Algebraic Formalisation of Program Development by Transformation[1]

Bernd Krieg-Brückner

FB3 Mathematik und Informatik, Universität Bremen
Postfach 330 440, D 2800 Bremen 33, FR Germany
email: mcvax!unido!ubrinf!bkb

A uniform treatment of algebraic specification is proposed to formalise data, programs, transformation rules, and program developments. It is shown by example that the development of 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.

1. Introduction

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 [1]. 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 [1], regular expressions over elementary steps [2], functional abstraction [3], and composition of logical inference rules [4, 5].

In Program Development by Transformation [6-8], the approach taken in the PROSPECTRA project [9, 10], 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 (chapters 2, 3), to derive basic transformation operations from these rules (chapter 4), to allow composition and functional abstraction (chapters 5, 6), and to regard development scripts as (compositions of) such transformation operations (chapter 7). Using all the results from program development based on algebraic specifications we can then reason about the development of 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. The Algebra of Programs

2.1. The Algebra of Data and the Algebra of Programs

In the PROSPECTRA project, loose algebraic specifications with partial functions and conditional equations [11, 12] are used to specify the properties of data and associated operations in PA^{nn}dA-S, the

[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)

PROSPECTRA Anna/Ada specification language [9, 10]. For example, the fact that, for the mathematical integers INT, (INT, *, 1) is a monoid could be specified as in (2.1-1).

Similarly, we can define the Abstract Syntax of a programming language such as PAnndA-S by an algebraically specified Abstract Data Type: trees in the Abstract Syntax correspond to terms in this algebra of (PAnndA-S) programs, non-terminals to sorts, tree constructor operations to constructor operations, etc. Most constructor operations are free, except for all operations corresponding to List or Sequence concatenation, see & in (2.1-2). In the case of STMT_SEQ, Empty corresponds to **null**; in Ada and & would correspond to ; in the concrete syntax for Pascal-like languages; cf. [13].

(2.1-1) Example: Algebra of Data: Monoid (INT, *, 1)

```
axiom ∀ X,Y,Z : INT ⇒
  (X * Y) * Z = X * (Y * Z),     1 * X = X,     X * 1 = X
```

(2.1-2) Example: (Syntactic) Algebra of Programs: Monoid (STMT_SEQ, &, Empty)

```
axiom ∀ R, S, T : STMT_SEQ ⇒
  ( R & S) & T = R & (S & T),    Empty & R = R ,    R & Empty = R
```

2.2. Concrete and Abstract Syntax; Notation

Although we are interested in 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, see (2.2-1). Schema variables (such as *P, E, EList, V*) are written with capital initials in the sequel, auxiliary and transformation functions in italics (cf. *OccursIn* in (3.3-2) and section 4.1). The brackets ⌈ ⌋ are used whenever a (nested) fragment of concrete syntax is introduced. Variables correspond in the example, repetition of (possibly distinct) actual parameter expressions *E* in ⌈ ⦃*E*, ⦄ ⌋ corresponds to the list *EList* of expressions (simplified parameter associations), etc. 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. Also, the typing and universal quantification of variables in equational axioms is usually omitted for brevity; it should be apparent from the context. The suffixes *List* and *Seq* are used to avoid ambiguity with variables for elements, , and ; are used to indicate list and sequence concatenation in the concrete syntax.

(2.2-1) Example: Correspondence between Concrete and Abstract Syntax: Procedure Call

```
⌈ P (⦃E, ⦄ V); ⌋   ≈     Call(P, EList & Exp(V))
```

2.3. Algebraic Semantics

In the approach of the algebraic definition of the semantics of a programming language (cf. [14]), 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 (see sections 3.3, 3.5 below). Each choice of model (each choice of sequentialisation by a compiler) is admissible. (In Ada, a program is erroneous, if the quasi-terminal semantic models for this program do not coincide.)

3. Transformation Rules

3.1. Examples of Transformation Rules

Consider the examples below; the transformation rules specify part of the transition from an applicative to an imperative language style, cf. [15-18]: the mapping of a function declaration to a procedure declaration with an additional result parameter (**out**), and the transformation of all calls (the body is not treated here).

(3.1-1) is actually a specialisation of a more general rule for the (arbitrary) Introduction ⇔ Elimination of a Declaration: here an additional declaration of a procedure P is introduced in the beginning of the transformation process, and an analogous rule would eliminate the declaration of F at the end.. Rule (3.1-2) transforms function calls on the right-hand-side of an assignment statement into procedure calls with out parameters. Rule (3.1-3) unnests expressions that might contain a call to F such that rule (3.1-2) can be applied. We are assuming a (sub)language (of Ada) where expression have no side-effects (see also section 3.2). Note that, in a proper formulation for these rules, the context has to be taken into account, for example to ensure that (3.1-2) is applied in the context of declarations for F and P as introduced by (3.1-1), or that the compatibility of declarations can be checked, for example the proper introduction of a new identifier P or X in (3.1-1). For lack of space, we are ignoring these considerations here; see [17,18] for a particular approach to a specification of context in an algebraic framework.

