Jean-Pierre Finance, Stefan Jähnichen,
Jacques Loeckx, Martin Wirsing (editors)

Logical Theory for Program Construction

Dagstuhl-Seminar-Report; 7
25.2. - 1.3.1991 (9109)

ISSN 0940-1121

Copyright © 1991 by IBFI GmbH, Schloß Dagstuhl, W-6648 Wadern, Germany
Tel.: +49-6871 - 2458
Fax: +49-6871 - 5942

Das Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI) ist eine gemeinnützige GmbH. Sie veranstaltet regelmäßig wissenschaftliche Seminare, welche nach Antrag der Tagungsleiter und Begutachtung durch das wissenschaftliche Direktorium mit persönlich eingeladenen Gästen durchgeführt werden.

Verantwortlich für das Programm:
Prof. Dr.-Ing. José Encarnaçao,
Prof. Dr. Winfried Görke,
Prof. Dr. Theo Härder,
Dr. Michael Laska,
Prof. Dr. Thomas Lengauer,
Prof. Ph. D. Walter Tichy,
Prof. Dr. Reinhard Wilhelm (wissenschaftlicher Direktor).

Gesellschafter: Universität des Saarlandes,
Universität Kaiserslautern,
Universität Karlsruhe,
Gesellschaft für Informatik e.V., Bonn

Träger: Die Bundesländer Saarland und Rheinland Pfalz.

Bezugsadresse: Geschäftsstelle Schloß Dagstuhl
Informatik, Bau 36
Universität des Saarlandes
W - 6600 Saarbrücken
Germany
Tel.: +49 -681 - 302 - 4396
Fax: +49 -681 - 302 - 4397
e-mail: dagstuhl@dag.uni-sb.de

Introduction

Today, many different approaches towards the use of formal methods in program construction exist, each having its favorite application domains, its advantages, and its drawbacks.

The aim of this workshop was to bring together experts to compare and evaluate their approaches and methods, especially with respect to their suitability for computer support. Great emphasis was put on discussions between the different approaches.

The workshop provided a forum for researchers and developers to gain awareness of current practical and experimental work across the breadth of the field.

Among the topics encompassed were:

- Specification languages and methods
- Algebraic algorithmic calculi
- Program development based on type-theory
- Program development based on logical theorem proving
- Formalization of methods in meta-calculi

Although, as intended, a broad variety of topics was covered, including e.g. non-monotonic reasoning as well as formal treatment of software reuse, some main issues can be identified:

- constructive languages and calculi related to Martin-Löf's type theory,
- formal semantics of algebraic and axiomatic specification languages,
- the combinatorial approach to functional program development related to the Bird-Meertens formalism.

We have the feeling that the workshop gave an actual overview of at least the European efforts in the field of formal program development and that it served as a source of mutual encouragement and suggestions.

The Organizers
Jean-Pierre Finance Stefan Jähnichen Jacques Loeckx Martin Wirsing

A Fine-Grain Approach to Sorted Logic

Jochen Burghardt

GMD Forschungsstelle Karlsruhe

The talk introduced a new constructor-based sort discipline on predicate logic that allows to express more sophisticated sorts than conventional approaches. Algorithms were given to calculate infimum and difference of two sorts, and to decide the inhabitance, the subsort, and the sort equivalence property. The range sort of a defined function (as opposed to a constructor function) is calculated from its defining equations using a mechanism of "sort rewriting". This leads to a more exact description than obtaining a signature from the user: different equations of the same function are usually assigned with different sorts. Thus, the sort discipline helps to select the "right" equations for inference steps (like narrowing), and cuts down the search space of a formal proof. This effect was demonstrated using an example from the area of data refinement and implementation proofs.

The Specification Language SPECTRUM

Franz Regensburger

Technische Universität München

This talk presented the general philosophy of the specification approach of the Munich project SPECTRUM. SPECTRUM is a language for predicate logic, and the specification development is completely performed in a predicate calculus based on natural deduction.

A framework for the development of specifications has been sketched. The general design decisions for the model semantics and the syntax of the abstract language have been outlined.

An interesting decision for the syntax is that formulae are defined in a way such that they are a proper subset of the terms of sort *Bool*. The property of a Boolean term to be also a formula may be decided by pure syntactic calculation, namely by attributation techniques.

The advantage of this function is that big formulae can be manipulated in an equational style as well as in the usual junctor-based natural deduction style.

A Refinement Case Study

Jean-Raymond Abrial

Paris

In this talk, I developed with great details a classical little example (the longest upsequence algorithm) of refinement from initial specification down to final code. I insisted on a few methodological points among which are the following:

- the importance of a sound mathematical preamble,
- the systematic usage of data refinement steps based on clear and intuitive technical decisions,
- the reusability of already specified and refined pieces of code.

The exercise is conducted using an homogeneous notational style based on abstract machines and generalized substitutions.

The Role of Programming Logics in Formal Program Construction

Werner Stephan

Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, D-7500 Karlsruhe, West Germany

Programming Logics are fundamental constituents of systems for logic-based program construction. In such a system we do not only generate a program, but also provide a formal proof of its correctness. In order to combine safety with flexibility the underlying logic should be expressive enough to formalize all interesting properties of programs and should allow many proof styles.

From a more technical point of view it is agreed that a suitable programming logic should represent programs in a form that is close to the original syntax and that the declarative system should allow domain independent reasoning.

After a brief discussion of two other well known programming logics an axiomatisation of recursive procedures (of any finite type) within the framework of Dynamic Logic is presented.

To make domain independent reasoning possible there are strong axioms for assignements and quantifiers (following Goldblatt's approach) and induction schemes for counters and environments. Counters and environments are auxiliary data structures for the axiomatisation of while loops and recursive procedures. They are kept separate from the data structures the programs compute on.

This axiomatisation is used as the logical basis of the Karlsruhe Interactive Verifier (KIV) which is a logic-based shell for program verification and program development.

