LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

Fifth Workshop on Specification of Abstract Data Types

Gullane, 1-4 September 1987

Fifth Workshop on
Specification of Abstract Data Types

ABSTRACTS

LFCS Report Series

ECS-LFCS-87-41
(also published as CSR-249-87)

October 1987

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Fifth Workshop on Specification of Abstract Data Types

— Foreword —

Don Sannella
Laboratory for Foundations of Computer Science
Department of Computer Science
University of Edinburgh

The Fifth Workshop on Specification of Abstract Data Types took place 1-4 September 1987 in the small town of Gullane, about 20 miles from Edinburgh. This series of workshops, including meetings in Sorpesee (1981), Passau (1983), Bremen (1984) and Warberg (1986), has become the chief international series of meetings devoted to this topic. Participants came from East and West Germany, Poland, Italy, the U.S., Spain, Portugal, France, the U.K., Norway, Switzerland, Denmark, Hong Kong and the Netherlands. Successors in Berlin (1988) and Dresden (1990) are already planned.

The algebraic specification of abstract data types has been a flourishing research topic in computer science since 1974. The main goal of work in this area is to evolve a methodology which would support the design and formal development of reliable software. The particular approach taken builds upon concepts from universal algebra and category theory.

The general feeling was that the workshop was extremely successful and that a number of very interesting new results and ideas were presented. This report contains abstracts of all the talks, most of which discussed very recent and as yet unpublished work.

Acknowledgements

The secretarial support of Joan Ratcliff and financial support from the British Council are gratefully acknowledged.

Good functors ... are those preserving philosophy!

Gilles Bernot
Université Paris-Sud, Orsay

The aim of this paper is to prevent the abstract data type researcher from an improper, abusive use of category theory. We mainly emphasize some unpleasant properties of the synthesis functor when dealing with so-called loose semantics in a hierarchical approach. All our results and counter-examples are very simple, nevertheless they shed light on many common errors in the abstract specification field.

We also summarize some properties of the category of models protecting predefined sorts.

Inheritance and reusability mechanisms in the PLUSS specification language

Michel Bidoit
L.R.I.
C.N.R.S. U.A. 410 & Université Paris-Sud
F - 91405 ORSAY Cédex, France

In this talk we briefly recall the main features of the PLUSS specification language (PLUSS means Proposal of a Language Usable for Structured Specifications). The original design of the PLUSS specification language was based on ASL and its semantics follows the "loose" approach. Thus the semantics of a specification is some class of (non-isomorphic) algebras, which correspond to the various possible implementations of the specification. The main originality of the PLUSS specification language is to state a careful distinction between completed specification components and specification components under design. By completed specification components we mean the following strong property: the class of possible implementations is fixed. Practically that means that such a specification component is either already implemented or may be implemented without taking care of its context (for instance the other components of a specification where this completed specification component is used). By specification component under design we mean a preliminary specification component where the signature and the axioms are not fully fixed and may be further refined: at this early stage, implementing the specification component is premature since the implementation choices may have to be reconsidered later, depending on the further refinements of the specification. These two kinds of specification components are differentiated by the keywords **spec** and **draft**.

According to this distinction between completed specification components and specification components under design, PLUSS offers primitives and mechanisms to describe "classes of implementations" as well as primitives and mechanisms to write and develop new specifications from other ones. As a consequence, the reusability concept should be split into two distinct ones: software components reusability (through their formal specifications) and specification components reusability. Specification reusability can be achieved by means of various specification-building primitives (such as enrichment and parameterization) and inheritance mechanisms.

Then we explain how far our intuition and needs about software reusability can be more or less reflected by the semantics of the enrichment specification-building primitive for completed specification components (this primitive is called use in PLUSS). We show that the loose approach is not powerful enough to fulfil our needs and we introduce a new approach, the stratified loose approach, which can be considered as a generalisation of both loose and initial semantics.

Acknowledgements: Many thanks are due to Marie-Claude Gaudel for fruitful discussions. This work is partially supported by ESPRIT project 432 METEOR and C.N.R.S. Greco de Programmation.

The Semantics of Generics in Pannda-S

Michael Breu
University of Passau

Abstract

Pannda-S is a specification language derived from the Ada/Anna language. Pannda-S allows loose specification of abstract data types (packages) and parameterized data types (generics). The semantics of generics is defined in such a way, that it can be viewed as the specification of a hierachical data structure, where the parameter plays the role of the primitive part, but need not to be term generated. Therefore concepts like hierachy consistency and persistency can be applied. Constraints as admissibility of an actual parameter algebra and hierachy faithfulness are expressed by consistency conditions.

COMMUNICATION BETWEEN AN INTERPRETER OF ABSTRACT SYNTAX TREES AND A TERM-REWRITING SYSTEM

by Xavier Burgués and Xavier Franch
Dept. Programació de Computadors
Facultat d'Informàtica de Barcelona

In the last years, a large number of research projects about programming environments have been developed. This kind of environments consider an underlying representation of programs, usually in an abstract syntax tree (AST for short) form. This internal representation is created and modified using a syntax-directed editor and/or a front-end compiler.

Another trend of current interest is the construction of term-rewriting systems that allow the execution of algebraic specifications.

In our department, we have investigated about both topics, working in two particular projects:

- a programming environment for the language Merlí, which is oriented to the use of abstract data types (ADTs for short) and program schemes; the kernel of this programming environment is based on the following tools: a syntax-directed editor, a front-end compiler, a schemes library system, a pretty-printer and an interpreter for ASTs;
- a term-rewriting system to execute an specification language defined as anotations for Merlí.

The next step we want to take is the connection of the interpreter of the first project with the rewriting system, which will allow the execution of programs that are partially specified (by equations) and partially implemented (in an imperative style).

Our talk will enumerate the different situations involved in this process, the problems that arise and some possible solutions.

The central issue when executing a program is the evaluation of functions. For every function evaluation we have to consider how the very function and its actual parameters have been defined (i.e.: specified, implemented or a mixture of both possibilities).

We distinguish six different cases one can expect to find in the process of evaluating a function using both the term-rewriting system and the interpreter of ASTs, related to every function's actual parameter, say **p**:

- when the function is specified, we consider three derived sub-cases:
  * **p** is an object from an specified ADT;
  * **p** is an implemented object corresponding to an ADT which is not the function's ADT;
  * **p** is an implemented object from the function's ADT; this would be an special case in which we have an ADT partially specified and partially implemented.
- when the function is implemented, we consider three analogous derived sub-cases:
  * **p** is also an implemented object;
  * **p** is an object from an specified ADT which is not the function's ADT;
  * **p** is an object from the function's ADT; this would be the already mentioned special case in which we have an ADT partially specified and partially implemented.

Every subcase involves a different kind of communication between the interpreter and the rewriting system.

General logic over an institution

R.M. Burstall
Department of Computer Science
University of Edinburgh

Stimulated by Bob Harper, we have been thinking about connections between the Edinburgh logical framework and the idea of institutions, both approaches to generality in logic. By analogy with Barwise's treatment of quantifiers in abstract logic we can show how to construct a new institution from an old one by adding a consequence relation and universally quantified variables. This mirrors part of the logical framework idea in institutional terms.

Formal specifications, prototyping and integration tests

C. CHOPPY
Laboratoire de Recherche en Informatique
U.A. C.N.R.S. 410
Université Paris-Sud, Bât 490
F-91405 ORSAY Cédex, FRANCE

Formal specifications came to be considered as crucial in software development ; this is due in particular to the fact that they require a very precise and non ambiguous definition of the problem to be solved from the beginning of the development. However, the applicability of formal specifications raises various problems : one needs a formalism that is accessible, a specification language to write legible specifications, a specification environment where various tasks such as editing, verification, etc. are automatized. The interest on formal specifications is enhanced when it turns out that they can be used at various stages in the software life cycle, such as prototyping, program construction, testing, and software reuse.

The context for this paper is that one uses a specification language that is modular and hierarchical (of course together with a user-friendly specification environment) and of an interpretation mechanism that allows to associate a computation, i.e. an evaluation, to a given specification and a series of operations on data. This evaluation mechanism is particularly interesting since it allows to perform rapid prototyping at the specification level.

When a complex system is designed, it is recommended to test each module individually and to test the integration of modules that are connected (this integration is often done step by step). Testing thus leads to realize "stub modules" and "driver modules" that simulate modules calling or called by the modules under test. The approach presented here is to use prototyping to help integration testing. Let us start with a hierarchical and modular formal specification of the system to be designed. Together with an evaluation procedure this specification may become executable and used as a prototype of the system. The idea here is to progressively replace specified modules by corresponding implemented modules. Performing an execution on such a system, where the integration is partial, requires mixing evaluation in the specified modules with evaluation in the programming language used in the implemented modules. With this kind of integration, modules are integrated in a context of other modules that are either specified or implemented ; it is then possible to do integration testing without having to realise stub or driver modules that would simulate an environment for the integrated modules.