(3.1-1) Trafo Rule: Introduction ⇔ Elimination of Procedure Declaration

```
declare
  ⦃D1⦄
  function F(⦃FP1;⦄) return R;

  ⦃D2⦄
begin
  ⦃S⦄
end;
```

such that
- *P does not occur in any of the D2, S*
- *P is not in conflict with other declarations*

```
declare
  ⦃D1⦄
  function F(⦃FP1; ⦄) return R;
  procedure P(⦃FP1; ⦄X: out R);
  ⦃D2⦄
begin
  ⦃S⦄
end;
```

(3.1-2) Trafo Rule: Assignment with Function Call ⇔ Procedure Call with **out** Parameter

```
V := F(⦃E, ⦄);
```

```
P(⦃E, ⦄ V);
```

(3.1-3) Trafo Rule: Nested ⇔ Sequential Evaluation of Expressions in Statements

```
⦅ return E;
| if E then ⦃S1⦄ ⟦else ⦃S2⦄ ⟧ end if;
| while E loop ⦃S1⦄ end loop;
| W := G(⦃E1, ⦄ E ⦃, E2⦄);
| Q(⦃E1, ⦄ E ⦃, E2⦄); ⦆
```

such that
- *V does not occur in any of the E1, E, E2, S1, S2*
- *V is not in conflict with other declarations*

```
declare
  V: T;
begin
  V := E;
⦅ return V;
| if V then ⦃S1⦄ ⟦else ⦃S2⦄ ⟧ end if;
| while V loop ⦃S1⦄ V := E; end loop;
| W := G(⦃E1, ⦄ V ⦃, E2⦄);
| Q(⦃E1, ⦄ V ⦃, E2⦄); ⦆
end;
```

Assuming suitable context conditions (in particular for (3.1-2)), each rule is applicable by itself and correct as an individual rule. In a combined transformation (or "development"), all these rules are taken together and applied exhaustively in reverse order; they will then transform the function F and all its calls to the procedure P and F can be eliminated. Before we come to the issue of application strategies etc. in chapter 6, let us consider some rules in more detail.

3.2. Uni-Directional Rules: Relations

If the evaluation of an expression might have side-effects, that is for a semantics with distinct quasi-terminal models (see section 2.2 above), the rules (3.1-1) to (3.1-3) would have to be interpreted from left to right. A *uni-directional* transformation rule is 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. [11]) and formalises the intuitive 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.

3.3. Bi-Directional Rules: Equations

The major kind of transformation rules we are interested in is the *bi-directional transformation rule*, a pair of semantically equivalent terms (in sense of section 2.2 above): an *equation* in the algebra of programs that is provable by deductive or inductive reasoning against the semantics. All rules in this paper are of this kind (indicated by ⇔). All considerations about interpreting equations as rewrite rules apply (confluence, termination, completion [12], etc.), cf. section 4.2.

(3.3-1) shows rule (3.1-2) as an equation and as a translation of the concrete syntax to terms in the semantic algebra of programs, cf. also (2.2-1); a similar translation for (3.1-1) would look very involved. (3.3-2) shows the assignment case of (3.1-3); applicability conditions have been formalised using auxiliary functions; they make the equation conditional (cf. section 4.1 for auxiliary functions having an implicit context parameter, such as TypeNameOf or TypeOf).

(3.3-1) Trafo Rule: Assignment with Function Call ⇔ Procedure Call: as Equation

```
AssignStmt(V, Call(F, EList)) = Call(P, EList & Exp(V))
```

```
⌈ V := F ( EList ); ⌋= ⌈ P ( EList, V ); ⌋
```

(3.3-2) Trafo Rule: Collateral ⇔ Sequential Evaluation of Expressions

```
¬ OccursIn(V, ⌈ EList1, E, EList2 ⌋) ∧ TypeNameOf( E ) = T ∧ ¬ EqName(V, W ) →
  ⌈ W := G ( EList1, E, EList2 ); ⌋= ⌈ declare V: T; begin V := E; W := G ( EList1, V, EList2 ); end; ⌋
```

3.4. Sets of Transformation Rules

We may have already noticed in section 3.1 that each rule in a set of rules achieves a certain (sub)goal (possibly only when applied exhaustively) that makes another rule applicable. We will come back to this issue in section 7.2. For the time being let us consider different sets of rules that achieve the same effect.

(3.3-2) is a rule analogous to a rule for the sequentialisation of collateral (or multiple) assignments into sequences of individual assignments (cf. [19]).The efficiency of the result depends on the sequence of application to subexpressions, and on the choice between two alternatives if an alternative analogous to (3.4-3) is added.

(3.4-1) is a simple-minded specialisation that forces a sequentialisation from left to right. This way, it does not avoid auxiliary variables, but it has the advantage of being simple to apply, with a fixed non-deterministic strategy (see section 7).

(3.4-1) Trafo Rule: Collateral ⇔ Sequential Evaluation of Expressions (Left to Right)

```
¬ OccursIn( V, ⌈ VList, E, EList ⌋)) ∧ TypeNameOf( E ) = T ∧ IsVarList( VList ) ∧ ¬ IsVar( E ) →
  ⌈ W := G ( VList, E, EList ); ⌋ = ⌈ declare V: T; begin V := E; W := G ( VList, V, EList ); end; ⌋
```

(3.4-2) is a generalisation of (3.3-2) that combines its effect with an "elimination" of common subexpressions in (3.5-1). If the desired goal is to unnest all function calls, then the effect is achieved by either rule (if applied exhaustively); (3.4-2), however, yields a more efficient result. Similarly, (3.4-3) combined to a rule set with (3.4-2), or, analogously, with (3.4-1) or (3.3-2), further enhances efficiency.

(3.4-3) is a specialisation of the assignment case that avoids introduction of variables. It is typical that a further strengthening of an applicability condition might lead to a specialisation that allows an improvement in efficiency. One might wish to add an additional condition ¬ *IsVar(E)* in (3.3-2), 3.4-2, 3), (3.5-1, 2) that restricts the application of the rule such that no trivial assignments (corresponding to renamings of program variables) are produced.