Integrating Various Approaches to Program Synthesis Using Dynamic Logic

Maritta Heisel

Universität Karlsruhe

The current situation in program synthesis can be characterized by the fact that there are several different approaches to this task which, in general, are incompatible. An integration of diffferent approaches has the advantage that the strong points of the isolated methods can be preserved, whereas their weaknesses can be eliminated to a large extent.

An integrated open synthesis system is presented. This system was designed and implemented within the logical framework of the Karlsruhe Interactive Verifier (KIV). This is a system for formal reasoning on imperative programs which uses the principle of tactical theorem proving.

The first step of the design was to formalize and implement various approaches to program synthesis known from the literature. This resulted in five separate strategies for program synthesis which are all available in the KIV environment.

In a second step, these strategies are integrated to form a homogeneous system. This system is characterized by the following technical features: (i) Programs are developed top-down. (ii) The first part of a compound statement should be developed first resulting in the advantage that more information is available for the development of the second part of the compound. (iii) As a necessary prerequisite for automation, non-logical information is made available to the system in addition to the program specification.

Apart from these technical points, the integrated system supports program development according to the following paradigm: The postcondition usually is represented as a conjunction. Then compound statements are developed to establish the parts

of the postcondition. In this process, the dependencies between the subgoals must be taken into account. Knowledge about the problem domain is incorporated by replacing the postcondition by a stronger one.

The integrated system is (i) formal, (ii) machine supported, (iii) abstract, because the logical rules of the system correspond to high-level design decisions, (iv) automatable, and (v) flexible, because it is designed as an open system: adding new rules is a routine activity.

References:

Formalizing ans Implementing Gries' Program Development Method in Dynamic Logic; to appear in Science of Computer Programming

Formal Program Development by Goal Splitting and Backward Loop Formation; Technical Report 32/90; Univ. Karlsruhe, Fak. für Informatik

The Complexity of Proving Program Correctness

Hardi Hungar

Universität Oldenburg

Everybody who has tried to prove the correctness of a program with respect to a formal specification knows that this is difficult even if the data domain is finite. It is already hard for while-programs, and it is even harder for programs with recursive procedures.

However, how hard is really? And what is the source of the difficulties? Are Hoare-style proof-systems adequate tools, or do they make the problem harder?

We answer these questions for partial correctness assertions about programs from different languages. We characterize the the complexities of spectra of partial correctness assertions (that is: of sets of those finite interpretations where an assertion is valid) for various programming languages. Then we show that there are not only decision procedures (of the determined complexities) but also proof constructing procedures, i.e., Turing-machines which do not only decide whether the (fixed) assertion is valid in the interpretation under investigation, but which also construct a proof in a Hoare-style system if it is valid.

The languages we considered include Clarke's language L4 which turns out to have a very hard (to decide) partial correctness theory (hyperexponential).

The main results are that Hoare-style systems are adequate, but that program correctness for complicated languages may be too hard to be handled in practice (hyperexponential).

Virtual Data Structures

S.D. Swierstra

Utrecht University, The Netherlands

In many algorithm first a data structure is constructed, which is then inspected at a later stage of the program. It was shown how in certain situations such intermediate data structures may be removed from the program using program transformation techniques. A calculational derivation was presented for an algorithm solving the "longest low segment" problem, where a segment is "low" when its maximum value is smaller than its length ($\uparrow / <^{\text{``}} \#$). The derivation was given using the Bird-Meertens Formalism.

Crucial design steps were:

- the choice of an appropriate generator for segments
- the refinement of the non-deterministic generator in order to treat all segments with the same maximum element at the same time.

The resulting algorithm runs in $O(n)$ time. Finally it was shown how a large class of similar problems could be solved by choosing appropriate substitutions for a set of operators characterizing the problem.

Calculating Programs by Equational Reasoning

Lambert Meertens

CWI Amsterdam & Utrecht University

Whereas we have formal languages for expressing algorithms in great detail – namely programming languages – no suitable formalisms exist when it comes to explaining how such solutions arise from the original problem. Common techniques in explaining algorithms are handwaving, using a mixture of pseudo-formal English and Pidgin ALGOL, and drawing pictures with snapshots of the algorithm in action.

Work has been in progress for about 15 years now to design a formalism for deriving programs by *calculation*, suitable for use in e.g. textbooks and the class room. The approach is to study diverse problems, both well-known ones and new ones, in order to identify the crucial algorithmic concepts involved, to devise notation, to formulate the algorithmic laws that hold, and, while doing so to build up a corpus of theories.

In the calculational style aimed at, equational reasoning is the basic mode: a sequence of expressions chained with (usually) the sign "=", in which each step is easily justified by appeal to one of the laws involving no more than pattern matching and substitution. The expressions can, e.g. be (a composition of) functional expressions, having a "mathematical" non-operational reading but usually also having computational content.

Basic equational laws can be obtained from the "unique properties" characterizing categorical limit and co-limit constructions. E.g., in the category of F-algebras (F an endofunctor on set, e.g.) the initial algebra in has a unique arrow (homomorphism) to any F-algebra, often denoted in diagrams with a dotted arrow. Leaving out type information, the unique property can be formulated as: $\forall\varphi :: \exists! h :: in; h = hF; \varphi$. This formulation is unsuitable for calculation. By choosing a *notation* for the unique arrow which depends functionally on φ, e.g. $(\!| \varphi |\!)$ we can reformulate: $h = (\!| \varphi |\!) \equiv \; in; h = hF; \varphi$. It thereby becomes possible to derive a diversity of non-trivial programs by straightforward calculation.

Calculation by Computer

Roland Backhouse

Eindhoven Technical University

A system (implemented by Paul Chisholm) providing support for interactive proof construction was briefly introduced and later demonstrated. The main concern is calculational (or transformational) style proof development, but a form of bottom-up natural deduction proof is also supported. The principles upon which the system is based are that it be flexible and easy to use, that proof is viewed as a syntactic editing process, and that the user decides the level of detail of a proof. Consequently, a traditional constraint of proof editors – that proofs be machine checkable – is not

enforced.

