Science of Computer Programming 9 (1987) 221-262
North-Holland

A SIMPLE CALCULUS FOR PROGRAM TRANSFORMATION (INCLUSIVE OF INDUCTION)*

Peter PEPPER

Fachbereich Informatik, Technische Universität Berlin, 1000 Berlin 10, Fed. Rep. Germany

Communicated by M. Sintzoff
Received November 1984
Revised March 1987

Abstract. A basic purpose of transformation systems is the application of 'correctness-preserving rules' in order to derive from given programs new, 'equivalent' ones. An important aspect here is the usage of induction principles, without which transformation systems would have too limited power.

The paper presents a formal system of 'transformation rules' that incorporates induction. This system is a kind of 'Gentzen-style calculus', impoverished, however, to a degree that just meets the needs of program transformation. Thus we achieve a basis for the design of transformation systems, which is both simple and sound.

Prologue

The calculus presented below is unusual in a number of respects.

Firstly, it is not motivated by any metamathematical or aesthetical considerations, but rather by purely pragmatic needs of a given software project, namely the *design of a transformation system.* Although a formal calculus is needed as the basis of such a system, this very fact shall be hidden from the actual user of the resulting system. (To this user, each activity shall have the appearance of a goal-directed transformation step, and not of a proof step in a calculus). Because of this hiding effect we can streamline the calculus precisely to the needs of program transformation, giving priority to ease of implementation over ease of reading. It is for this reason that we content ourselves, e.g., with just two levels of nested Horn clauses rather than working with arbitrary nesting; analogously, we use special 'scheme variables' and 'indeterminates' in the place of arbitrary universal and existential quantification (in order to save the associated technical problems with binding and scoping). So, whenever we speak of 'simplicity' in connection with our calculus, we have implementation costs in mind.

* This work was carried out within the Sonderforschungsbereich 49, Programmiertechnik, Munich.

Secondly, we do not primarily aim at those properties that logicians classically try to achieve for their calculi:

- *Completeness* of the calculus could at best be attained relative to the completeness of the underlying atomic predicates. Since—in our case—these predicates typically express equivalences of programs, this completeness is lacking for any reasonably powerful programming language.
- *Expressive power* in the sense of 'primitive vs. general recursion', 'arithmetical hierarchy', etc. is no criterion for us; we measure the expressive power solely by the requirements of our software project.
- Questions of *decidability* of satisfiability and the *complexities* of the corresponding decision procedures are of no relevance in our context.

The only metalogical property that we actually do have to require of our calculus is, of course, *soundness*. In addition, *minimization* of the rules/axioms is a highly attractive aim in our situation; but in case of conflict we would give preference to naturalness over minimality.

Thirdly, it is not our ambition to invent new paradigms of logical calculi; on the contrary, we are anxious to utilize as much of existing calculi as possible. Thence, our calculus is obtained by curtailing known logical systems of those parts that are not needed in our context.

Finally, the framework for the design of our calculus is program transformation, a method for *constructing* new programs from given ones by virtue of strictly formalized rules. So it is just natural that the underlying calculus will be one of *constructive logic*.

We believe that the contribution of this paper (if any) is the demonstration that and how a concrete software project can be based on a formal calculus, even if certain aspects of this calculus may look peculiar to logicians.

0. Introduction: The paradigm of program transformation

Program transformation means the stepwise development of programs

$$P_0 \xrightarrow{T_1} P_1 \xrightarrow{T_2} P_2 \longrightarrow \cdots \longrightarrow P_{n-1} \xrightarrow{T_n} P_n$$

where the individual transitions $T_i : P_{i-1} \to P_i$ are done according to strictly formal rules. In this paper we present a formal calculus that specifies the central activities that take place during a transformational program development. (For more information about the methods and principles of program transformation we refer to the literature, in particular to [2] and some papers in [32]; further references can be found in these two books.)

Example 0.1. Consider the following transformation rule which codifies a standard technique for 'recursion removal'. We use here a PASCAL-like notation for programs. $A[x]$, $B[x]$, etc. stand for arbitrary expressions that possibly contain occurrences of the identifier x. (The notation of transformation rules is that of [2].)

function $f(x:\mathbf{m})$: **r**;
  if $B[x]$ **then** $f := D[x]$
    else $f := E[A[x], f(C[x])]$ **endif**

$\Downarrow$

$\lceil \forall a, b, c: E[a, E[b, c]] \equiv E[E[a, b], c]$
$\lfloor$ New$[z]$, New$[x_0]$

function $f(x_0:\mathbf{m})$: **r**;
  if $B[x_0]$
    then $f := D[x_0]$
    else var $(x:\mathbf{m}, z:\mathbf{r}) := (C[x_0], A[x_0])$
      while $\neg B[x]$
      do $(x, z) := (C[x], E[z, A[x]])$
      enddo;
      $f := E[z, D[x]]$
  endif

The above rule states that the original declaration of the function f can be replaced by the new declaration, provided that the expression E is associative. The predicate *New*$[z]$ actually is an abbreviation for a collection of predicates *NotOccurs*$[z, B]$, *NotOccurs*$[z, D], \ldots$, which together guarantee that no name clashes are introduced.

To demonstrate the application of such rules we consider the following function that searches for a minimal element of a nonempty sequence:

function *MinSearch*$(s:\mathbf{sequ})$:**elem**;
  if *length*$(s) = 1$ **then** *MinSearch* $:=$ *s*.1
    else *MinSearch* $:=$ *min*(*s*.1, *MinSearch*(*s.rest*)) **endif**

A *matching* of the input template of the aforementioned rule with this program yields the *instantiation*

$$\theta = \{f \triangleq \mathit{MinSearch}, x \triangleq s, \ldots, E[x, y] \triangleq \mathit{min}(x, y), C[x] \triangleq s.\mathit{rest}\}.$$

(The notion of instantiation is the usual one that is known from *unification* problems; see e.g. [28, 34]). So, by applying the rule we obtain the new program:

function *MinSearch*$(s_0:\mathbf{sequ})$:**elem**;
  if *length*$(s_0) = 1$
    then *MinSearch* $:=$ s_0.1
    else var $(s:\mathbf{sequ}, m:\mathbf{elem}) := (s_0.\mathit{rest}, s_0.1)$;
      while $\neg$*length*$(s) = 1$
      do $(s, m) := (s.\mathit{rest}, \mathit{min}(m, s.1))$
      enddo;
      MinSearch $:=$ *min*$(m, s.1)$
  endif

This transition is correct, since *min* is indeed associative; that is, the instantiated formula

$$\forall a, b, c: \quad \mathit{min}(a, \mathit{min}(b, c)) \equiv \mathit{min}(\mathit{min}(a, b), c)$$

holds.

There are also transformations that need *context information.* This can sometimes be specified with the help of slightly more complex applicability conditions:

Example 0.2. Consider the transformation rule

if B **then** E_1 **else** E_2 **endif**

$$\Downarrow \quad \begin{cases} B \equiv \mathit{true} \Rightarrow E_1 \equiv F_1 \\ B \equiv \mathit{false} \Rightarrow E_2 \equiv F_2 \end{cases}$$

if B **then** F_1 **else** F_2 **endif**

This rule states that "E_1 can be replaced by F_1 in the **then**-branch, provided that the rule

$$\begin{array}{l} E_1 \\ \Downarrow \quad \{ \; B \equiv \mathit{true} \\ F_1 \end{array}$$

is valid" (analogously for E_2/F_2). This shows that our templates $\Downarrow$[may play the role of genuine transformations as well as that of applicability conditions. In order not to overload this notation we prefer in the sequel a more logic-oriented style and write the above rule as

$$\frac{\begin{array}{l} B \equiv \mathit{true} \rightarrow E_1 \equiv F_1 \\ B \equiv \mathit{false} \rightarrow E_2 \equiv F_2 \end{array}}{\textbf{if } B \textbf{ then } E_1 \textbf{ else } E_2 \textbf{ endif} \equiv \textbf{if } B \textbf{ then } F_1 \textbf{ else } F_2 \textbf{ endif}}$$

The step from the transformation paradigm to the logic paradigm simply comes from a slight change of viewpoint: In the sequence

$$P_0 \xrightarrow{T_1} P_1 \xrightarrow{T_2} P_2 \longrightarrow \cdots \longrightarrow P_{n-1} \xrightarrow{T_n} P_n$$

a transition from program P_i to program P_{i+1} now is read as the statement of the property "P_i is transformable into P_{i+1}" (where 'transformable' usually means 'semantically equivalent'). And the presence of applicability conditions $A_1, \ldots, A_n$ just means that we need to verify $A_1, \ldots, A_n$ in order to deduce the validity of the transition.

In the sequel, we present a formal system that allows us to define precisely the basic activities that take place in a transformational program development. In this calculus, one can see quite clearly how the following tasks are to be performed:
- application of transformations,
- verification of applicability conditions,
- use of induction principles,
- derivation of new rules.

Our study aims above all at a *unified* treatment of the actual transformation steps and the verification of applicability conditions. Moreover, it should be possible to

transform not only programs but also program schemes in order to allow the derivation of new transformation rules from existing ones.

One may view the calculus given below as a mathematical modelling of the transformation paradigm. It may thus serve as a kind of *requirements specification for transformation systems.* In the development of this calculus, our freedom of design choices is constrained by the following two requirements:

- The calculus must not become unduly complex.
- The system must be sound and robust.

