Algebraic Specification and Functionals for Transformational Program and Meta Program Development[1]

Bernd Krieg-Brückner

FB3 Mathematik und Informatik, Universität Bremen
Postfach 330 440, D 2800 Bremen 33, FR Germany
email: bkb@sun1.informatik.uni-Bremen.de

A methodology for PROgram development by SPECification and TRAnsformation is described. Formal requirement specifications are the basis for constructing *correct* and efficient programs by gradual transformation.

A uniform treatment of algebraic specification is presented to formalise data, programs, transformation rules, in fact the program development process itself. It is shown by example that the development of meta programs, for example an efficient transformation algorithm incorporating the effect of a set of transformation rules, is analogous to program development: the transformation rules act as specifications for the transformation algorithms, and the negation of their applicability conditions as development goals.

The paper focusses on the combination of functional programming and algebraic specification and reasoning, leading to a considerably higher degree of abstraction, avoiding much repetitive development effort, the use of homomorphic extension functionals, homomorphic properties for reasoning about correctness and optimisation, and the recursion schema of homomorphic extension as a program development strategy and as an induction schema for proofs.

1. Introduction

1.1. The PROSPECTRA Methodology of Program Development

The project "PROgram devlopment by SPECification and TRAnsformation aims to provide a rigorous methodology for developing *correct* Ada software and a comprehensive support system. It is a cooperative project between Universität Bremen, Universität Dortmund, Universität Passau, Universität des Saarlandes (all D), University of Strathclyde (GB), SYSECA Logiciel (F), Dansk Datamatik Center (DK), and Alcatel Standard Electrica S.A. (E), and is sponsored by the Commission of the European Communities under the ESPRIT Programme, ref. #390 (see [1, 2]).

The *methodology* integrates program construction and verification during the development process. User and implementor start with a formal specification, the interface or "contract". This initial specification is then gradually transformed into an optimised machine-oriented executable program. The final version is obtained by stepwise application of transformation rules. These are carried out by the system, with interactive guidance by the implementor, or automatically by compact transformation tools. Transformation rules form the nucleus of an extendible knowledge base.

The strict methodology of Program Development by Transformation (based on the CIP approach, see e.g. [3, 4]) is completely supported by the *system*. Any kind of activity is conceptually and technically regarded as a transformation of a "program" at one of the system layers. This provides for a uniform user

[1] The research reported herein has been partially funded by the Commission of the European Communities under the ESPRIT Programme, Project #390, PROSPECTRA (PROgram development by SPECification and TRAnsformation)

interface, reduces system complexity, and allows the construction of system components in a highly generative way.

The PROSPECTRA methodology, its objectives, the development model, algebraic specification and transformational program development model are briefly summarised in chapter 2.

1.2. Algebraic Specification and Functionals

This paper focusses on a particular extension to classical algebraic specification: the use of functionals. Chapter 3 describes the combined advantages of functional programming and algebraic specification and reasoning: a considerably higher degree of abstraction, avoiding much repetitive development effort, the use of homomorphic extension functionals, homomorphic properties for reasoning about correctness and optimisation, and the recursion schema of homomorphic extension as a program development strategy and as an induction schema for proofs.

For the purpose of exhibiting the use of functionals, a notation with explicit Curry-ing to allow partial parameterisation is used. The system allows a switch from a more conventional Ada oriented notation to this more symbolic style using, for example, —> instead of **return**. The functional composition operator ∘ and the identity function id are assumed to be universally defined.

1.3. Formalisation of Transformational Program Development

Chapter 4 and 5 deal with meta-program development, that is with the application of algebraic specification and functionals to the definition of program transformation tactics, in fact to a formalisation of the program development process itself.

Various authors have stressed the need for a formalisation of the software development process: the need for an automatically generated transcript of a development "history" to allow re-play upon re-development when requirements have changed, containing goals of the development, design decisions taken, and alternatives discarded but relevant for re-development [29]. A *development script* is thus a formal object that does not only represent a documentation of the past but is a plan for future developments. It can be used to abstract from a particular development to a class of similar developments, a *development method*, incorporating a certain strategy. Approaches to formalise development descriptions contain a kind of development program [29], regular expressions over elementary steps [30], functional abstraction [31] and composition of logical inference rules [32, 33].

In Program Development by Transformation [3-5], an elementary development step is a *program transformation*: the application of a transformation rule that is generally applicable; a particular development is then a sequence of rule applications. The question is how to best formalise rules and application (or inference) strategies.

The approach taken in this paper is to regard transformation rules as equations in an algebra of programs, to derive basic transformation operations from these rules, to allow composition and functional abstraction, and to regard development scripts as (compositions of) such transformation operations. Using all the results from program development based on algebraic specifications and functionals we can then reason about the development of meta programs, i. e. transformation programs or development scripts, in the same way as about programs: we can define requirement specifications (development goals) and implement them by various design strategies, and we can simplify ("optimise") a development or development method before it is first applied or re-played.

2. PROgram development by SPECification and TRAnsformation

2.1 Objectives

Current software developments are characterised by ad-hoc techniques, chronic failure to meet deadlines because of inability to manage complexity, and unreliability of software products. The major objective of the PROSPECTRA project is to provide a technological basis for developing *correct* programs. This is achieved by a methodology that starts from a formal specification and integrates verification into the development process.

The initial *formal requirement specification* is the starting point of the methodology. It is sufficiently rigorous, on a solid formal basis, to allow verification of correctness during the complete development process thereafter. The methodology is deemed to be more realistic than the conventional style of a posteriori verification: the construction process and the verification process are broken down into manageable steps; both are coordinated and integrated into an implementation process by *stepwise transformation* that guarantees a priori correctness with respect to the original specification. Programs need no further debugging; they are correct by construction with respect to the initial specification. Testing is performed as early as possible by *validation* of the formal specification against the informal requirements (e.g. using a prototyping tool).

Complexity is managed by abstraction, modularisation and stepwise transformation. Efficiency considerations and machine-oriented implementation detail come in by conscious design decisions from the implementor when applying pre-conceived transformation rules. A long-term research aim is the incorporation of goal orientation into the development process. In particular, the crucial selection in large libraries of rules has to reflect the reasoning process in the development.

Engineering Discipline for Correct SW: The PROSPECTRA project aims at making software development an engineering discipline. In the development process, ad hoc techniques are replaced by the proposed uniform and coherent methodology, covering the complete development cycle. Programming knowledge and expertise are formalised as transformation rules and methods with the same rigour as engineering calculus and construction methods, on a solid theoretical basis.

Individual transformation rules, compact automated transformation scripts and advanced transformation methods are developed to form the kernel of an extendible knowledge base, the method bank, analogously to a handbook of physics. Transformation rules in the method bank are proved to be correct and thus allow a high degree of confidence. Since the methodology completely controls the system, reliability is significantly improved and higher quality can be expected.

Specification: Formal specification is the foundation of the development to enable the use of formal methods. High-level development of specifications and abstract implementations (a variation of "logic programming") is seen as the central "programming" activity in the future. In particular, the development of methods for the derivation of constructive design specifications from non-constructive requirement specifications is a present focus of research.

The abstract formal (e.g. algebraic) specification of requirements, interfaces and abstract designs (including concurrency) relieves the programmer from unnecessary detail at an early stage. Detail comes in by gradual optimising transformation, but only where necessary for efficiency reasons. Specifications are the basis for adaptations in evolving systems, with possible replay of the implementation from development histories that have been stored automatically.

Programming Language Spectrum: Ada and Anna: Development by transformation receives increased attention world-wide, see [6]. However, it has mostly been applied to research languages. Instantiating the general methodology and the support system to Ada [7] and Anna (its complement for formal specification, see [8]) make it realistic for systems development including concurrency aspects. $PA^{nn}dA$, the PROSPECTRA Anna/Ada subset, covers the complete spectrum of language levels from formal specifications and applicative implementations to imperative and machine-dependent representations. Uniformity of the language enables uniformity of the transformation methodology and its formal basis.

Stepwise transformations synthesise Ada programs such that many detailed language rules necessary to achieve reliability in direct Ada programming are obeyed by construction and need not concern the program developer. In this respect, the PROSPECTRA methodology may make an important contribution to managing the complexity of Ada.

Research Consolidation and Technology Transfer: The PROSPECTRA project aims at contributing to the technology transfer from academia to industry by consolidating converging research in formal methods, specification and non-imperative "logic" programming, stepwise verification, formalised implementation techniques, transformation systems, and human interfaces.

Industry of Software Components: The portability of Ada allows pre-fabrication of software components. This is explicitly supported by the methodology. A component is catalogued on the basis of its interface. Formal specification in Anna gives the semantics as required by the user; the implementation is hidden and may remain a company secret of the producer.

Ada/Anna and the methodology emphasise the *pre-fabrication* of generic, universally (re-)usable, *correct* components that can be instantiated according to need. This will invariably cut down production costs by avoiding duplicate efforts. The production of perhaps small but universally marketable components on a common technology base will also assist smaller companies in Europe.

Tool Environment: Emphasis on the development of a comprehensive support system is mandatory to make the methodology realistic. The system can be seen as an integrated set of tools based on a minimal Ada Program Support Environment, e.g. the planned ESPRIT Portable Common Tool Environment (PCTE). Because of the generative nature of system components, adaptation to future languages is comparatively easy. Existing environments only support the conventional activities of edit, compile, execute, debug.

The support of correct and efficient transformations is seen as a major advance in programming environment technology. The central concept of system activity is the application of transformations to trees. Generator components are employed to construct transformers for individual transformation rules and to incorporate the hierarchical multi-language approach of PA^{nn}dA (PROSPECTRA Anna/Ada), TrafoLa (the language of transformation descriptions), and ControLa (the command language). Generators increase flexibility and avoid duplication of efforts; thus the overall systems complexity is significantly reduced. Choosing Ada/Anna as a standard language, and standard tool interfaces (e.g. PCTE), will ensure portability of the system as well as of the newly developed Ada software. A brief overview of the system can be found in [1]; see also [9-11].