A Relational Theory of Datatypes

Roland Backhouse

Eindhoven Technical University

Research was reported into a theory of datatypes based on the calculus of relations. A fundamental concept is the notion of "relator" which is an adaptation of the categorical notion of functor. Axiomatisations of polynomial relators (that is, relators built from the unit type and the disjoint sum and cartesian product relators) are given, following which the general class of initial datatypes is studied. Among the topics discussed are natural polymorphism, junctivity, and continuity properties.

Relations as a Program Development Language

Bernhard Möller

University of Augsburg

We use relations as elements of a language in which to specify and develop programs. The main emphasis is on algebraic laws for the language constructs which are to be used in transforming specifications into efficient programs. Our approach is characterized by the following particularities:

1. We use relations of arbitrary arities. Relations of arities $\geqslant 2$ are used as non-deterministic functions with tuples as arguments and results. Unary relations, i.e. sets of singleton tuples or, equivalently, of single elements, correspond to types. The two nullary relations (the one consisting of the empty tuple and the empty one) play the rôle of Boolean values. This also allows easy definitions of assertions and conditionals.

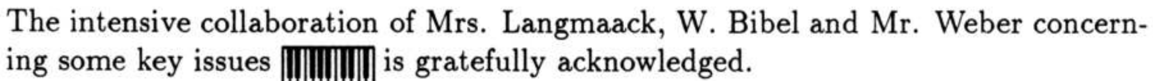
2. Relations may be of higher order, i.e. contain other relations as tuple components. This also allows parameterized and dependent types.

3. We allow nested tuples as elements of relation.

Essential operations on relations are (besides union, intersection, and difference) junction and join. Junction encompasses concatenation and composition; as special cases we obtain image and reverse image as well as tests for emptiness, membership, intersection, and equality. Special cases of the join yield restriction.

All operations are monotonic wrt. inclusion. Hence, the Knaster-Tarski fixpoint theorem provides a semantics for recursive definitions of relations, in particular of types. By (3) we obtain general tree-like data types in this way. The principle of computational induction allows proofs about recursively defined types or relations.

The algebraic properties of the operators are illustrated with the derivations of a simple reachability algorithm from its specification.

The intensive collaboration of Mrs. Langmaack, W. Bibel and Mr. Weber concerning some key issues  is gratefully acknowledged.

Time Analysis, Cost Equivalence and Program Refinement

D. Sands

Imperial College, London SW7

Techniques for reasoning about extensional properties of functional programs are well-understood, but methods for analysing the underlying intensional, or operational properties have been much neglected. This talk presents the development of a simple but practically useful calculus for time analysis of non-strict functional programs with lazy lists.

An operational model is used to induce a set of equations which form the basis of a calculus for reasoning about time cost. However, in order to buy-back some equational properties lacking from this calculus, we develop a non-standard notion of operational equivalence, cost equivalence. By considering time cost as an "observable" component of the evaluation process, we define this relation by analogy with Park's definition of bisimulation in CCS. This formulation allows us to show that cost equivalence is a contextual congruence (and therefore substitutive w.r.t. the calculus) and provides a uniform method for establishing cost-equivalence laws. Cost equivalence is interesting in its own right a similar notion of program refinement arises naturally, and implications for program transformation are briefly considered.

Coping With Requirement Freedoms

Martin S. Feather

USC/Information Sciences Institute, Marina Del Rey, California, USA

Formal methods for software development employing formal specifications are being used to good effect in a number of real-world situations. Two key factors that impede the even more extensive application of these methods are the difficulty of manipulating formal specifications, and the difficulty of constructing specifications. Manipulation has been widely studied (verification, analysis and program transformation research); construction has received less attention.

We argue that there may be a wide gap between the natural statement of a task's requirements and a formal specification of the same. To understand this gap, we identify "freedoms" that requirements typically exhibit, but which specifications cannot tolerate (e.g. inconsistency, incompleteness). We also consider the processes that are applied to construct and use formal specifications. Comparing the freedoms against the processes, we determine the capabilities required of those processes.

We than sketch some small steps toward these ends:

- Specification construction by incremental elaboration in several (mostly) independent directions (using "evolution transformations" to perform the elaboration steps), followed by comparisons to identify and resolve interdependencies, and combination to achieve an all-inclusive final specification.
- "Idealized" description of a task from several different points of view, followed by negotiation and compromise to resolve the inconsistencies.

Both of these illustrate the wealth of opportunity that exists for techniques, tools and methods to support specification construction by coping with requirement freedoms.

(Joint Work with Stephen Fickas of the University of Oregon, Eugene)

The IO-Graph-Method

Gerd Neugebauer

TH Darmstadt, West Germany

Two essential subproblems of automatically generating programs from vage ideas are the algorithm design (program synthesis) and algorithm implementation (program transformation). The IO-Graph Method originally was developed to solve the problem of algorithm implementation. To do so it incorporates the sequentially execution of the desired program together with the usage of moded variables, i.e. in most real programs variables are used either as inputs or as outputs. As natural consequence some properties of variables have to be taken into account, e.g. variables can not be taken as input before they are bound to values.

In addition to the given problem description - in first order logic - the description of the executable predicates has to be available. This information is stored in the so-called environment. Such considerations led to an algorithm to determine all executable orderings of literals in the special case of a single clause.

This method of algorithm implementation is then combined with a method for algorithm design - the LOPS approach.The combination is demonstrated with the square example, i.e. the relation between two natural numbers x, y is given by $y = x^2$. For this specification different distributions of input and output variables are taken into account. Different algorithms are obtained by exploiting the modes and the choices which can be taken during the algorithm design phase.

Proof and Program Transformation in Type Theory: Some Remarks

Didier Galmiche

C.R.I.N, Universitè Nancy I, Nancy, France

This talk aims at presenting the notion of proof and program transformation in type theory and more precisely in programming with proofs frameworks. In such frameworks, programs obtained from proofs (by extraction) are not always efficient and the relationship between programs and proofs has to be studied through construction and transformation steps.

Considering constructive type theory, we study transformation of programs through proofs using a λ-abstraction generalization strategy and connections with data structure transformers with a view to deriving more interesting programs (from an efficiency point of view).