The latter requirement just says that the system should prohibit the derivation of invalid results (rather than rely on a disciplined use by the programmer). This constraint sometimes conflicts with the first one, which states that we should, e.g., avoid predicate logic when propositional logic will do, or that we should abandon full propositional logic when positive implicational logic suffices. The calculus presented below is a compromise between these two goals. It can be seen essentially as a system of two levels of Horn clause logic (see Example 0.2), that is, as a Gentzen-like (intuitionistic) system, or as a system of 'consequence logic' in the sense of [25].

Remark. The terminology in the literature is quite diffuse here. We use an amalgamation of wordings that can be found in [15, 23, 25, 19, 20, 36, 21, 14, 13, 22, 26, 24, 17]. Moreover, we try to stress by our use of terminology the fact that we do not have full propositional/predicate logic here, but rather a system that is streamlined towards the needs of program transformation.

The remainder of the paper is organized as follows: The first part describes our 'calculus of transformations': Section 1 introduces the basic notions of this calculus, such as 'term', 'formula', 'clause', 'inference', and so forth; the treatment is based on an algebraic view of programming languages. Section 2 discusses some aspects of its realization by a mechanical transformation system, and Section 3 gives the rationale for the particular design of the calculus. The second part of the paper considers the incorporation of the two major induction principles that are available in Computer Science: Section 4 presents 'computational induction' and its derivatives, and Section 5 discusses 'term induction'. Some of the underlying semantical considerations are briefly discussed in Appendix A.

Part 1. A calculus of program transformations

In the sequel we give a formal foundation for program transformations, based on algebraic principles. Moreover, we discuss some aspects of the technical realization of this calculus in computer-aided systems.

1. The definition of the calculus

In this section we define the syntactic constructions of our calculus as well as the admissible deduction rules. The semantic intuition behind the various constructs is given only informally; a more formal treatment is deferred to Appendix A.

1.1. An algebraic view of programs and transformations

In connection with the semantics of programing languages and in particular with program transformation, algebraic techniques have turned out to be most valuable. (For details we have to refer to the literature, e.g. [9, 10, 16].) We merely sketch here those aspects that are relevant for the following considerations.

Example. We view the syntax of programming languages as 'signatures', and thus programs as 'terms'. For example, the program fragment

while $x > 0$ **do** $x := x - y$ **enddo**

corresponds to the term

$$loop(apply(gt, x, zero), assign(x, apply(sub, x, y))).$$

(The relationship to parse trees is obvious.)

Terms

We start from the following algebraic basis:

- PL is the *signature* of a programming language (in the sense of [16]);
- V is a (countable) set of symbols, called *scheme variables*;
- $\hat{X}$ is a (countable) set of symbols, called *indeterminates*;
- a *program* is a well-formed term from the term algebra $W[\text{PL}]$;
- a *program scheme* is a well-formed term from $W[\text{PL}; V \cup \hat{X}]$;
- an *instantiation* $\theta: V \to W[\text{PL}; V \cup \hat{X}]$ assigns terms to scheme variables; $t\theta$ denotes the application of θ to the scheme variables of term t; that is, for a term t containing the scheme variables $x_1, \ldots, x_n$ the term $t\theta$ has the form $t[t_1/x_1, \ldots, t_n/x_n]$, where $t_i = \theta(x_i)$.

Semantic interpretation. We presuppose the existence of a semantic model for the programming language at hand:

- The *semantic model* M is a PL-algebra (in the sense of [10, 16]);
- $\mathbb{M}: W[\text{PL}] \to M$ is the 'morphism from syntax to semantics' that assigns to every well-formed program term its semantic interpretation.

In the case of nondeterministic programs, the model M is, of course, a relational algebra (see Appendix A.)

Notation. We let $a, b, c, \ldots, r, s, t$ range over terms from $W[\text{PL}; V \cup \hat{X}]$, $u, v, \ldots, z$ range over scheme variables from V, and $\hat{u}, \hat{v}, \ldots, \hat{z}$ range over indeterminates from $\hat{X}$.

Remarks. (i) The distinction of V and $\hat{X}$ has mainly technical reasons (see Section 3): instantiations θ are restricted to scheme variables from V. Thus, the indeterminates from $\hat{X}$ provide us with certain aspects of universal quantifications.

(ii) We use the notion 'term' for programs as well as for program schemes.

(iii) We admit only instantiations $t\theta$ that yield well-formed terms.

Formulas

Now we add the level of formulas over terms.

A **formula** is of the form

$$P[t_1, \ldots, t_n]$$

where P is a *predicate* symbol of arity n, and where $t_1, \ldots, t_n \in W[\text{PL}; V \cup \hat{X}]$ are terms.

Semantic interpretation. The definition of the predicates $P[\ldots]$ is part of the definition of the underlying programming language. Based on such a definition, the notions of *validity* and *satisfiability* then are defined as usual: The atomic formula $P[t_1, \ldots, t_n]$ is *valid*, iff for any substitution of the scheme variables and indeterminates occurring in the $t_1, \ldots, t_n$ by ground terms from $W[\text{PL}]$ the resulting ground formula $P[t'_1, \ldots, t'_n]$ evaluates to 'true'. (Analogously for satisfiability.)

Notation. We let $A, B, C, \ldots$ range over atomic formulas, and $\mathbb{A}, \mathbb{B}, \ldots$ over lists (i.e. conjunctions) of atomic formulas.

Remark. Conceptually (and pragmatically), one may distinguish between 'syntactic predicates' that are defined over $W[\text{PL}]$, and 'semantic predicates' that are defined over the semantic model M. (The former are generally decidable, whereas the latter usually are not—cf. [10].)

Examples of predicates. The following list illustrates syntactic and semantic predicates. Note that in the case of syntactic predicates we admit auxiliary functions (of functionality $W[\text{PL}] \to W[\text{PL}]$), an equality symbol, and (finite) sets. For further details on syntactic and semantic predictates we refer to [1, 7, 8].

Examples of syntactic predicates are:

- $Occurs[x, t]$. ⟪The identifier x occurs in the term t⟫.
- $m = Type[E]$. ⟪The expression E has type m⟫.
- $F = Declaration[f]$. ⟪The identifier f has the term F as the right-hand side of its declaration⟫.
- $GloVars[p] \subseteq \{v_1, \ldots, v_n\}$. ⟪Procedure p has at most the global variables $v_1, \ldots, v_n$⟫.

The most important semantic predicates are:

- $Equivalent[t_1, t_2]$, also denoted $t_1 \equiv t_2$. ⟪The programs t_1 and t_2 are semantically equivalent⟫.

- *Descendant*$[t_1, t_2]$, also denoted $t_1 \supseteq t_2$. ⟪Viewed as sets/relations, the interpretation of the (nondeterministic) program t_2 is a subset of the interpretation of t_1 (cf. [30])⟫.
- *LessDefined*$[t_1, t_2]$, also denoted $t_1 \sqsubseteq t_2$. ⟪The interpretation of program t_1 is 'less defined' than the interpretation of program t_2 (cf. [26, 35])⟫.
- *Defined*$[t]$. ⟪The interpretation of program t yields a defined value; that is, $\perp \notin \mathbb{M}[t]$⟫.
- *Determinate*$[t]$. ⟪The interpretation of program t yields a uniquely determined value; that is, $\text{card}(\mathbb{M}[t]) = 1$⟫.
- *Continuous*$[cn[\cdot]]$. ⟪The interpretation of $cn[\cdot]$—which is a relation—is continuous with respect to the Egli-Milner ordering (see Appendix A)⟫.

Remark. The binary semantic predicates *Equivalent*, *Descendant*, and *LessDefined* are the genuine focus of attention in a transformation system, since they realize the transitions $T_i: P_i \to P_{i+1}$. By contrast, the additional semantic predicates *Defined*, *Determinate*, *Continuous*, and so forth are less amenable to treatment within the framework of such systems. Therefore one often uses corresponding syntactic predicates that are at least sufficient to guarantee the desired semantic properties. So *Determinate* is frequently guaranteed by the absence of nondeterminate operators, and *Defined* by the absence of recursion/iteration; *Continuous* is deduced from the fact that certain language constructs are known to be continuous (as part of their semantic definition) and that the composition of continuous constructs is again continuous. Note that the semantic predicates *Defined* and *Determinate* are closely related to the 'equality test' $\cdot = \cdot$ (which is a strict, boolean-valued operation); we have

$$(e = e) \equiv true \;\to\; Defined[e]$$

$$(e = e) \equiv true \;\to\; Determinate[e]$$

where e is a (non-functional) expression, since for undefined e the equality test yields $\perp$, and for nondeterminate e it may yield true as well as false.

Simple equations such as

$$m = Type[E] \quad \text{or} \quad F = Declaration[f],$$

where the left-hand sides are scheme variables and the right-hand sides are applications of certain (language-dependent) functions, are used in the envisaged transformation system to calculate the respective instances automatically rather than requiring the user to provide them.

1.2. The calculus of clauses

The basic idea of our calculus is of constructive nature: Following the central principle of Gentzen-style calculi we focus on the notion that "a certain fact is derivable under certain assumptions".