2.2 The Development Model

Consider a simple model of the major development activities in the life of a program:

Requirements Analysis

- **Informal** Problem Analysis
- **Informal** Requirement Specification

Development

- **Formal** Requirement Specification ⇑ *Validation*
- **Formal** Design Specification ⇑ *Verification*
- **Formal** Construction *by Transformation* ⇑ *Verification*

Evolution

- Changes in Requirements ⇒ **Re-Development** ⇑

The *informal requirements analysis* phase precedes the phases of the *development* proper, at the level of formal specifications and by transformation into and at the level(s) of a conventional programming language such as Ada.

After the program has been installed at the client, no maintenance in the sense of conventional testing needs to be done; "testing" is perfomed *before* a program is constructed, at the very early stages, by validation of the formal requirement specification against the informal requirements.

The *evolution* of a program system over its lifetime, however, is likely to outweigh the original development economically by an order of magnitude. Changes in the informal requirements lead to re-development, starting with changes in the requirement specification. This requires re-design, possibly by replay of the original development (which has been archived by the system) and adaptation of previous designs or re-consideration of previously discarded design variations.

2.3 Algebraic Specification

A requirement specification defines *what* a program should do, a design specification *how* it does it. The motivations and reasons for design decisions, the *why*'s, are recorded along with the developments.

Requirement specifications are, in general, non-constructive; there may be no clue for an algorithmic solution of the problem or mapping of abstract to concrete (i.e. predefined) data types. It is essential that the requirement specification should not define more than the *necessary* properties of a program to leave room for design decisions. It is intentionally vague or *loose* in areas where the further specification of detail is irrelevant or impossible. In this sense, loose specification replaces non-determinancy, for example to specify an unreliable transmission medium in a concurrent, distributed situation [27, 28].

From an economic point of view, overspecification may lead to substantial increase in development costs and in efficiency of execution of the program since easier solutions are not admissable. If the requirement specification is taken as the formal *contract* between client and software developer, then there should perhaps be a new profession of an independent *software notary* who negotiates the contract, advises the client on consequences by answering questions, checks for inconsistencies, resolves unintentional ambiguities, but guards against overspecification in the interest of both, client and developer. The answer of questions about properties of the formal requirement specification correspond to a *validation* of the informal requirement specification using a prototyping tool.

Design specifications specify abstract implementations. They are constructive, both in terms of more basic specifications and in the algorithmic sense. If the requirement specification is loose and allows a set of models, then the design specification will usually restrict this set, eventually to one (up to isomorphism); it is then callled monomorphic.

(2.3-1) Abstract Type: Booleans (as in STANDARD)

```
package BOOLS is
   type BOOLEAN is private;
   false:    BOOLEAN;
   true:     BOOLEAN;
   "¬":      BOOLEAN ⟶  BOOLEAN;
   "∧":      BOOLEAN ⟶  BOOLEAN ⟶  BOOLEAN;
   "∨":      BOOLEAN ⟶  BOOLEAN ⟶  BOOLEAN;
axiom for all x, y, z: BOOLEAN =>
   true ≠ false,      ¬false = true,     ¬true = false,
   x ∧ false = false, x ∨ true = true,   x ∧ true = x,    x ∨ false = x,    x ∨ ¬x = true, x ∧ ¬x = false,
   ¬¬x = x,           x ∨ y = ¬(¬x ∧ ¬y),     x ∧ y = ¬(¬x ∨ ¬y),
   x ∧ (y ∧ z) = (x ∧ y) ∧ z,   x ∧ y = y ∧ x,   x ∧ x = x,   x ∧ (y ∨ z) = (x ∧ y) ∨ (x ∧ z),
   x ∨ (y ∨ z) = (x ∨ y) ∨ z,   x ∨ y = y ∨ x,   x ∨ x = x,   x ∨ (y ∧ z) = (x ∨ y) ∧ (x ∨ z);
end BOOLS;
```

As an example take the specification of Booleans (2.3-1). Several axioms here are redundant and can be derived from others. The axioms in the last two lines specify important properties of Booleans, but they are non-operational, whereas the other equations can be interpreted as rewrite rules (suitably modified and extended by dual equations corresponding to the commutativity axioms), see also (4.3-4) below. We might want to use completion techniques to derive operational functions. At the same time, the specification shows the limitations of the rewriting approach since the application of the idempotence law cannot be guaranteed in general.

The next example is the specification of lists in (2.3-2). Note that we may have two views of lists: either constructed by empty and cons or by empty, "&" and single. Depending on the view, lists have different algebraic properties; in the second case those of a monoid. This aspect becomes important when defining functions on them, as we will see below. The definition of the selectors head and tail insures uniqueness of models up to isomorphism.

(2.3-2) Abstract Type: Lists

```
generic
  type ITEM is private;
  eq:      ITEM —>  ITEM —>  BOOLEAN
           :: for all x, y, z: ITEM => eq x x,  eq x y = eq y x,  eq x y ∧ eq y z → eq x z;
package LISTS is
  type LIST is private;
  empty:  LIST;
  cons:   ITEM —>  LIST —>  LIST;
--empty:  LIST;                                  as before
  "&":    LIST —>  LIST —>  LIST;
  single: ITEM —>  LIST;
axiom for all x, y, z: LIST =>
  empty & x = x,   x & empty = x,   x & (y & z) = (x & y) & z;
  isEmpty: LIST —>                  BOOLEAN;
  head:    (x: LIST :: ¬ isEmpty x) —> ITEM;
  tail:    (x: LIST :: ¬ isEmpty x) —> LIST;
axiom for all e: ITEM; l: LIST =>
  isEmpty empty = true,      isEmpty (cons e l) = false,
  head (cons e l) = e,       tail (cons e l) = l,
  head ((single e) & l) = e, tail ((single e) & l) = l;
end LISTS;
```

(2.3-3) Abstract Type: Sets

```
generic
  type ITEM is private;
  eq:      ITEM —>  ITEM —>  BOOLEAN
           :: for all x, y, z: ITEM => eq x x,  eq x y = eq y x,  eq x y ∧ eq y z → eq x z;
package SETS is
  type SET is private;
  empty:     SET;
  "∪":       SET —>  SET —>  SET;
  singleton: ITEM —>  SET;
  "∈":       ITEM —>  SET —>  BOOLEAN;
axiom for all a, b: ITEM; x, y, z: SET =>
  x ∪ empty = x,  x ∪ (y ∪ z) = (x ∪ y) ∪ z,  x ∪ y = y ∪ x,  x ∪ x = x,
  a ∈ empty = false,   a ∈ (singleton b) = eq a b,   a ∈ (x ∪ y) = (a ∈ x) ∨ (a ∈ y);
end SETS;
```

Sets (2.3-3) are an example of a loose specification (cf. also the priority queues in [41]). The operations $\cup$ and $\in$ are intentionally loosely specified. Consider sets represented as lists, for a moment. If several

elements in a sequence of $\cup$'s are equivalent w. r. t. eq, these elements may or may not be multiply represented. Also, the order in which elements are added may or may not result in distinct model representations.

In the so-called *initial model* all terms are considered to be different unless they can be shown to be equal by equational reasoning (we are only considering the case of first order functions here). The SETs

$x \cup (y \cup z)$ and $(x \cup y) \cup z$ or $x \cup y$ and $y \cup x$ or $x \cup x$ and x

have distinct representations and a search is performed when $\in$ is applied.

In the *terminal model* all terms have the same representation unless they can be shown to be different. The resp. SETs in the pairs above have only one, the canonical representation; this means that a search has to be made for overlapping elements in both sets upon $\cup$.

Any model will do. Betwen the two extremes (searching all multiply represented elements upon $\in$ in the initial and eliminating multiple elements upon $\cup$ in the terminal model) lie other admissible models, for example one using a hash table. It should also be emphasised that not only a list-like implementation is possible as might be suggested. Consider the slight extension of an order relation being given on the elements. Then an implementation of an ordered set using, for example, a binary or balanced tree representation is admissable. Similarly, any search algorithm, for example binary search, is equivalent for the initial model. Note the analogy between binary search and a binary search tree: the same idea is once represented in the algorithm and once in the data structure.

Partial Functions: Note that head and tail are *partial functions,* they are only defined if the pre-condition on the parameter holds (cf. [13]). Similarly, a pre-condition on cons could be introduced, stating, for example, that the length should be less than some number MAX_SIZE. cons then becomes a *partial constructor function;* LISTS then defines bounded lists (cf. [41]). Corresponding definedness premisses must then also be included in the equations.

When introducing limitations such as bounds in a methodological step, it is desirable not to have to introduce the definedness premises explicitly; they should be included implicitly by a transformation. In fact it can be argued that they should not be shown explicitly in the text of the specification at all (see [14]) so that they will not clutter the definition of the "normal cases". They are necessary, however, when reasoning about the definedness of equations during program development and verification. The cluttering problem can be solved by subtypes used to abbreviate such conditions, possibly leading to a more efficient way of checking for definedness. See also [41, 15, 16] for the introduction of exceptions that arise from partialities in the operations and have their counterpart in the delay conditions of monitor tasks.

2.4 Correctness

Notions of Correctness: Various notions of correctness have been distinguished in the literature, in particular in an algebraic framework: total, partial, and robust correctness (see also [17, 18]). It is expected that all three will arise but each has a well-defined place in the methodology.