Using the Calculus of Constructions in Order to Synthetise Correct Programs

Benjamin Werner

INRIA - Rocquencourt, (Project FORMEL) France

The Calculus of Constructions, a polymorphic λ-calculus with dependent types, is a by-now well known formalism. Its expressiveness, as well for algorithms as for logical/mathematical reasoning, makes it a reasonable base for program development, where the specifications and their proofs are written in the same formalism.

The idea is to define a notion of realisability between the specifications (i.e. types of the CoC of the form $\forall x : T \exists y : T' \;\; P(x, y)$) and a target language which corresponds approximately to the functional kernel of ML. An extraction algorithm maps proofs of specifications to programs of this language and erases the non-computational parts of the proof.

It is possible to use extensions of the programming language by proving that they can be viewed as realisations of axioms in CoC. In that way, it is possible (resp. should soon be possible) to produce programs using machine integers, general recursion, control structures, etc.

Several examples have been developed, and encourargingly reasonable programs obtained. Recent progress in proof synthesis allows us to hope that the user comfort of the system should soon be enhanced.

The theoretical background of this talk is the work of Christine Paulin-Mohring.

A Survey of the Categorical Semantics of Dependent and Polymorphic Types

Thomas Streicher

Fakultät Mathematik Informatik, Universität Passau, D-8390 Passau, West Germany

We define in a modular way the categorical semantics of type theoretic concepts as appearing in LF, AUTOMATH, Calculus of Constructions, Martin-Löf Type Theory etc. That means for any type-theoretic concept we give a corresponding categorical

condition.

Finally we discuss several dependence and independence results.

Independence results can be proven by changing the choice of propositional types in the model of realisability sets.

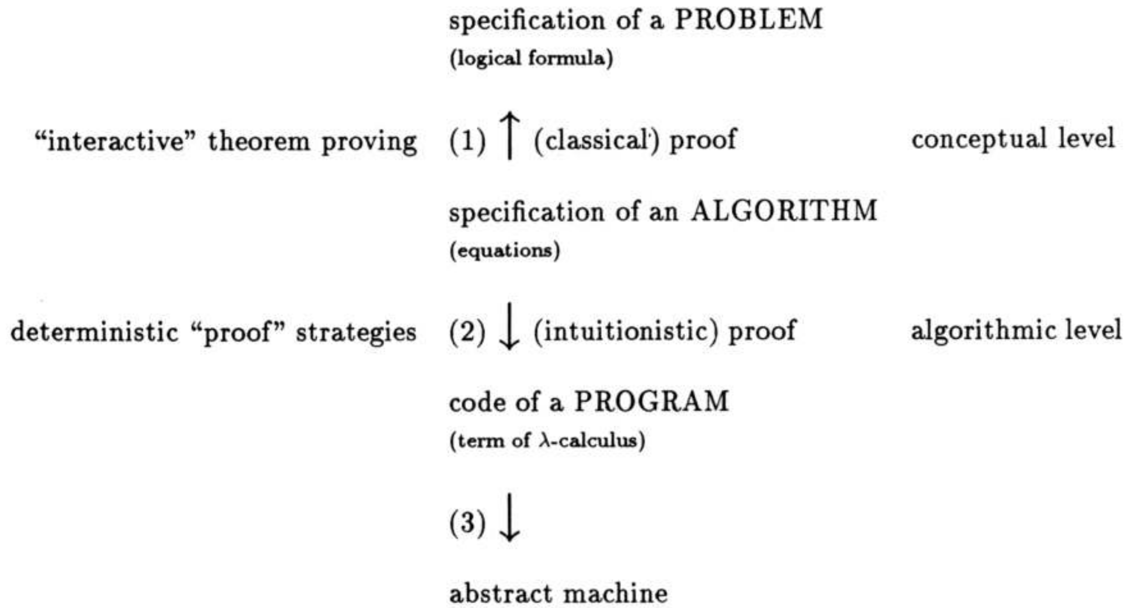
Propre: An Experimental Language for Programming with Proofs

Michel Parigot

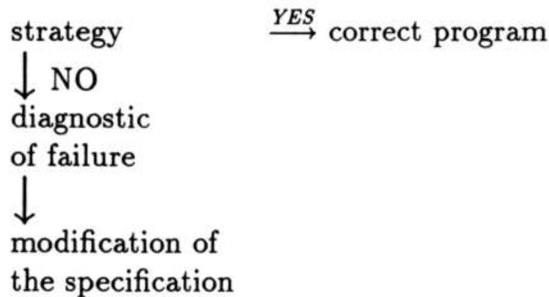
CNRS, Université Paris 7, France

The main originality of the system - as a so called "Theory of Types" - is probably to be based on usual logic, with usual syntax and usual semantics! But there are some mathematics behind in order to justify the approach. The system is actually in a "primitive" state and will be enriched using a "call by need" methodology. Implementation is only partly done.

As well-known the extraction of programs from constructive proofs of high-level specifications of problems allow to build correct programs. But it leads to a basic difficulty: how to recognize at the level of proofs the corresponding algorithms (an algorithm is not only a function, it represents a particular way of computing which is usually not specified in the problem). In PROPRE the specification of algorithms is considered as a basic part of the system. The general scheme of the language is the following:



- At level 2 one needs a strict control on the proof (as representing a way of computing) and therefore uses deterministic strategies to prove theorems and extract programs. A deterministic strategy is a deterministic algorithm which either produces a proof of a predefined form (and thus a program) or given a diagnostic of failure (with respect to the predefined form) suggesting modifications of the specification. This leads to a notion of interactive correctness:

  Many deterministic strategies have been designed and implemented by Pascal MANOURY and Marianne SIMONOT.

- At level 1 no control is really needed and even classical proofs are allowed. The idea is to confront, in many directions, specifications of algorithms and specifications of problems. We plan to use at this level different existing theorem provers, but nothing has been yet implemented.

- At level 3 an abstract machine based on a mathematical model has been implemented by Christophe RAFFALLI.

