinite tree. Whence many results concerning classes of interpretation and families of computation domains.

Here infinite trees also play a role, in fact a crucial role. For the link between a semantics defined in an ordered structure and the semantics defined in a topological structure lies in the fact that the set of infinite trees $M^{\infty}(F, V)$ has both an ordered structure and a topological structure which are closely related (in fact an increasing function is order continuous iff it is continuous for the topology). The free complete F-magma $M^{\infty}(F, V)$ thus appears as the mother structure in which the phenomena of computation can be better described.

SOME THEORETICAL RESULTS CONCERNING PROLOG

K.R. Apt (Rotterdam)

A program in the programming language PROLOG is a set of definite clauses. A definite clause is a construct of the form $A \leftarrow B_1 \& \ldots \& B_n$ where $A, B_1, \ldots, B_n$ are atomic formulae and is to be understood as $\forall x_1, \ldots, x_k \ (B_1 \& \ldots \& B_n \supset A)$. A program is activated by a goal statement which is a construct of the form $\leftarrow A_1 \& \ldots \& A_n$ (understood as $\forall x_1, \ldots, x_\ell \ \neg (A_1 \& \ldots \& A_n)$).

From a goal

$$\leftarrow A_1 \& \ldots \& A_i \& \ldots \& A_n \qquad \text{where } n \geq 1,\ 1 \leq i \leq n$$

with selected atomic formula A_i one derives a goal

$$\leftarrow (A_1 \& \ldots \& A_{i-1} \& B_1 \& \ldots \& B_m \& A_{i+1} \& \ldots \& A_n)\,\Theta$$

for every clause $A \leftarrow B_1 \& \ldots \& B_m$ $(m \geq 0)$ such that A_i and A are unifiable with most general unifier Θ. A derivation (computation) successfully terminates if an empty clause is derived. Whether a computation successfully terminates depends on the selection rule and on the search algorithm used to search for an empty clause in the tree of goal statements.

The above can be seen as a special type of a resolution (called SLD-resolution). We prove the soundness and completeness of this resolution: a goal $\leftarrow N$ is a root of a tree containing an empty clause iff $S \cup \{\leftarrow N\}$ is inconsistent. Whether a tree with a goal $\leftarrow N$ as a root contains an empty clause is independent on the selection rule used.

A COMPLETE SET OF ASSERTIONS ON CONCURRENT SYSTEMS

Antoni Mazurkiewicz (Warsaw)

A simple concurrent system consisting of a finite number of sequential (nondeterministic) components is considered. Each component is built up from finitely many actions joined together by their input and output places. Two different components have disjoint sets of places; some actions, however, can be common for two (or more) components. The only synchronization between components is the requirement that an action, common for several components is to be performed by them coincidently (at the same time). This model is very close to the "path expression" model [Campbell, Lauer]. A configuration of the system is a vector of places, one place is taken from one component. The system can move from one configuration to another producing a history [Shields & Lauer 78] - a vector of strings, each string (an individual history) being a sequence of actions generated by one component.

We say that an assertion set for the system is given, if to each place a predicate (over histories) is assigned. An assertion set is said to be sound, if for any configuration, reachable from a given initial configuration, the conjunction of predicates to its places is true for corresponding histories. An assertion set is said to be complete, if whenever the conjunction of predicates assigned to a configuration is true for a vector of strings, this configuration is reachable from the initial one and the vector of strings forms a corresponding history.

It is shown that for any concurrent system as above a sound and complete assertion set can be constructed in such a way that the following requirements are satisfied:

(i) (1^{st} locality principle). Only local conditions need to be satisfied by predicates to form an assertion set; such a local condition concerns exclusively predicates assigned to places around an action.

(ii) (2^{nd} locality principle). Values of (history) variables in the scope of predicates assigned to places of a component are not changed when this component is suspended and other components are allowed to run.

Predicates constructed depend not only on individual history variables but also on "skew" history variables intended to keep so-called "told" and "heard" histories. These variables are the only ghost variables in predicates forming assertion sets.