Partial Correctness: The *revision* of a specification of unbounded lists to one of bounded lists as described above implies a relationship of *partial correctness* of the latter to the former. A program not using lists longer than MAX_SIZE remains correct under the revision; in general, the pre-condition has to be proved for every call on cons to maintain (total) correctness. The generation of such verification conditions upon the revision can be automated by the system.

Implementation: Certain abstract type (schemata) that correspond to predefined Ada type (constructors, selectors, other auxiliary functions and their algebraic specification), for example **record**, are standard in PA^{nn}dA (see [19]). For example, the usual free term constructions (lists, trees) are available. They are implemented in Ada in a standard way and turned into an Ada text automatically as an alternative (standard Ada) notation for the package defining the abstract type. We assume that a standard Ada implementation using access types (pointers) and allocators is still considered to be "applicative" at this level of abstraction and that side-effects of allocation will be eliminated during the development process by explicit storage allocation whenever required.

Robust Correctness: Total correctness preserves the complete set of models. In practice, we are not so much interested in total correctness; the notion of implementation has to be generalised

(a) to allow a *smaller set of models* for the implementation for a loose requirement specification, and

(b) to allow that operations in the implementation are *more defined* (for example totally defined or raising exceptions) than those in the requirement specification, cf. [41] for an example.

Integration of Construction and Verification: Not only is the program construction process formalised and structured into individual mechanisable steps, but the verification process is structured as well and becomes more manageable. If transformation rules are correctness-preserving, then only the applicability of each individual rule has to be verified at each step. Thus a major part of the verification, the verification of the correctness of each rule, need not be repeated.

Verification then reduces to verification of the applicability of a rule, and program versions are correct by construction (with respect to the original requirement specification). This stepwise proof is expected to be much easier than a corresponding proof of the final version.

As an alternative to proving the applicability conditions as they arise, the system can keep track of the verification conditions generated and accumulate them till the (successful) end of the development. This way, no proofs are necessary for "blind alleys", with the danger that the supposedly correct development sequence leading to the final version turns out to be a "blind alley" itself, if the proof fails. But even if we consider all proofs required from the developer (with assistance from the system) together, they are still much less complicated than a monolithic proof of the final version.

2.5 Transformational Program Development

The Transformational Development Model: Each transition from one program version to another can be regarded as a transformation in an abstract sense. It has a more technical meaning here: a transformation is a development step producing a new program version by application of an individual transformation rule, a compact transformation script, or, more generally, a transformation method invoking these. Before we come to the latter two, the basic approach will be described in terms of the transformation rule concept.

A transformation rule is a schema for an atomic development step that has been pre-conceived and is universally trusted, analogously to a theorem in mathematics. It embodies a grain of expertise that can be transferred to a new development. Its application realises this transfer and formalises the development process.

Transformations preserve correctness and therefore maintain a tighter and more formalised relationship to prior versions. Their classical application is the construction of optimised implementations by transformation of an initial design that has been proved correct against the formal requirement specification. Further design activity then consists in the selection of an appropriate rule, oriented by development goals, for example machine-oriented optimisation criteria.

Language Levels: We can distinguish various language levels at which the program is developed or into which versions are transformed, corresponding to phases of the development:

- formal requirement specification: loose equational or predicative specifications
- formal design specification: specification of abstract implementation
- applicative implementation: recursive functions
- imperative implementation: variables, procedures, iteration by loops
- flowchart implementation: labels, (conditional) jumps
- machine-oriented implementation: machine operations, registers, storage as array of words

All these language levels are covered by PA^{nn}dA (the first two by PA^{nn}dA-S), cf. [15, 16, 18, 19].

The (interactive) deduction of constructive design specifications from non-constructive requirement specifications, a kind of *program synthesis,* can be supported by complex transformation strategies and

tools. The enhancement of the Knuth-Bendix completion technique, for example, receives major attention in the PROSPECTRA project (see [20, 21]).

Current research focusses on the development activities at the specification level, demanding most creativity from the developer. This language level is perhaps the most important programming language of the future (supported by a prototyper, it is a kind of logic programming language). Many developments at lower levels can also be expressed at the specification level, for example "recursion removal" methods transforming into tail-recursive functions [22].

As an example, consider the rule in (2.5-1), expressed here at the specification level; the rule could be adapted further to apply to equations with constructors on the left instead of selectors on the right-hand sides. Assume that we want to derive a body for length. We first remove constructors on the left-hand-sides with the aid of the selectors and is_empty defined for LIST, see [23]. length is not in tail-recursive form: the addition of 1 still has to be made upon return from the recursion. By applying the transformation rule in (2.5-1), however, we can embed it into a function len that is tail-recursive, see (2.5-2). len can thus be transformed into a local loop, see (2.5-3). Note that the applicability condition, namely that + is an associative operation with neutral element 0, has to be proved with the aid of the system; but see also section 3.2 below.

*(2.5-1) **Transformation Rule:** Linear Recursion with Associative Operation to Tail Recursion*

```
  f: S —> R;

axiom for all x: S =>

  B x →   f x = T x,
¬ B x →   f x = op (f (H x)) (K x);
such that
  axiom for all x, y, z: R =>
    op x n = x,    op x (op y z) = op (op x y) z;
  f does not occur in T, H, K
```

```
  f: S —> R;
  g: S —> R —> R;
axiom for all x: S; y: R =>
          f x = g x n,
  B x →   g x y = op (T x) y,
¬ B x →   g x y = g (H x) (op (K x) y);
```

*(2.5-2) **Transformation:** Linear Recursion with Associative Operation to Tail Recursion:* length

```
  length:      LIST —>    INTEGER;

axiom for all x: LIST =>

  isEmpty x →   length x = 0,
¬ isEmpty x →   length x = length (tail x) + 1;
```

```
  length:  LIST —>    INTEGER;
- -: len:  LIST —>    INTEGER;
axiom for all x: LIST ; r: INTEGER =>
                    length x = len x 0,
  isEmpty x →   len x r = 0 + r,
¬ isEmpty x →   len x r = len (tail x) (1 + r);
```

(2.5-3) Ada Program: Applicative and Imperative Body of length *(with Unfold of* len*)*

```
function LENGTH (X: LIST) return INTEGER is
begin
  if IS_EMPTY (X) then
    return 0;
  else
    return LENGTH(TAIL(X)) + 1;
  end if;
end LENGTH;
```

```
function LENGTH (X: LIST) return INTEGER is
  V: LIST:= X; R: INTEGER := 0;
begin
  while not IS_EMPTY(V) loop
    V := TAIL(V);   R := 1+R;
  end loop;
  return R;
end LENGTH;
```

Catalogues of Transformation Rules: Some catalogues of transformation rules have been assembled for various high-level languages. Of particular interest is the structured approach of the CIP group. The program development language CIP-L is formally defined by transformational semantics (see [4, 5]), mapping all constructs in the wide spectrum of the language to a language kernel. Here, the kernel is

PA^{nn}dA-S, cf. [18]. These basic transformation rules have an axiomatic nature; compact rules for program development can be derived from them in a formal way.

Transformation Rules, Scripts, and Methods: Individual transformation rules are generalised to transformation *scripts*: sets of transformations rules applied together, possibly with local tactics that increase the efficiency. The long term research goal is to develop transformation *methods* that relieve the programmer from considerations about individual rules to concentrate on the goal oriented design activity. A transformation method is thus a set of rules or scripts with a global application strategy.

Calculus of Transformational Development: In analogy to an algebraic "calculus of data", a transformation rule is an axiom or theorem in a calculus of transformation. In fact we can regard the basic transformation rules as equations in the semantic algebra of program terms, using algebraic semantics [12, 24]. Alternatively, we can prove the correctness of a basic rule against a given semantics of the (kernel) language.

More complex derived transformation rules actually used in development can then either be proved as equational or inductive theorems or, if the basic rules are loose (or the given semantics is, for example with respect to the order of evaluation), then transformation rules may introduce design decisions in analogy to design specifications, and are robustly correct. They must, of course, be consistent with the basic rules.

Current research is concerned with such a calculus of transformation rules and their composition to complex development terms, representation of development strategies etc, see [25, 26] and chapters 4, 5 below.

3 Functionals

3.1 Methodological Advantages

Functionals, i. e. higher order functions with functions as parameters and/or results (cf. [34-38, 42-44]), allow a substantial reduction of re-development effort, in the early specifications and all subsequent developments. This aspect of functional abstraction is in analogy to parameterised data type specifications such as generics in Ada.

It is an interesting observation that many if not most definitions of functionals have a restricted form: the functional argument is unchanged in recursive calls. A functional together with its (fixed) functional parameters can then always be considered as a new function symbol (corresponding to an implicit instantiation), or it can be explicitly expanded. Functionals of this restricted form can be transformed to Ada generics; instantiation is then explicit, cf. [25]. In the sequel, we will restrict ourselves to this case. In the presence of overloading, a functional that is locally defined to a parameterised specification has an analogous effect as a polymorphic functional, cf. (3.1-1).

(3.1-1) Functional: *Map of a unary function over Lists*

```
  ... inside LISTS
  Map: (ITEM —> ITEM) —> LIST—> LIST;
axiom for all f: ITEM —> ITEM; e: ITEM; l: LIST =>
  Map f empty      = empty,
  Map f (cons e l) = cons (f e) (Map f l);
```

Thus the major advantage of functionals appears, at first glance, to be "merely" one of abbreviation. In contrast to generics, tedious explicit instantiation is avoided for functional parameters, in particular for partial parameterisation ("Curry'ing"). However, working with functionals quickly leads to a new style of programming (i. e. specification and development) at a considerably higher degree of abstraction. As we shall see below, much repetitive development can be reduced to the application of homomorphic extension functionals.