An Overview of another logical framework

Bengt Nordström

Chalmers & University of Göteborg, Sweden

The idea behind a logical framework is to have a formal system in which it is possible to express many different theories. We can look at Martin-Löf's logical framework as a small programming language (it has only two ways of constructing programs) with a simple but rich type structure.

The ground types are *Set* (also called *Prop*) and $El(A)$ (if $A \in Set$). The objects in *Set* are (monomorphic) sets and the objects in $El(A)$ are the elements of A (i.e. objects whose value is a canonical element in A). If A is a type and B a family of types indexed by $x \in A$ then $(x \in A)B$ is a type whose objects are functions f such that $f(a) \in B[a/x]$ whenever $a \in A$.

A program consists of:

declaration of constants
expression

where the declaration of constants is a description of a theory (primitive constants used to express formation and introduction rules, implicitly defined constants expresses elimination rules with the step between the definiendum to the definiens being contraction, explicitly defined constants used to express lemmas or derived rules).

Abstract Data Types in the Polymorphic λ-Calculus

Kurt Sieber

University Saarbrücken, West Germany

The ultimate goal of our work is to develop the foundations of program verification for a "realistic" powerful programming language. Such a programming language should in particular contain a module concept, so that large programs can be built from samll reusable units (and their correctness can be derived from the specifications of these units).

We have developed our own programming language for this purpose. Its type system is essentially the same as for Girard's system F_ω, but we have added several features which are useful in a real programming language like recursive types, imperative features, exception handling, ... Following Mitchell/Plotkin's idea that "abstract types have existential type", modules are given a type and are thus first class objects. We have extended this idea to parametrized abstract data types: They can be considered as functors (= functions mapping modules to modules) of higher order existential types (with $\exists$-quantification over type constructors instead of types). This view allows us to program with parametrized abstract datatypes in a very flexible way. An example which illustrates this flexibility: The parametrized ADT "Set" can be used to implement the parametrized ADT "Nested Set" (nested in the same sense as LISP-lists).

Current state of our work: The programming language is implemented. A formal description will be available in some months. Currently we are taking the first steps towards program verification.

A Meta-Calculus for Formal System Development

Matthias Weber

Universität Karlsruhe, West Germany

This talk presents the definition and basic properties of the DEVA meta-calculus, a language designed for the formalisation of development methods and developments. The design of DEVA was influenced by results on the development of logical frameworks such as ELF, CoC, or the languages investigated in the AUTOMATH project.

The definition of DEVA is presented in five pieces starting from the following two approximations:

- simple typed λ-calculus
- λ-calculus with hierarchical, λ-structured types

The five pieces of DEVA gradually introduce operations to structure developments and methods.

Further, an implicit level is defined in which it is allowed to omit components of developments.

Finally, the talk has presented basic properties of DEVA:

- Church-Rosser property for closed expressions.
- Closure of type-validity against reduction and validity.
- Strong normalisation for type-valid expressions.
- Decidability of type-validity.

An experiment in Formal Software Development: Using the B Theorem Prover on a VDM Case study

Yves Ledru

Université Catholique de Louvain, Louvain-la-Neuve, Belgium

The B tool is generic theorem prover developed by J.R. Abrial. This talk has presented how the prover has been instantiated to the logic of VDM (Vienna Development Method). The result of this instantiation is an environment for the formal proof of VDM developments. It has been applied succesfully on several case studies. The nature of the support provided by this environment is two-fold: the validity of the formal development is ensured but also automatic generation of the VDM proof obligations is provided. These proof obligations are then discharged as ordinary theorems with the assistance of the tool.

One of the by-products of this work is a better understanding of the formal method and in particular, the macroscopical structure of developments and the hidden proof rules of the method.

A full paper on this subject has been published on the proceedings of the 12 th International Conference on Software Engineering (Nice, 1990, IEEE publisher).

Specifying Systems of Structured Polymorphic Specifications

Martin Wirsing

Universität Passau & Forwiss, Bayern, West Germany

Starting from an algebraic specification approach a uniform framework for specifying data structures and configurations of systems was presented. The framework covers flat algebraic specifications with higher-order functions and shallow polymorphism. Structured specifications are introduced with the help of three specification operations for which semantics, a "module algebra" and modular proof systems are developed. The concept of dependent types is used to define parameterised specifications and classes of specification schemata, so-called polymorthic specifications. Due to the introduction of a second level, configurations of modules can be abstractly specified; relationships between different classes of objects such as specifications, signatures or parameterised specifications can be described.

(Joint work with Jacek Leszczytowski, Polish academy of Science, Warsaw)

Inheritance with exceptions - On the meaning of "but"

Pierre-Yves Schobbens

U.C.Louvain, Belgium

We show here why exceptions to a general rule can improve the expressive power of loose algebraic languages (such as Wirsing (1986)), allowing to better capture informal requirements and better explain formal specifications. We propose to this end a new (non-monotonic) logical connective, "but", which links two (first-order) formular (or specification modules), the first one being the general rule, the second one exceptions that takes precedence over the general rule. We indicate that a model-theoretic definition of this connective offers several advantages:

- abstraction from the syntax;
- easy integration in model theoretic specification languages (Wirsing 1986, Brauer 1985)

• enforced consistency (contrast with Reiter (1980), KL-ONE, ...).

Such a definition is proposed, or top of "correspondences" (a variant of homomorphisms), which can be sketched as choosing the models of the exceptions that are the "closest" to the set of models of the general rule. A simple algorithm is given for solving S *but* $E \models \Phi$ in the conjunctive sublanguage - which essentially first looks for an answer in E, and in S only if E fits to answer the question.

The audience is invited to look for further properties of this connective.

The Algebraic Module Specification and Interconnection Language ACT TWO

Werner Fey

Technische Universität Berlin, West Germany

The specification language ACT TWO - as successor of ACT ONE - is based on algebraic module specifications, (parameterized) initial specifications and requirements specifications with their interconnection mechanisms renaming, extension, union and actualization. As a kernel language ACT TWO is intented for the horizontal structuring of modular systems mainly on the design level.