NONDETERMINISTIC KAHN NETWORKS

David Park (Warwick)

Let Σ be a finite alphabet. $(K_\Sigma, \sqsubseteq)$ is the domain with elements $K_\Sigma = \Sigma^* \cup \Sigma^\omega$ and with the prefix ordering $u \sqsubseteq v \Leftrightarrow (\exists w)v = uw$. $(P(K_\Sigma), \subseteq)$ is the conventional power-set of K_Σ (not the power-domain). A nondeterministic Kahn network over Σ is a system of defining relations, of the form $W_i \in f_i(W_{i_1} \ldots . W_{i_{n_i}})$, $1 \leq i \leq N$, where the functions $f_i : K^{n_i} \to P(K_\Sigma)$ are associated with computing stations and the variables W_i stand for (histories of) communication lines. Such a sequence of relations may be used to specify an asynchronous computing network, with devices communicating through buffers. The deterministic case, that each f_i produces just singleton sets, was presented in [Kahn: IFIP 74 Proceedings 471-5].

There it was convincingly argued that the f_i must be continuous functions in the sense of Scott. The operational behaviour was then obtained abstractly as the least fixpoint μf, for the $f : K_\Sigma^N \to K_\Sigma^N$ got by combining the f_i. A similar analysis for the general case has long presented an open problem, most importantly when some f_i is a (nondeterministic) "fair merge" of sequences, an interleaving guaranteed on infinite sequences to absorb all of each sequence. This paper presents a solution, arguing in four stages:

1. Conceptual Problems: what is the nondeterministic analog of "continuity" ? Let D,E be bounded-complete ω-algebraic domains. Writing $w \leftarrow X$ for $(\exists x)[w<x \& x \in X]$ and using standard notions from [Plotkin: SIAM J. Comp.5, 452-487]

   Defn: $f : D \to P(D)$ is coherent iff (i) $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq_M f(y)$
   (ii) if $w \leftarrow f(x)$ then $(\exists v)[v<x \& w \leftarrow f(v)]$

   Theorem: f is coherent iff (i) above & (ii') if $u \in f(x)$, there exist
   $v_0<v_1<v_2< \ldots\ldots$, $w_0<w_1<w_2< \ldots\ldots$ with $\sqcup v_i = x$, $\sqcup w_i = u$ and $w_i \leftarrow f(v_i)$, all i.

Theorem: if all values of f are singletons then f is coherent iff $\bar{f}(x) = \{g(x)\}$, all x, for some continuous g.

Defn: the finite behaviour of $f : D \to P(E)$ is the relation $\{e_1 \ldots e_n\} \sqsubseteq_M f(d)$ between finite elements of D and finite sets of finite elements of E.

Theorem: There is a unique maximal coherent $\bar{f} : D \to P(E)$ with the finite behaviour of some coherent f, such that $\bar{f}(x) \supseteq f(x)$ for all f with that behaviour. $\bar{f}$ can be obtained from the unique continuous $f_M : D \to \mathcal{P}_M(E)$ (to the Plotkin powerdomain) determined by the behaviour, by taking $\bar{f}(x)$ as the maximal representative of the equivalence class $f_M(x)$, for each x.

2. Defining 'fair merge': Let $\mathcal{R}^3(K_\Sigma)$ be the Boolean algebra of 3-ary relations on K_Σ; $\nu X.\mathcal{F}(x)$, $\mu X.\mathcal{F}(x)$, for monotone $\mathcal{F}: \mathcal{R}^3(K_\Sigma) \to \mathcal{R}^3(K_\Sigma)$ are resp. the maximal and minimal fixpoint operators. The graph of 'fair merge' is:

$$\nu X.\ (\mu Y.(\mathcal{F}_L(Y) \cup \mathcal{F}_R(\mu Z.(\mathcal{F}_L(X) \cup \mathcal{F}_R(Z)))))$$

where
$$\mathcal{F}_L(W) = \{(\lambda, y, \lambda) \mid y \in K_\Sigma\} \cup \{(\sigma x, y, \sigma z) \mid \sigma \in \Sigma,\ (x,y,z) \in W\}$$
$$\mathcal{F}_R(W) = \{(y, \lambda, \lambda) \mid y \in K_\Sigma\} \cup \{(x, \sigma y, \sigma z) \mid \sigma \in \Sigma,\ (x,y,z) \in W\}$$

λ denoting the null string .