This paper defines a framework where this kind of mixed evaluation is possible and studies the problems raised by the mixing of evaluation. This approach requires in particular to examine how and under which conditions data can be transfered from one type of evaluation to the other. A notion of *granularity* of implementation is defined to identify circumstances where it is necessary to implement more than one module at the same time. The examples are specified using algebraic specifications ; the programming languages that are considered for these examples are Pascal and Lisp. However this approach is applicable to other formal specification languages and programming languages.

Program design from generic specifications

Silvia Clerici
Facultat d'Informàtica
Universitat Politècnica de Catalunya
Barcelona, Spain

We describe our current project for building, ordering, and handling large specifications, in such a way that programs may be derived from them.

Our goal is to design a methodology for integrating, in the same conceptual framework, data types and problem solving schemes, and to implement the corresponding environment as a support for building specifications with this method.

In order to derive automatic implementations, constraints on the way of building specifications are imposed. Specifically, only restricted forms of "basic constructors" are allowed. These constraints, together with a "class-subclass" mechanism, provide criteria for organizing a "generic specifications base".

AXIS

Derek Coleman
HP Labs
Hewlett Packard Ltd
Filton Road
Bristol BS12 6QZ

Axis is an algebraic specification language designed at HP Laboratories Bristol and intended for industrial use. The language is based on Clear and OBJ and has parameterised modules with loose interface specifications together with module combinators USING, SUM, RENAME and DERIVE.

The implementation was developed in a rigorous manner using Axis and contains a compiler, rewrite engine and a specification database. Axis is already finding use inside Hewlett Packard. This talk will examine application of algebraic specification to software development and other areas.

ALGEBRAIC DATA TYPE AND PROCESS SPECIFICATIONS BASED ON PROJECTION-SPACES

H. Ehrig

ABSTRACT

In order to provide an algebraic semantics for recursively defined (nonterminating) processes there is a well-known metric approach leading from process algebras to complete process algebras. In order to obtain an algebraic specification and completion of such algebras it is sufficient and much more convenient to consider instead of metric spaces only spaces with a suitable family of projections, called projection spaces. Projection spaces and algebras are shown to be a suitable basis for an algebraic semantics of combined data type and process specifications.

SYNTAX AND SEMANTICS OF ACT TWO

Werner Fey
Technical University of Berlin

ABSTRACT

The language ACT TWO for the specification of modular systems is based on module specifications, parameterised specifications, requirements specifications, and their structuring concepts union, extension, actualization and renaming.

Similar to the specification language ACT ONE also for ACT TWO the abstract syntax is given by an extended BNF-grammar and the full semantics is defined on two levels: The first level is that of specifications and the second level is that of algebras and functors. Only correct specifications on the first level with compositional semantics on the second level are denoted, if some semantical context conditions are satisfied.

PROOF-THEORETIC SEMANTICS OF SPECIFICATION-BUILDING OPERATIONS

José Fiadeiro and Amílcar Sernadas

IST/INESC
Rua Alves Redol, 9, 3º
1000 Lisboa, Portugal

Building on the work of Goguen and Burstall on institutions and on Tarski's notion of deductive system, a formalism for manipulating theories in an arbitrary logic is presented. Its main contribution is the formalisation of the semantics of theory-building operations, including the so-called "data-constraining", on top of the underlying logic's derivability relation. For that purpose, the notion of π-institution is proposed as a proof-theoretic counterpart to the notion of institution, replacing the satisfaction relation by a primitive consequence operator. Clause logic with resolution (in a slightly extended form) and many-sorted first-order equational logic with Goguen and Meseguer's calculus have been proved to be π-institutions.

The interest of this approach is, we believe, not just a formal one. For instance, some authors advocate that underlying each layer of the refinement process should be a proof theory, instead of a model theory, but, up to now, no alternative framework to the institutional one has been proposed for supporting this point of view. Also, facing the increasing demand for specific formalisms for dealing with particular classes of problems, it seems more appropriate to allow their definition in terms of the intended consequence relation instead of requiring the definition of a particular notion of model. For instance, when defining the "behaviour" of the theories of a certain logic, we usually have in mind the mechanisms that define this behaviour in a more direct and "syntactic" way, for instance in terms of "when does a certain theory imply another one". Besides, during the construction of a specification, the "end user" will be facing an inference engine, that is to say, its interface with the logic will be given through the proof mechanisms that are put at its disposal. Of course, this does not mean that having a correct notion of model does not help guiding intuition.

Like in institutions, Category Theory is used for formalising the concepts that are introduced. The main constructs underlying this formalism build heavily on the classification of theory morfisms. The notions of conservative and true (in the sense of Ehrich) morphisms developed within the equational institution are adapted to π-institutions and compared to each other. Conservative morphisms play an important role, in particular in the definition of an image factorisation system for the category of theories from an image factorisation system of the signature category. The existence of such an image factorisation system for the signature category (together with finite cocompleteness) is an important measure of the "expressive power" of the π-institution in what concerns specification building. For any π-institution satisfying these properties, it is possible to define the notions of minimal and canonical interpretation of a theory, thus providing the basis for the semantics of "data-constraining". Moreover, adopting the category of its minimal interpretations as the model class of a theory, it is shown that a liberal institution can be obtained from every π-institution where the notions of canonical interpretation and free model (in the sense of Goguen and Burstall) coincide.

On top of this framework, the different levels of specification building distinguished by Sanella and Tarlecki (presentation, theory and model levels) have been analysed for the traditional Clear-like operations (including data constraining). This classification is ortogonal to the proof/model-theoretic one. In the classical institutional framework, the semantics of these operations is model-theoretic, even at the theory level. On the other hand, adopting the proposed notion of model, the semantics of a specification in the π-institutional framework can also be given at the model level. At this level, the semantics of a specification consists of a collection of theories (in the proof-theoretic sense). As argued above, this seems to be a more adequate denotation of a specification than any collection of models (in the traditional model theoretic sense).

A semantics for conceptual modelling has also been developed in the π-institutional framework following the approach proposed by Amílcar and Cristina Sernadas. In fact, this was the starting point to the development of the proposed formalism. Conceptual Modelling Approaches can be seen to introduce derivation mechanisms within the underlying π-institution's category of theories, and formal conceptual model development can be seen as a *derivation* using these mechanisms, thus depicting an explicit structure for the resulting conceptual schema. This topic is too broad in itself to be focused.

Simulation of Pr/E nets by term rewriting and some related problems.

Harald-Reto Fonio
GMD Birlinghoven
St Augustin, FRG

Joining the contribution of H.W.Schmidt, I will point out way of describing the process development of Pr/E nets by applying rewrite rules being derived from the firing rules of the net. Term rewrite methods come in in a natural way, since Pr/E nets are a synthesis of abstract data types and Petri net theory. Here I am especially interested in a suitable construction of these rules.

Concerning the construction of term rewrite rules we compose rules w.r.t. some visible subspecification SPEC contained in a larger specification SPEC', that means w.r.t. the source of the arrow $SPEC \to SPEC'$ in the finitely cocomplete category CATSPEC of specifications. In order to derive the rules for nets we assume for some Petri net $N = (S,T,F)$ with finite sets S and T of S- and T-elements and a flow relation F in $S \times T \cup T \times S$, we assign to each $x \in S \cup T$ an arrow $SPEC(x) \to SPEC'(x)$ in CATSPEC, and to each element $(x,y) \in F$ a double arrow $SPEC(x) \leftarrow SPEC(x,y) \to SPEC(y)$. In order to obtain the total specification for the whole net and the total visible part we take for the families

$\{SPEC'(x) \leftarrow SPEC(x) \leftarrow SPEC(x,y) \to SPEC(y) \to SPEC'(y)\}_{(x,y) \in F}$ and

$\{SPEC(x) \leftarrow SPEC(x,y) \to SPEC(y)\}_{(x,y) \in F}$ the star pushout (/Eh 78/ for the case of graph objects); SPEC(x,y) is the common visible interface of SPEC(x) and SPEC(y). As Pr/E nets model distributed systems, star pushout are the natural combination tool for the local parts (e.g. /Fo 84/). Moreover we assume that for any x in $S \cup T$ the specification SPEC(x) contains some distinctive variable X over some boolean algebra the specification of which is contained in SPEC(x) (such boolean algebras may be $\{1,0\}$, powersets, products or coproducts of boolean algebras). These variables are supposed to be place holders (or ports) for tokens and events in the net N, and they may be endowed with an arity (e.g. $X = (X_1,X_2)$ of arity 2). We may identify these variables with the nodes of the net N. Partially defined predicates may restrict the domain of definition of these variables. Finally to each arc $(x,y) \in F$ a substitution σ is associated which is defined for the arities in x and y and w.r.t. the visible specification SPEC(x,y); this substitution σ assigns to each variable x and y some term with or without variables in the boolean part of SPEC(x,y). For example $\sigma(x)$ may denote some set; we assume here $\sigma(y)$ to be a singleton for $y \in T$. Net markings denoting actual states of the net N appear as substitutions of the S-elements, too. Finally for all $x \in S$ the boolean terms $\sigma(x)$ are combined to a term of the coproduct of the booelan algebras underlying all the S-elements.