It is this aspect, that many functions should have the property of being homomorphisms, that goes beyond the correctness properties expressible in standard functional programming (in Miranda, for example). There, one tends to think only in terms of free term algebras. Here, we have the whole power of algebraic specification available to state, for example, that the properties of a monoid hold and are preserved by a (homomorphic) function, indeed by a functional for a whole class of applications. Development (optimising transformations etc.) need be made only once for the functional. In fact, the recursion schema of homomorphic extension (see [37]) provides a program development strategy ("divide and conquer", cf. [39]) and an induction schema for proofs.

We will see further below, how important these homomorphic extension functionals are for the concise definition of program development tactics.

3.2 Homomorphisms and Homomorphic Extension Functionals

The functional Map of (3.1-1) is a special case of a more general homomorphic extension functional, see (3.2-1). LinHom and BinHom correspond to the two different views one may have of list construction and thus correspond to a program development strategy by linear "divide and conquer" or binary partitioning, respectively. Map can, of course, be defined either way as an automorphism (i. e. a homomorphism to the same structure).

*(3.2-1) **Functional:** Homomorphic Extension Functionals over Lists*

```
... inside LISTS
generic
   type T is private; type TL is private;
package LIST_HOM is
   LinHom:    TL —> (T —> TL —> TL) —> (ITEM —> T) —> LIST—> TL;
axiom for all emptyTL: TL; consTL: T —> TL —> TL; h: ITEM —> T; e: ITEM; l: LIST =>
   LinHom emptyTL consTL h empty     = emptyTL,
   LinHom emptyTL consTL h (cons e l) = consTL (h e) (LinHom emptyTL consTL h l);
end LIST_HOM;
```

```
generic
   type TL is private;
package LIST_MONOID_HOM is
   BinHom:    (n: TL) —> (op: TL —> TL —> TL) —> (ITEM —> TL) —> LIST —> TL
              :: for all x, y, z: TL => op x n = x,    op n x = x,    op x (op y z) = op (op x y) z
axiom for all n: TL; op: TL —> TL —> TL; h: ITEM —> TL; e: ITEM; x, y: LIST =>
   BinHom n op h empty     = n,
   BinHom n op h (x & y)   = op (BinHom n op h x) (BinHom n op h y),
   BinHom n op h (single e) = h e;
end LIST_MONOID_HOM;
```

```
package AUTO is new LIST_HOM (ITEM, LIST); use AUTO;
Map: (ITEM —> ITEM) —> LIST—> LIST;
axiom Map = LinHom empty cons;
```

Note that BinHom requires, that the algebraic structure mapped into has the properties of a monoid (actually, an injection function corresponding to single is combined with the function h, now from ITEM to TL). In this case we can transform BinHom using the monoid properties of lists and employ an analogous recursion removal transformation to (2.5-1) that is only applicable, if op and n form a monoid (cf. [42, 22]), see (3.2-2).

In functional programming, such a global optimisation is not possible since we could not be sure that the binary operation is associative in general; there is no way to state such a requirement in a standard functional programming language. In conventional programming or algebraic specification without functionals we would have to separately prove the property and optimise for each case (each instance of the functional).

(3.2-2) ***Functional:*** *Optimisation using Algebraic Properties and Recursion Removal Transformation*

```
  ... inside LIST_HOM
  axiom for all n: TL; op: TL —> TL —> TL; h: ITEM —> TL; e: ITEM; y, z: LIST =>
                BinHom n op h empty          = n,
                BinHom n op h ((single e) & y)  = op (h e) (BinHom n op h y),
¬ isEmpty z →   BinHom n op h z              = op (h (head z)) (BinHom n op h (tail z));
```

```
    BinH2: (n: TL) —> (op: TL —> TL —> TL) —> (ITEM —> TL) —> TL —> LIST —> TL
           :: for all x, y, z: TL => op n x = x,   op x n = x,   op x (op y z) = op (op x y) z;
  axiom for all n: TL; op: TL —> TL —> TL; h: ITEM —> TL; e: ITEM; r: TL; z: LIST =>
                BinHom n op h z          = BinH2 n op h n z,
                BinH2   n op h r empty   = op n r,
¬ isEmpty z →   BinH2   n op h r z       = BinH2 n op h r (op (h (head z)) r) (tail z);
```

As an example for the instantiation of a homomorphic extension functional, see (3.2-3) for length (cf. (2.5-2, 3) above).

(3.2-3) ***Functional:*** *Homomorphism over Lists as Instantiation of Homomorphic Extension Functional*

```
  package toINT_HOM is new LIST_MONOID_HOM (INTEGER); use toINT_HOM ;
  length:     LIST —>   INTEGER;
- -:one:      ITEM —>   INTEGER;
axiom for all e: ITEM =>  one e = 1;   length = BinHom 0 "+" one
```

(3.2-4) ***Functional:*** *Homomorphic Extension Functionals for Predicates over Lists*

```
  package toBOOL_HOM is new LIST_MONOID_HOM (BOOLEAN); use toBOOL_HOM ;
  Exist:      (ITEM —> BOOLEAN) —> LIST —>   BOOLEAN;
  ForAll:     (ITEM —> BOOLEAN) —> LIST —>   BOOLEAN;
  isElem:     ITEM —>   LIST —>   BOOLEAN;
axiom for all x: ITEM; a, b: LIST =>
  Exist = BinHom false "∨",   ForAll = BinHom true "∧",   isElem x = Exist (eq x);
```

(3.2-5) ***Functions:*** *Homomorphic Implementation of Sets as Lists: Initial Model*

```
  ... inside SETS   with LISTS;
private
  package ITEM_LISTS is new LISTS (ITEM, eq); use ITEM_LISTS;   - - enrichment
  type SET is new LIST;
axiom for all x: ITEM; a, b: SET =>
  empty = ITEM_LIST.empty,   a ∪ b = a & b,   singleton x = single x,   x ∈ a = isElem x a;
```

(3.2-6) ***Functions:*** *Homomorphic Implementation of Sets as Lists: Terminal Model*

```
  ... inside SETS   with LISTS;
private
  package ITEM_LISTS is new LISTS (ITEM, eq); use ITEM_LISTS;   - - enrichment
  type SET is new LIST;
  package toLIST_HOM is new SET_HOM (SET); use toLIST_HOM;
- -: elimElem:  SET —> ITEM —> SET;
axiom for all x: ITEM; a, b: SET =>
  empty = LIST (empty),   singleton x = single x,   x ∈ a = LIST (isElem x a),
  isElem x a → elimElem a x = empty,   ¬ isElem x a → elimElem a x = single a
  a ∪ b = a & Hom empty "&" (elimElem a) b;
```

Similarly, existential and universal quantification of a predicate over a list can be defined by homomorphic extension of the predicate over lists, using the algebraic properties of Booleans, see (3.2-4). isElem can be defined this way as an example. Note the use of partial parameterisation for eq (on INTEGER).

Homomorphisms over sets are defined in the same way; there are, however, additional restrictions on Hom guaranteeing the preservation of the algebraic properties of sets (cf. (2.3-3)). The implementation of sets as list can be defined homomorphically; (3.2-5) gives the initial model (elements are repeated and $\in$ must search among all these), and (3.2-5) gives the terminal model (elements are only inserted if they do not yet occur; $\in$ need only search in this smaller list, the "real set").

4 Formalisation of Progam Transformations

4.1 The Syntactic Algebra of Programs

We can define the Abstract Syntax of a programming language such as PA^{nn}dA-S by an algebraically specified Abstract Data Type: trees in the Abstract Syntax correspond to terms in this algebra of (PA^{nn}dA-S) programs, non-terminals to sorts, tree constructor operations to constructor operations, etc., see (4.1-1, 2). Most constructor operations are free, except for all operations corresponding to List or Sequence concatenation.

Although we are interested in the operations of the abstract syntactic algebra of programs, it is often more convenient to use a notation for *phrases* (program fragments with schema variables) of the *concrete syntax* corresponding to appropriate *terms* (with variables) in the algebra. Phrases provide a concise notation for large terms. The brackets ⌜ ⌟ are used whenever a (nested) phrase of the concrete syntax is introduced. In this paper, we are not concerned with notational issues at the concrete syntax level nor with the (non-trivial) translation of phrases from concrete to abstract syntax.

Specifications of abstract types such as those in (4.1-1, 2), including selectors and other auxiliary operations, are automatically constructed from a given abstract syntax specification in the present PROSPECTRA system.

(4.1-1) Abstract Type: Abstract Syntax for Expressions and Expression Lists

```
with NAMES, LISTS;  use NAMES;
package EXPS is
  type EXP is private;
  package EXP_LISTS is new LISTS(EXP);  use EXP_LISTS;  subtype EXP_LIST is EXP_LISTS.LIST;
  mkName:   NAME —>      EXP;            - - concrete phrase: ⌜ n ⌟
  mkTuple:  EXP_LIST —> EXP;             - - concrete phrase: ⌜ el ⌟  if empty or single, otherwise: ⌜ ( el ) ⌟
  mkCall:   EXP —> EXP_LIST —> EXP;  - - concrete phrase: ⌜ e el ⌟
  ... a definition of selectors and their axioms is omitted for brevity
  ... homomorphisms etc. see below
end EXPS;
```

(4.1-2) Abstract Type: Abstract Syntax for Statements and Statement Sequences

```
with NAMES, EXPS, LISTS;  use NAMES, EXPS;
package STMTS is
  type STMT is private;
  package STMT_SS is new LISTS(STMT); use STMT_SS; subtype STMT_SEQ is STMT_SS.LIST;
  mkIf:     EXP —>  STMT_SEQ —>  STMT_SEQ —>  STMT;
  - - concrete phrase: ⌜ if e then sst else sse end if; ⌟
  ... a definition of further constructors, selectors and their axioms is omitted for brevity
end STMTS;
```

4.2 Transformation Rules: Equations in the Semantic Algebra of Programs

In the approach of the algebraic definition of the semantics of a programming language (cf. [12]), an evaluation function or interpretation function from syntactic to semantic domains is axiomatised. The equational axioms of such functions induce equivalence classes on (otherwise free) constructor terms. In other words, we can prove that two (syntactic) terms are *semantically equivalent,* in a context-free way or possibly subject to some syntactic or semantic pre-conditions. Such a proof can of course also be made with respect to some other style of semantic definition for the language. Thus we obtain a *semantic algebra* of programs in which transformation rules are equations as a quotient algebra of the *abstract syntactic algebra* in which only equations for & exist.