(3.4-2) Trafo Rule: Nested ⇔ Sequential Evaluation of Expressions

```
OccursIn( E, Exp ) ∧ ¬ OccursIn( V, Exp ) ∧ TypeNameOf( E ) = T ∧ ¬ EqName( V, W ) →
  ⌈ W := Exp; ⌋ = ⌈ declare V: T; begin V := E; W := SubstByIn( E, V, Exp ); end; ⌋
```

(3.4-3) Trafo Rule: Nested ⇔ Sequential Evaluation of Expressions (Re-Use of Variables)

```
OccursIn( E , Exp ) ∧ ¬ OccursIn( V, Exp ) ∧ TypeOf( E) = TypeOf( V ) →
  ⌈ V := Exp; ⌋ = ⌈ V := E; V := SubstByIn( E, V, Exp ); ⌋,
```

The important observation is that all these (sets of) rules, when applied exhaustively, yield semantically equivalent sequentialisations. With respect to some efficiency metrics where minimisation of assignments and variable usage is a concern, however, they behave quite differently. Moreover, the order of application of a general rule (rather than simple-minded application from left-to-right) becomes of great importance. Each application strategy yields a different syntactic (normal) form.

3.5. Derivation of Transformation Rules

Let us now derive (3.4-2) from the generally applicable rule (3.5-1) as a start, see (3.5-2). We apply the usual derivation steps of substitution, application of an equational law, renaming of variables, etc.

(3.5-1) Trafo Rule: Multiple ⇔ Single Evaluation of Same Subexpression

```
IsSimpleStmt( S ) ∧ OccursIn( E, S ) ∧ ¬ OccursIn( V, S ) ∧ TypeNameOf( E ) = T →
  S = ⌈ declare V: T; begin V := E; SubstByIn( E, V, S ) end; ⌋
```

(3.5-2) Trafo Rule Derivation: Nested ⇔ Sequential Evaluation of Expressions

```
IsSimpleStmt( ⌈ W := Exp; ⌋) ∧ OccursIn( E, ⌈ W := Exp; ⌋) ∧ ¬ OccursIn( V, ⌈ W := Exp; ⌋)
  ∧ TypeNameOf( E ) = T →
  ⌈ W := Exp; ⌋ = ⌈ declare V: T; begin V := E; SubstByIn( E, V, ⌈ W := Exp; ⌋) end; ⌋
```

```
OccursIn( E, Exp ) ∧ ¬ OccursIn( V, Exp ) ∧ ¬ EqName( V, W ) ∧ TypeNameOf( E ) = T →
  ⌈ W := Exp; ⌋ = ⌈ declare V: T; begin V := E; W := SubstByIn( E, V, Exp ); end; ⌋
```

We continue in (3.5-3) by instantiating E in (3.5-2) by a call to F and therefore restricting the application of the general rule to only those cases that are necessary for the subsequent application of rule (3.3-1).

(3.5-3) Trafo Rule Derivation: Nested ⇔ Sequential Evaluation of Call to F

```
OccursIn(⌈ F ( EList ) ⌋, Exp ) ∧ ¬ OccursIn( V, Exp ) ∧ ¬ EqName( V, W ) ∧ TypeNameOf(⌈ F ( EList ) ⌋) = T →
  ⌈ W := Exp; ⌋ = ⌈ declare V: T; begin V := F ( EList ); W := SubstByIn(⌈ F ( EList ) ⌋, V, Exp ); end; ⌋
```

Now we apply an equational law, namely the transformation rule (3.3-1), to the function call to F on the right hand side. This way we finally end up with rule (3.5-4) that combines the effect of rules (3.1-2) and (3.1-3).

(3.5-4) Trafo Rule Derivation: Nested Evaluation of Function Call ⇔ Procedure Call

```
OccursIn(⌈ F ( EList ) ⌋, Exp ) ∧ ¬ OccursIn( V, Exp ) ∧ ¬ EqName( V, W ) ∧ TypeNameOf(⌈ F ( EList ) ⌋) = T →
  ⌈ W := Exp; ⌋ = ⌈ declare V: T; begin P ( EList, V ); W := SubstByIn(⌈ F ( EList ) ⌋, V, Exp ); end; ⌋
```

4. Transformation Operations

4.1. Auxiliary Operations, Applicability Predicates

We note a number of auxiliary functions and predicates in the above equations, such as *OccursIn* or *TypeOf*. These can be structurally defined (cf. [19]) and must hold over subterms or over a larger context of the actual rule application. Such functions correspond to derived or inherited attributes, resp., in an implementation of transformation rules by attributed tree transformations. They could be considered as auxiliary functions and predicates that are pre-defined in a transformation system for the language in question. The precise role of the implicit context parameter in these equations is presently unresolved (see also [17])

Analogously, additional auxiliary functions and predicates can be defined and tailored to the transformation rules and operations under consideration. Consider (4.1-1) and (4.1-2): they define applicability conditions of the rules (3.3-1) and (3.5-3), resp. In (4.1-2), the occurrence of a call to F (F shall be fixed by the context of the rule application) with some actual parameter list EList is postulated in X; in effect, EList is existentially quantified.