For some $(x,y) \in F$ we call x a preelement of y and y a postelement of x. For an T-element y with a preelement x, a postelement x' and for substitutions σ assigned to (x,y) and σ' assigned to (y,x') we first try to unify $\sigma(y)$ and $\sigma'(y)$ by a suitable unifier σ'', if exists. Assuming an actual marking σ''' for all the S-elements, the firing rule for y causes subtraction $\sigma(\sigma''(s(x))$ from $\sigma'''(x)$ and addition of $\sigma''(\sigma'(x'))$ to $\sigma'(x')$, corresponding to the axiom of the boolean algebras for that places. Moreover $\sigma''(\sigma(x))$ should be contained in $\sigma'''(x)$ and $\sigma''(\sigma'(x'))$ should be disjoint to $\sigma'(x')$. Concerning these constructive steps see /Fo 87/ for more dedails.

Even though the firing rules are of local nature, net states are usually presented as products w.r.t. the S-elements. In order to catch this local behaviour by rewrite rules, I introduced coproducts of the boolean algebras involved. Without knowing from each other S. Kaplan did the same step for the derivation of the rewrite rules in question; I became aware of this fact when obtaining a paper very recently (/Ka 87/). First steps into this direction have been done in /ScPa 85/ for the denotations of process terms, but unfortunately his intentions remained dark to me. I first introduced coproducts in order to define minimal subobjects for the description of conflicts, which is of course not simple when using products. This led me in a natural way to the representation of the terms as direct sums of the S-elements.

The ideas listed above may be reflected by the example of the five dining philosophers denoted by the set $\{1,2,3,4,5\}$, where i may pick up in an action pu(i) or put down in an action pd(i) the forks $\{f(i),f(i+1)\}$, where $f(i+5) = f(i)$. Given now suitable actual sets "th_1" of thinking and "ea_1" of eating philosophers and "fo_1" of forks, the fact that the philosopher i tries the action pu(i) means :

apply the rule pu(i) defined by

$$th + fo + ea \rightarrow th\backslash\{p\} + fo\backslash\{f(i),f(i+1)\} + ea\backslash\{p\}$$

to the actual term $ea_1 + fo_1 + th_1$ under the precondition $\{p\}$ in th, $\{f(i),f(i+1)\}$ in fo and $\{p\}$ not in ea.

References :

/Eh 87/ Ehrig, H. :
Introduction to the algebraic theory of graph grammars ; LNCS 73, 1979, 1 - 69.

/Fo 84/ Fonio, H.- R. :
Amalgamation of graph transformations with applications to synchronisation in distributed systems ; Diplomarbeit, TU Berlin,Dec. 1984.

/Fo 87/ Fonio, H.- R. :
Symbolic execution of nets using term rewrite systems ; Graspin techn. paper, GMD 32/1, GMD, 1987.

/Ka 87/ Kaplan, S. :
Rewriting with a nondeterministic choice operator ; L.R.I. U.A. 410 of the C.N. R.S., Doc. Nr. METEOR/t4/LRI/3-6, Universite des sciences,Orsay,1987; to appear in T.C.S. 1987.

/ScPa 85/ Schmidt, H.W., Papazoglou,M. :
Abstract modular implementation of processes by predicate event systems ; Graspin technical paper ;GMD22/2, GMD,1985.

A new completion procedure for conditional equational specifications

Harald Ganzinger
Fachbereich Informatik
Universität Dortmund
D-4600 Dortmund 50, W. Germany

The paper presents a new completion procedure for conditional equations. The work is based on the notion of reductive conditional rewriting. The procedure has been designed to also handle nonreductive equations that are generated during completion. The paper in particular presents techniques for simplification of conditional equations and rules, so that the procedure terminates on more specifications. The correctness proofs which form a substantial part of this paper employ recursive path orderings on proof trees, an extension of the ideas of Bachmair, Dershowitz and Hsiang to the conditional case.

How to improve the efficiency of the interpretation of algebraic specifications

Horst Hansen

We present a way to improve the efficiency of the interpretation of equational algebraic specifications. Instead of using an interpreter which is based on common rewrite-techniques on terms we use an interpreter which has terms with access points (i.e. terms together with substitutions) as the basic underlying data-structure. We define some general operations on terms with access points (f.i. 'move an access point'), which can be implemented in a way that their execution needs constant time on any term. Our interpreter uses these general operations together with normal rewriting steps. The new interpreter is called correct w.r.t. a normal interpreter, if the results of any interpretation are identical after carrying out all substitutions involved in the term with access points. Access points may be chosen freely. It can be shown that any correct new interpreter needs no more rewrite steps to produce the result of any computation. We present several examples, in which the number of required rewrite-steps decreases.

Institutions and the Edinburgh Logical Framework

Robert Harper
Edinburgh University

Both the Edinburgh Logical Framework and Burstall and Goguen's institutions address the question "what is a logical system?" LF provides one answer to this question in the form of a general theory of formal systems and inferential activity (such as proof checking and proof development). Institutions provide another answer in the form of an abstract account of model theory. It is natural to consider the relationship between the two points of view. In this talk we consider the problem of representing a logic in LF from an institutional point of view. First, we sketch a model theory for LF and show that it forms an institution. Then we define a notion of representation of one institution in another (in particular, in LF) and establish the basic theorems that characterize such representations. Finally, we discuss the introduction of higher-order judgements into an institution, and establish a complete correspondence between these judgements and their representation in LF.

WP Observational Algebra Specifications

Magne Haveraaen
Department of Informatics
Allegt. 55
N-5007 Bergen, NORWAY

Dijkstra's Weakest Precondition (WP) predicate transformer is well suited for describing the semantics of imperative constructs, as well as developing and proving properties of imperative programs. Specifications using WP usually ignore data types, and assume the existence of some primitive types, e.g. integers and arrays.

Algebraic Specifications (AS), however, are well suited for defining Abstract Data Types (ADT), but are mostly used in a functional framework. There are several approaches to AS. In the Observational approach the only thing that matters is what we can observe of the value contained in an ADT. This ultimately reduces to a boolean value, or, in other words, a predicate.

Here a technique that combines these two methods is shown. The purpose is to make it easy to define ADTs in an imperative language, and the technique is tentatively called "WP Observational Algebra Specifications". It allows us to combine the WP method for program development with refinements of data type specifications, enhancing the usefulness of both methods. The use of WP to define ADTs has also been described by Würges in [2].

In these definitions of ADTs, as in imperative languages in general, procedures that change a program's state is the norm. In fact any partial operator must be written as a procedure, and only total operators may be defined as functions. The specifications will thus show how a sequence of one or more procedures alter the state of a program's variables. In many instances a sufficiently complete specification can be written following the guidelines given by Guttag and Horning in [1]. Other ADTs are more easily specified using informal guidelines, e.g. as those suggested in [2], together with an explicit proof of sufficiently completeness.

To simplify both specifications and proofs, restrictions are placed on the parameters to procedures. As the specifications only consider the external behavior of a procedure call, the resulting proof rules are as simple as those associated with the assignment statement. These restrictions also simplify the proof that an implementation of a procedure satisfies its specification, as most of the predicates resolving problems associated with parameter passing are not needed.

To illustrate the specifications, some variations of the ADT queue is shown. The standard specification of an "unbounded queue" is used as a starting point, and then simple variations produce the more realistic, in terms of a possible implementation, specifications:

1. A bounded queue, with a possibly unknown, but fixed, maximal length. Corresponds to a statically allocated queue, or a dynamically allocated queue, which is allowed

to grow to a certain length, e.g. bounded by some percentage of the free memory at the time of its creation.

2. A queue with a monotonically increasing maximal length. Corresponds to a dynamically allocated queue, which never releases any of its elements, but recycles them internally.

3. A queue with a variable maximal length. Corresponds to a dynamically allocated queue which frees and allocates elements as needed. It may thus not be able to grow to any specific length, not even the same length as it had earlier on in the same program execution.

Lastly, the refinement method is used to show how a small program may be developed, and some problems with the practical use of formal specifications to define ADTs are pointed out.

[1] John Guttag & J. J. Horning: The algebraic Specification of Abstract Data Types. Acta Informatica, Vol. 10, 1978, pp. 27-52.