3. Negative length buffers: $W \in$ merge $(0^\omega, W)$ appears to have anomalous solutions, e.g. $W = w0^\omega$, any $w \in \Sigma^*$. The anomaly disappears on adopting a nonstandard operational notion - permitting any 'queue' on a communication line to take negative length, in which case it is to be taken as a sequence of predictions by the consumer of the producer's future behaviour, which must conform to them if the computation is to be permitted. The set of tuples satisfying the defining relations (in the sense obtained by writing '$\in$' for '$\in$') is shown to be the operational behaviour assuming this notion.

4. The hat trick: to restore the standard operational notion of buffer, one can adjust the functions f_i so that negative queues do not occur. Add a new symbol '$\wedge$' to Σ; $\hat{\Sigma} = \Sigma \cup \{\wedge\}$. If $w \in K_{\hat{\Sigma}}$, doff(w) is the sequence obtained by deleting '$\wedge$' from w. Given $f : K_\Sigma^n \to P(K_\Sigma)$ (coherent) a coherent $\hat{f} : K_{\hat{\Sigma}}^n \to P(K_{\hat{\Sigma}})$ can be otained satisfying:

(i) if each w_i is finite, $w \in \hat{f}(w_1 \ldots w_n)$, then length (w) > length(w_i)

(ii) if $w \in \hat{f}(w_1 \ldots w_n)$, then $\text{doff}(w) = f(\text{doff}(w_1) \ldots \text{doff}(w_n))$.

The intended behaviour, with standard operational notions, is then $\{(\mathrm{doff}(w_1)\ldots\mathrm{doff}(w_N))\,|\,w_i \in \hat{f}_i(w_{i1}\ldots w_{in_i})\}$, assuming each f_i is coherent.

[This device was suggested by Wadge].

AN EXERCISE IN PROOF RULE JUSTIFICATION

J.W. de Bakker (Amsterdam)

Abstract not received.

FAIR OR-TREES AND NON DETERMINISTIC PROGRAM SCHEMES

M. Nivat (Paris)

A non deterministic program scheme (ndrps)S is a system of defining equations S $\varphi_i(v_1,\ldots,v_{n_i}) = \tau_i$, $i = 1,\ldots,N$

where $\tau_i \in M(F \cup \Phi \cup \{\underline{or}\}, \{v_1,\ldots,v_{n_i}\})$

The choice operator $\underline{or}$ is interpreted operationally by giving the rules

$$\underline{or}\,(t_1,t_2) \to t_1 \qquad \underline{or}\,(t_1,t_2) \to t_2$$

which are applicable at all steps of a computation sequence.

The most natural idea to give a semantics of a ndrps S is to first look at the vector of trees $\vec{T} = \langle T_1,\ldots,T_N\rangle$ which are generated by S in $M^\infty(F \cup \{or\}, V)$ according to the standard theory of deterministic recursive program scheme. Afterwards one tries to break the $\underline{or}$'s that is to define a morphism Θ: $M^\infty(F \cup \{\underline{or}\}, V) \to \mathcal{P}(M^\infty(F, V))$ The morphism Θ is easily defined on finite trees by

$$\Theta(v) = \{v\}$$

$$\Theta(f(t_1,\ldots,t_n)) = f(\Theta(t_1),\ldots,\Theta(t_n))$$

$$\Theta(\underline{or}(t_1,t_2)) = \Theta(t_1) \cup \Theta(t_2)$$

And we try to extend it by continuity using on $M^\infty(F \cup \{\underline{or}\}, V)$ the topology induced by the Micielski-Taylor metric d and on $\mathcal{P}(M^\infty(F,V))$ the Hausdorff metric defined from d between closed subsets.

A necessary and sufficient condition to define $\Theta(T)$, where T is infinite is that

$\forall \varepsilon \ \exists \eta \ d(t,T) < \eta$ and $d(t',T) < \eta \Rightarrow d(\Theta(t),\Theta(t')) < \varepsilon$

We say that Θ has bounded oscillation at T.