Note that the semantic specification may be intentionally loose, that is some semantic aspects such as the order of evaluation of expressions in a call may be intentionally left unspecified. From an algebraic point of view this means that several distinct semantic models exist for the loose semantic specification. Usually, these form a lattice between the initial model on top (where all terms are distinct that cannot be proven to equal) and the terminal model at the bottom (where all terms are the same that cannot be proven to differ). In some cases, unique initial and terminal models may not exist: if expressions may have side-effects, for example, several (quasi-terminal) models exist according to particular sequentialisations of evaluation (cf. [13]). Each choice of model (each choice of sequentialisation by a compiler) is admissible.

(4.2-1) shows examples of transformation rules for if_statements, analogous to the algebraic properties of a non-strict function ifThenElse (there are more rules about nesting etc. that are omitted here).

(4.2-1) Transformation Rules (in the Semantic Algebra): if_statements

axiom for all e: EXP; sst, sse: STMT_SEQ =>
⌈ **if** true **then** *sst* **else** *sse* **end if**; ⌋ = *sst*, ⌈ **if** false **then** *sst* **else** *sse* **end if**; ⌋ = *sse*,
⌈ **if not** *e* **then** *sst* **else** *sse* **end if**; ⌋ = ⌈ **if** *e* **then** *sse* **else** *sst* **end if**; ⌋ ;

A *uni-directional* transformation rule corresponds to a relation between semantic models such that each model in the range is a robustly correct implementation of some model in the domain; thus it corresponds to a semantic inclusion relation in a model-oriented sense. Again this notion is taken from the theory of algebraic specification (cf. [13] or the converse relation as the approximation relation ≤ on (transformation) functions in [36]). It formalises the notion of correctness with respect to some implementation decision that narrows implementation flexibility or chooses a particular implementation. These rules are of course not invertible (a decision cannot be reversed) and, interpreted as rewrite rules, are not confluent in general. In this paper, we restrict our attention to bi-directional rules although most considerations generalise.

The major kind of transformation rules we are interested in is the *bi-directional transformation rule*, a pair of semantically equivalent terms: an *equation* in the semantic algebra of programs that is provable by deductive or inductive reasoning against the semantics. All rules in this paper are of this kind. All considerations about interpreting equations as rewrite rules apply (confluence, termination, completion [21], etc.).

Note that we can apply all the power of the algebraic framework to transformation rules specified in this way, for example the deduction of new rules using equational or inductive reasoning, even completion techniques.

4.3 Basic Transformations: Operations in the Syntactic Algebra of Programs

From each transformation rule or set of related rules, that is equations in the semantic algebra, an elementary transformation operation can be constructed in a straightforward way as a partial function in the *abstract syntactic algebra*, see (4.3-1): it maps to a normal form in the quotient algebra corresponding to the equations. Each equation is considered as a rewrite rule from left to right (or from right to left), and, if the system of rewrite rules is confluent, yields a corresponding normal form. The function corresponds to an identity in the semantic algebra and achieves a kind of normalisation in the syntactic algebra.

(4.3-1) shows an example of a basic transformation function derived from a single transformation rule: *swapIf* swaps the then_ and else_parts of a conditional if the condition is of the form **not** *e*.

(4.3-1) Basic Transformation Function: swapIf

```
  swapIf:      (s: STMT :: is_swapIf s) —> STMT;
axiom for all e: EXP; sst, sse: STMT_SEQ =>
  swapIf ⌈ if not e then sst else sse end if; ⌋ = ⌈ if e then sse else sst end if; ⌋;
```

Similarly, a basic applicability predicate can be derived from the transformation rule (possibly including contextual or semantic applicability conditions in addition to the syntactic ones). Note that the **others** can be expanded using simple syntactic predicates (to be defined jointly with the Abstract Syntax), cf. example below.

(4.3-2) Basic Applicability Predicate: is_swapIf

```
  is_swapIf:      STMT —> BOOLEAN;
axiom for all e: EXP; sst, sse: STMT_SEQ; s: STMT =>
  (is_swapIf ⌈ if not e then sst else sse end if; ⌋ = true,   others → is_swapIf s = false);
```

Other basic transformation functions and predicates are defined analogously below. The applicability condition is_elimIf has been simplified using structural reasoning. More simplification rules could of course be used on Booleans.

(4.3-3) Basic Transformation Function: elimIf

```
  is_elimIf:    STMT —> BOOLEAN;
  elimIf:       (s: STMT :: is_elimIf s) —> STMT;
axiom for all e, e1: EXP; sst, sse: STMT_SEQ =>
  elimIf ⌈ if true then sst else sse end if; ⌋ = sst, elimIf ⌈ if false then sst else sse end if; ⌋ = sse,
  is_elimIf ⌈ if e then sst else sse end if; ⌋ = eq e ⌈ true ⌋ ∨ eq e ⌈ false ⌋,
  not isIfStmt s → is_elimIf s = false;
```

(4.3-4) Basic Transformation Functions: elimNot *and* deMorgan

```
  is_elimNot:   EXP —> BOOLEAN;
  is_deMorgan:  EXP —> BOOLEAN;
  elimNot:      (e: Exp :: is_elimNot e) —>    EXP;
  deMorgan:     (e: Exp :: is_deMorgan e) —> EXP;
axiom for all x, y: EXP =>
     elimNot ⌈ not not x ⌋ = x,      elimNot ⌈ not false ⌋ = ⌈ true ⌋,  elimNot ⌈ not true ⌋ = ⌈ false ⌋,
  (is_elimNot ⌈ not not x ⌋ = true,  is_elimNot ⌈ not false ⌋ = true,  is_elimNot ⌈ not true ⌋ = true,
  others → is_elimNot x = false),
  deMorgan ⌈ not x and not y ⌋ = ⌈ not (x or y) ⌋,   deMorgan ⌈ not x or not y ⌋ = ⌈ not (x and y) ⌋,
  (is_deMorgan ⌈ not x and not y ⌋ = true,          is_deMorgan ⌈ not x or not y ⌋ = true,
  others → is_deMorgan x = false);
```

If we want to apply elementary transformations over a larger context with some tactics (see below), we need to extend the domain of a partial function to larger terms, as in (4.3-5) for tdeMorgan. The first equation corresponds to the previous definition for deMorgan. The second extends the definition to the identity over EXP, negating the applicability condition; cf. also Try in (4.4-1) below.

(4.3-5) Basic Transformation Functions: Totalisation of deMorgan

```
  tdeMorgan:    EXP —> EXP;
axiom for all x: EXP =>
  is_deMorgan x → tdeMorgan x = deMorgan x,      ¬ is_deMorgan x → tdeMorgan x = x;
```

4.4 Transformation Functionals: Homomorphic Extensions and Tactics

In analogy to tacticals in [38], we might call some transformation functionals *transformals* since they embody application tactics or strategies. Consider for example (4.4-1): if some transformation function *f* and its applicability condition *p* are given, then *Try* provides a totalisation or extension to identity if *p* does not hold.

(4.4-1) Functional: Try

```
  Try:  (EXP —> BOOLEAN) —>  (EXP —> EXP) —>  EXP —>  EXP;
axiom for all p: EXP —> BOOLEAN; f: EXP —> EXP; x: EXP =>
¬ p x →   Try  p f x = x,
  p x →   Try  p f x = f x,
```

```
  tdeMorgan = Try is_deMorgan deMorgan;
```

More important for application tactics are *homomorphic extension* functionals (see [25]), in this case the structural extension of the effect of a (local) transformation or predicate over larger terms. In (4.4-2, 3), *Hom* and *Hext* extend a function *fn* on names over expressions; they are similarly defined for statements below. (4.4-3) shows an example of an instantiation: a function that counts the number of occurrences of a given name in an expression; note that it uses the homomorphic extension functionals of lists for nested expression lists.

(4.4-2) Functionals: Basic Homomorphic Extensions for Expressions and Expression Lists

```
  ... inside EXPS
  generic
    type TN is private;  type TE is private;  type TEL is private;
  package EXP_HOM is
    Hom:        (TN —> TE) —>  (TEL —> TE) —>  (TE —> TEL —> TE) —>
                (NAME —> TN) —>  (EXP_LIST —> TEL) —>  EXP —>  TE;
  axiom for all fName: TN —> TE; fTuple: TEL —> TE; fCall: TE —> TEL —> TE;
                fn: NAME —> TN; fel: EXP_LIST —> TEL; n: NAME; e: EXP; el: EXP_LIST =>
    Hom fName fTuple fCall fn fel (mkName n)   = fName (fn n),
    Hom fName fTuple fCall fn fel (mkTuple el) = fTuple (fel el),
    Hom fName fTuple fCall fn fel (mkCall e el) = fCall (Hom fName fTuple fCall fn fel e) (fel el);
  end EXP_HOM;
```

```
  package AUTO is new EXP_HOM (NAME, EXP, EXP_LIST); use AUTO;
  Hext:   (NAME —> NAME) —>  (EXP_LIST —> EXP_LIST ) —>  EXP —>  EXP;
axiom  Hext = Hom mkName mkTuple mkCall;
```

(4.4-3) Instance: Number of Occurrences of a Name in an Expression or Expression List

```
  package toINT_HOM is new EXP_HOM (INTEGER, INTEGER, INTEGER); use toINT_HOM ;
  package LISTtoINT_HOM is new LIST_MONOID_HOM (INTEGER); use LISTtoINT_HOM ;
  numOcc:   NAME —>  EXP —>   INTEGER;
--:numName:  NAME —>  NAME —>  INTEGER;
axiom for all n, m: NAME =>
  numName n n = 1,      ¬ eq n m → numName n m = 0,
  numOcc n = Hom id id "+" (numName n) (BinHom 0 "+" (numOcc n));
```

Sweep is a homomorphic extension functional from expressions to expressions; it applies the basic function *f* to every subexpression, cf. (4.4-4, 5). Substitution can be defined using this tactic, see (4.4-6).