[2] Harald Würges: A specification Technique Based on Predicate Transformers. Acta Informatica, Vol. 15, 1981, pp. 425-445.

MUTUALLY RECURSIVE ALGEBRAIC DOMAIN EQUATIONS

Anne Elisabeth Haxthausen
Department of Computer Science
Technical University of Denmark
DK-2800 Lyngby

This talk discusses an algebraic analogy to domain equations as known from denotational semantics. In the algebraic approach domains are considered as algebras, i.e. as collections of sets and operations on these. Given algebraic specifications of primitive domains like *booleans* and *natural numbers*, and parameterized algebraic specifications of domain constructors like *union* and *product* new domains may be defined by means of algebraic domain equations. In a work done by Ehrich and Lipeck (1983) the case of one recursive algebraic domain equation is treated. Here we consider the general case of n mutually recursive algebraic domain equations. This is roughly speaking a set of parameterized algebraic specifications applying instances of each other in a recursive fashion. As for $n = 1$ strongly persistent algebraic domain equations have an initial solution. We define an operation taking as argument a set of recursive algebraic domain equations and giving an algebraic specification of the initial solution by a new parameter passing mechanism called *recursive parameter passing*.

Towards a Kernel Language for Reusable Components

Martin Wirsing and Rolf Hennicker

Universität Passau
Lehrstuhl für Informatik
Postfach 2540
D - 8390 Passau

In the practice of software development often new code is produced even in cases, where software components already exist solving similar (or even the same) tasks. It is obvious that the effectivity of software development and the reliability of software could be improved, if already existing software components would be reused as many times as possible.
In the talk a formal definition of reusable (software) components is given and a kernel language for reusable components is presented. The definition of reusable components is based on the following idea:
If one wants to reuse a piece of software, it is necessary to have an exact understanding what this piece of software really does. Therefore a reusable component should be at least a pair, consisting of an abstract specification of the program or the subroutine one wants to reuse and of the concrete implementation. In our approach we consider reusable components not only as pairs of specifications (with an implementation relation in between) but more general as trees of specifications, where any children node is an implementation of the father node.
To formalize the notion of reusable components, we use the algebraic specification language ASL for describing the nodes of reusable components and we use the implementation relation for ASL-specifications for the arrows of the trees of specifications.
To define the language it is shown that all ASL-operations extend in a natural way to operations on reusable components. In addition we define several basic operations for manipulating reusable components.
In order to describe relationships between reusable components an implementation relation for reusable components is introduced. Informally this implementation relation says that a reusable component r2 is an implementation of a reusable component r1, if r2 is a refinement of r1 (up to ASL-implementations of the nodes of r1).

A Note on Inductive Inference for Solving Termination Problems

Klaus P. Jantke
Computing Centre
Humboldt University Berlin
German Democratic Republic

Muffy Thomas
Dept. of Computing Science
University of Stirling
Stirling, Scotland

We consider the use of inductive inference techniques for solving the problem of non-termination when applying the Knuth-Bendix (KB) completion procedure.

Inductive inference is a mathematical theory of learning objects of some underlying well-defined class from some incomplete information; for example, from some example computations. We investigate the situation where we are given a non-canonical rewriting system, X say, and we would like to find an enrichment of X , Y say, such that $X \cup Y$ is canonical. The enriched rewriting system $X \cup Y$ should be complete with respect to X and consistent with respect to a subset of the sorts of X.

The canonical rewriting system is generated by applying inductive inference techniques in different ways: for synthesising new operator definitions and for finding weakest common generalisations of infinitely many critical pairs. The approach offers a completely new methodology for improving the power of Knuth-Bendix algorithms.

The motivation for considering this problem is proofs by consistency in particular kinds of theories containing associative-commutative constructors. The application of the inductive inference techniques to a particular example enables us to show that a certain property holds for the example and this leads us to conjecture that the property holds for all rewriting systems belonging to a particular class.

Transformations on Specifications

Stefan Kahrs
Universitaet Bremen
Fachbereich Mathematik/Informatik
Bibliotheksstrasse
D-2800 Bremen

The approaches of formal program development and algebraic specification are combined to something like a formal specification development. The development starts with a requirement specification and ends - after several development steps - with another (sometimes executable) specification. Each step preserves - in a specific sense - the semantics of the previous specification.

To avoid explicit correctness proofs for each development step in each development the single steps are performed by the use of transformation rules. These rules are (should be) proven correct once for all. Beside some general remarks on correctness of transformation rules one (!) correctness concept of specification transformations is discussed.

Algebraic specification of concurrent systems : a proposal

Stéphane Kaplan
L.R.I. Bât. 490
Université des Sciences
F-91405 Orsay Cedex (France)

We consider an extension of the abstract data type formalism to the specification of concurrent systems. We introduce the key-concept of *process specification*. In such a specification, processes act on data via an application operator. Processes are defined by composition of atomic actions (described by their action on the environment), and non-atomic actions (described by composition of smaller actions). The operators are the non-deterministic, sequential and parallel composition.

Relative Completeness in Algebraic Specifications

Lin Huimin
Laboratory for Foundations of Computer Science
Department of Computer Science
University of Edinburgh

August 19, 1987

Abstract

A notable phenomenon in the theory of algebraic specifications is the divergence of different approaches, namely the initial algebra approach final algebra approach, and generated model approach. The notions of behavioral equivalence and sufficient completeness are closely related to the latter two.

In this papper, a concept called *relative completeness* is proposed. Roughly speaking, a relatively complete specification is a maximal consistent specification with respect to the initiality restriction on its base specification. In the context of many-sorted equational logic, it makes sense to talk about relative completeness in different sorts. We call a specification *base complete* if it is relatively complete in each base sort (sort of the base specification), and *complete* if it is relatively complete in every sort. It turns out that base completeness is weaker than sufficient completeness, but it still guarantees the existence of final models.

A complete specification has a unique (up to isomorphism) generated model which preserves its base models, so its initial and final models coincide. This is not necessarily true for base complete specifications, whose models are unique only up to behavioural equivalence. But a base complete specification can always be uniquely extended to a complete one and the initial models of the unique extension coincide with the final model of the original specification. Such extension can be characterised as equalizing 'indistinguishable' terms. These results reveal the relationship between the initial and final model semantics.

The specification environment OBSCURE

Part I: An informal introduction

Jacques Loeckx
Universität des Saarlandes
Fachbereich 10 - Informatik
D — 6600 Saarbrücken

OBSCURE consists of a specification language together with an environment for it. The present talk is devoted to a brief description of the environment.

The *OBSCURE* environment is a program consisting of a specification unit and a verification unit. With the help of a command language the user induces the design unit to stepwise generate specifications. The verification unit allows to prove properties of these specifications. In the talk a protocol of a session with the design unit will be discussed in some detail.

Using Graph Rewriting as Operational Semantics for Algebraic Specifications

M. Löwe
Technical University Berlin

ABSTRACT

Term rewriting systems have been proposed as operational semantics for algebraic specifications. Sometimes, however, rewriting on terms, i.e. on trees or collapsed trees, is very inefficient. It is very often impossible to increase the speed of evaluation in a way that the performance is sufficient for industrial purposes.

We propose the use of graph rewriting techniques as operational semantics for algebraic specifications. To connect specifications with graph grammars (we refer to the Berlin algebraic approach), we translate signatures and equations into graph grammar productions. This translation is shown to be correct.

This graph grammar may be optimized by changing and adding some productions in a certain specification-consistent way. We make this notion precise in this paper and give small examples where this technique leads to a significant reduction of evaluation steps.

Thus, without leaving the field of rewriting, we are able to express even sophisticated optimizations in a theoretically well-founded way such that correctness of the optimized version can be shown with well-known proof techniques.

We concentrate on the implementation of a *process specification* by another, and provide a proof method for this. Proving the validity of an implementation then amounts to establishing properties similar to the ones required for classical implementation proofs, together with properties stating that no more interleavings are *observed* in the implementation than in the specification using the observers of the latter (*serializability* condition).

Several examples have been developed, among which a distributed object oriented language specified at two levels of abstraction.

J U N G L E E V A L U A T I O N

Hans-Jörg Kreowski
Universität Bremen

joint work with Annegret Habel and Detlef Plump

The evaluation of algebraic, functional or logical programs may be extremely slow. If the solved problems are hard, nothing can be done. But what if the framework must be blamed?

Consider, for example, the specification of a function generating a totally balanced binary tree (without labels) of height n if n is the input.

```
generate = nat + bintree +
   opns: GENERATE: nat → bintree
   eqns: GENERATE(0) = BIN(EMPTY,EMPTY)
         GENERATE(SUCC(N)) = BIN(GENERATE(N),GENERATE(N))
```

The evaluation of $\mathrm{GENERATE}(\mathrm{SUCC}^n(0))$ within the framework of term or tree rewriting consumes space (and time or a number of processors) exponential in n. This unfortunate behavior is easily avoided as long as the two identical subterms of the right-hand side of the second equation do not cause a duplication of computation.