(4.1-1) Auxiliary Operation: Applicability Predicate *IsAssignFCall*

```
IsAssignFCall( ⌈ V := G( EList ) ⌋) = EqName(F, G),
¬ IsCall( Exp ) → IsAssignFCall( ⌈ V := Exp ⌋) = FALSE,
¬ IsAssignStmt( Stmt ) → IsAssignFCall( Stmt ) = FALSE
```

(4.1-2) Auxiliary Operation: Applicability Predicate *ContainsNestedFCall*

```
ContainsNestedFCall( X ) = OccursIn( ⌈ F( EList ) ⌋, X )
```

4.2. Transformation Operations: Homomorphisms

An elementary transformation operation can be constructed from (transformation rules, that is) equations in the semantic algebra in a straightforward way as a partial function in the *abstract syntactic algebra*, see (4.2-1): 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 normalisation in the syntactic algebra.

(4.2-1) Trafo Operation: Assignment with Function Call to Procedure Call

$$\mathit{TFCallToProc}(\lceil V := F(\mathrm{EList}); \rfloor) = \lceil P(\mathrm{EList}, V); \rfloor$$

4.3. Extension of the Domain

If we want to apply elementary transformations over a larger context with some tactics (see chapter 6.2, 7), we need to extend the domain of a partial function to larger terms, as in (4.3-1) for TFCallToProc. The first equation corresponds to the previous definition for TFCallToProc. The second extends the definition to the identity over STMT , negating the applicability condition denoted by the predicate IsAssignFCall, see (4.1-1); cf. also Try in section 7.3.

(4.3-1) Trafo Operation: Extension to STMT

$$\begin{array}{rl} & \mathit{TFCallToProc}(\lceil V := F(\mathrm{EList}); \rfloor) = \lceil P(\mathrm{EList}, V); \rfloor, \\ \neg\, \mathit{IsAssignFCall}(\mathrm{Stmt}) \rightarrow & \mathit{TFCallToProc}(\mathrm{Stmt}) = \mathrm{Stmt} \end{array}$$

5. Development of Transformation Operations

5.1. Loose Specification

(5.1-1) Trafo Operation: Extension to STMT (Loose Specification)

$$(\ \mathit{TFCallToProc}(\lceil V := F(\mathrm{EList}); \rfloor) = \lceil P(\mathrm{EList}, V); \rfloor)\ \vee\ (\mathit{TFCallToProc}(\mathrm{Stmt}) = \mathrm{Stmt}\ )$$

Compared with (4.3-1), the compact definition of (5.1-1) is also semantically correct since TFCallToProc is a homomorphism and therefore all values in the equivalence class denoted by the original rule are acceptable. Loose specifications allow several distinct (that is non-isomorphic) models. In this case the $\vee$ operator between equations has been used to allow an additional degree of freedom over classical horn-clause specifications, analogous to non-determinacy (for one approach cf. [20]). This version specifies a class of functions (one being the "syntactic" identity in the term algebra); the more explicit definition of (4.3-1) specifies a single function mapping to a canonical form: for each non-trivial application the function call is actually changed to a procedure call.

Such a simple definition is often convenient at the start of the development of a transformation operation to characterise its effect before turning to considerations of termination, efficiency etc.

5.2. Requirement and Design Specifications

In general, we would like to start with a *requirement specification* of a transformation operation before considering a particular *design specification*, possibly several design alternatives (cf. also section 3.5). The same kind of reasoning as in program development can be applied. Any of the designs is then either formally derived from or proved to be a (robustly) correct implementation of the requirement specification (cf. [10-12]).

As an example, consider the extension of the effect of TFCallToProc over STMT_SEQ. We can characterise the desired effect as in (5.2-1): TFCallToProc should be applied to every element of a sequence (alternatively: to some arbitrary element). (5.2-2) and (5.2-3) show two divide and conquer strategies for achieving this (cf. [21]), depending on the basic operations available on STMT_SEQ, a partition or left linear structural decomposition strategy is applied. In fact, we can abbreviate such strategies by functional abstraction using a functional as in (5.2-4), see section 6.1.

(5.2-1) Trafo Operation: Extension over STMT_SEQ: *Requirement Specification*

$$\begin{array}{l} \mathit{TFCallStmts}(\mathit{Empty}) = \mathit{Empty}, \\ 0 < I \wedge I \leq \mathit{Length}(\mathrm{SSeq}) \rightarrow\ \mathit{Select}(\mathit{TFCallStmts}(\mathrm{SSeq}), I) = \mathit{TFCallToProc}(\mathit{Select}(\mathrm{SSeq}, I)) \end{array}$$

(5.2-2) Trafo Operation: Extension over STMT_SEQ: *Design Specification:* Partition

```
TFCallStmts (Empty) = Empty,
TFCallStmts (SSeq1 & S & SSeq2) = TFCallStmts (SSeq1) & TFCallToProc (S) & TFCallStmts (SSeq2)
```

(5.2-3) Trafo Operation: Extension over STMT_SEQ: *Design Specification:* Linear Decomposition.

```
TFCallStmts (Empty) = Empty,
TFCallStmts (Add(S,R)) = Add(TFCallToProc (S), TFCallStmts (R))
```

(5.2-4) Trafo Operation: Extension over STMT_SEQ: *Design Specification:* Functional

```
TFCallStmts (SSeq) = MapStmtSeq (TFCallToProc )(SSeq)
```

6. Functionals

6.1. Restricted Functionals

Let us focus on this issue of functional abstraction in more detail. Higher order functions allow a substantial reduction of re-development effort (in analogy to parameterised data type specifications), just as in program development (cf. [22-24]).

Thus we can abstract the homomorphic extension over statement sequences in (5.1-1) to (5.1-3) to a (partially parameterised or "Curry'd") functional MapStmtSeq. (6.1-1) shows the signature, an abstract requirement specification and a particular design specification by partition.