A **clause** is of the form

$$A_1, \ldots, A_n \to B \quad \text{(or short: } \mathbb{A} \to B\text{)},$$

where $A_1, \ldots, A_n, B$ are atomic formulas. The formulas in the set $\mathbb{A} = \{A_1, \ldots, A_n\}$ are called *antecedents*, the formula B is called *consequent.*

Interpretation. From a deduction-theoretic point of view, a clause expresses the fact that we can deduce the consequent B from the antecedents $A_1, \ldots, A_n$.

Notation. We let $\alpha, \beta, \gamma, \ldots$ range over clauses, and $\Phi, \Psi, \ldots$ over sets of clauses. (As usual, the symbol ',' binds tighter than the symbol '$\to$'.)

Remark. Our use of the symbol '$\to$' follows the notation of Lorenzen [25]. Manna [26] uses '$\Rightarrow$' (which we have avoided, since we do not want to consider the figure "$\mathbb{A} \to B$" as a statement which—as a whole—is true if only $\mathbb{A}$ is false). Manna [26] calls our clauses 'assumption formulas', while Kleene [23] uses the word 'sequent' (as a translation of the German word 'Sequenz' that is used in [15]). Finally, Kleene [23] uses the word "succedent" instead of consequent.

Examples for clauses. The following list shows that not only classical transformation rules (such as Example 0.1) can be expressed as clauses:

(1) $\mathit{Defined}[x] \to \mathit{Equivalent}[\mathit{cond}(x, y, y), y]$,
(2) $\mathit{Descendant}[x, y],\ \mathit{Equivalent}[y, z] \to \mathit{Descendant}[x, z]$,
(3) $\mathit{Defined}[x],\ \mathit{LessDefined}[x, y] \to \mathit{Defined}[y]$,
(4) $\mathit{Occurs}[x, r] \to \mathit{Occurs}[x, \mathit{cond}(r, s, t)]$,
(5) $\to \mathit{Type}[\mathit{apply}(f, a)] = \mathit{ResultType}[f]$.

The validity of all these assertions must be established by the semantic definition of the programming language under consideration. (Note that (3) only holds in flat domains.)

Deductions

Let Γ be a given basic set of clauses ('assumptions'). Following, e.g., the terminology of [23] or [14] we define as *deductions* (from Γ) in our calculus sequences $\Delta = \langle F_1, \ldots, F_n \rangle$ of formulas such that for every k, $0 \leq k \leq n$, either

(i) the formula $F_k \in \Gamma$, or

(ii) the clause $(F_{i_1}, \ldots, F_{i_m} \to F_k)$ is in Γ, with $1 \leq i_j < k$. (That is, the formulas $F_{i_1}, \ldots, F_{i_m}$ occur before F_k in the deduction sequence Δ.)

(iii) $F = F_n$.

So a deduction consists of formulas that are in the set Γ of *assumptions* (step (i)) or that are immediate consequences of preceding formulas in the sequence (step (ii)).

Notation. As usual, we write

$$\Delta : \Gamma \vdash F$$

to express the fact that Δ is a deduction of the formula F from the assumptions Γ. (We omit Δ if it is clear from the context or if it is not relevant.)

Inference-rules

Next we want to set up rules that allow us to derive new clauses from given ones. We call—following [25]—such rules 'admissible', if any deduction Δ that is rendered possible by the new clause can be translated into a deduction Δ' that uses only the old clauses. formally:

Let $[R]$ be a rule that yields a clause β from clauses $\alpha_1, \ldots, \alpha_n$. Then $[R]$ is *admissible* iff for any set of clauses Γ, formula F, and deduction Δ there exists a deduction Δ' such that

$$\Delta : \Gamma \vdash F \quad \text{implies} \quad \Delta' : \Gamma' \vdash F,$$

where

$$\Gamma' = (\Gamma \backslash \{\beta\}) \cup \{\alpha_1, \ldots, \alpha_n\}.$$

There are three basic rules that are admissible in any calculus and that therefore lead to a 'meta-calculus' in the sense of [25]. These rules essentially date back to [15]. (Note that $\mathbb{A}$ and $\mathbb{B}$ stand for sets and thus may be empty.)

[I] *Tautologies*:
(i) $A \to A$
This is an axiom scheme for the meta-calculus.

[II] *Extension* of the antecedent:
(i) From $\mathbb{B} \to C$ infer $\mathbb{A}, \mathbb{B} \to C$.

[III] *Cut* (modus ponens):
(i) From $\mathbb{A} \to B$ and $B, \mathbb{B} \to C$ infer $\mathbb{A}, \mathbb{B} \to C$.

Admissibility of rules [I]-[III]

For the rules [I], [II], and [III] the admissibility is shown, e.g., in [25, pp. 40–46]. For instance, in the case of the (simplified) Cut rule [III]

"from $A \to B$ and $B \to C$ infer $A \to C$",

one demonstrates this as follows:

Let $\Delta = \langle F_1, \ldots, F_n \rangle$ be a deduction from Γ that uses the clause $(A \to C)$. Suppose that this clause is applied in the deduction Δ to derive the formula $F_k = C$ from the formula $F_i = A$ (with $i < k$). Then we insert into Δ in front of the formula F_k the formula B. The result obviously is a legal deduction Δ' from $\Gamma \backslash \{(A \to C)\} \cup \{(A \to B), (B \to C)\}$.

1.3. The calculus of inferences

We can make the inference rules of the previous section into formal objects themselves. This way we obtain a new calculus (one level 'higher' than the previous one), in which we can formalize the derivation of new clauses from given ones.

An **inference** is of the form

$$\frac{\begin{array}{c} A_{11}, \ldots, A_{1n_1} \to B_1 \\ \vdots \\ A_{m1}, \ldots, A_{mn_n} \to B_m \end{array}}{A_1, \ldots, A_n \to B} \qquad \begin{array}{c} \alpha_1 \\ \vdots \\ \left(\text{short } \dfrac{\alpha_m}{\beta} \text{ or } \dfrac{\Phi}{\beta}\right) \end{array}$$

where the $\alpha_1, \ldots, \alpha_m, \beta$ are clauses. The clauses α_i in the set $\Phi = \{\alpha_1, \ldots, \alpha_m\}$ are called *premises*, the clause β *conclusion.*

Interpretation. An inference represents an admissible rule in the sense of Section 1.2.

Remark. The above notation is used, e.g., in [15] or [23]. Some authors, e.g. Lorenzen [25], use the symbol '$\vdash$' to express inferences. Others, e.g., Shoenfield [36], Hilbert and Ackermann [20], or Gries [17], use a verbal form such as "from ... infer ...". In analogy to [27] (although our emphasis is quite different from theirs) we refer to the clauses in the premise as *goals*, and to the conclusions as (conditional) *assertions.*

Examples of inferences. A first example of an inference has already been given in the introduction. Other examples are induction rules, e.g. for the data type SET:

$$\frac{\begin{array}{l} P[\![emptyset]\!] \\ P[\![singleton(\hat{x})]\!] \\ P[\![\hat{r}]\!],\ P[\![\hat{s}]\!] \to P[\![union(\hat{r}, \hat{s})]\!] \end{array}}{P[\![s]\!]}$$

where $P[\![s]\!]$ denotes an atomic formula in the scheme variable s, for instance $Equivalent[\![union(r, s), union(s, r)]\!]$. Note the use of indeterminates and scheme variables in the above rule (which will be explained later on).

But also elementary transformations can be given in the form of inferences (rather than as clauses):

$$\frac{Defined[\![b]\!]}{Equivalent[\![cond(a, cond(b, r, s), u), cond(b, cond(a, r, u), cond(a, s, u))]\!]}$$

Meta-deductions

In the new calculus we have again a notion of 'deduction', which is defined in complete analogy to the one from the previous section. We just have to consistently replace 'formula' by 'clause' and 'clause' by 'inference'.

Meta-inferences

As in the case of clauses, we can again derive new inferences from given ones by virtue of 'meta-inferences'. There are four basic rules that are admissible here and that therefore lead to a 'meta-calculus' in the sense of [25]. Rule [IV] connects the two levels of inferences and meta-inferences. (Note that $\mathbb{A}$ and $\mathbb{B}$ as well as Φ and Ψ stand for sets and thus may be empty.)

[I] *Tautologies*:

(ii) $\dfrac{\alpha}{\alpha}$

This is an axiom schema for the meta-calculus.

[II] *Extension* of the antecedent/premise:

(ii) From $\dfrac{\Psi}{\alpha}$ infer $\dfrac{\Phi, \Psi}{\alpha}$.

[III] *Cut* (modus ponens):

(ii) From $\dfrac{\Phi}{\alpha}$ and $\dfrac{\overset{\alpha}{\Psi}}{\gamma}$ infer $\dfrac{\overset{\Phi}{\Psi}}{\gamma}$.

[IVa] *Importation*:

$$\text{From } \frac{\overset{\Phi}{\to A}}{\mathbb{B} \to C} \text{ infer } \frac{\Phi}{A, \mathbb{B} \to C}.$$

[IVb] *Exportation*:

$$\text{From } \frac{\Phi}{A, \mathbb{B} \to C} \text{ infer } \frac{\overset{\Phi}{\to A}}{\mathbb{B} \to C}.$$

In addition to these general rules, there is one further meta-inference that is oriented towards our underlying notion of program schemes:

[V] *Instantiation*:

$$\text{From } \frac{\Phi}{\alpha} \text{ infer } \frac{\Phi\theta}{\alpha\theta}$$

where θ is an instantiation of *scheme variables* by terms.