The recipe is clear: Represent functional expressions by structures with shared substructures rather than by terms or trees. This is the basic idea of graph grammar approaches to the evaluation of functional expressions as studied by Ehrig, Padawitz, Rosen, Staples and - most recently - by Barendregt et al. Jungle evaluation, as introduced in this talk, provides a new and alternative proposal in this line of research. Some details are characteristic of jungle evaluation:

(1) Jungles are recursively defined (hyper)graphs so that structural induction is available.

(2) Jungle evaluation is intentionally related to the evaluation of algebraic specifications.

(3) Jungle evaluation can be seen as graph rewriting in the sense of the "Berlin" - approach so that various known results on graph grammar derivations can be applied.

(4) Jungle evaluation comprises modes of non-sequential rewriting.

NB: If you understand the intuition behind a "tree" in the sense of graph theory and computer science, you may think of a "jungle" as a "forest" of coalesced "trees".

The specification environment OBSCURE

Part II: Formal description

Thomas Lehmann
Fachbereich 10 – Informatik
Universität des Saarlandes
D — 6600 Saarbrücken

OBSCURE consists of a specification language together with an environment for it. The present talk is devoted to the description of the specification language.

Syntactically the *OBSCURE* specification language is a language of terms in the sense of [BHK 86]. Each term is characterized by its imported and exported signature, and is interpreted as a function sending an algebra over the imported signature into a set of algebras over the exported signature. It is shown that the context conditions of the language guarantee that this function is well-defined.

[BHK 86] Bergstra, J.A., Heering, J., Klint, P., "Module Algebra", *Report CS-RXXX, Centre for Mathematics and Comp. Sc., Amsterdam* (1986), submitted for publication

Soundness and Completeness of the Birkhoff Many-Sorted Equational Calculus

Vincenzo Manca and Antonino Salibra
Dipartimento di Informatica - Universitá di Pisa
Corso Italia 40 - 56100 Pisa
Italy

Abstract

Goguen and Meseguer (see [GM 81]) have shown that the many-sorted extension of the Birkhoff one-sorted equational calculus is unsound when many-sorted algebras with empty carrier sets are admitted. They proposed adorning equations with explicit declarations of variables, and gave sound and complete rules for a general equational calculus (for updated presentations, discussions and related results see [GM 85], [GM 86], [ELM 86]).

In this work we present another approach, alternative to that proposed by Goguen and Meseguer that makes sound and complete the many-sorted extension of the Birkhoff equational calculus. The possibility of maintaining the classical rules is obtained by introducing a new notion of satisfiability, called *strong* satisfiability. In fact, we are able to prove the following main theorem:

Theorem 1 (*Soundness and Completeness*) Let E be a set of equations on a set V of variables.

$$E \models_S t_1 \equiv t_2 \quad \text{iff} \quad E \vdash_{Bir} t_1 \equiv t_2,$$

where $\models_S$ is the strong satisfiability and $\vdash_{Bir}$ is the Birkhoff many-sorted equational calculus.

Now we summarize the other results presented in the paper.

We study an easier notion of satisfiability $\models_W$, called *weak* satisfiability, and give sufficient and necessary conditions in order to reduce the strong satisfiability to the weak one. We provide two different kinds of results: the first ones relate to the syntactic structure of equations, the other ones relate to signatures. In fact, ***strongly sensible signatures***, which generalize sensible signatures introduced by Huet and Oppen (see [HO 80]), and ***uncritical equations*** are the tools by means of which we obtain the equivalence between the strong and the weak satisfiability. The following theorem synthesizes some results about.

Theorem 2 Let Σ be a signature. The following conditions are equivalent:

(i) Σ is strongly sensible;

(ii) For any Σ-algebra $\mathcal{A}$ and any terms $t_1,t_2 \in T_\Sigma(V)$, the equation $t_1 \equiv t_2$ is uncritical in $\mathcal{A}$;

(iii) For any Σ-algebra $\mathcal{A}$ and any terms $t_1,t_2 \in T_\Sigma(V)$, $\mathcal{A} \models_W t_1 \equiv t_2$ iff $\mathcal{A} \models_S t_1 \equiv t_2$.

We give also an analysis of several satisfiability notions presented in the literature, and a comparison with the ours. Besides, strongly sensible signatures guarantee both the equivalence between the strong satisfiability and the $\models_{HO}$ satisfiability (introduced by Huet and Oppen in [HO 80]), and the completeness of the Birkhoff equational calculus for the $\models_{HO}$ satisfiability. The main results of this analysis is constituted by the following theorems.

Theorem 3 Let Σ be a signature. Then, Σ is strongly sensible iff, for any Σ-algebra $\mathcal{A}$ and Σ-equation $t_1 \equiv t_2$ on variables V,

$$\mathcal{A} \models_S t_1 \equiv t_2 \text{ iff } \mathcal{A} \models_{HO} t_1 \equiv t_2.$$

Theorem 4 The Huet and Oppen satisfiability verifies the Birkhoff completeness theorem relatively to a signature Σ *iff* Σ is a strongly sensible signature.

Finally, we introduce the notion of *invariant presentation* which yields the following theorem.

Theorem 5 Let Σ be a strongly sensible signature. The Birkhoff calculus and the Goguen and Meseguer calculus are equivalent with respect to the presentation (Σ,V,E) *iff* the presentation (Σ,V,E) is invariant.

As a corollary, we obtain an interesting theorem of [GM 85] that characterizes the signatures Σ for which the two calculi are equivalent with respect to *any* presentations (Σ,V,E).

References

[ELM 86] Ehrig H., Loeckx J. & Mahr B.: *A Remark on the Equational Calculus for Many-Sorted Algebras with Possibly Empty Carrier Sets*, Bulletin of the EATCS, n.30, October 1986.

[GM 81] Goguen J.A. & Meseguer J.: *Completeness of Many-Sorted Equational Logic*, SIGPLAN Notices, Vol.16, n.7, 1981, pp. 24-32.

[GM 85] Goguen J.A. & Meseguer J.: *Completeness of Many-Sorted Equational Logic*, Houston Journal of Mathematics Vol.11, n.3, 1985, pp.307-334. Preliminary version is appeared in: SIGPLAN Notices,Vol.16, n.7,1981, pp.24-32.

[GM 86] Goguen J.A. & Meseguer J.: *Remarks on Remarks on Many-Sorted Equational Logic*, Bulletin of the EATCS, n. 30, October 1986.

[HO 80] Huet G. & Oppen D.C.: *Equations and Rewrite Rules: a Survey*, in R. Book (ed), Formal Language Theory: Perspective and Open Problems, Academic Press, 1980, pp. 349-405.

Specification and design of concurrent programs

Dominique Mery
Univ. de Metz

We present an extension of the system due to Van Nguyen, Gries and Owicki and try to integrate it into our proof systems. We think that our proof system can be seen as a specification system and we try to solve some combinatorial aspects due to the interferences. Our final goal is to find a new approach more appropriate to the concurrent programming. The algebraic framework seems to offer a good frame.

Combining Facets of Actions

Peter D. Mosses

Computer Science Department
Aarhus University
Ny Munkegade
DK–8000 Aarhus C, Denmark

Action Notation [1,2,3] has various sub-notations, called *facets*, for functional, imperative, binding and communicative actions. Each facet has a tractable (limiting complete) algebraic specification. These are to be combined to give a (limiting complete) algebraic specification of the full action notation.

The approach taken is to identify actions with *synchronized tuples* of single-faceted actions. The specifications, although algebraic, are reminiscent of Structural Operational Semantics: a subsort of actions corresponds to "configurations".

[1] Peter D. Mosses. Modularity in action semantics. 1987. Preliminary version, presented at a CSLI Workshop on Semantic Issues in Human and Computer Languages, Half Moon Bay, California, March 1987. Available from the author.

[2] Peter D. Mosses and David A. Watt. The use of action semantics. In *Proc. IFIP TC2 Working Conference on Formal Description of Programming Concepts III (Gl. Avernæs, 1986)*, North-Holland, 1987.

[3] David A. Watt. An action semantics of Standard ML. 1987. To appear in Proc. MFPLS'87.

Completeness of Implementation Criteria for Nondeterministic Data Types

Tobias Nipkow
Department of Computer Science
The University
Manchester M13 9PL

The implementation relationship between nondeterministic data types is defined in terms of the induced implementation relationship between the program that use the data types. Thus the notions of *loose*, *robust* and *partial* implementations carry over from programs to data types.