It is known that if we define $\Theta(T)$, for all such T's, by

$$\Theta(T) = \lim \Theta(t_n) \text{ for all } t_n \text{ which converges towards } T$$

then Θ is continuous at T. Results from Arnold,Nivat can be applied namely the set of results of all computations of the ndprs S at $\varphi_1(v_1,\ldots,v_{n_1})$ equal to $\Theta(T_1)$ if Θ has bounded oscillation at T_1. We are lead to give conditions for Θ to have bounded oscillation at T and prove that this is true if T is a fair or-tree i.e. satisfies on all infinite branch of T, for all depth n, there exists an F-symbol of depth greater than n. In other words T does not contain an "infinite or-branch".

From this we easily derive that T_1 is fair if and only if S is a Greibach-scheme which is also the condition for the theorem on the equivalence of operational and greatest fixpoint semantics of Arnold,Nivat.

In conclusion:-the Greibach condition on S stems out from a necessary condition of continuity to extend the natural mapping of or-trees into sets of ordinary trees - the use of the metric topology allows us to write local continuity conditions which appear naturally in the theory.

MODULAR DENOTATIONAL SEMANTICS

Peter Mosses (Aarhus)

Denotational semantics is a fine tool for analysing and describing the fundamental concepts of programming languages. However, denotational semantics (DS) has not become popular for giving formal definitions of the meanings of "realistic" languages: the difficulty of reading and writing the definitions seems to increase much faster than the complexity of the languages. I consider that this is due to the lack of modularity in DS descriptions.

The proposed approach is based on the work of Burstall and Goguen: it is to structure DS by specifying a theory corresponding to each "concept" used. For example, the concept of a store (update, contents, new) can be described by one theory, and that of an environment or symbol-table (bind, find) by another theory. More significantly, the operation of sequencing "actions" can be described without prejudging the issue of whether jumps are to be allowed, thus one can avoid the whole semantic specification being permeated by the choice between the "direct" and "continuation" style.

One might "derive" all the individual theories from the Scott theory of domains. However, I would argue that the reader of a DS wants to

be told (at least some of) the laws which the primitive semantic operations obey. This, together with the desire to build directly on Burstall and Goguen's work on algebraic theories, encourages me to keep to equational algebraic specifications, defining semantic operators implicitly by equations expressing their influences on each other.

I would not claim that such implicit specifications are particularly easy to construct. What makes the effort worthwhile is that it should be possible to re-use (most of) the specifications time and time again, in describing different programming languages. I have worked out some algebraic theories which I believe to have this generality.

Once one has put together the specifications of such primitive semantic concepts as are necessary, what remains is to specify the main semantic functions. Here, I keep to the denotational rule: the meaning of a phrase of a programming language should be equated with some composition of the meanings of its subphrases. Thus, close contact with DS is maintained, and one can build on the experience gained in the last 1o years. In fact, I would like a standard DS to be an interpretation of the semantic theory.

Another hoped-for feature of the proposed approach is the ability to handle abstract syntax, context-sensitive constraints, syntactic proof-rules and various sorts of semantics just one formalism: that of (structured) algebraic theories. This, together with the conceptual simplicity of algebraic equations, might make formal specifications accessible to those who need them most: the designers, implementors and users of programming languages.

REASONING ABOUT FAIRNESS IN UNSCHEDULED ACTOR-SYSTEMS

A. Kerp, W. Oberdörster, P. Raulefs (Bonn)

A method for reasoning about fairness in the CSSA-actor model of computation is developed.

In this model, we only make very weak and general assumptions about concurrent computations in actor systems:

(1) Computation is done by a society of independent, concurrent sequential machines we call actors.

(2) Actors may communicate by sending messages to each other. An actor may only transmit a message to an actor it is acquainted with. Actor systems form communication networks having actors as nodes and acquaintance links as directed edges. Since actors

may be dynamically created or destroyed, and acquaintance links may be communicated among actors, the communication network formed by an actor may dynamically change throughout the course of a computation.

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

Although assumption (3) is quite realistic, it very much complicates reasoning about concurrent systems of sequential processes when compared to models assuming synchronized communication as a primitive. This is due to the fact that whenever messages are under way, there is no way of knowing anything about the order of their arrival at target actors, so we have to consider all possibilities. It is our intention to investigate concurrent actor systems under this extremely weak assumption first, and then determine in which way more restricted communication primitives reduce the complexity of the semantical model and the difficulty of reasoning about such systems.