Interpretation. The soundness of the above meta-inferences is shown Appendix A.

Remark. In the practical realization of a transformation system these meta-inferences are realized by basic algorithms of the system. This means in particular that

there is no general schema matching on this level and consequently no need for 'variables for clauses' and the like.

Higher-level rules

The rules for clauses can now be written as formal inferences:

$$[\mathrm{I(i)}]: \frac{}{A \to A}.$$

$$[\mathrm{II(i)}]: \frac{\mathbb{B} \to C}{\mathbb{A}, \mathbb{B} \to C}.$$

$$[\mathrm{III(i)}]: \frac{\begin{array}{c}\mathbb{A} \to B \\ B, \mathbb{B} \to C\end{array}}{\mathbb{A}, \mathbb{B} \to C}.$$

All three inferences are derivable from the remaining meta-inferences (see Appendix A).

So our clauses represent a 'calculus of level 1', and the inferences represent a 'calculus of level 2'. Both levels are connected to each other through the rules [IVa, b]. Obviously, the same kind of relation holds again between inferences and meta-inferences. So the meta-inferences (the logical connective of which is written here as "from ... infer ...") form a 'calculus of level 3', which is related to the calculus of inferences by virtue of meta-meta-inferences analogous to the rules [IVa, b].

This construction can be continued to an arbitrary extent, leading to the 'consequence calculus' (German: Konsequenzkalkül) of Lorenzen [25]. Our specific application area (namely that of program transformation) allows us, however, to stop at level 2.

Remark. The thus constructed logic coincides with the intuitionistic calculus of 'positive implicational logic' (which is described in detail, e.g., in [21]).

1.4. Derived meta-inferences

We have kept the basic meta-inferences in Section 1.3 as simple as possible in the interest of easier readability. But in practical applications it is interesting to have more flexible variants available. Thus we have the following 'derived meta-inferences' (in the sense of [26]), which form an equally expressive system. (For the proofs of these derived meta-inferences, see Appendix A.)

[II*] *Extension* of selected antecedents:

$$\text{From } \frac{\Phi}{\mathbb{A} \to A \atop \mathbb{B} \to B} \text{ infer } \frac{\Phi}{\mathbb{A}, C \to A \atop \mathbb{B}, C \to B}.$$

(More generally, one can add antecedents to the conclusion and simultaneously to zero or more premises.)

Note that an extension in the conclusion is always possible (according to [II(i)]), whereas an extension in a premise necessitates a simultaneous extension in the conclusion.

[III*] *Modified cut*:

$$\text{From } \frac{\Phi}{\mathbb{A} \to A} \text{ and } \frac{\begin{matrix}\mathbb{A},\mathbb{B} \to A \\ \Psi\end{matrix}}{\gamma} \text{ infer } \frac{\begin{matrix}\Phi \\ \Psi\end{matrix}}{\gamma}.$$

[IVa*] *Modified importation*:

$$\text{From } \frac{\begin{matrix}\Phi \\ \mathbb{D} \to A\end{matrix}}{\mathbb{B} \to C} \text{ infer } \frac{\Phi}{A, \mathbb{B} \to C}.$$

Derived meta-inferences (language-dependent)

The predicates (such as *Equivalent*, *Determinate*, etc.) that are defined as part of the semantic specification of the programming language at hand exhibit certain properties that can be represented in the form of (language-dependent) clauses or inferences. The most important such properties are transitivity and monotonicity (see, e.g., [9]), which can be represented in the form of clauses. (In an algebraic approach to the specification of programming languages these clauses act as 'extra-logical axioms", that is, as properties which characterize the specific theory under consideration.)

Due to their distinguished role, certain axioms may be 'lifted' to the level of 'derived meta-inferences' (in the sense of [26]). More precisely, the system should contain procedures that apply such axioms automatically, either forward or backward ('goal reduction').

[VI] '*Transitivity*' *in assertions* (language-dependent): For illustration we consider the combination of the predicates $\cdot \supseteq \cdot$ (i.e. *Descendant*$[\cdot, \cdot]$ and $\cdot \equiv \cdot$ (i.e. *Equivalent*$[\cdot, \cdot]$). By virtue of Exportation and Modus ponens the language axiom

$$r \equiv s, s \supseteq t \;\to\; r \supseteq t$$

leads to the meta-inference

$$\text{From } \frac{\Phi}{\mathbb{A} \to r \equiv s} \text{ and } \frac{\Psi}{\mathbb{B} \to s \supseteq t} \text{ infer } \frac{\begin{matrix}\Phi \\ \Psi\end{matrix}}{\mathbb{A}, \mathbb{B} \to r \supseteq t}.$$

[VII] '*Transitivity*' *in goals* (language-dependent): As before, we consider the combination of the predicates $\cdot \supseteq \cdot$ and $\cdot \equiv \cdot$, but now applied to goals

rather than to assertions. The same axiom as in rule [VI] gives rise to the meta-inference

$$\text{From } \frac{\Phi}{\mathbb{A} \to r \supseteq t}_{\alpha} \text{ and } \frac{\Psi}{\mathbb{B} \to r \equiv s} \text{ infer } \frac{\Phi, \Psi}{\mathbb{A} \to \mathbb{B} \to s \supseteq t}_{\alpha}.$$

[VIII] *Monotonicity* (language-dependent): All language constructs should be monotone with respect to the relations $\cdot \equiv \cdot$, $\cdot \supseteq \cdot$, $\cdot \sqsubseteq \cdot$ (see [9]). so we have for example the language axiom (for any context $cn[\cdot]$)

$$r \supseteq s \to cn[r] \supseteq cn[s],$$

which yields the corresponding meta-inference

$$\text{From } \frac{\Phi}{\mathbb{A} \to r \supseteq s} \text{ infer } \frac{\Phi}{\mathbb{A} \to cn[r] \supseteq cn[s]}$$

for any context $cn[\cdot]$ from $W[\text{PL}; V \cup \hat{X}]$.

Remark. One gets variants of these meta-inferences for every possible combination of predicates like *Descendant*, *LessDefined*, and *Equivalent*—provided 'transitivity' actually holds. In other words, these meta-inferences realize applications of the following axioms:

	assertion	assertion		new assertion
resp.	new goal	assertion		given goal
resp.	assertion	new goal		given goal
	$r \equiv s$,	$s \equiv t$	$\to$	$r \equiv t$,
	$r \equiv s$,	$s \supseteq t$	$\to$	$r \supseteq t$,
	$r \equiv s$,	$s \sqsubseteq t$	$\to$	$r \sqsubseteq t$;
		$\vdots$		
	$r \sqsubseteq s$,	$s \sqsubseteq t$	$\to$	$r \sqsubseteq t$.

Note that the consequent always is the 'lower bound' of the two antecedents according to the partial order $\{Descendant < Equivalent, LessDefined < Equivalent\}$.

Similarly, we can reduce goals that are unary predicates such as *Determinate* or *Defined*. Here we realize the following rules:

new goal	assertion		given goal
Def[r],	$r \equiv s$	$\to$	Def[s],
Def[s],	$r \equiv s$	$\to$	Def[r],
Def[r],	$r \sqsubseteq s$	$\to$	Def[s],
Det[r],	$r \equiv s$	$\to$	Det[s],
Det[s],	$r \equiv s$	$\to$	Det[r],
Det[r],	$r \supseteq s$	$\to$	Det[s].

The importance of the derived rules [VI]-[VIII] lies in the following fact: The focus of attention in a transformation system is the derivation of new programs from given ones. In our terminology this means the establishment of assertions of the kind

$$\frac{\Phi}{\mathbb{A} \to r \equiv s}, \quad \frac{\Phi}{\mathbb{A} \to r \supseteq s}, \quad \text{or} \quad \frac{\Phi}{\mathbb{A} \to r \sqsubseteq s}$$

(if possible, with empty $\mathbb{A}$ and Φ). The derived meta-inferences [VI]-[VIII] just show, how such assertions can be combined with other assertions, or how they can be used in the reduction of premises (goals) to simpler subgoals (that is, in the verification of applicability conditions).

1.5. The role of free variables

It is well known from studies in formal logic that free variables have to be handled with great care (see, e.g., the extensive discussions in [23, pp. 94–151] or in [21, pp. 86–94 and 150–154]). The point is that the free variables must remain unaffected throughout a complete proof. To see the problem, we briefly present a wrong derivation.

Example (An illegal derivation). As has been demonstrated in the introduction, there are transformation rules that require certain algebraic properties as their applicability conditions. So let us suppose that we could infer the equivalence of two given functions f and g under the proviso that some operation 'A' has a neutral element 'e'. In predicate logic such a situation might be expressed as follows:

$$\forall x: ((\forall z: A[z, e] \equiv z) \Rightarrow f(x) \equiv g(x, e));$$

that is, we have a universal quantification on the left-hand side of an implication.