Three variations on the simulation relationship (half a bisimulation) between data types are shown to be sound implementation criteria. However not all of them are complete. The relationship between the soundness and completeness of implementation criteria and the expressiveness of the observing programming languages is discussed.

Using multiple views for software specification

Celso Niskier
Department of Computing
Imperial College of Science and Technology

Recently attention has been given to the idea of combining specification styles according to problem characteristics. In our approach, specifications written in different languages (or even logics) are seen as different theories expressing particular *views* of reality, each one comprised of limited aspects (properties) of the object to be modeled. The use of multiple views seems to capture the essence of abstraction hidden in the choice of formalisms. The work on PRISMA (Portugese for prism) is an attempt to investigate these ideas, through the formalization of heuristics for association between different views. In this context, heuristics can be understood as partial functions relating sets of properties in one view to sets of properties in another view. Therefore, they represent a dynamic account of the problem of interpretation between theories in different logics. The PRISMA paradigm is illustrated by means of an example, and further developments are also discussed.

Behavioral semantics for algebraic specification languages

M. Pilar Nivela, Fernando Orejas
Facultat d'Informàtica, Universitat Politècnica de Catalunya
Barcelona, Spain

Based on the notion of behavioural equivalence introduced in [1], we explore the constructs needed to give a complete semantic definition to an algebraic (equational) specification language.

Essentially, what we consider the adequate *institution* to deal with behavioural semantics is presented. Then, the semantics (at the model and at the specification levels) of some of the most usual constructs, especially parameter passing, are defined and compatibility results are shown. The ideas to be presented seem to put together in a nice way notions as implementation and parameterization.

[1] Meseguer, J. - Goguen, J. "Initiality, Induction and Computability". Algebraic Methods in Semantics, M. Nivat and J. C. Reynolds Eds. , Cambridge University Press, 1.985, pp. 459-541.

Abstract Data Types Meet Knowlegde Representation for Natural Language Understanding

Christoph Beierle and Udo Pletat
IBM Deutschland GmbH
Bereich Wissenschaft - LILOG
Postfach 80 08 80
D-7000 Stuttgart 80
West Germany

Abstract

In the recent years unification based grammar formalisms have been accepted as a powerful technique for analyzing natural language.
Labelled graphs appear as representations of the results of the syntatic analysis as well as the semantic construction extracting the meaning from natural language texts. These graphs can be considered to define sets together with a collection of attributes. This leads to many-sorted knowledge representation formalisms forming the basis of natural language understanding systems.

Many-sorted equational logic constitutes the logical backbone of abstract data type specifications. Motivated by this link we present the knowledge representation language L_{LILOG}. This language integrates

- sort defining graph structures
- order-sorted first order predicate logic
- equational data type specification.

For the Horn logic kernel of L_{LILOG} we introduce an initial semantics as well as its operational semantics based on SLDE-resolution and graph unification.
This language constitutes a logic programming language offering data types and inheritance.

Relating Subsorting and Partiality

Axel Poigné

Dept of Computing
Imperial College
London SW7 2BZ[1]

Subsorting eliminates a certain amount of partiality, e.g. in

```
spec STACK² is
sorts $data, nestack < stack
ops  empty :  → stack
       push : stack  data → nestack
       pop : nestack → stack
       top : nestack → data
var  s : stack, d : data
eqns   pop(push(s,d)) = s
         top(push(s,d)) = d.
```

Though elegant the specification refuses terms such as pop(pop(push(push(empty,d),d') which may well have a meaning in that it reduces to pop(push(empty,d). This observation suggests to split a specification into a "syntax" and "data" component

```
spec STACK  is
syntax sorts  DATA, STACK
        ops  empty :  → STACK
                push : STACK  DATA → STACK
                pop : STACK → STACK
                top : STACK → DATA
sorts $data < DATA, nestack < stack < STACK
data     ops  empty :  → stack
                push : stack  data → nestack
                pop : nestack → stack
                top : nestack → data
var  s : stack, d : data
eqns   pop(push(s,d)) = s
         top(push(s,d)) = d.
```

Terms are **well-formed** if they are syntactically correct with regard to the syntax signature. Well-

---

[1] New address: GMD F2G2, Postfach 1240, Scloß Birlinghofen, D-5205 Sankt Augustin 1
em: ap @ gs2.gmd.uucp

[2] To anticipate a common misunderstanding: stacks are most of the time used to point out the deficiencies of algebraic specifications.

formed terms are defined only if they are equivalent to a **well-typed** term, i.e. a term syntactically correct with regard to the data signature. Pragmatically, well-formedness can be checked at compile time while well-typedness can in general be established at run time only. Classified algebra reformalise this intuition by reintroducing partiality via unary **type predicates**.

spec STACK **is**
sorts <u>DATA</u>, <u>STACK</u>
ops empty : → <u>STACK</u>
push : <u>STACK</u> <u>DATA</u> → <u>STACK</u>
pop : <u>STACK</u> → <u>STACK</u>
top : <u>STACK</u> → <u>DATA</u>
types $_ε<u>data</u> : <u>DATA</u>, _ε<u>nestack</u> , _ε<u>stack</u> : <u>STACK</u>
var s : <u>STACK</u>, d : <u>DATA</u>
axioms s ε <u>nestack</u> ⊢ s ε <u>stack</u>
⊢ empty ε <u>stack</u>
s ε <u>stack</u>, d e <u>data</u> ⊢ push ε <u>nestack</u>
s ε <u>stack</u> ⊢ pop ε <u>stack</u>
s ε <u>stack</u> ⊢ top ε <u>data</u>
s ε <u>stack</u>, d e <u>data</u> ⊢ pop(push(s,d)) = s
s ε <u>stack</u>, d e <u>data</u> ⊢ top(push(s,d)) = d

The talk discusses the semantics of such specifications based on the Scott-Fourman view of partiality and relates classified specifications to other concepts of subsorting[3].

---

3 An extended abstract with the same title is available.

On operational semantics of behavioural canons

Horst Reichel
TU Magdeburg
Department of Mathematics
PSF 124
Magdeburg, DDR - 3010

Behavioural canons have two properties of special interest with respect to operational semantics: equational partiality and behavioural validity of conditional equations. We investigate the consequences of these properties to the conditional narrowing algorithm developed by H. Hussman. It turns out that equational partiality causes some problems. These problems arise from the facts that first the reflexivity of the rewrite relation is only valid for evaluable terms, and that second the most general unifier of two terms may be a substitution producing nonevaluable terms even if there are unifiers which produce evaluable terms. Consequently, the conditional narrowing algorithm is no longer an iteration of the narrowing step followed by a final application of the unification step. The necessity of proving that a substitution produces evaluable terms brings in a second recursion. On the other side, behavioural validity simplifies the operational semantics, since rule application may be restricted to terms which produce values in visible sorts. In this way operational semantics works in the initial algebra of the equivalence class of behaviourally equivalent algebras.

Mathematics versus implementations of partial evaluators

Marek Ryćko
Institute of Computer Science
Polish Academy of Sciences
P.O. Box 22
00-901 Warsaw PKiN, Poland

The talk will be devoted to the mathematical analysis of the concept of partial evaluation.

First, an idea of partial evaluation will be introduced informally on several examples. The examples will illustrate a basic difference between trivial and nontrivial partial evaluation programs. Then several variants of partial evaluators will be defined formally and their possible applications will be discussed. These applications include compiling, compiler generation and the generation of many other interesting programs. Different partial evaluators will be compared with respect to their applicational power.

The talk will be concluded with remarks about the state of the art of the existing implementations of partial evaluators. Possible research directions in the field will be discussed as well.

Implementations revisited

Don Sannella
Department of Computer Science
University of Edinburgh

The program development process is viewed as a sequence of implementation steps leading from a specification to a program. Based on an elementary notion of refinement, two notions of implementation are studied: constructor implementations which involve a construction "on top of" the implementing specification, and abstractor implementations which additionally provide for abstraction from some details of the implemented specification. These subsume most formal notions of implementation in the literature. Both kinds of implementations satisfy a vertical composition and a (modified but still acceptable) horizontal composition property. All the definitions and results generalize to the framework of an arbitrary institution, and it is possible to change institutions at any point in the program development process, implementing a specification from one institution by a specification in another institution.

Predicate-event nets for reasoning about nonsequential behaviour

Heinz Wilhelm Schmidt
GMD Birlinghoven
St Augustin, FRG

High-level Petri nets and algebraic specification are combined in Predicate- event (PrE) nets. The states and actions of these nets are predicates interpreted by operations of an algebra. PrE nets lend themselves for verifying lifeness and safeness of non-sequential behaviour on a high-level. The correct implementation of such specifications almost in the standard algebraic sense preserves lifeness and safeness.