(4.4-4) Functionals: Transformation Tactics for Expressions: Sweep

```
  Sweep:    (EXP —> EXP) —>  EXP —>  EXP;
axiom  Sweep f = Hom (f ∘ mkName) (f ∘ mkTuple) (f ∘ mkCall) id (Map (Sweep f));
```

(4.4-5) Transformation Function: Sweep of deMorgan

```
  everydeMorgan:  EXP —> EXP;
axiom  everydeMorgan = Sweep tdeMorgan;
```

(4.4-6) Functionals: Transformation Tactics for Expressions: Substitution

```
  package NametoExp_HOM is new EXP_HOM (EXP, EXP, EXP_LIST); use NametoExp_HOM ;
  NtoExp:    (NAME —> EXP) —> (EXP_LIST —> EXP_LIST ) —> EXP —> EXP;
axiom  NtoExp = Hom id mkTuple mkCall;
```

```
  substByIn: NAME —> EXP —> EXP —> EXP;
- -:substName: NAME —> EXP —> NAME —> EXP;
axiom for all n, m: NAME; x: EXP =>
  substName n x n = x,      ¬eq n m → substName n x m = mkName m,
  substByIn n x = NtoExp (substName n x) (Map (substByIn n x));
```

```
  substByIn: EXP —> EXP —> EXP —> EXP;
- -:subst:    EXP —> EXP —> EXP —> EXP;
axiom for all x, y, z: EXP =>
  subst y x y = x,      ¬eq y z → subst y x z = z,
  substByIn y x = Sweep (subst y x);
```

SweepP is a similar homomorphic extension functional for predicates, see (4.4-7). The definition of could, of course, also be generalised to a general functional for predicates or pairs.

(4.4-7) Functionals: Homomorphic Predicates for Expressions

```
  SweepP:  (BOOLEAN —> BOOLEAN —> BOOLEAN) —> (EXP_LIST —> BOOLEAN) —>
           (EXP —> BOOLEAN) —> EXP —> BOOLEAN;
axiom for all op: BOOLEAN —> BOOLEAN —> BOOLEAN; pel: EXP_LIST —> BOOLEAN;
           p: EXP —> BOOLEAN; e: EXP; el: EXP_LIST =>
  SweepP op pel p (mkName n)  = p (mkName n),
  SweepP op pel p (mkTuple el) = op (p (mkTuple el)) (pel el),
  SweepP op pel p (mkCall e el) = op (p (mkCall el)) (op (SweepP op pel p e) (pel el));
```

```
  eq:        EXP —> EXP —> BOOLEAN;
axiom for all n, m: NAME; x, y: EXP; a, b: EXP_LIST =>
  eq (mkName n) (mkName m)  = eq n m,
  eq (mkTuple a) (mkTuple b)  = eq a b,
  eq (mkCall x a) (mkCall y b)  = eq x y ∧ eq a b,
  others →  eq x y = false;
```

```
  Exist:     (EXP —> BOOLEAN) —> EXP —> BOOLEAN;
  ForAll:    (EXP —> BOOLEAN) —> EXP —> BOOLEAN;
  occursIn:  EXP —>              EXP —> BOOLEAN;
axiom for all p: EXP —> BOOLEAN; e: EXP =>
  Exist p  = SweepP "∨" (Exist (Exist p)) p,        -- Exist on EXP_LIST!
  ForAll p = SweepP "∧" (ForAll (ForAll p)) p,      -- ForAll on EXP_LIST!
  occursIn e = SweepP "∨" (Exist (occursIn e)) (eq e);
```

Finally, analogous definitions are made for statements, see (4.4-8, 9). In fact, general definitions of homomorphic extension functionals could be constructed automatically for a given abstract syntax, in the same way as the construction of an algebraically specified type in the present PROSPECTRA system.

*(4.4-8) **Functionals:** Homomorphisms for Statements and Statement Sequences*

```
  ... inside STMTS
  generic
    type TE is private;  type TS is private;  type TSS is private;
  package STMT_HOM is
    Hom:        (TE —> TSS —> TSS —> TS) —>
                (EXP —> TE) —>  (STMT_SEQ —> TSS) —>  STMT —>  TS;
  axiom for all flf: TE —> TSS —> TSS —> TS; fe: EXP —> TE; fss: STMT_SEQ —> Tss;
                e: EXP; sst, sse: STMT_SEQ =>
    Hom flf fe fss (mklf e sst sse)   = flf (fe e) (fss sst) (fss sse);
  end STMT_HOM;
```

```
  package AUTO is new STMT_HOM (EXP, STMT, STMT_SEQ); use AUTO;
  Hext:    (EXP —> EXP) —>  STMT —>  STMT;
axiom for all fe: EXP —> TE =>
  Hext fe = Hom mklf fe (Map (Hext fe));
```

```
  Sweep:    (EXP —> EXP) —>  (STMT —> STMT) —>  STMT —>  STMT;
axiom for all fe: EXP —> EXP; fs: STMT —> STMT =>
    Sweep fe fs = Hom (fs ∘ mklf) fe (Map (Sweep fe fs));
```

```
  SweepP:   (BOOLEAN —> BOOLEAN —> BOOLEAN) —>  (STMT_SEQ —> BOOLEAN) —>
            (STMT —> BOOLEAN) —>  STMT —>  BOOLEAN;
axiom for all op: BOOLEAN —> BOOLEAN —> BOOLEAN; pss: STMT_SEQ —> BOOLEAN;
            pe: EXP —> BOOLEAN; ps: STMT —> BOOLEAN; e: EXP; sst, sse: STMT_SEQ =>
  SweepP op pss pe ps (mklf e sst sse)   = op (ps (mklf e sst sse)) (op (pe e) (op (pss sst) (pss sse)));
```

```
  Exist, ForAll:   (EXP —> BOOLEAN) —>  (STMT —> BOOLEAN) —>  STMT —>  BOOLEAN;
  occursIn:        STMT —>  STMT —>  BOOLEAN;
- -: constFalse:   EXP —>  BOOLEAN;
axiom for all pe: EXP —> BOOLEAN; ps: STMT —> BOOLEAN; e: EXP; s: STMT =>
  Exist pe ps   = SweepP "∨" (Exist (Exist pe ps)) pe ps,      - - Exist on STMT_SEQ!
  ForAll pe ps  = SweepP "∧" (ForAll (ForAll pe ps)) pe ps,    - - ForAll on STMT_SEQ!
  occursIn s    = SweepP "∨" (Exist (occursIn s)) constFalse (eq s), constFalse e = false;
  ... and so on, analogously to expressions
```

5 Formalisation of Transformational Program Developments

5.1 Development Scripts: Composite Transformation Functions

Since we can regard every elementary program development step as a transformation, we may conversely define a *development script* to be a composition of transformation operations (including application strategies for sets of elementary transformation operations). In this view we regard a development script as a *development transcript* (of some constant program term) to formalise a concrete development history, possibly to be re-played, or as a *development method* abstracting to a class of analogous programs.

5.2 Development Goals: Requirement Specifications

A *development goal*: it is a requirement specification for a development script, that is a transformation function emplying a certain transformation strategy, yet to be designed. It can be considered to be a characteristic predicate for the respective transformation function or the post-condition of the application of some set of transformation rules. For example, we can state the desired goal for normalisation of statements or Boolean expressions as in (5.2-1).

(5.2-1) Development Goals: Normalisation of Expressions and Statements

```
  not_normSubExp, not_normExp:   EXP —>  BOOLEAN;
  not_normStmt:                  STMT —> BOOLEAN;
axiom for all x: EXP; s: STMT =>
  not_normSubExp x = is_elimNot x ∨ is_deMorgan x,    not_normExp = Exist not_normSubExp,
  not_normSTMT = Exist is_elimIf;
```

Often, the application of some set of rules requires the satisfaction of some pre-condition established by (previous exhaustive application of) some other set of rules, i. e. as the post-condition of this set of rules. Note that such intermediate conditions never need to be checked operationally as long as it can be shown that they are established by previous application of other rules.

If these conditions can be defined structurally (or "syntactically"), as in our example, then they characterise certain *normal forms*. This leads to a substantial improvement in the modularisation of sets of rules and separation of concerns, consequently ease of verification. Transformation functions having structural normal forms as applicability conditions correspond to Wile's syntax directed experts [40].

5.3 Development Tactics: Transformals

Exhaustive application of some set of rules can be expressed by suitable transformals. *While* can be used to apply a transformation function *f* as long as some condition *p* holds. Similarly, *Iterate* iterates a local transformation function *f* as long as some local condition *p* holds somewhere, see (5.3-1). These transformals correspond to a kind of "Markov algorithm" tactics when generalised to sets of rules.