For predicate logic we have the so-called 'generalization rule' (cf. [36]):

$$\frac{B}{\forall x: B}.$$

So it would suffice to prove $A[z, e] \equiv e$ with the 'free variable' z in order to establish our desired equivalence $f(x) \equiv g(x, e)$ by way of modus ponens. Yet, the resulting inference

$$\frac{A[z, e] \equiv e}{f(x) \equiv g(x, e)}$$

is not valid in our calculus: When we interpret 'A', e.g., as multiplication, we can apply the instantiation $\theta = \{z \triangleq 0, e \triangleq 2, A[\cdot, \cdot] \triangleq \cdot * \cdot\}$ and obtain

$$\frac{0 * 2 = 0}{f(x) \equiv g(x, 2)}.$$

This is certainly not the intended effect, since '2' is not a neutral element for multiplication.

The only purpose of the above universal quantifier (in our context) is to prohibit illegal instantiations. To achieve this very effect, we have introduced the set $\hat{X}$ of indeterminates. So our inference actually must read ('generalization on constants', see [14] or [26]):

$$\frac{A[\hat{z}, e] \equiv \hat{z}}{f(x) \equiv g(x, e)}$$

with $\hat{z} \in \hat{X}$ being an indeterminate.

Note that—as in the case of universal quantification—we have to prove also here the validity of the premise:

$$A[\hat{z}, e] \equiv \hat{z},$$

which can also be achieved by proving

$$A[x, e] \equiv x$$

and then instantiating with $\theta = \{x \triangleq \hat{z}\}$. (See also [21] or [25].)

So, whenever we want to translate a predicate-logic formula with 'local' universal quantifiers into our formalism, we simply have to introduce indeterminates in the place of the bound variables.

Note that without such a device as our indeterminates we would not be able to express properties such as

"if $P[\![a]\!]$ holds for all values a, then Q holds"

(cf. [21, pp. 93–94]). On the other hand, our design allows us to achieve this effect without going into full predicate logic. Since indeterminates cannot be substituted for, they are necessarily 'held constant throughout subsidiary deductions' (in the terminology of [23]).

For the same reasons we have to limit the Instantiation Rule [V] to inferences. Here the resulting substitutions cannot do any harm with respect to subsidiary (meta-)deductions, because on this level we do not work with the 'meta-version' of the Importation Rule [IVa] (which corresponds to the well-known 'Deduction Theorem').

On the other hand, we cannot apply instantiations on the level of clauses; that is, an inference like

$$\frac{\mathbb{A} \to B}{\mathbb{A}\theta \to B\theta}$$

is not valid! If we would allow inferences such as the one above, the distinction between scheme variables and indeterminates would vanish. (I am grateful to one of the referees for pointing out this effect.) By using the inference

$$\frac{P[\![x]\!]}{P[\![\hat{x}]\!]}$$

we could infer from any inference of the kind

$$\frac{P[\![\hat{x}]\!]}{Q}$$

by the Cut Rule the new inference

$$\frac{P[\![x]\!]}{Q}.$$

However, the converse direction of the above inference is valid! That is, indeterminates in the premise can be replaced by scheme variables in the conclusion:

[Va] *Substitution* of indeterminates by scheme variables

$$\frac{\alpha\hat{\theta}}{\alpha}$$

where $\hat{\theta}$ associates some scheme variables to corresponding indeterminates.

Example: Using this mechanism, we can make, e.g., the associativity of a certain operation '$\cdot$' available for further transformations:

$$\frac{\hat{u}\cdot(\hat{v}\cdot\hat{w})\equiv(\hat{u}\cdot\hat{v})\cdot\hat{w}}{u\cdot(v\cdot w)\equiv(u\cdot v)\cdot w}.$$

An example for the application of this inference is given in Section 3 below.

Note that rule [Va] above is just the counterpart of the well-known Substitution Theorem of predicate logic (cf. [36]):

$$\vdash(\forall x\colon B[x])\Rightarrow B[a].$$

2. Representation of transformation tasks in the calculus

Now we demonstrate how the central tasks in transformational program developments can be explained in terms of our calculus.

Note that by virtue of the rules [IVa] and [IVb] we can make use of the following *simplifications*:

(i) Every transformation rule (such as Examples 0.1 and 0.2) is an inference.

(ii) The conclusion of every transformation rule is a clause with empty antecedent; that is, every inference has the form

$$\frac{\Phi}{A}$$

where A is an atomic formula.

Activity 1: Genuine transformation steps

Let $\rightarrowtail$ ('is transformable into') stand for any binary semantic predicate $Q[\cdot, \cdot]$, and suppose that a program development has already proceeded through k versions of a given program, that is,

$$p_1 \rightarrowtail p_2 \rightarrowtail \cdots \rightarrowtail p_k,$$

(where different occurrence of '$\rightarrowtail$' may stand for different predicates) and that we now want to apply the rule

$$\begin{array}{l} a \\ \Downarrow[\alpha_1, \ldots, \alpha_m \\ b \end{array}$$

which in our framework reads

$$[\mathrm{A}] \qquad \frac{\alpha_1, \ldots, \alpha_m}{a \rightarrowtail b}.$$

However, we usually just want to apply this rule to a small fragment t of $p_k = cn[t]$. Then we have to match the input template a against the fragment t of the program p_k, yielding—if successful—an instantiation θ. Thus we obtain the new version p_{k+1} by instantiating b accordingly, that is, $p_{k+1} =_{\mathrm{def}} cn[b\theta]$. In order to guarantee the legality of the transformation step, we will have to verify the—instantiated—applicability conditions $\alpha_1\theta, \ldots, \alpha_m\theta$. All this is expressible in our calculus as follows:

Let $p_k =_{\mathrm{def}} cn[t]$ be given. Let $\theta =_{\mathrm{def}} Match(a, t)$, that is, $a\theta = t$. Then we apply the Instantiation Rule [V] and obtain the new inference

$$[\mathrm{A}'] \qquad \frac{\alpha_1\theta, \ldots, \alpha_m\theta}{t \rightarrowtail b\theta}.$$

With the Monotonicity Rule [VIII] this becomes (since $p_k = cn[t]$)

$$[\mathrm{A}''] \qquad \frac{\alpha_1\theta, \ldots, \alpha_m\theta}{p_k \rightarrowtail cn[b\theta]}.$$

So our development has been extended to

$$\begin{array}{l} p_1 \rightarrowtail p_2 \rightarrowtail \cdots \rightarrowtail p_k \rightarrowtail p_{k+1} =_{\mathrm{def}} cn[b\theta] \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad | \\ \qquad\qquad\qquad\qquad\qquad\quad \alpha_1\theta, \ldots, \alpha_m\theta. \end{array}$$

Note that the transitions $p_i \rightarrowtail p_{i+1}$ are annotated by the goals to be verified (which just is another graphical representation for inferences).

Summing up, a genuine transformation step consists of a composition of the three basic operations

- matching,
- instantiation,
- monotonicity.

Variations on the theme. The central aspect of the above process is the *instantiation* of a certain inference with an instantiation θ. Besides matching there are two other ways, in which such a θ is derived:

(i) *Indefinite rules.* These are rules, where the output template contains scheme variables that are not present in the input template (and thus cannot be instantiated through matching). An example is given by rules for recursion removal, which require the existence of suitable inverses for operations (cf. [2, 12, 33]). Application of such a rule to a program that uses the operation *succ* from the type NAT thus yields the intermediate inference (cf. [4])

$$\frac{\begin{array}{c}\cdots\\ g(succ(\hat{x}))=\hat{x}\end{array}}{rec[succ] \rightarrowtail tailrec[g]}.$$

Now the user can provide the instantiation *pred* for *g*—or the system may find *pred* by browsing through the relevant data types—thus achieving

$$\frac{\begin{array}{c}\cdots\\ pred(succ(\hat{x}))=\hat{x}\end{array}}{rec[succ] \rightarrowtail tailrec[pred]}.$$

Since this law exists in the data type NAT, that is, since the assertion

$$pred(succ(\hat{x})) \equiv \hat{x}$$

is valid, the premise in the above inference 'vanishes' (see below).

(ii) *General programs.* Of course, one can use in the place of general pattern matching special-purpose algorithms that produce the envisaged instantiation θ. Typical candidates are the fold- and unfold-rules from [11]:

$$\begin{array}{cc}\textbf{unfold} & \textbf{fold}\\ \dfrac{\begin{array}{c}F = Declaration[f]\\ x = Form\ Param[f]\end{array}}{f(a) \sqsubseteq F_x^a} & \dfrac{\begin{array}{c}F = Declaration[f]\\ x = Form\ Param[f]\end{array}}{F_x^a \sqsupseteq f(a)}.\end{array}$$

If we apply the fold rule to a given expression E, we must check whether there is a suitable expression a such that $F_x^a = E$. In other words, here we have to treat the parameters of the body of f like scheme variables during the matching of E with F.

More sophisticated algorithms could be employed in the verification of applicability conditions (see below). Here it may be interesting to cut a predicate like

$$r \sqsupseteq t$$

into two predicates

$$r \sqsupseteq s \sqsupseteq t.$$

This means that a suitable term s should be found based on the forms of both r and t.