Data Abstraction and the Correctness of Modular Programming

Oliver Schoett

Institut für Informatik, Technische Universität München,
Postfach 20 24 20, 8000 München 2, West Germany

A theory of data abstraction in modular programming is outlined that explains why this technique leads to correct programs.

Data abstraction allows users and implementers to take different views of a specification: While users may depend on a specification as it is, implementers need not provide program entities that satisfy the specification, but merely a "representation" of such entities. This means that the users may be supplied with program entities that do not satisfy the specification, and so it is necessary to explain why their code functions correctly nevertheless.

The theory shows that data abstraction leads to correct programs if the modules of the programs are "stable", and it is suggested that programming languages for data abstraction should guarantee stability of their modules. The stability criterion corresponds closely to the intuitive idea of "limited access" to encapsulated data types and to "representation independence" properties of the typed λ-calculus.

The theory is developed in the general framework of an "institution" and uses an abstract notion of "representation". Specifically, the institution of partial many-sorted algebras is considered and the representation relations "behavioural inclusion", "behavioural equivalence", and "standard representation" (the popular concept based on abstraction functions) are studied. These representation relations are characterized by certain many-sorted relations between algebras, called "correspondences", which provide useful practical proof methods for the correctness of data representations.

Behavioural equivalence is found to be superior to standard representation in that "representation bias" of a specification no longer restricts its range of implementations, and in that it allows more constructs to be included in a data abstraction programming language.

Reference

Schoett, Oliver: Data Abstraction and the Correctness of Modular Programming. PhD Thesis, Department of Computer Science, University of Edinburgh 1987 (= CST-42-87, also published as ECS-LFCS-87-19).

Specification and Verification of Hardware using OBJ

Vicky Stavridou
Department of Computer Science
University of Manchester
Oxford Road
Manchester, M13 9PL

July 22, 1987

Abstract

OBJ is an executable specification language based on equationally defined abstract data types. So far, it has been used successfully in a number of software engineering projects. This talk will concentrate on OBJ as a unified framework in which hardware designs can be specified, simulated and verified.

Combinational devices which are usually modelled as first order functions can easily be specified and simulated in OBJ. Specifying sequential devices requires higher order functions since their input and output signals are functions from times to values. Although OBJ does not support higher order functions, sequential circuits can be modelled easily, by exploiting the powerful parameterisation facilities of the language.

More interestingly, formal verification of circuits can be carried out within OBJ as well, thus eliminating the need for a separate theorem prover. This is achieved by using Hsiang's axiomatisation of the propositional calculus. Signals are then viewed as mappings from times onto propositions. Theorems about circuits (properties and/or equivalence) are formulae in this axiomatisation, which are reduced to truth (or falsehood) using OBJ's rewrite rule engine.

Early experiments with this technique have yielded encouraging results. We have successfully verified Dtype and RS flip-flops, delay units, hierarchies of adders, a twisted ring counter, a CMOS adder etc. We are currently experimenting with verification of circuits which require inductive arguments. This work was performed in collaboration with Dr J A Goguen at SRI International and is part of the HAVE (HArdware VErification) project at the Department of Computer Science, University of Manchester.

Algebraic specifications with built-in domain constructors

Bernhard Möller, Andrzej Tarlecki and Martin Wirsing

We study some theoretical aspects of the algebraic specifications of higher-order data types, where data of some sorts are defined explicitly as functions between lower-level data. It should be stressed that this extension of the usual (first-order) algebraic specification method is essential for many applications. We show that some of the standard results (e.g. the existence of initial model) carry over. The key idea is to consequently use the "generation principle" and to consider reachable higher-order algebras.

However, the situation is different with terminal, fully abstract models. An additional condition is required to guarantee extensionality of the fully abstract model in the higher-order case.

We view this analysis as just an example of using simultaneously the techniques of algebraic specifications (axioms plus the initiality or terminality requirement) with those of denotational semantics (explicit definition of function spaces and of the corresponding application operations). We show that similar results hold, for example, for the standard power-set construction, and give a more general, abstract formulation of such a mixed technique.

AN ABSTRACT DATA TYPE
FOR STRUCTURED SYSTEM DEVELOPMENT TOOLS *

T.H. Tse

Centre of Computer Studies & Applications

University of Hong Kong

Hong Kong

Structured system development methodologies are currently recognized as the most popular tools in information system development. A complex system can be specified in a top-down and graphical fashion, thereby enabling practitioners to visualize the target systems and communicate with users much more easily than by means of traditional methods.

In fact, the structured methodologies have been designed by quite a number of distinct

* This project is supported in part by a Hong Kong & China Gas Research Grant, a CICHE Visitorship of the British Council and a University of Hong Kong Conference Grant.

authors, each employing a number of tools which are different in graphical outlook. Examples are DeMarco data flow diagrams, Jackson structure diagrams, Jackson structure text, system specification diagrams, system implementation diagrams, Warnier/Orr diagrams and Yourdon structure charts. Different tools are found to be suitable for different stages of a typical system life cycle. In other words, a specification must be converted from one form to another during the development process. Unfortunately, however, the tools are only manual systems derived from the experience of the authors. Little attempt has been made in proposing a formal framework behind them. The transition from one tool to another is only done manually. Automatic development aids tend to be *ad hoc* and tool-dependent. There is a need, therefore, in providing a unified framework for the structured tools and a computed-aided means of mapping one tool to another.

To solve the problem, an abstract data type has been defined as a unifying framework behind the popular structured tools. It links up DeMarco data flow diagrams, Jackson structure text and Yourdon structure charts by means of homomorphisms and equations. These three structured tools have been selected for study because they have entirely different physical appearances: they are in the forms of flowgraphs, trees and texts respectively. A prototype system to implement the abstract data type has been developed. It enables users to draw a hierarchy of DeMarco data flow diagrams, review them to an appropriate level of detail, and zoom in/zoom out to lower/higher levels when required. It stores the data flow diagrams internally as abstract data types, and then converts them automatically into Yourdon structure charts and Jackson structure text. The system has been implemented on a Macintosh Plus using Turbo Pascal.

Because of the linkage between abstract data types and initial algebras, the specifications can be validated using algebraic interpreters and the correctness of programs proved through algebraic theory. For example, a UMIST OBJ has been used to validate the abstract data type generated by the prototype system. On the other hand, the mathematical concepts can be transparent to system designers who do not want to be involved with complex theories.

The Algebraic Specification of Semicomputable Algebras.

J.Vrancken.

Abstract.

The notion of a (semi-)computable algebra was introduced by Rabin and Mal'cev in the early sixties. It means that the algebra can be written as the surjective image of a number algebra in which the functions are recursive and the equality relations are recursive or recursively enumerable. In computer science many-sorted algebras are used as a means to model data-types and ever since the early seventies the question has been studied how to specify algebras by means of equations. The question which algebras can be specified, can be answered to a certain extent by referring to the computability properties of an algebra. It soon turns out that even very simple algebras can not be specified right away but need the use of extra (hidden) functions or even extra (hidden) sorts. For instance the natural numbers with zero, successor and squaring. Bergstra and Tucker have proven that computable algebras can always be specified with extra functions, they gave an upper bound for the number of extra functions and they gave a proof of the specifiability of single-sorted semicomputable algebras. In our paper we give a proof for the many-sorted case from which follows an upper bound of atmost one extra sort.

The proofs runs roughly as follows : for the semic. algebra A with signature Σ we take a coordinatization $\alpha : C \rightarrow A$ in which C has only one sort (existence follows by means of Gödel numberings). The inverse images under α of the identity relations on the sorts of A are recursively enumerable (by definition of semicomputability) and can be enumerated by recursive functions. Now we can construct an algebra B as the disjoint union of A and C together with the function α and the enumeration functions. A is obviously the restriction of B to the signature Σ and the function α happens to be specifiable by means of the enumeration functions. This makes the whole of B specifiable without extra sorts.

RECURSIVE TYPES and POINTERS
(Abstract)

Eric G. Wagner
Mathematical Sciences Department
IBM T. J. Watson Research Center
Yorktown Heights, NY / USA

The paper presents a rigorous algebraic formulation, and proof, of the folk theorem to the effect that all the recursive types defined using "products and sums" (e.g., NATURAL-NUMBERs, STACKs, TREEs, etc.) can be implemented using pointers. We give an algebraic formulation of recursive types and their operations. A collection of recursive types, together with their inherent operations, form a special kind of algebra. We also give an algebraic formulation of imperative programming languages with pointers and variables. Roughly speaking, a memory state corresponds to an algebra, and any operation which "creates pointers" corresponds to a transformation on algebras. Thus the two mathematical frameworks are rather different. What we show though is that "the usual" implementation of recursive types by pointers works, from a mathematical point of view, because, in a sense to be explained in the full paper and sketched below, it induces an algebra which is homomorphic to the desired algebra of recursive types.