Usually, methods are developed to synchronize a system of concurrent processes by adding schedulers to the system. Distributing the actions done by schedulers among individual processes leads to more efficient yet not necessarily less well-structured solutions. We consider the problem of reasoning about fairness without the presence of explicit schedulers.

Our approach consists in looking at particular combinators between actor systems that allow to form more complicated actor systems. The identification merge combinator maps two actor systems into a new one by identifying two actors of same type (= same actor script) in each system.

Based on a denotational semantics we develop a proof theory for static actor systems (no actors are created/destroyed throughout the course of a computation) that allows to establish fairness of such identification merges.

This method is illustrated at a particularly simple solution of the "dining philosophers"-problem: We present two actor scripts for forks and philosophers s.t. for any $n \geq 2$, a fair system of n philosophers and forks can be uniformly generated. The fairness proof consists of considering philosophers/two-forks - triangles, and (1) deriving the fairness of isolated triangles, followed by (2) deriving the fairness of the identification merge of two triangles, and (3) inferring the fairness of the entire system.

This example suggests a uniform proof method for showing fairness of "regularly" constructed actor systems which we call actor crystals.

A TOPOLOGICAL CLOSURE OF FREE X-CATEGORIES

Günter Hotz (Saarbrücken)

x-Categories were introduced in [Ho] 1965 to develop an algebraic calculus for representations of switching functions by boolean networks. The morphisms of free x-categories represent these networks. Here we generalize this theory to an algebraic calculus of infinite networks, which together with a functorial interpretation represent infinite functions. The infinite networks note how to compute the values of the functions for arguments by means of certain elementary operations. In the case of computable functions one may look at these networks as universal enfoldings of programs to compute the function.

Mathematically we construct a topological closure of the free x-categories. This closure is induced by a partial ordering $<$ of the morphism $\mathfrak{M}$ of the free x-category $\mathfrak{F}$. $<$ is defined by "derivations" which are generated by substitutions of variables of the generator set of $\mathfrak{F}$ by morphisms of $\mathfrak{F}$ with the same domain and codomain - in my lecture I restricted to substitution of the empty word. We get a T_0-space $\overline{\mathfrak{M}}$. The operations x and o can uniquely be defined on $\overline{\mathfrak{M}}$. We construct in this way a continuous x-category $\overline{\mathfrak{F}}$ which contains $\mathfrak{F}$ as a sub-x-category. The trace of the topology on the "terminal morphisms" seems to be a Hausdorff space.

Now one finds for explicit recursive equations solutions in $\overline{\mathfrak{F}}$. One can use the construction of Wolfgang Weidner given in his dissertation to construct an algebraic closure of $\mathfrak{F}$ in $\overline{\mathfrak{F}}$. His constructions use a finer topology as our's is. But this topology is too fine to construct a space in which all "enfoldings" are convergent.

Literature:

[Ho] G. Hotz: Eine Algebraisierung des Syntheseproblems von Schaltkreisen I und II, EIK Vol I, pp. 185-2o5, (2o9-231), 1965

[W] W. Weidner: Der algebraische und topologische Abschluß freier x-Kategorien, Dissertation an der Universität des Saarlandes, 1977.

REGULAR ALGEBRAS

Jerzy Tiuryn (Aachen)

The idea laying behind regular algebras is that when dealing with fixed point semantics of programming languages one never needs to

work with complete lattices or cpo's equipped with (directed-) continuous maps, but rather with structures where certain lub's exist, and with maps preserving these lub's. More to the point, there are structures (connected with semantics of non-deterministic procedures) with operations being not continuous but still having a very nice property allowing to solve an arbitrary finite system of fixed-point equations, built up from these operations, within ω steps of iterations. Briefly speaking, ordered algebras with this property are called regular algebras.

The lecture was a kind of a survey of results on regular algebras got by the author since 1975/76. The following topics were mentioned during the lecture:

1. The analogy with continuous algebras.
2. Examples of phenomena leading to non-continuous functions having the above-mentioned properties.
3. Free regular algebras and regular polynomials.
4. Varieties of regular algebras defined by equalities as well as by inequalities between regular polynomials.
5. Connections with rational algebraic theories.