(6.1-1) Functional: Extension over STMT_SEQ

```
function MapStmtSeq ( G: function (S: STMT) return STMT )
                        return function (SSeq: STMT_SEQ) return STMT_SEQ;
---------------------------------------------------------------------------
axiom for all G: function (S: STMT) return STMT; SSeq, SSeq1, SSeq2: STMT_SEQ; I: NATURAL ⇒
   MapStmtSeq (G)(Empty) = Empty,
   0 < I ∧ I ≤ Length(SSeq) → Select ( MapStmtSeq (G)(SSeq), I) = G (Select (SSeq, I));
---------------------------------------------------------------------------
   MapStmtSeq (G)(Empty) = Empty,
   MapStmtSeq (G)(SSeq1 & S & SSeq2) = MapStmtSeq (G)(SSeq1) & G(S) & MapStmtSeq (G)(SSeq2)
```

It is an interesting observation that most definitions of such 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; therefore the conformance to the theory of algebraic specification is evident in the restricted case. (Functionals of this restricted form can be transformed to Ada generics; instantiation is then explicit, cf. [19].) 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 the same effect as a polymorphic functional.

6.2. Homomorphic Extension Functionals

In fact, most of these functionals have the nature of *homomorphic extension* functionals (see [25]), in this case the structural extension of the effect of a (local) transformation or predicate over larger terms. In (6.2-2,3), *SomeWhere* and *SomeWherePred* extend a total function *F* or predicate *P* on simple statements over compound statements and statement sequences; they would be similarly defined for, and over, other kinds of terms. *SomeWhere* is a homomorphic extension functional from terms to terms, and *SomeWherePred* from terms to BOOLEAN.

Compare the differences in the definition of the homomorphic extension functionals *SomeWhere* and *EveryWhere:* note that the ∨ operator between equations has been used (cf. section 5.1) to indicate arbitrary choice between then or else part, for example. Thus the function denoting a particular occurrence of an application of *F* is in the specified class of functions. *AnyWherePred* is analogous to *SomeWherePred* but uses **and** instead of **or**.

(6.2-2) Functional: *SomeWhere* over Statements

```
IsSimpleStmt(Stmt) → SomeWhere (F) (Stmt) = F (Stmt),
( SomeWhere (F)(   ⌈ SSeq1; SSeq2 ⌋) = ⌈ SomeWhere (F) (SSeq1); SSeq2 ⌋ )  v
  (SomeWhere (F)(  ⌈ SSeq1; SSeq2 ⌋) = ⌈ SSeq1; SomeWhere (F) (SSeq2) ⌋ ),
( SomeWhere (F)(   ⌈ if B then SSeq1 else SSeq2 end if; ⌋ ) =
                   ⌈ if B then SomeWhere (F) (SSeq1) else SSeq2 end if; ⌋ )  v
  ( SomeWhere (F)( ⌈ if B then SSeq1  else SSeq2  end if; ⌋ ) =
                   ⌈ if B then SSeq1  else SomeWhere (F) (SSeq2)  end if; ⌋ ),
SomeWhere (F)(     ⌈ while B loop SSeq  end loop; ⌋ ) =
                   ⌈ while B loop SomeWhere (F) (SSeq) end loop; ⌋
```

(6.2-3) Functional: *SomeWherePred* over Statements

```
IsSimpleStmt(Stmt) → SomeWherePred (P) (Stmt) = P (Stmt),
SomeWherePred (P)(⌈SSeq1; SSeq2 ⌋) =
     SomeWherePred (P) (SSeq1) or SomeWherePred (P) (SSeq2),
SomeWherePred (P)(⌈ if B then SSeq1 else SSeq2 end if; ⌋ ) =
     SomeWherePred (P) (SSeq1) or SomeWherePred (P) (SSeq2),
SomeWherePred (P)(⌈ while B loop SSeq  end loop; ⌋ ) = SomeWherePred (P) (SSeq)
```

(6.2-4) Functional: *EveryWhere* over Statements

```
IsSimpleStmt(Stmt) → EveryWhere (F) (Stmt ) = F (Stmt),
EveryWhere (F)(  ⌈ SSeq1; SSeq2 ⌋) = ⌈ EveryWhere (F) (SSeq1) ; EveryWhere (F) (SSeq2) ⌋
EveryWhere (F)(  ⌈ if B then SSeq1 else SSeq2 end if; ⌋ ) =
                 ⌈ if B then EveryWhere (F) (SSeq1)  else EveryWhere (F) (SSeq2) end if; ⌋ ,
EveryWhere (F)(  ⌈ while B loop SSeq  end loop; ⌋ ) =
                 ⌈ while B loop EveryWhere (F) (SSeq) end loop; ⌋
```

7. Developments

7.1. 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.

7.2. Development Goals: Requirement Specifications

We have already stated in chapter 3.6 that the application of some set of rules often requires the satisfaction of some pre-condition established by (exhaustive application of) some other set of rules. Conversely, this condition can be considered to be a required post-condition of the previous set of rules, or a characteristic predicate for the respective transformation function. Let us call such a condition a *development goal*: it is a requirement specification for a function yet to be designed; see example in section 7.5 below.

If these conditions can be defined structurally (or "syntactically"), as we indeed hope will mostly be the case, 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. Note that 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. Transformation functions having structural normal forms as applicability conditions correspond to Wile's syntax directed experts [26].