Activity 2: Compactification of development histories

Suppose that we have a development sequence of the form

$$\begin{array}{ccccc} p_1 & \rightarrowtail & p_2 & \rightarrowtail \cdots \rightarrowtail & p_k \\ | & & | & & \\ \Phi_1 & \cdots & \Phi_{k-1} & & \end{array}$$

but that we are actually interested in the relationship between p_1 and p_k. (Recall that '$\rightarrowtail$' may stand for different semantic predicates such as $\cdot \cong \cdot$, $\cdot \supseteq \cdot$, or $\cdot \sqsubseteq \cdot$.) So we have to apply repeatedly a compactification of the following kind:

Consider

$$\begin{array}{ccccccc} \cdots \rightarrowtail & p_{i-1} & \rightarrowtail & p_i & \rightarrowtail & p_{i+1} & \rightarrowtail \cdots \\ & | & & | & & & \\ & \Phi_{i-1} & & \Phi_i & & & \end{array}$$

which is nothing but another representation for the two inferences

$$\frac{\Phi_{i-1}}{p_{i-1} \rightarrowtail p_i}, \qquad \frac{\Phi_i}{p_i \rightarrowtail p_{i+1}}.$$

Now we apply the Transitivity Rule [VI] (provided that the two semantic relations $p_{i-1} \rightarrowtail p_i$ and $p_i \rightarrowtail p_{i+1}$ are indeed compatible) and obtain the new inference

$$\frac{\Phi_{i-1}, \Phi_i}{p_{i-1} \rightarrowtail p_{i+1}},$$

or, in graphical notation,

$$\begin{array}{ccccccc} \cdots \rightarrowtail & p_{i-1} & \rightarrowtail & p_{i+1} & \rightarrowtail \cdots \\ & | & & & \\ & \Phi_{i-1}, \Phi_i & & & \end{array}$$

So it is seen that Compactification of Developments is nothing but an application of the Transitivity Rule [VI].

Activity 3: Verification of applicability conditions

Suppose that we have a development step of the form

$$\begin{array}{ccccccc} \cdots \rightarrowtail & p_i & \rightarrowtail & p_{i+1} & \rightarrowtail \cdots \\ & | & & & \\ & \Phi_i = \{\alpha_1, \ldots, \alpha_m\} & & & \end{array}$$

and that we want to 'verify' the goal α_1. Suppose moreover that α_1 is of the most general form $\mathbb{A} \rightarrow A$. So we are actually dealing with an inference of the form

$$[*] \qquad \frac{\begin{array}{c}\mathbb{A} \rightarrow A \\ \alpha_2, \ldots, \alpha_m\end{array}}{p_i \rightarrowtail p_{i+1}}.$$

According to the Cut Rule [III], we can simplify this inference [*] to

$$[**] \qquad \frac{\alpha_2, \ldots, \alpha_m}{p_i \rightarrowtail p_{i+1}},$$

if we are able to derive the inference

$$[\#] \qquad \frac{\alpha_2, \ldots, \alpha_m}{\mathbb{A} \rightarrow A}.$$

Because of the Importation Rule [IVa] it suffices to establish the inference

$$[\#\#] \qquad \frac{\mathbb{A}\quad \alpha_2, \ldots, \alpha_m}{A}.$$

Technically speaking, the command "Verify α_1" initiates a subdevelopment where the clauses $\mathbb{A}, \alpha_2, \ldots, \alpha_m$ are made temporarily available as 'assumptions', until the desired conclusion A has been achieved. The logical justification of the correctness of this proceeding is that from [##] we infer [#], which can be applied to [*] in order to yield the simplified [**].

Activity 4: Reduction of goals

A complete verification of applicability conditions (as shown above) is not always possible, in particular when dealing with program schemes rather than with programs. But we may still be able to reduce certain goals to simpler subgoals. We presuppose the same situation as above, but now look more closely at the structure of the formula A. So we have a situation like

$$[*] \qquad \frac{r \rightarrowtail t \quad \alpha_2, \ldots, \alpha_m}{p_i \rightarrowtail p_{i+1}}.$$

As shown above, we start a development for the term r (or for t) under the assumptions $\alpha_2, \ldots, \alpha_m$. (This is a 'normal transformation activity of the kinds 1 and 2.) In the most general case, this subdevelopment establishes (after compactification) an inference of the form

$$[\#] \qquad \frac{\beta_1, \ldots, \beta_n, \alpha_2, \ldots, \alpha_m}{r \rightarrowtail s}$$

(where the $\beta_1, \ldots, \beta_n$ are a collection of additional applicability conditions that were encountered during the development from r to s).

By the Transitivity Rule [VII] this leads from [*] to the new inference

$$[**] \qquad \frac{s \rightarrowtail t \quad \beta_1, \ldots, \beta_n, \alpha_2, \ldots, \alpha_m}{p_i \rightarrowtail p_{i+1}}$$

(provided that the semantic relations in $r \rightarrowtail t$ and $r \rightarrowtail s$ are indeed compatible). So the goal

$$r \rightarrowtail t$$

has been replaced by the subgoals

$$(s \rightarrowtail t), \beta_1, \ldots, \beta_n.$$

Activity 5: 'Claims'

During a program development it will sometimes happen that one needs (for an activity of kind 1) a rule that does not yet exist in the catalogue. However, one cannot expect the user to always abandon his current development in such a situation in order to first establish the required rule. The system rather should allow him to 'claim' the validity of the rule.

In the calculus this just means that one adds the tautology (according to meta-inference [I])

$$\frac{}{A \rightarrow A} \quad \text{or} \quad \frac{\alpha}{\alpha}.$$

By the mechanisms demonstrated before, the use of such a tautology will establish the clauses A and α, respectively, as goals that remain to be verified.

3. Rationale

In this section we give the motivation for the particular design of the calculus outlined in the previous sections.

Recall that we have two 'global' motivations: The calculus

- must have sufficient *expressive power* to model the major activities that take place in transformational program development,
- should be as *simple* as possible, in particular with respect to its technical realization in a mechanical system.

It is the second requirement that rules out the use of full predicate logic here (in order to save binding and the associated α-reductions), and it is the combination of both requirements that leads to the distinction of scheme variables V and indeterminates $\hat{X}$.

The sufficiency of the expressive power has been shown in the previous section (and will be rounded off by the induction principles in Part 2 below). So we concentrate here on the other issues.

Universal quantification vs. indeterminates

Since the concept of 'indeterminates' is one of the unconventional features of our calculus, a further illustration of its purpose may be helpful. Consider two recursive

functions f and g that are characterized by the following equivalences (where A, B, C, D are terms and '$\cdot$' is some binary operation):

(1) $\quad \vdash f(x) \equiv \textbf{if } B \textbf{ then } f(A) \cdot C \textbf{ else } D \textbf{ endif},$

(2) $\quad \vdash g(x, y) \equiv f(x) \cdot y.$

We want to show (as part of a well-known paradigm for recursion removal—cf. [2]—that the following equivalence also holds:

(*) $\quad \vdash g(x, y) \equiv \textbf{if } B \textbf{ then } g(A, C \cdot y) \textbf{ else } D \cdot y \textbf{ endif},$

provided that '$\cdot$' is associative.

The corresponding deduction proceeds as follows:

We start from the associativity of '$\cdot$':

(3) $\quad \vdash u \cdot (v \cdot w) \equiv (u \cdot v) \cdot w.$

By the Instantiation Rule [V] we obtain

$$\vdash f(A) \cdot (C \cdot y) \equiv (f(A) \cdot C) \cdot y.$$

By the Monotonicity Rule [VIII] this yields

$$\vdash \textbf{if } B \textbf{ then } f(A) \cdot (C \cdot y) \textbf{ else } D \cdot y \textbf{ endif}$$

$$\equiv \textbf{if } B \textbf{ then } (f(A) \cdot C) \cdot y \textbf{ else } D \cdot y \textbf{ endif}.$$

One of the fundamental axioms for **if**-constructs then leads to

$$\vdash \textbf{if } B \textbf{ then } (f(A) \cdot C) \cdot y \textbf{ else } D \cdot y \textbf{ endif}$$

$$\equiv \textbf{if } B \textbf{ then } (f(A) \cdot C) \textbf{ else } D \textbf{ endif} \cdot y.$$

Now an application of (1) using the Transitivity Rule [VI], yields

$$\vdash \textbf{if } B \textbf{ then } f(A) \cdot (C \cdot y) \textbf{ else } D \cdot y \textbf{ endif} \equiv f(x) \cdot y.$$

Finally, by applying (2) twice, again using the Transitivity Rule [VI] we obtain

$$\vdash \textbf{if } B \textbf{ then } g(A, C \cdot y) \textbf{ else } D \cdot y \textbf{ endif} \equiv g(x, y),$$

which is the desired result.

If we want to freeze this development into the single inference

$$\frac{\begin{array}{l}\hat{u} \cdot (\hat{v} \cdot \hat{w}) \equiv (\hat{u} \cdot \hat{v}) \cdot \hat{w} \\ f(x) \equiv \textbf{if } B \textbf{ then } f(A) \cdot C \textbf{ else } D \textbf{ endif} \\ g(x, y) \equiv f(x) \cdot y\end{array}}{g(x, y) \equiv \textbf{if } B \textbf{ then } g(A, C \cdot y) \textbf{ else } D \cdot y \textbf{ endif}},$$

we have to convert the scheme variables u, v, w from (3) into indeterminates $\hat{u}$, $\hat{v}$, $\hat{w}$, since we have applied a 'local' instantiation to them during the proof. Therefore we must prohibit that other instantiations are applied to u, v, w in the context of the overall inference. Formally, this is achieved by applying rule [Va] (from Section 1.5) to the equation (3) above.