ON *-ALGEBRAS

Klaus Indermark (Aachen)

According to their arities the operations on an algebra A form a heterogeneous (many-sorted) algebra: the derived algebra $D(A)$.

As this construction is reiterated in the algebraic analysis of functions defined by auxiliary recursion on higher functional types, one has to use troublesome indices for the carrier sets of $D^n(A)$. Here, we suggest a simple method to overcome that notational burden. Disregarding arities we allow an operation to have an arbitrary number of arguments. This is no restriction insofar as arities can be simulated by means of $\perp \in A$. On the other hand, many operations share this property, as e.g. projections, composition and resolution operators.

To be precise, let $\underline{Ops}(A) := \{f \mid f:A^* \to A\}$ for any set A.
If Ω is a set of operation symbols (without rank!) and $\varphi:\Omega \to \underline{Ops}(A)$, then $A := \langle A;\varphi \rangle$ is called an Ω-algebra. The class $\underline{Alg}_\Omega$ of Ω-algebras contains for each set X $F_\Omega(X)$, freely generated by X.
Similarly, if A is a cpo and Ops(A) is restricted to continuous operations, we get the subclass $\underline{Alg}^c_\Omega$ of complete Ω-algebras containing the free complete $F^c_\Omega(X)$ for any generating set X. As in the ranked case, the carrier sets $F_\Omega(X)$ and $F^c_\Omega(X)$ are representable by finite and in-

finite trees. However, labelling by operation symbols is unrestricted.

If $X = \{x_1, x_2, \ldots\}$, then any $t \in F^c_\Omega(X)$ represents an operation on each $A \in \underline{Alg}^c_\Omega$: the derived operation of t on A.
This derivation operator $\underline{derop}_A : F^c_\Omega(X) \to \underline{Ops}(A)$ is defined by

$\underline{derop}_A\,(t)\ (a_1..a_n) := \overline{[a_1..a_n, \perp]}\,(t)$ where $[a,\perp](x_i) := \underline{if}\ i \leq \underline{lg}(a)$ $\underline{then}\ a_i\ \underline{else}\ \perp$ and $\overline{}$ denotes the homomorphic extension.

Next, we consider rational schemes, since they play a central role in the analysis of higher type recursion and, moreover, they serve for studying iterative control structures.

Extending Ω to $R(\Omega) := \Omega \cup \{\Pi_i \mid i \geq 1\} \cup \{¢, \mathbb{R}\}$ we define

$Rat_\Omega := F_{R(\Omega)}$ as the algebra of rational Ω-schemes. For any interpretation $A \in \underline{Alg}^c_\Omega$ we construct $\underline{Ops}(A) := \langle \underline{Ops}(A); \ldots' \rangle \in Alg^c_{R(\Omega)}$

where F' means left-composition, Π_i' projection, ¢' composition and $\mathbb{R}'$ resolution of regular equations with parameters. The unique $R(\Omega)$-homomorphism $h_{Ops(A)} : Rat_\Omega \to Ops(A)$ describes the semantics of Rat_Ω with respect to A.
Then, we extend $F^c_\Omega(X)$ to $F^c_\Omega(X)^r \in \underline{Alg}^c_{R(\Omega)}$ such that $\underline{derop}_A$ becomes an $R(\Omega)$-homomorphism. Hence, the semantics $h_{Ops(A)}$ splits into

$Rat_\Omega \xrightarrow{\underline{tree}} F^c_\Omega(X)^r \xrightarrow{\underline{derop}_A} Ops(A)$. A rational tree $t \in \underline{tree}(\underline{Rat}_\Omega)$ represents an equivalence class of rational schemes. From their regular and equational normal forms we infer degrees measuring the iteration complexity of a rational tree. We briefly discuss some relationship to the star-height hierarchy of regular languages and the Kosaraju-hierarchy of repeat-exit schemes that has also been studied by G. Cousineau.