7.3. Development Tactics: Transformals

In analogy to tacticals in [27], we might call some transformation functionals *transformals* since they embody application tactics or strategies, cf. *SomeWhere* and *EveryWhere* in section 6.2 above. Consider for example (7.3-1): if some transformation function *F* and its applicability condition *C* (including structural applicability, see section 7.2) are given, then *Try* provides a totalisation or extension to identity if *C* does not hold (cf. section 4.3).

(7.3-1) Functional: *Try*

```
¬ C(X) →  Try (F, C) (X) = X,
  C(X) →  Try (F, C) (X) = F(X)
```

IterateWhile can be used to apply a transformation function *F* as long as some condition *C* holds. Similarly, *IterateSomeWhile* iterates a local transformation function *F* as long as some local condition *C* holds somewhere.

(7.3-2) Functional: *IterateWhile, IterateSomeWhile*

```
¬ C(X) →  IterateWhile (F, C) (X) = X,
  C(X) →  IterateWhile (F, C) (X) = IterateWhile (F, C) (F(X))
TrySomeWhere(F, C) (X) = SomeWhere(Try (F, C)) (X),
IterateSomeWhile (F, C) (X) = IterateWhile (TrySomeWhere(F, C), SomeWherePred(C)) (X)
```

Note that we are dealing with loose specifications here (see section 5.1 above). *F* is applicable iff *C* holds and *Try* yields the identity otherwise. Thus *SomeWhere(Try (F, C))* includes a lot of unwanted identity solutions even if *F* was applicable somewhere since *SomeWhere* may not pick the right position for the application of *F*. We would expect a stronger specification such as *TrySomeWhere* in (7.3-3) which does indeed yield a proper application of *F* if it is applicable anywhere, that is a kind of *Try(SomeWhere(F),SomeWherePred(C))* in which *SomeWhere(F)* is always well-defined. Note that *TrySomeWhere* is still loosely specified if, for example, *F* can be successfully applied in several sub-sequences; also, it still yields the identity if *F* can nowhere be applied. This conforms with the original specification and intentions.

(7.3-3) Functional: *TrySomeWhere* over Statements

```
¬SomeWherePred(C) (X) → TrySomeWhere (F, C) (X) = X,
IsSimpleStmt(Stmt) ∧ C(Stmt) → TrySomeWhere (F) (Stmt) = F (Stmt),
( SomeWherePred(C) (SSeq1) →
    TrySomeWhere (F, C)(⌈ SSeq1; SSeq2 ⌋) = ⌈ TrySomeWhere (F, C) (SSeq1); SSeq2 ⌋ )  ∨
  (SomeWherePred(C) (SSeq2) →
    TrySomeWhere (F, C)(⌈ SSeq1; SSeq2 ⌋) = ⌈ SSeq1; TrySomeWhere (F, C) (SSeq2) ⌋ ),
... and so on analogously
```

As far as possible, we would like to achieve the same strategic effect (the same development goal) by transformals such as *IterateSomeWhile(F, C), Sweep(Try (F, C))* or even *EveryWhere(Try (F, C))* by different, increasingly more efficient, application tactics (cf. (7.3-3, 4), (6.2-3)). A transformation from one tactic to another is possible by development rules, see the next section and an example in section 7.5.

(7.3-4) Functional: *Sweep* over Statements

```
IsSimpleStmt(Stmt) → Sweep (F) (Stmt ) = F (Stmt),
Sweep (F)( ⌈ SSeq1; SSeq2 ⌋) = F (⌈ Sweep (F) (SSeq1) ; Sweep (F) (SSeq2) ⌋ ),
Sweep (F)( ⌈ if B then SSeq1 else SSeq2 end if; ⌋ ) =
        F( ⌈ if B then Sweep (F) (SSeq1) else Sweep (F) (SSeq2) end if; ⌋ ),
Sweep (F)( ⌈ while B loop SSeq end loop; ⌋ ) =
        F( ⌈ while B loop Sweep (F) (SSeq) end loop; ⌋ )
```

7.4. Development Rules: Equational Properties

Development rules, that is equational properties of development scripts, allow us to express and to reason about design alternatives or *alternative development tactics* (see (7.5-4) below), and to *simplify developments* by considering them as algebraic terms in the usual way. (7.4-1) gives us some simple properties of *Try*. The second follows immediately from the associativity of *AND* . Similarly, (7.4-2) follows from the definition of *Try* and *Iterate*. (7.4-3) shows the development of such a rule step by step, using rule (7.4-2). It may be used to simplify iterated application into bottom-up one-sweep application.

(7.4-1) Development Rule: "Associativity" of *Try*

```
Try(Try(F, AND(C1, C2)) (X) = Try(Try(F, C1), C2) (X),
AND(C1, C2) (X) = C1(X) and C2(X)
---
Try(Try(F, C1), C2) (X) = Try(Try(F, C2), C1) (X)
```

(7.4-2) Development Rule: Elimination of *Try* and *IterateWhile*

```
C(X) ∧¬ C(F(X)) →   Try (F, C) (X) = F(X),
C(X) ∧¬ C(F(X)) →   IterateWhile (F, C) (X) = F(X)
```

(7.4-3) Development Rule Derivation: *Iterate* ⇔ *Sweep*

```
IterateSomeWhile (F, C) (X) = IterateWhile (TrySomeWhere(F, C), SomeWherePred(C)) (X)
```

```
IterateSomeWhile (F, C) (X) = IterateWhile(Sweep(Try(F, C)), SomeWherePred(C)) (X)
```

```
SomeWherePred(C) (X) ∧¬ SomeWherePred(C) ( Sweep(Try(F, C)) (X) ) →
  IterateWhile(Sweep(Try(F, C)), SomeWherePred(C)) (X) = Sweep(Try(F, C)) (X)
```

```
SomeWherePred(C) (X) ∧¬ SomeWherePred(C) ( Sweep(Try(F, C)) (X) ) →
  IterateSomeWhile (F, C) (X) = Sweep(Try(F, C)) (X)
```

(7.4-4) shows an example: *SwapIf* swaps the then and else parts of a conditional if the condition is of the form **not** *B*. If *B* is not normalised, that is if it may contain further **not** prefixes, then an iteration is necessary. Otherwise a sweep suffices; we can prove the pre-condition of (7.4-3), that is that a single application at every node is enough.