The functional semantics of ACT TWO is defined on two levels: the presentation and the model level. Semantical context conditions define the desired sublanguage of correct and consistent specifications with compositional semantics. The language definition of ACT TWO is given in [Fey 88]. The theoretical foundation of the underlying specification concepts and interconnection mechanisms can also be found in [EM 85, 90].

[Fey 88] - W. Fey, Pragmatics, Concepts, Syntax, Semantics, and Correctness Notions of ACT TWO: An Algebraic Module Specification and Interconnection Language. Techn. Report No 1988/26, TU Berlin, FB 20

[EM 85, 90] - H. Ehrig, B. Mahr, Fundamental of Algebraic Specifications Vol. 1+2. EATCS Monographs on Theoretical Computer Science, Vol. 6 + Vol. 21, Springer-Verlag (1985,1990).

The Specification System OBSCURE: An Overview

Jacques Loeckx

University of Saarbrücken, West Germany

The system OBSCURE consists of a specification language and an environment.

The specification language[1, 3] differs from languages such as Clear or ACT-ONE in at least three respects. First, it is a language for parametrized algebras rather than for algebras. Next, it handles with models rather than theories. Finally, it is nearly institution-independent.

The environment [3] supports the editing and rapid prototyping of specifications. It allows infix and mixfix notation, overloading and lazy evaluation. The induction prover INKA developed at the University of Karlsruhe is to be integrated into the environment in the very next future.

[1] Th. Lehmann, J. Loeckx, "The specification Language of OBSCURE" in D. Sannella, A.Tarlecki (eds.), "Recent trends in abstract data types", LNCS 332, 1988

[2] Th. Lehmann, J. Loeckx. "OBSCURE, a specification language for abstract data types", Internal Rep., Universität Saarbrücken, Nov. 1990. Submitted for publication.

[3] J. Fuchs et al, "The manual of OBSCURE", Internal Report, Universität Saarbrücken, Febr. 1991.

Software-Retrieval Based on Logical Specifications of Functions

Sigi Meggendorfer

Intellektik - AI Research Group, Technical University Munich, West Germany

Within the SW Retrieval task it is necessary to propose a pragmatical point of view on formal specification techniques. These techniques and methods should support the SW designer in efficient development of large SW-Systems and therefore be treatable in practice.

Our approach to SW retrieval given here takes place at the lowest pre-algorithmical specification level of functions and modules. It is founded on a frame based specification formalism decribing as much properties of a function as are available. In order to retrieve pieces of code we demand the existance of a library of SW components consisting of a small piece of code and a related detailed specification. To solve the retrieval task we have to compare the specification of the desired function (called target) with all given specifications in the library (called sources).

One important part of a specification, called the semantics attribute, is represented by using full first order predicate logic in a well known pre- and postcondition definition style for functions. We introduce a formal definition of the reuse relation based on the semantics of functions which has a so-called input completeness and output correctness properly, following the intuition of reusability.

This formal framework with the related inference mechanism (a theorem prover; we use SETHEO) is embedded in a KL-ONE like knowledge representation system which supports taxonomical reasoning with a classifier. This embedding results in a connection of the theorem prover and the classifier yielding several advantages for the original theorem prover task to show reusability.

We will discuss some ideas and problems of this framework.

Introduction

A Fine-Grain Approach to Sorted Logic Jochen Burghardt

The Specification Language SPECTRUM Franz Regensburger

A Refinement Case Study J.-R. Abrial

The Role of Programming Logics in Formal Program Construction Werner Stephan

Integrating Various Approaches to Program Synthesis Using Dynamic Logic Maritta Heisel

The Complexity of Proving Program Correctness Hardi Hungar

Virtual Data Structures S.D. Swierstra

Calculating Programs by Equational Reasoning Lambert Meertens

Calculation by Computer Roland Backhouse

A Relational Theory of Datatypes Roland Backhouse

Relations as a Program Development Language Bernhard Möller

Time Analysis, Cost Equivalence and Program Refinement D. Sands

Coping With Requirement Freedoms Martin S. Feather

The IO-Graph-Method Gerd Neugebauer

Proof and Program Transformation in Type Theory: Some Remarks Didier Galmiche

Using the Calculus of Constructions in Order to Synthetise Correct Programs Benjamin Werner

A Survey of the Categorical Semantics of Dependent and Polymorphic Types Thomas Streicher

Propre: An Experimental Language for Programming with Proofs Michel Parigot

An Overview of another logical framework Bengt Nordström

Abstract Data Types in the Polymorphic λ-Calculus Kurt Sieber

A Meta-Calculus for Formal System Development Matthias Weber

An experiment in Formal Software Development: Using the B Theorem Prover on a VDM Case study Yves Ledru

Specifying Systems of Structured Polymorphic Specifications Martin Wirsing

Inheritance with exceptions - On the meaning of "but" P.-Y. Schobbens

The Algebraic Module Specification and Interconnection Language ACT TWO Werner Fey

The Specification System OBSCURE: An Overview Jacques Loeckx

Software-Retrieval Based on Logical Specifications of Functions Sigi Meggendorfer

Jean-Raymond Abrial
26 rue des Plantes
F-75014 Paris
France
phone: (1) 45428349

Prof. Dr. Roland Backhouse
Dept. of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513
NL-5600 MB Eindhoven
Netherlands
phone: +31 40 472744
email: wsinrcb@win.tue.nl
Fax: +31 40 436685

Prof. Dr. Wolfgang Bibel
Technische Hochschule Darmstadt
Fachbereich 20 Informatik
Alexanderstr. 10
W-6100 Darmstadt
phone: +49-6151-16-2100
email: xiiswbib@ddathd21.bitnet
Fax: +49-6151-16-5326

Jochen Burghardt
GMD Forschungsstelle
an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1
W-7500 Karlsruhe 1
phone: +49-721-6622-45
email: burghard@gmdka.uucp
Fax: +49-721-6622-968