List of participants

ALBER, K. / Lehrstuhl für Informatik / Gaußstraße 12 / D-3300 Braunschweig

APT, D. / Kamer H 5-12 / Burgemeester Oudlaan 50 / Erasmus Universiteit / NL-Rotterdam

ARNOLD, A. / Université de Poitiers / Laboratoire d'Informatique / Bâtiment de Mathématiques / 40, av. du Recteur Pineau / F-86022 Poitiers-Cédex

de BAKKER, J. / Mathematisch Centrum / 2e Boerhaavestraat 49 / NL-1091 Amsterdam

BIBEL, W. / Institut für Informatik / TUM / Arcisstraße 21 / D-8000 München

BOEHM, C. / Istituto Matematico 'G. Castelnuovo' / Piazzale delle Scienze / I-00815 Roma

CLAUS, V. / Lehrstuhl für Informatik / Postfach 500 500 / D-4600 Dortmund 50

COURCELLE, B. / Université de Bordeaux 1 / Mathématiques-Informatique / 351 Cours de la Libération / F-33405 Talence-Cédex

COUSINEAU, G. / Institut de Programmation / Université Paris VI / 4, Place Jussieu - Cédex 05 / F-75230 Paris

CREMERS, A. / Lehrstuhl für Informatik / Postfach 500500 / D-4600 Dortmund 50

DAMM, W. / Lehrstuhl für Informatik II / Technische Hochschule / Büchel 29-31 / D-5100 Aachen

EHRICH, H. / Lehrstuhl für Informatik / Postfach 500500 / D-4600 Dortmund 50

FEHR, E. / Lehrstuhl für Informatik II / Technische Hochschule / Büchel 29-31 / D-5100 Aachen

GAUDEL, M. / IRIA Domaine de Voluceau / Rocquencourt / B.P. 105 / F-78150 Le Chesnay

GUESSARIAN, I. / Laboratoire d'Inform. Théorique et Programmation / Université Paris 7 / Tour 45-55 / 2, place Jussieu / F-75005 Paris

HOTZ, G. / Fachbereich 10 - Informatik / Universität des Saarlandes / D-6600 Saarbrücken

INDERMARK, K. / Lehrstuhl für Informatik II / Technische Hochschule / Büchel 29-31 / D-5100 Aachen

KRÖGER, F. / Institut für Informatik / TUM / Arcisstraße 21 / D-8000 München 2

LANGMAACK, H. / Institut für Informatik / Christian-Albrecht-Universität / Olshausenstraße 4o-6o / 23oo Kiel 1

LEHMANN, S. / Fachbereich 1o - Informatik / Universität des Saarlandes / D-66oo Saarbrücken

LOECKX, J. / Fachbereich 1o - Informatik / Universität des Saarlandes / D-66oo Saarbrücken

MAZURKIEWICZ, A. / Computer Centre / Polish Academy of Sciences PKIN, P.O. Box 22 / PL-00.9o1 Warszawa

MILNER, R. / Computer Science Department / Edinburgh University / Mayfield Road / GB-Edinburgh EH 93 JZ

MOSSES, P. / Dept. of Computer Science / University of Aarhus / DK-Aarhus

NEWEY, M. / Computer Center / Australian National University / Canberra Act/ Australia 26oo

NIVAT, M. / Université Paris 7 / Tour 45 - 55 / 2, place Jussieu / F-75oo5 Paris

OLNHOFF, T. / Institut für Informatik / Azenbergstraße 12 / D-7ooo Stuttgart 1

PARK, D. / University of Warwick Coventry / GB-Warwickshire CV 47 AL

RAOULT, J. / Université de Paris Sud-Orsay / L.R.I. / Bâtiment 49o F-914o5 Orsay Cedex

RAULEFS, P. / Institut für Informatik / Wegelerstraße 6 / D-53oo Bonn

REISSIG, W. / Lehrstuhl für Informatik II / Technische Hochschule / Büchel 29-31

de ROEVER, W. / University of Utrecht / Dept. of Computer Science / Budapestlaan 8, P.B. 8oo12 / NL-35o8 TA Utrecht

SIEBER, K. / Fachbereich 1o - Informatik / Universität des Saarlandes / D-66oo Saarbrücken

TIURYN, J./Lehrstuhl für Informatik II / Technische Hochschule / Büchel 29-31 / D-51oo Aachen

WADSWORTH, C. / Dept. of Computer Science / King's Building / University of Edinburgh / GB-Edinburgh EH 9 3 JZ