(7.4-4) Trafo Function: *SwapIf*

```
SwapIf ( ⌈ if not B then SSeq1 else SSeq2 end if; ⌋ ) =
        ⌈ if B then SSeq2 else SSeq1 end if; ⌋
SwapIfApplicable ( ⌈ if B then SSeq1 else SSeq2 end if; ⌋ ) = IsUnaryNotExp(B),
SwapIfApplicable ( ⌈ while B loop SSeq end loop; ⌋ ) = FALSE, ...
---
SwapEveryIf(X) = IterateSomeWhile (SwapIf, SwapIfApplicable) (X)
```

```
¬ SomeWherePred (SeveralNots) (X) → SwapEveryIf(X) = Sweep (Try(SwapIf, SwapIfApplicable)) (X)
```

(7.4-5) shows the transition from sweep application to application at the "leaves" of a term only, if (we can prove that) the applicability condition of a transformation function *F* implies that it is applicable at "leaves" only (denoted by *IsSimple*). Here, a "leaf" is a subterm that is not further broken down by *EveryWhere*. *EveryWhere* is simpler than *Sweep* since the (structure of the) term itself is untouched and needs not be reconstructed, only "leaves" are transformed. We can think of it as simultaneous application at all "leaves", and, indeed, a parallel implementation would be possible.

(7.4-5) Development Rule: *Sweep* ⇔ *EveryWhere* over Simple Construct

```
Sweep(Try(F, IsSimple)) (X) = EveryWhere(F) (X)
Sweep(Try(Try(F, C), IsSimple)) (X) = EveryWhere(Try(F, C)) (X)
```

(7.4-6) looks not very meaningful just by itself. However, we may be able to simplify considerably by introducing specialised functions with local iteration tactics for each case, see example (7.5-5) below. In particular, local iteration can then often be expressed by explicit structural recursion in the range of one iteration step.

(7.4-6) Development Rule: Global ⇔ Local Iteration

```
IterateSomeWhile (F, C) (X) = Sweep(IterateSomeWhile (F, C)) (X)
```

7.5. A Development of a Development Script

As an example of a particular development, consider the development goals in (7.5-1), based on the applicability predicates in (4.1-1, 2). *NoNestedCall(F)* is a development goal for a transformation functional *TUnnestEveryCall(F)* in (7.5-2) corresponding to exhaustive application of rule (3.1-3) or the various rules for unnesting of expressions in the assignment statement, (3.3-2) to (3.4-3). These are not specific for *F*, but achieve the desired goal nevertheless since they unnest every call. (3.5-3) is specific; it could be made even more specific by insisting in *NoNestedCall* on nestedness, excluding that the assigned expression is a direct call to *F*. This way, the goal is satisfied with minimal changes.

At the same time, *NoNestedCall(F)* is part of *NoCall(F)* and thus pre-condition for *TElimFunctDecl(F)*. We can easily prove that *TEveryCallToProc(F, P)* and *IntroProcDecl(F, P)* are invariant w.r.t. *NoNestedCall(F)*, thus it is satisfied as a pre-condition of *TElimFunctDecl(F)*. This way, *TFunctToProc(F, P)* is correctly defined in (7.5-2) as a composition of the separate transformation functions. We assume that *F* is well-defined in *Block*.

(7.5-1) Development Goals: Function to Procedure

```
NoNestedCall(F)(X) = ¬ SomeWherePred(ContainsNestedCall(F))(X),
NoCall(F)(X) = NoNestedCall(F)(X) ∧ ¬ SomeWherePred(IsAssignCall(F))(X)
```

(7.5-2) Development Script: Function to Procedure

```
TUnnestEveryCall(F)(Block) =      IterateSomeWhile (TUnnestCall(F), ContainsNestedCall(F))(Block),
TEveryCallToProc(F, P)(Block) =   IterateSomeWhile (TCallToProc(F, P), IsAssignCall(F))(Block),
NoCall(F)(Block) →
   TElimFunctDecl(F)(Block) =     SomeWhere (TElimDecl(F))(Block)
-----------------------------------------------------------------------------------------------
TFunctToProc(F, P)(Block) =
   TElimFunctDecl(F) ( TEveryCallToProc(F, P) ( IntroProcDecl(F, P) ( TUnnestEveryCall(F) (Block) )))
```

Let us now try to apply some of the development rules of section 7.4 above to simplify iteration. Indeed, *TEveryCallToProc* can be simplified to *EveryWhere* iteration on simple statements since we can show that *IsAssignCall* implies *IsAssignStmt*, therefore *IsSimpleStmt*, see (4.1-1) and (7.5-3).