Dr. Martin S. Feather
University of Southern California
Information Sciences Institute
4676 Admiralty Way
Marina Del Rey CA 90292, USA
phone: +1-213-822-1511,ext. 246
email: feather@vaxa.isi.edu
Fax: +1-213-823-6713

Dr. Werner Fey
Technische Universität Berlin
Fachbereich 20, Sekr. FR 6-1
Institut für Software und Theoretische Informatik
Franklinstraße 28-29
D-1000 Berlin 10
phone: 030 314 25812 or 030 314 73510

Prof. Dr. Jean-Pierre Finance
CRIN-CNRS & INRIA Lorraine
Boite Postale 239
F-54506 Vandoeuvre lès Nancy, France
phone: (33) 83 91 2112
email: finance@loria.crin.fr
Fax: (33) 83 41 3079

Dr. Didier Galmiche
CRIN-CNRS
Boite Postale 239
F-54506 Vandoeuvre-lès-Nancy, France
phone: (33) 83 91 20 00 ext. 2869
email: galmiche@loria.crin.fr

Maritta Heisel
Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
Postfach 69 80
W-7500 Karlsruhe 1
phone: 0721-608-4212
email: heisel@ira.uka.de

Dr. Hardi Hungar
Universität Oldenburg
Fachbereich 10 - Technische Informatik
Postfach 2503
Ammerländer Heerstraße 114-118
W-2900 Oldenburg
phone: (0441) 798-2372
email: hardi.hungar@arbi.informatik.uni-oldenburg.de

Prof. Dr. Stefan Jähnichen
Universität Karlsruhe
Institut für Programmstrukturen und Datenorganisation
Postfach 69 80
W-7500 Karlsruhe 1
phone: +49-721-662210
email: jaehn@gmdka.uucp
Fax: +49-721-6622-968
or:
GMD Karlsruhe
Vincenz-Prießnitz-Str. 1
W-7500 Karlsruhe 1
phone: +49-721-662210

Dr. Christoph Kreitz
Technische Hochschule Darmstadt
Fachbereich 20 Informatik
Alexanderstr. 10
W-6100 Darmstadt
phone: +49-6151-162863
email: xiisckre@ddathd21.bitnet
Fax: +49-6151-165326

Prof. Dr. Hans Langmaack
Inst. für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel
Preußerstr. 1-9, Haus II
W-2300 Kiel 1
phone: 0431/5604-28 or 27
email: hl@eausun.uucp
Fax: 0431/566143

Y. Ledru
Unité d'Informatique
Université Catholique de Louvain
Place Sainte Barbe, 2
B-1348 Louvain-la-Neuve, Belgium
phone: +32 10 47 31 50
email: yl@info.ucl.ac.be
Fax: +32 10 45 03 45
Dr. Jacek Leszczylowski
Institue of Computer Science
Polish Academy of Sciences
P.O. Box 22
00-901 Warszawa PKiN, Poland
phone: +48 (22) 357716 (home)
email: jacek@ida.liu.se
Fax: +48 (22) 200114 (office)

Prof. Dr. Wei Li
Dept. of Computer Science
Beijing University of
Aeronautics and Astronautics
Beijing 100083, P.R. China
phone: 0086-1-201 6994
Fax: 0086-1-201 5347
z.Zt.
Fachbereich 14 - Informatik
Lehrstuhl Prof. Dr. Hotz
Universität des Saarlandes
W-6600 Saarbrücken 11

Prof. Dr.-Ing. Jacques Loeckx
Fachbereich 14 - Informatik
Universität des Saarlandes
Im Stadtwald 15
W-6600 Saarbrücken
phone: (0681) 302 3435
email: loeckx@cs.uni-sb.de

Prof. Dr. Lambert Meertens
CWI
Dept. of Algorithmics & Architecture
P.O. Box 4079
NL-1009 AB Amsterdam, Netherlands
phone: +31 20 592 4141
email: lambert@cwi.nl
Fax: +31 20 592 4199
or:
Utrecht University
P.O. Box 80 089
NL-3508 TB Utrecht, Netherlands
phone: +31 30 534040

email: lambert@cwi.nl
Fax: +31 30 513791

Sigi Meggendorfer
TU München, Institut für Informatik
Forschungsgruppe KI
Augustenstr. 46, RGB
W-8000 München 2
phone: 089/521096
email: sigi@lan.informatik.tu-muenchen.dbp.de

Prof. Dr. W. Menzel
Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
Postfach 69 80
W-7500 Karlsruhe 1
phone: 0721/608 3919
email: menzel@ira.uka.de

Prof. Dr. Bernhard Möller
Institut für Mathematik
Universität Augsburg
Universitätsstraße 2
W-8900 Augsburg
phone: +49-821-598-2164
email: moellerb%uniaug@ira.uka.de
Fax: +49-821-598-2200

Gerd Neugebauer
Technische Hochschule Darmstadt
Fachbereich 20 Informatik
Alexanderstr. 10
W-6100 Darmstadt
phone: 06151/16 5382
email: xiisgneu@ddathd21.bitnet

Prof. Dr. Bengt Nordström
Department of Computer Science
University of Göteborg - Chalmers
S-412 96 Göteborg, Sweden
phone: +46-31-721033
email: bengt@cs.chalmers.se
Fax: +46-31-165 655

Prof. Dr. Michel Parigot
Université Paris 7, CNRS URA 753
2, place Jussieu
F-75251 Paris Cedex 05, France
phone: +33-1-44-27-37-68
email: parigot@frmap711
Fax: +33-1-44-27-69-35

Prof. Dr. Helmut A. Partsch
Faculty of Mathematics and Informatics
Nijmegen University
Toernooiveld 1
NL-6525 ED Nijmegen, Netherlands
phone: +31-80-652258/653410
email: helmut@cs.kun.nl
Fax: +31-80-553450

Prof. Dr. Peter Pepper
Technische Universität Berlin
Fachbereich 20 -- Informatik
Franklinstraße 28-29, Sekr. FR 5-6
D-1000 Berlin 10
phone: (030) 314-73470
email: pepper@opal.cs.tu-berlin.de