1. Recursive Types: A *recursive type specification*, $\mathscr{R}$, consists of disjoint finite sets G, R, and V, a designated object $1 \in G$., and, where $S = G \cup R \cup V$, a mapping $\mu: R \cup V \to S \otimes S$. Intuitively the elements of G are *given* or *ground*, types, the elements of R and V are recursively defined *record* and *variant* types, respectively.

Define $\sum(\mathscr{R})$, the *signature for* $\mathscr{R}$, to be the $(S^* \times S)$-indexed family of sets $\sum(\mathscr{R}) = < \sum(\mathscr{R})_{w,s} \mid w \in S^*, s \in S >$ where, for each $r \in R$, $\sum(\mathscr{R})_{r,\mu(r)_1} = \{ p_r: r \to \mu(r)_1 \}$, $\sum(\mathscr{R})_{r,\mu(r)_2} = \{ q_r: r \to \mu(r)_2 \}$; for each $v \in V$, $\sum(\mathscr{R})_{\mu(v)_1,v} = \{ \kappa_r: \mu(v)_1 \to v \}$, $\sum(\mathscr{R})_{\mu(v)_2,v} = \{ \lambda_r: \mu(v)_2 \to v \}$; and $\sum(\mathscr{R})_{w,s} = \emptyset$, for all other w and s. Given an *assignment* $\alpha: G \to |\mathbf{Set}|$, let $\mathscr{R}(\alpha)$ be the "least", partial $\sum(\mathscr{R})$-algebra, A (i.e. $\sum(\mathscr{R})$-algebra defined over the category, **Pfn**, of sets and partial functions), such that, A_1 is "the" singleton set; for each $r \in R$, $A_r = A_{\mu(r)_1} \otimes A_{\mu(r)_2}$ (Cartesian product) with projections $(p_r)_A: A_r \to A_{\mu(r)_1}$ and $(q_r)_A: A_r \to A_{\mu(r)_2}$; and, for each $v \in V$, $A_v = A_{\mu(v)_1} + A_{\mu(v)_2}$ (coproduct) with injections $(\kappa_v)_A: A_{\mu(v)_1} \to A_v$ and $(\lambda_v)_A: A_{\mu(v)_2} \to A_v$.

Example: Take $G = \{1\}$, $R = \emptyset$, $V = \{N\}$, and $\mu(N) = \{ <1,N> \}$, then $\mathscr{R}(\alpha)$ is the algebra of the natural numbers with $\kappa_{\mathscr{R}(\alpha)} = 0$, and $\lambda_{\mathscr{R}(\alpha)} = \text{suc}$, the successor function.

We exploit the universal properties of the coproduct, the existence of fixpoints in **Pfn**, and the distributive law $((A+B) \otimes C \cong (A \otimes C) + (B \otimes C))$, to define *derived operators* in addition those defined in terms of composition and product tupling. For example, writing $[f, g]: A+B \to C$ for the coproduct mediator of $f: A \to C$ and $g: B \to C$, we can define the predecessor function on N, as pred $= [0, 1_N]: (1+N) \to N$, and we can define the addition function, add, as the fixpoint add $= [\pi_2, \text{add} \circ \text{suc}]: ((1+N) \otimes N) \to N$, where π_2 denotes the projection, $\pi_2: (1 \otimes N) \to N$. (To make this fully precise we need the framework given in [1].)

2. Pointer Algebras: A *Pointer presentation* consists of a recursive type specification, $\mathscr{R} = < G, R, V, 1, \mu >$ together with a relation ρ on $G+R+V$. An $<\mathscr{R}, \rho>$-algebra consists of an $\mathscr{R}$-algebra A, with an additional carrier,

$$A_M = \prod <(A_s \to A_t) \mid (s,t) \in \rho>$$

(so that for $m \in A_M$, $m = < m_{s,t}:A_s \to A_t \mid (s,t) \in \rho >$) and, for each pair $(s,t) \in \rho$, an operation $Pr_{s,t}A_M \times A_s \to A_t$ such that $Pr_{s,t}(m, a) = m_{s,t}(a)$, plus a designated element σ of A_M. Think of $(s,t) \in \rho$ as meaning that "the elements of A_s are pointers to elements of A_t"; think of A_M as the set of all possible memory states; think of $Pr_{s,t}$ as the operation which, given a memory state m and a pointer p, fetches the value in m pointed at by p; and, think of σ as the current memory state.

In this framework, the action of "creating a pointer and pointing it at a given value" can be captured by a ρ-indexed family, NEW, of operations such that, given an $<\mathscr{R}, \rho>$-algebra A, $<s, t> \in \rho$, and $a \in A_t$, then $NEW_{s,t}(A, a)$ is the $<\mathscr{R}, \rho>$-algebra, B, that results from freely adjoining a new element, $\nu = \nu(A,a)$ to A_s (so, $B_s = A_s + 1$, and other carriers change accordingly) and taking σ_B to be the extension of σ_A in which $\sigma_B(\nu) = a$.

3. Implementing recursive types using pointers: Given $\mathscr{R} = <G, R, V, 1, \mu>$, we implement it by means of a pointer presentation $Pt(\mathscr{R}) = <<G+R+V, R, V, 1, \mu^{\#}>, \rho>$ which results when, for each $X \in R \cup V$, we add a new sort $\uparrow X$, form $\mu^{\#}$ by changing the X's in the target of μ to $\uparrow X$'s, and put $<X\uparrow, X>$ in ρ. This corresponds to a common pointer implementation of recursion.

Given $\alpha:G \to |\mathbf{Set}|$, let $I(\mathscr{R}, \alpha)$ be the $Pt(\mathscr{R})$-algebra given by the assignment $\beta:G+R+V \to |\mathbf{Set}|$ such that $\beta(g) = \alpha(g)$, and $\beta(r) = \beta(v) = \emptyset$. $I(\mathscr{R}, \alpha)$ corresponds to the "memory state before any pointers have been created."

Define $\mathbf{R}(\mathscr{R}, \alpha)$ then to be the set of all $Pt(\mathscr{R})$-algebras reachable from $I(\mathscr{R}, \alpha)$ by repeated applications of $NEW_{s,t}$.

Given $\mathscr{R} = <G, R, V, 1, \mu>$, and $\alpha:G \to |\mathbf{Set}|$ define $\mathbf{B} = \mathbf{B}(\mathscr{R}, \alpha)$ to be the $\sum(\mathscr{R})$-algebra where for each $s \in G \cup R \cup V$, $\mathbf{B}_s = \{ (A, a) \mid A \in \mathbf{R}(\mathscr{R}, \alpha), \text{ and } a \in A_s \}$. and, for any $r \in R$, or $v \in V$,

$$(p_r)_B(A, a) = \begin{cases} (A, \sigma_A((p_r)_A(a))) \text{ if } \mu(r)_1 \in R \cup V \\ (A, (p_r)_A(a)) \text{ if } \mu(r)_1 \in G \end{cases}$$

$$(\kappa_v)_B(A, a) = \begin{cases} (NEW(A,a), (\kappa_v)_{NEW(A,a)}(\nu(A, a))), \text{ if } \mu(v)_1 \in R \cup V \\ (A, (\kappa_v)_A(a)) \text{ if } \mu(v)_1 \in G \end{cases}$$

and q_r, and λ_v are similarly defined.

Note that while **B** is a $\sum(\mathscr{R})$-algebra, it is not an $\mathscr{R}$-algebra since we do not have, for example, that $\mathbf{B}_r \cong \mathbf{B}_{\mu(r)_1} \otimes \mathbf{B}_{\mu(r)_2}$. However we do have the following.

Proposition: There exists a surjective $\sum(\mathscr{R})$-homomorphism $h:\mathbf{B} \to \mathscr{R}(\alpha)$. □

This is still not enough. However, by further exploiting the structure underlying **B**, (using the framework developed in [1]) we can enrich **B** by adding a small number of operations (and operators), corresponding, on **B**, to operations on pointers, and, under h to the basic constructions used to build the derived operators in $\mathscr{R}(\alpha)$. This gives us the following result.

Theorem: For any derived operator expression $e(x_1,\dots,x_n)$ in $\mathscr{R}(\alpha)$ there exists in derived operator $E(x_1,\dots,x_n)$ in **B** (extended) such that for any $A \in \mathbf{R}$, and $a_1,\dots,a_n$ in A

$$h(E_B((A, a_1),\dots,(A, a_n)) = e_{\mathscr{R}(\alpha)}(h(A, a_1),\dots,h(A, a_n)).$$ □

[1] Wagner, E. G., "Semantics of Block Structured Languages with Pointers," Proceedings of the 3rd Annual Workshop on the Mathematical Foundations of Programming Language Semantics, April 1987, New Orleans, LA. (To appear in Springer LNCS).