(7.5-3) Development Script Derivation: Simplification of *TEveryCallToProc*

```
TEveryCallToProc(F, P) (Block) =      EveryWhere(Try(TCallToProc(F, P), IsAssignCall(F))) (Block)
```

We can try to specialise to local iteration analogously to (7.4-6), see (7.5-4) for the major cases. Note that explicit recursion on *TUnnest* instead of "deep" iteration is used for loops, *Init*, and particularly for assignments. As a minor point, *TUnnest* is not yet quite complete since the obligation to find some *F* that is not in conflict still has to be fulfilled. Improvements analogously to (3.5-3) could still be made, avoiding the unnesting of the whole expression in favour of those subexpressions containing a call to *F*.

(7.5-4) Functional: *TUnnestEveryCall*: Local Iteration

```
TUnnestEveryCall(F) (Block)  =  Sweep(Try(TUnnest(F), ContainsNestedCall(F))) (Block)
------------------------------------------------------------------------------------------
ContainsNestedCall(F) (E) ∧ ¬ OccursIn(V, E)  →
   TUnnest(F) (⌈ return E ; ⌋)  =  Init(V, E) (⌈ return V ; ⌋),
ContainsNestedCall(F) (E) ∧ ¬ OccursIn(V, E) ∧ ¬ OccursIn(V, SSeq1 ) ∧ ¬ OccursIn(V, SSeq2 )  →
   TUnnest(F) ( ⌈ if E then SSeq1 else SSeq2 end if; ⌋ )  =
      Init(V, E) ( ⌈ if V then SSeq1 else SSeq2 end if; ⌋ ),
ContainsNestedCall(F) (E) ∧ ¬ OccursIn(V, E) ∧ ¬ OccursIn(V, SSeq )  →
   TUnnest(F) ( ⌈ while E loop SSeq  end loop; ⌋ )  =
      Init(V, E) (⌈ while V loop SSeq; TUnnest(F) ( ⌈ V := E; ⌋ ) end loop; ⌋ ),
OccursIn(⌈ F( EList ) ⌋, Exp) ∧ ¬ OccursIn(V, Exp)  →
   TUnnest(F) (⌈ W := Exp; ⌋ )  =  Init(V,⌈ F( EList ) ⌋,) (⌈ W := SubstByIn(⌈ F( EList ) ⌋, V, Exp); ⌋ ),
... analogously for procedure call
------------------------------------------------------------------------------------------
¬ OccursIn(V, E) ∧ ¬ OccursIn(V, Stmt) ∧ TypeNameOf( E)  =  T  →
   Init(V, E) (Stmt)  =  ⌈ declare V : T; begin TUnnest(F) ( ⌈ V := E; ⌋ ) Stmt end; ⌋ )
```

We note that it makes no difference in (7.5-2) whether to introduce the procedure declaration before or after normalisation of the function calls. This can been expressed by a re-ordering property as in (7.5-4). Thus we can combine *TEveryCallToProc(F, P) TUnnestEveryCall(F)* into one functional by unfold-fold of *TEveryCallToProc*, in other words by applying *TEveryCallToProc* to every call to F in the range of *TUnnestEveryCall* directly. This combination had been done for assignments in (3.5-4); the only change in (7.5-4) would be to unfold *Init* in the assignment case and to replace the assignment to *V* by a procedure call.

(7.5-5) Development Rule: Reordering Property of Transformations

```
TUnnestEveryCall(F)  ( IntroProcDecl(F, P) (Block) ) =  IntroProcDecl(F, P)  ( TUnnestEveryCall(F)  (Block) )
```

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. [9,10,15]. Intermediate loose specifications could be made operational by some functional language with non-deterministic pattern matching and backtracking. Such a language is presently designed and implemented in the PROSPECTRA project; see [16] for a first approach.

8. Conclusion

It has been demonstrated that the methodology for program development based on the concept of algebraic specification of data types 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 algebraic 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 [26]. 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.

Acknowledgements

I wish to thank M. Broy, H. Ganzinger, B. Gersdorf, S. Kahrs, D. Plump, and Z. Qian for helpful criticism and comments.

References

[1] 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.

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

[3] Feijs, L.M.G., Jonkers, H.B.M, Obbink, J.H., Koymans, C.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.

[4] 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.

[5] 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.

[6] Bauer, F.L., Berghammer, R., Broy, M., Dosch, W., Geiselbrechtinger, F., Gnatz, R., Hangel, E., Hesse, W., Krieg-Brückner, B., Laut, A., Matzner, T., 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,* 1985.

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

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

[9] Krieg-Brückner, B., Hoffmann, B., Ganzinger, H., Broy, M., Wilhelm, R., Möncke, U., Weisgerber, B., McGettrick, AA.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.

[10] 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.

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

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

[13] Hoare, C.A.R., Hayes, I.J., He, J.F., Morgan, C.C., Roscoe, A.W., Sanders, J.W., Sorensen, I.H., Spivey, J.M., Sufrin, B.A.: Laws of Programming. *CACM 30: 8* (1987) 672-687.

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

[15] Krieg-Brückner: 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

[16] Heckmann, R.: A Functional Language for the Specification of Complex Tree Transformation. *this volume.*

[17] Qian, Z.: Structured Contextual Rewriting. Proc. Int'l Conf. on Rewriting Techniques and Applications (Bordeaux). *LNCS 256* (1987) 168-179.

[18] Qian, Z.: Recursive Presentation of Program Transformation. PROSPECTRA Report M.1.1.S1-SN-17.1, Universität Bremen (1986).

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

[20] Broy, M.: Predicative Specification for Functional Programs Describing Communicating Networks. *IPL 25* (1987) 93-101.

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

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

[23] 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.

[24] 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.

[25] 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.

[26] Wile, D. S.: Organizing Programming Knowledge into Syntax Directed Experts. USC/ISI manuscript (1987).

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