Franz Regensburger
TU München
Arcisstr. 21
W-8000 München 40
phone: 089/2105-8194
email: regensbu@lan.informatik.tu-muenchen.de

Dr. David Sands
Imperial College
180 Queen's Gate
London SW7 2BZ, U.K.
phone: (071) 5895111 or 4993
email: ds@doc.ic.ac.uk

Dr. Kurt Sieber
Fachbereich 14 - Informatik
Universität des Saarlandes
Im Stadtwald 15, Bau 36
W-6600 Saarbrücken 11
phone: 0681/302 3235
email: sieber@cs.uni-sb.de

Martin Simons
GMD Forschungsstelle
Vincenz-Prießnitz Straße 1
W-7500 Karlsruhe 1
phone: 0721/662221
email: simons@karlsruhe.gmd.dbp.de

Prof. Dr. Michel Sintzoff
Université de Louvain
Unité d'Informatique
place Sainte-Barbe 2
B-1348 Louvain-la-Neuve, Belgium
phone: +32-10-473150
email: ms@info.ucl.ac.be
Fax: +32-10-450345

Dr. Thomas Streicher
Universität Passau
Fakultät für Informatik
Innstr. 33, Postfach 2540
W-8390 Passau
phone: 0851/509-353
email: streiche@unipas.fmi.uni-passau.de
Fax: 0851/509-497

Prof. Dr. S. Doaitse Swierstra
Utrecht University
Dept. of Computer Science
P.O. Box 80 089
NL-3508 TB Utrecht, Netherlands
phone: +31-30-53-3962
email: swierstra@cs.ruu.nl
Fax: +31-30-513791

Matthias Weber
Universität Karlsruhe
Institut für Programmstrukturen und Datenorganisation
Postfach 69 80
W-7500 Karlsruhe 1
phone: +721/608-4386
email: nick@gmdka.uucp
or:
GMD Forschungsstelle
Vincenz Prießnitz Str. 1
W-7500 Karlsruhe 1
phone: +721/662236
email: nick@gmdka.uucp
Fax: +721/6622968

Benjamin Werner
INRIA - Domaine de Voluceau-Racquencourt
BP 105
F-78153 Le Chesnay Cedex, France
phone: (1) 39 63 52 31
email: werner@marguax.inria.fr

Prof. Dr. Martin Wirsing
Universität Passau
Fakultät für Mathematik und Informatik
Gebäude FMI
Innstr. 33
W-8390 Passau
phone: 0851/509345
email: wirsing@forwiss.uni-passau.de
Fax: 0851/509497
or:
Bayer. Forschungszentrum für Wissensbasierte Systeme
Forschungsgruppe Programmiersysteme
Innstr. 33
W-8390 Passau
phone: 0851/509705
email: wirsing@forwiss.uni-passau.de
Fax: 0851/509497

Bisher erschienene und geplante Titel:

W. Gentzsch, W.J. Paul (editors):
Architecture and Performance, Dagstuhl-Seminar-Report; 1, 18.-20.6.1990; (9025)

K. Harbusch, W. Wahlster (editors):
Tree Adjoining Grammars, 1st. International Worshop on TAGs: Formal Theory and Application, Dagstuhl-Seminar-Report; 2, 15.-17.8.1990 (9033)

Ch. Hankin, R. Wilhelm (editors):
Functional Languages: Optimization for Parallelism, Dagstuhl-Seminar-Report; 3, 3.-7.9.1990 (9036)

H. Alt, E. Welzl (editors):
Algorithmic Geometry, Dagstuhl-Seminar-Report; 4, 8.-12.10.1990 (9041)

J. Berstel , J.E. Pin, W. Thomas (editors):
Automata Theory and Applications in Logic and Complexity, Dagstuhl-Seminar-Report; 5, 14.-18.1.1991 (9103)

B. Becker, Ch. Meinel (editors):
Entwerfen, Prüfen, Testen, Dagstuhl-Seminar-Report; 6, 18.-22.2.1991 (9108)

J. P. Finance, S. Jähnichen, J. Loeckx, M. Wirsing (editors):
Logical Theory for Program Construction, Dagstuhl-Seminar-Report; 7, 25.2.-1.3.1991 (9109)

E. W. Mayr, F. Meyer auf der Heide (editors):
Parallel and Distributed Algorithms, Dagstuhl-Seminar-Report; 8, 4.-8.3.1991 (9110)

M. Broy, P. Deussen, E.-R. Olderog, W.P. de Roever (editors):
Concurrent Systems: Semantics, Specification, and Synthesis, Dagstuhl-Seminar-Report; 9, 11.-15.3.1991 (9111)

K. Apt, K. Indermark, M. Rodriguez-Artalejo (editors):
Integration of Functional and Logic Programming, Dagstuhl-Seminar-Report; 10, 18.-22.3.1991 (9112)

E. Novak, J. Traub, H. Wozniakowski (editors):
Algorithms and Complexity for Continuous Problems, Dagstuhl-Seminar-Report; 11, 15-19.4.1991 (9116)

B. Nebel, C. Peltason, K. v. Luck (editors):
Terminological Logics, Dagstuhl-Seminar-Report; 12, 6.5.-18.5.1991 (9119)

R. Giegerich, S. Graham (editors):
Code Generation - Concepts, Tools, Techniques, Dagstuhl-Seminar-Report; 13, , 20.-24.5.1991 (9121)

M. Karpinski, M. Luby, U. Vazirani (editors):
Randomized Algorithms, Dagstuhl-Seminar-Report; 14, 10.-14.6.1991 (9124)

J. Ch. Freytag, D. Maier, G. Vossen (editors):
Query Processing in Object-Oriented, Complex Object, and Nested Relation Databases, Dagstuhl-Seminar-Report; 15, 17.-21.6.1991 (9125)

M. Droste, Y. Gurevich (editors):
Semantics of Programming Languages and Model Theory, Dagstuhl-Seminar-Report; 16, 24.-28.6.1991 (9126)