(5.3-1) Development Tactics: While, Iterate

```
  While, Every, Iterate: (EXP —> BOOLEAN) —> (EXP —> EXP) —> EXP —> EXP;
axiom for all p: EXP —> BOOLEAN; f: EXP —> EXP; x: EXP =>
¬ p x →   While p f x = x,
  p x →   While p f x = While p f (f x),
          Every p f x = Sweep (Try p f) x,
          Iterate p f x = While (Exist p) (Every p f) x;
```

(5.3-2) shows some examples; for statements, iteration is defined analogously with an extra parameter for the homomorphic extension of iteration over expressions. Note, however, that the composition *compose_iter* does not achieve the desired effect yet. It must be iterated again.

(5.3-2) Application of Development Tactics: iter_normExp, iter_normStmt

```
  iter_deMorgan, iter_elimNot, compose_iter, iter_normExp: EXP —> EXP;
axiom for all x: EXP =>
  iter_deMorgan x  = Iterate is_deMorgan deMorgan x,
  iter_elimNot x   = Iterate is_elimNot elimNot x,
  compose_iter     = iter_deMorgan ∘ iter_elimNot,
  iter_normExp     = Iterate not_normSubExp compose_iter;
  iter_normStmt:   STMT —> STMT;
axiom iter_normStmt   = Iterate is_elimIf iter_normExp (Try is_elimIf elimIf);
```

(5.3-3) provides an improved version with a combined sweep of either *deMorgan* or *elimNot*. In fact, we could have proved immediately on the transformation rules that the transformations on expressions are mutually independent and correspond to a confluent and terminating system of rewrite rules; thus we could have used one basic transformattion function. This had purposely not been done to show the effect of combination of several transformations in an iteration strategy.

(5.3-3) Application of Development Tactics: iter_normExp

```
iter_normExp:   EXP —>  EXP;
TryBoth:        (EXP —> BOOLEAN) —>  (EXP —> EXP) —>
                (EXP —> BOOLEAN) —>  (EXP —> EXP) —>  EXP —>  EXP;
axiom for all p1, p2: EXP —> BOOLEAN; f1, f2: EXP —> EXP; x: EXP =>
  iter_normExp  = While not_normExp  (Sweep (TryBoth is_deMorgan deMorgan is_elimNot elimNot)),
  p1 x → TryBoth p1 f1 p2 f2 x = f1 x,     p2 x → TryBoth p1 f1 p2 f2 x = f2 x,
  p1 x ∧¬ p2 x → TryBoth p1 f1 p2 f2 x = x;
```

5.4 Development Rules: Equations over Tactics

We would like to improve the transformation tactics even further. As far as possible, we would like to achieve the same strategic effect (the same development goal) by different, increasingly more efficient, application tactics. A transformation from one tactic to another is possible by development rules, see the (5.4-1). *Development rules,* that is equational properties of development scripts, allow us to express and to reason about design alternatives or *alternative development tactics*, and to *simplify developments* by considering them as algebraic terms in the usual way. (5.4-2) shows the development of a derived rule by equational reasoning. It may be used to simplify iterated application into a single bottom-up one-sweep application. This rule, and an analogous rule for statements, is used in (5.4-3) to simplify our example since we can prove the premise.

(5.4-1) Development Rule: Elimination of While

```
axiom for all p: EXP —> BOOLEAN; f: EXP —> EXP; x: EXP =>
p x ∧¬ p (f x) → While p f x = f x
```

(5.4-2) Development Rule Derivation: Iterate ⇔ Every ⇔ Sweep

```
                                    Iterate  p f x = While (Exist p) (Every p f) x,   - - definition of While
Exist p x ∧¬ Exist p (Every p f x) → Iterate  p f x = Every p f x,                    - - elimination of While
Exist p x ∧¬ Exist p (Every p f x) → Iterate  p f x = Sweep (Try p f x);              - - definition of Every
```

(5.4-3) Derivation: iter_normExp, iter_normStmt

```
axiom  iter_normExp = Sweep (TryBoth is_deMorgan deMorgan is_elimNot elimNot),
       iter_normStmt = While (Exist is_elimIf) (Sweep iter_normExp (Try is_elimIf elimIf)),
       iter_normStmt = Sweep iter_normExp (Try is_elimIf elimIf);
```

We have converged more and more to the development of a complete specification of a set of efficient transformation functions that can be directly translated into a recursive applicative program in some language, cf. [1, 2, 16]. Intermediate specifications could be made operational by some functional language with non-deterministic pattern matching and backtracking. Such a language is presently being designed and implemented in the PROSPECTRA project; see [11] for a first approach.

6. Conclusion

It has been demonstrated that the methodology for program development based on the concept of algebraic specification of data types, with functionals, and program transformation can be applied to the development of transformation algorithms; in the semantic algebra of programs, equations correspond to bi-directional transformation rules. Starting from small elementary transformation rules that are proved correct against the semantics of the programming language, we can apply the usual equational and inductive reasoning to derive complex rules; we can reason about development goals as requirement specifications for transformation operations in the syntactic algebra and characterise them as structural normal forms; we can implement transformation operations by various design alternatives; we can optimise them using al-

gebraic properties; we can use composition and functional abstraction; in short, we can develop *correct*, efficient, complex transformation operations from elementary rules stated as algebraic equations.

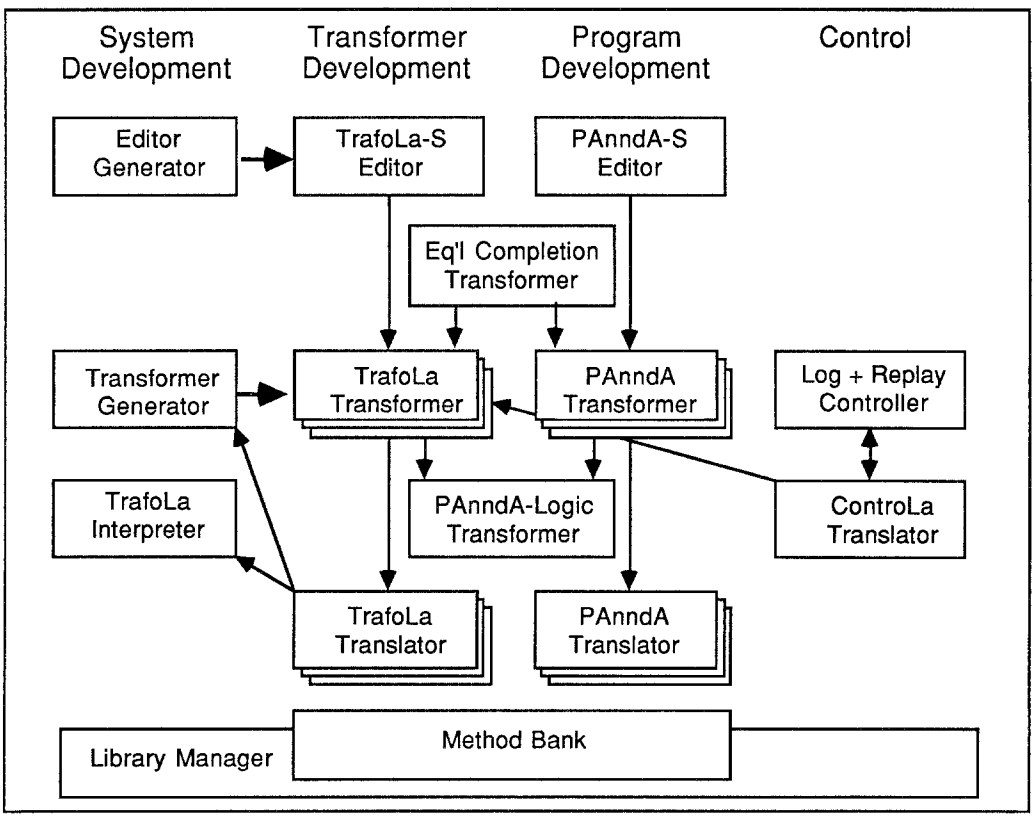
Moreover, we can regard development scripts as formal objects: as (compositions of) such transformation operations. We can specify development goals, implement them using available operations, simplify development terms, re-play developments by interpretation, and abstract to development methods, incorporating formalised development tactics and strategies. The abstraction from concrete developments to methods and the formalisation of programming knowledge as transformation rules + development methods will be a challenge for the future.

Many questions remain open at the moment. One is a suitable separation of a large set of known rules into subsets such that each can be handled by dedicated tactics with an improved efficiency over the general case, and coordinated by an overall strategy; these correspond to the "syntax-directed experts" of [40]. Another is the strategy questions: the selection of a development goal (sometimes expressible as a normal form) based on some efficiency or complexity criteria.

There is a close analogy to the development of efficient proof strategies for given inference rules (transformation rules in the algebra of proofs). Perhaps the approach can be used to formalise rules and inference tactics in knowledge based systems.

Since every manipulation in a program development system can be regarded as a transformation of some "program" (for example in the command language), the whole system interaction can be formalised this way and the approach leads to a uniform treatment of programming language, program manipulation and transformation language, and command language.

(6-1) PROSPECTRA System Architecture



This approach is presently exploited in the PROSPECTRA system (cf. 6-1)). The specification language PAnndA-S of programs (with Ada as a target) is also used as a transformation specification language TrafoLa-S. In this case, an abstract type schema to define Abstract Syntax is predefined, and translation to an internal applicative tree manipulation language is automatic. Work on a more powerful language with higher order matching and functionals is going on [11]. Various development strategies such as fold-unfold, variable abstraction, finite differencing, rewriting, narrowing and unification with a set of equations interpreted as rewrite rules, etc. have been implemented and are being complemented by other strategies and methods. Work on a translation between the command language and TrafoLa has started to allow the translation of development histories (replay is already possible).