Syntactic predicates

Whereas it is the purpose of a transformational program development to establish one of the semantic relationships ($\equiv, \supseteq, \sqsubseteq$) between the initial version p_1 and the final version p_n of a given program, the syntactic predicates are only needed in certain applicability conditions, mainly in order to guarantee the observance of context conditions. Although it is possible to specify them axiomatically, e.g.

$$Occurs[x, B] \to Occurs[x, \textbf{if } B \textbf{ then } E \textbf{ else } F \textbf{ endif}],$$

they will in practice be implemented by simple (recursive) algorithms over the terms from $W[\text{PL}]$.

The only complication arises, when program schemes from $W[\text{PL}; V \cup \hat{X}]$ are to be transformed. Then the evaluation of a predicate like $Occurs[x, t]$ will usually not yield true or false but rather one or more predicates $Occurs[x, t_1], \ldots,$ $Occurs[x, t_n]$ for subterms t_i of t, the conjunction of which is equivalent to the original predicate.

Remark. Due to the simplicity of the predicates it is tempting to try to incorporate them into the language (more precisely, to enrich the language PL to a language EPL $\supseteq$ PL), rather than regarding them as being on the same (meta-) level as the semantic predicates. The reason why this does not work is illustrated by the following trivial example: The program fragment

$$\textbf{if } x+1 > x \textbf{ then } z := z * 2 \textbf{ else } x := x - 1 \textbf{ endif}$$

is semantically equivalent to (and thus can be transformed into) the fragment

$$z := z * 2,$$

(under the assumption that x has a defined value). However, the syntactic predicates

$$Occurs[x, \textbf{if} \ldots \textbf{endif}] \quad \text{and} \quad Occurs[x, z := z * 2]$$

are obviously different. So 'correct' transformations may be forbidden in particular contexts. (This just means that syntactic predicates are not monotonic with respect to the relation "is transformable into".)

Relationship to further proof principles

So far our only proof principle is that of 'modus ponens' (cf. rule [III(ii)]). However, in mathematics there are further important proof methods such as 'modus tollens', 'reductio ad absurdum' or 'case distinction' (see also [15]).

Reductio ad absurdum (proof by contradiction):

$$\text{From } \frac{\Phi}{A \to B} \text{ and } \frac{\Phi}{A \to \neg B} \text{ infer } \frac{\Phi}{\to \neg A}.$$

Modus tollens:

$$\text{From } \frac{\Phi}{A \to B} \text{ infer } \frac{\Phi}{\neg B \to \neg A}.$$

Case distinction:

$$\text{From } \frac{\Phi}{\to A \vee B} \text{ and } \frac{\Phi}{A \to C} \text{ and } \frac{\Phi}{B \to C} \text{ infer } \frac{\Phi}{\to C}.$$

We cannot include these rules into our calculus, since we do not even have the negation symbol '$\neg$' or the disjunction symbol '$\vee$' at our disposal. The reason for these omissions simply is that our calculus focusses on the notion of "is transformable into". In this context it does not make any sense to work with the negated form "is not transformable into" or with the ambiguous form "is transformable into . . or into . . .". (Recall that our aim just is to formalize existing concepts of program transformation by way of a calculus, and not to invent a system for program verification.)

Moreover, we can always introduce suitable pairs of predicates such as, e.g., *Defined*[. . .] and *Undefined*[. . .] (as it is usually also done in PROLOG-programming). Incidentally, we thus meet Griss' requirements for 'negationless mathematics' (cf. [19]).

Part 2. Induction principles

In connection with programming, there are two major induction principles available:

- *computational induction*, which is based on the fixed-point theorem of Kleene,
- *structural induction*, which is based on Noetherian orderings.

Computational induction is justified by the principle of 'approximation of functions'. It has been introduced by D. Scott, and it comprises in principle also 'recursion induction' [30] and 'fixpoint induction' [31].

Structural induction comprises as its most important special case *term induction* which is based on the generation principle for algebraic types (cf. [2, 37]).

We consider each of these principles in turn and demonstrate how they can be cast into the underlying *induction scheme*

$$\frac{\vdash A_{\text{init}} \quad \vdash A_{\text{induction step}}}{\vdash A}.$$

In our treatment, we are not interested in the greatest possible generalizations of these rules but—on the contrary—aim at technically simple (but still useful) instances.

All our induction principles ultimately go back to the principle of Noetherian induction. This principle is based on Noetherian partial orders (see e.g. [3]): A poset $(S, \prec)$ is *Noetherian* (well-founded), if every nonvoid subset of S has a minimal element. (Equivalently: if every descending chain is finite.)

So the following (first-order) formula is valid in a Noetherian poset $(S, \prec)$:

$$(*) \qquad (\forall a\colon (\forall b, b \prec a\colon A[\![b]\!]) \Rightarrow A[\![a]\!]) \Rightarrow (\forall x\colon A[\![x]\!]).$$

Accordingly, the following (first-order) inference is valid:

$$\frac{(\forall b, b \prec \hat{a}\colon A[\![b]\!]) \to A[\![\hat{a}]\!]}{A[\![x]\!]}.$$

(Note that $\hat{a}$ is an indeterminate, whereas x is a scheme variable.)

In this general form, the 'local' universal quantifier still exceeds our formalism. This problem could be overcome by using the meta-inference

$$\text{From } \frac{\hat{b} \prec \hat{a} \to A[\![\hat{b}]\!]}{A[\![\hat{a}]\!]} \quad \text{infer} \quad \frac{}{A[\![x]\!]}.$$

However, this would mean to introduce meta-inferences as formal objects into our calculus. The resulting addition of a further level together with the corresponding 'meta-meta-inferences' would have severe impacts on the interaction between inferences, scheme variables, and indeterminates (see Section 1.5). Thence we refrain from this extension to our system. But we can realize special instances of Noetherian induction, namely 'stepwise Noetherian induction' (cf. [26]): Let for any $x \in S$ the (possibly empty) set $\{pred_1(x), \ldots, pred_{n(x)}(x)\}$ denote all immediate predecessors of x with respect to $\prec$. Then we have the following inference-scheme:

[IX] *Stepwise induction*:

$$[*] \qquad \frac{\begin{array}{l} A[\![pred_1(\hat{a})]\!], \ldots, A[\![pred_{n_{\hat{a}}}(\hat{a})]\!] \to A[\![\hat{a}]\!] \\ \text{(one such clause for every combination of } pred\text{-operations)} \end{array}}{\to A[\![x]\!]}.$$

In the remainder of this part we give a number of instances of the above inference-scheme, where the operations $pred_i$ as well as the predicate A are concretely specified. Note that the validity of all these rules depends on the semantics of the programming language under consideration. So they are not part of our calculus but rather examples of 'extra-logical axioms' that can be accommodated in connection with our calculus.

Note. The following induction rules demonstrate that two features of our calculus are indeed mandatory: We need at least two levels of entailment, and we need

indeterminates in addition to scheme variables; otherwise the following induction rules could not be represented within the calculus.

4. Computational induction

The validity of the principle of computational induction stems from the fact that the meaning of a recursive function is the least fixed point of the equation

$$f = F[f]$$

where $F[\cdot]$ is the body of the function (or procedure), and that this fixed point is the limit of the sequence

$$\bot, F[\bot], F^2[\bot], \ldots, F^k[\bot], \ldots,$$

where $\bot$ is the totally undefined function/relation. In those cases, where $F[\cdot]$ is not continuous but merely monotone, the sequence has to be extended into the transfinite ordinals. In this case, the Noetherian induction uses the ordinal numbers as underlying well-founded set. (The following considerations are based on [13, 22, 30, 31, 35].) In the sequel we merely list the appropriate rules of inference; their soundness is discussed in Appendix A.

We presuppose that we are given—as a representative for systems of n functions—two function declarations (analogously for procedures) of the form

function $f(x)$; $F[f, g](x)$,

function $g(y)$; $G[f, g](y)$

and that we need to establish an assertion of the kind

$$R[f, g] \rightarrowtail S[f, g]$$

for 'embeddings' $R[\cdot, \cdot]$ and $S[\cdot, \cdot]$, and with $\rightarrowtail$ standing for one of the predicates $\equiv, \supseteq, \sqsubseteq$.

Scott induction

For the two functions f and g as defined above and with $\rightarrowtail$ standing for $\equiv$ or $\sqsubseteq$, the following rule of inference is valid (where $\hat{h}_1, \hat{h}_2 \in \hat{X}$ are indeterminates):

$$\frac{\begin{array}{l} \mathit{Continuous}[R[\cdot, \cdot]], \mathit{Continuous}[S[\cdot, \cdot]] \\ R[\bot, \bot] \rightarrowtail S[\bot, \bot] \\ R[\hat{h}_1, \hat{h}_2] \rightarrowtail S[\hat{h}_1, \hat{h}_2] \\ \quad \rightarrow R[F[\hat{h}_1, \hat{h}_2], G[\hat{h}_1, \hat{h}_2]] \rightarrowtail S[F[\hat{h}_1, \hat{h}_2], G[\hat{h}_1, \hat{h}_2]] \end{array}}{R[f, g] \rightarrowtail S[f, g]}.$$

Note that only the embeddings $R[\cdot, \cdot]$ and $S[\cdot, \cdot]$ need to be continuous (in order to establish the validity of $R[F^\alpha[\bot, \bot], G^\alpha[\bot, \bot]] \rightarrowtail S[F^\alpha[\bot, \bot], G^\alpha[\bot, \bot]]$ also for limit ordinals α). For the function bodies F and G monotonicity suffices.