Acknowledgements

I wish to thank B. Gersdorf, J. v. Holten, S. Kahrs, D. Plump, R. Seifert, and Z. Qian for helpful comments.

References

[1] Krieg-Brückner, B., Hoffmann, B., Ganzinger, H., Broy, M., Wilhelm, R., Möncke, U., Weisgerber, B., McGettrick, A.D., Campbell, I.G., Winterstein, G.: PROgram Development by SPECification and TRAnsformation. *in:* Rogers, M. W. (ed.): *Results and Achievements,* Proc. ESPRIT Conf. '86 . North Holland (1987) 301-312.

[2] Krieg-Brückner, B.: Integration of Program Construction and Verification: the PROSPECTRA Project. in: Habermann, N., Montanari, U. (eds.): Innovative Software Factories and Ada. Proc. CRAI Int'l Spring Conf. '86. *LNCS 275* (1987) 173-194.

[3] Bauer, F.L.: Program Development by Stepwise Transformations - The Project CIP. **in**: Bauer, F. L., Broy, M. (eds.): Program Construction. *LNCS 69,* Springer 1979.

[4] Bauer, F.L., Berghammer, R., Broy, M., Dosch, W., Gnatz, R., Geiselbrechtinger, F., Hangel, E., Hesse, W., Krieg.-Brückner, B., Laut, A., Matzner, T.A., Möller, B., Nickl, F., Partsch, H., Pepper, P., Samelson, K., Wirsing, M., Wössner, H.: *The Munich Project CIP, Vol. 1: The Wide Spectrum Language CIP-L. LNCS 183,* Springer 1985.

[5] Bauer, F.L., Ehler, H., Horsch, B., Möller, B., Partsch, H., Paukner, O., Pepper, P.,: *The Munich Project CIP, Vol. 2: The Transformation System CIP-S. LNCS 292,* Springer 1987.

[6] Partsch, H., Steinbrüggen, R.: Program Transformation Systems. *ACM Computing Surveys 15* (1983) 199-236.

[7] Reference Manual for the Ada Programming Language. ANSI/MIL.STD 1815A. US Government Printing Office, 1983. Also **in**: Rogers, M. W. (ed.): *Ada: Language, compilers and Bibliography.* Ada Companion Series, Cambridge University Press, 1984.

[8] Luckham, D.C., von Henke, F.W., Krieg-Brückner, B., Owe, O.: Anna, a Language for Annotating Ada Programs, Reference Manual. *LNCS 260,* Springer (1987).

[9] Bertling, H., Ganzinger, H.: A Structure Editor Based on Term Rewriting. **in**: Proc. 2nd ESPRIT Technical Week, Brussels (1985) 455-466.

[10] Möncke, U., Weisgerber, B., Wilhelm, R.: Generative Support for Transformational Programming. **in**: Proc. 2nd ESPRIT Technical Week, Brussels (1985) 511-528.

[11] Heckmann, R.: A Functional Language for the Specification of Complex Tree Transformations. *in:* Proc. European Symposium On Programming '88, *LNCS 300* (1988) .

[12] Broy, M., Pepper, P., Wirsing, M.: On the Algebraic Definition of Programming Languages. *ACM TOPLAS 9* (1987) 54-99.

[13] Broy, M., Wirsing, M.: Partial Abstract Types. *Acta Informatica 18* (1982) 47-64.

[14] Owe. O.: An Approach to Program Reasoning Based on a First Order Logic for Partial Functions. Research Report No. 89, Institute of Informatics, University of Oslo, 1985.

[15] Krieg-Brückner, B.: Transformation of Interface Specifications. **in:** Kreowski, H.-J. (ed.): *Recent Trends in Data Type Specification.* Informatik Fachberichte 116, Springer 1985, 156-170.

[16] Krieg-Brückner, B.: Systematic Transformation of Interface Specifications. *in:* Meertens, L.G.T.L. (ed.): *Program Specification and Transformation,* Proc. IFIP TC2 Working Conf. (Tölz '86). North Holland (1987) 269-291.

[17] Broy, M., Möller, B., Pepper, P., Wirsing, M.: Algebraic Implementations Preserve Program Correctness. *Science of Computer Programming 7* (1986) 35-53.

[18] Breu, M., Broy, M., Grünler, Th., Nickl, F.: PA^{nn}dA-S Semantics. PROSPECTRA Study Note M.2.1.S1-SN-1.3, Universität Passau, 1988.

[19] Kahrs, S.: PA^{nn}dA-S Standard Types. PROSPECTRA Study Note M.1.1.S1-SN-11.2, Universität Bremen, 1986.

[20] Ganzinger, H.: Ground Term Confluence in Parametric Conditional Equational Specifications. **in:** Brandenburg, F.J., Vidal-Naquet, G., Wirsing, M.(eds.): Proc. 4th Annual Symp. on Theoretical Aspects of Comp. Sci., Passau '87. *LNCS 247* (1987) 286-298.

[21] Ganzinger, H.: A Completion Procedure for Conditional Equations. Techn. Bericht No. 243, Fachbereich Informatik, Universität Dortmund, 1987 (to appear in *J. Symb. Comp.*)

[22] Bauer, F.L., Wössner, H.: *Algorithmic Language and Program Development.* Springer 1982.

[23] Kahrs, S.: From Constructive Specifications to Algorithmic Specifications. PROSPECTRA Study Note M.3.1.S1-SN-1.2, Universität Bremen, 1986.

[24] Pepper, P.: A Simple Calculus of Program Transformations (Inclusive of Induction). *Science of Computer Programming 9: 3* (1987) 221-262.

[25] Krieg-Brückner, B.: Formalisation of Developments: An Algebraic Approach. *in:* Rogers, M. W. (ed.): *Achievements and Impact.* Proc. ESPRIT Conf. 87. North Holland (1987) 491-501.

[26] Krieg-Brückner, B.: Algebraic Formalisation of Program Development by Transformation. *in:* Proc. European Symposium On Programming '88, *LNCS 300* (1988) 34-48.

[27] Broy, M.: Predicative Specification for Functional Programs Describing Communicating Networks. *Information Processing Letters 25:2* (1987) 93-101.

[28] Broy, M.: An Example for the Design of Distributed Systems in a Formal Setting: The Lift Problem. Universität Passau, Tech. Rep. MIP 8802 (1988).

[29] Wile, D. S..: Program Developments: Formal Explanations of Implementations. *CACM 26:* 11 (1983) 902-911. *also in:* Agresti, W. A. (ed.): *New Paradigms for Software Development.* IEEE Computer Society Press / North Holland (1986) 239-248.

[30] Steinbrüggen, R..: Program Development using Transformational Expressions. Rep. TUM-I8206, Institut für Informatik, TU München, 1982.

[31] Feijs, L.M.G., Jonkers, H.B.M, Obbink, J.H., Koymans, p.P.J., Renardel de Lavalette, G.R., Rodenburg, P.M.: A Survey of the Design Language Cold. *in:* Proc. ESPRIT Conf. 86 (Results and Achievements). North Holland (1987) 631-644.

[32] Sintzoff, M.: Expressing Program Developments in a Design Calculus. *in:* Broy, M. (ed.): *Logic of Programming and Calculi of Discrete Design.* NATO ASI Series, Vol. F36, Springer (1987) 343-365.

[33] Jähnichen, S., Hussain, F.A., Weber, M.: Program Development Using a Design Calculus. *in:* Rogers, M. W. (ed.): *Results and Achievements,* Proc. ESPRIT Conf. '86 . North Holland (1987) 645-658.

[34] Bird, R.S.: Transformational Programming and the Paragraph Problem. *Science of Computer Programming 6* (1986) 159-189.

[35] Broy, M.: Equational Specification of Partial Higher Order Algebras. *in:* Broy, M. (ed.): *Logic of Programming and Calculi of Discrete Design.* NATO ASI Series, Vol. F36, Springer (1987) 185-241.

[36] Möller, B.: Algebraic Specification with Higher Order Operators. *in:* Meertens, L.G.T.L. (ed.): *Program Specification and Transformation,* Proc. IFIP TC2 Working Conf. (Tölz '86). North Holland (1987) 367-398.

[37] von Henke, F.W.: An Algebraic Approach to Data Types, Program Verification and Program Synthesis. *in:* Mazurkiewicz, A. (ed.): Mathematical Foundations of Computer Science 1976. *LNCS 45* (1976) 330-336.

[38] Gordon, M., Milner, R., Wadsworth, Ch.: Edinburgh LCF: A Mechanised Logic of Computation. *LNCS 78* .

[39] Smith, D.R.: Top-Down Synthesis of Divide-and-Conquer Algorithms. *Artificial Intelligence 27*:1 (1985) 43-95.

[40] Wile, D. S.: Organizing Programming Knowledge into Syntax Directed Experts. Proc. Int'l Workshop on Advanced Programming Environments (Trondheim). *LNCS 244* (1986) 551-565.

[41] Krieg-Brückner, B.: The PROSPECTRA Methodology of Program Development. *in:* Zalewski (ed.): Proc. IFIP/IFAC Working Conf. on HW and SW for Real Time Process Control (Warsaw). North Holland (1988) 257-271.

[42] Bird, R., Wadler, Ph.: *Introduction to Functional Programming*. Prentice Hall 1988.

[43] Nickl, F., Broy, M., Breu, M., Dederichs, F., Grünler, Th.: Towards a Semantics of Higher Order Specifications in PA^{nn}dA-S. PROSPECTRA Study Note M.2.1.S1-SN-2.0, Universität Passau, 1988.

[44] Karlsen, E., Joergensen, J., Krieg-Brückner, B.: Functionals in PA^{nn}dA-S. PROSPECTRA Study Note S.3.1.C1-SN-10.0, Dansk Datamatic Center, 1988.