Scott induction for $\supseteq$ requires in addition the premise

$$\mathit{Continuous}[\![F[\cdot,\cdot]]\!], \qquad \mathit{Continuous}[\![G[\cdot,\cdot]]\!].$$

Note. In a technical realization one has to avoid our "for the functions ... as defined above", in order to make the rule self-contained. So one needs the additional premises

$$F = \mathit{Declaration}[\![f]\!], \qquad G = \mathit{Declaration}[\![g]\!],$$

with a syntactic function $\mathit{Declaration}[\![\cdot]\!]$ that yields the right-hand side of the declaration of an identifier.

Recursion induction [30]

Let f and g be defined as above. Let moreover

$$\textbf{function } h(x);\ H[h](x)$$

be given. The following rule of inference is valid (where $\hat{x} \in \hat{X}$ is an indeterminate):

$$\frac{\begin{array}{l}\mathit{Defined}[\![h(\hat{x})]\!] \\ f(\hat{x}) \equiv H[f](\hat{x}) \\ g(\hat{x}) \equiv H[g](\hat{x})\end{array}}{f(x) \equiv g(x)}.$$

Note that one can often use f or g itself in the place of h. (The problem with this rule is, of course, the definedness proof.)

'Transformational' induction

This is a special case of Scott induction, suggested in [5] for its technical simplicity. We consider simplified functions

$$\textbf{function } f(x);\ F[f](x),$$

$$\textbf{function } g(x);\ G[g](x)$$

and a simpler assertion that is to be established, namely (for $\rightarrowtail$ being $\equiv$ or $\sqsubseteq$)

$$f \rightarrowtail S[g].$$

Then Scott induction simplifies to (with an indeterminate $\hat{h} \in \hat{X}$)

$$\frac{\begin{array}{l}\mathit{Continuous}[\![S[\cdot]]\!] \\ \bot \rightarrowtail S[\bot] \\ F[S[\hat{h}]] \rightarrowtail S[G[\hat{h}]]\end{array}}{f \rightarrowtail S[g]}.$$

Fixpoint induction [31]

For the relation $\sqsubseteq$ there is a special case of Scott induction: Consider the definition

function $f(x)$; $F[f](x)$

and a functional expression E. Then we have the valid assertion

$$F[E] \sqsubseteq E \rightarrow f \sqsubseteq E$$

and thus the rule

$$\frac{F[E] \sqsubseteq E}{f \sqsubseteq E}.$$

(The proof simply follows from the fact that $\perp \sqsubseteq E$, and thus by monotonicity $F[\perp] \sqsubseteq F[E] \sqsubseteq E$; hence $F^{\alpha}[\perp] \sqsubseteq E$ for all ordinals α.)

5. Structural induction

In the sequel we consider an instance of stepwise induction that is based on the 'generation principle' of algebraic data types. This leads to the meta-rule of term induction. (The following considerations are based on the theory of algebraic types—in the form described, e.g., in [37] or [2].)

Term induction

The principle of term induction relies on the *generation principle* for algebraic types (cf., e.g., [2, 37]): A sort **elem** that is defined by an algebraic type comprises exactly those objects that are generable with the operations of the type. This leads to the idea of a *constructor set* (cf. [18]): A set of operations (with range **elem**) is called a constructor set, if all objects of **elem** are generable with these operations only. Clearly, the set of all operations with range **elem** constitutes a constructor set (at least for types that introduce only one new sort). But for the term induction we are, of course, interested in 'minimal' constructor sets. (Unfortunately, the property of being a minimal constructor set is in general undecidable.)

For example, in NAT the constant *zero* and the operation *succ* form a minimal constructor set. All other operations, such as *pred*, *add*, *mult*, etc., can be defined in terms of these two constructor operations. Similarly, in the type BINTREE the operations *emptytree* and *cons* are constructors, while *left* and *right* can be left out.

We rely on the validity of a particular kind of lemmas for the given type T:

Decomposition lemmas. A *Decomposition Lemma* for a sort **elem** is of the form (where the **with**-notation indicates a restricted domain for the quantification, and

where the operations $\{c_1, \ldots, c_m\}$ form a constructor set)

$$\begin{array}{l}\forall x \in \textbf{elem}: \\ \exists x_1, \ldots, x_{n_1} \textbf{ with } Q_1[x_1, \ldots, x_{n_1}]: \; x \equiv c_1(x_1, \ldots, x_{n_1}) \;\vee \\ \qquad\qquad \vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \\ \exists x_1, \ldots, x_{n_m} \textbf{ with } Q_m[x_1, \ldots, x_{n_m}]: \; x \equiv c_m(x_1, \ldots, x_{n_m})\end{array}$$

(where in each existential quantification zero or more x_i may again be of sort **elem**).

We derive from the validity of the above lemma the validity of the following induction rule. (Note that this actually is an instance of the 'proof by case distinction' that was mentioned at the end of Section 3.)

Decomposition induction. Given the Decomposition Lemma above, we obtain the valid induction rule (where $\hat{x}_{ij}^*$ stand for those x_{ij} that are again of sort **elem**)

$$\frac{\begin{array}{l}Q_1[\hat{x}_{11}, \ldots, \hat{x}_{1n_1}] \equiv \mathit{true},\, A[\![\hat{x}_{11}^*]\!], \ldots, A[\![\hat{x}_{1j}^*]\!] \to A[\![c_1(\hat{x}_{11}, \ldots, \hat{x}_{1n_1})]\!] \\ \qquad\qquad \vdots \\ Q_m[\hat{x}_{m1}, \ldots, \hat{x}_{mn_m}] \equiv \mathit{true},\, A[\![\hat{x}_{m1}^*]\!], \ldots, A[\![\hat{x}_{mj}^*]\!] \to A[\![c_m(\hat{x}_1, \ldots, \hat{x}_{mn_m})]\!]\end{array}}{A[\![x]\!]}.$$

So we see that every algebraic type that is entered into the system should come equipped with a collection of suitable (and verified) Decomposition Lemmas, since these provide the essence of important induction rules.

Examples for decomposition lemmas. Let us consider again the two examples NAT and BINTREE. Here we obtain the rules

$$\frac{\begin{array}{l}A[\![\mathit{zero}]\!], \\ A[\![\hat{x}]\!] \to A[\![\mathrm{succ}(\hat{x})]\!]\end{array}}{A[\![x]\!]}$$

and similarly

$$\frac{\begin{array}{l}A[\![\mathit{emptytree}]\!], \\ A[\![\hat{u}]\!],\, A[\![\hat{v}]\!] \to A[\![\mathit{cons}(\hat{u}, \hat{x}, \hat{v})]\!]\end{array}}{A[\![t]\!]}.$$

To see both the variety of possible Decomposition Lemmas and the role of the additional predicates Q for domain restrictions, consider the following Decomposition Lemma for SET (where "$r < s$" is the predicate "$\forall x \in r, y \in s: x < y$"):

$$\begin{array}{l}\forall s \in \textbf{set}: \\ s \equiv \emptyset \;\vee \\ \exists x \in \textbf{elem}: s \equiv \{x\} \;\vee \\ \exists u, v \in \textbf{set with } u \neq \emptyset \wedge v \neq \emptyset \wedge u < v: s \equiv u \cup v.\end{array}$$

This lemma essentially says we can partition any non-degenerate set s into a set of small elements and a set of large elements. Thus we obtain the rule (which is useful, e.g., in 'Quicksort')

$$\frac{\begin{array}{l} A[\![\emptyset]\!], \\ A[\![\{\hat{x}\}]\!] \\ \hat{u} \neq \emptyset, \hat{v} \neq \emptyset, \hat{u} < \hat{v}, A[\![\hat{u}]\!], A[\![\hat{v}]\!] \to A[\![\hat{u} \cup \hat{v}]\!] \end{array}}{A[\![s]\!]}.$$

(For reasons of readability we have omitted the '$\equiv$ *true*' for boolean-valued terms.)

There are many variations of this paradigm. Well known are partitionings into 'almost equally large' subsets, i.e.

$$card(u) \div card(v) \leqslant 1$$

(where $card(u)$ gives the cardinality of u, and $i \div j$ is the 'symmetric difference' $\max(i,j) - \min(i,j)$), or partitionings into a singleton set and the remainder set, i.e.

$$card(u) = 1.$$

This shows that the use of Decomposition Lemmas is more powerful than merely working on the basis of constructor sets, because we can utilize more powerful assumptions during the proofs.

Exercursus: How to verify decomposition lemmas

In the previous sections we have seen how to derive induction rules from existing Decomposition Lemmas. So there remains the problem of obtaining suitable Decomposition Lemmas.

We must realize, however, that our calculus is designed as a formal basis for program transformation and not as a tool for theorem proving (in the framework of algebraic types). So we presuppose in general that the laws of a type are entered into the transformation system as 'extra-logical axioms' (usually in the form of clauses), the validity of which has been shown elsewhere. Nevertheless, we can perform at least a limited class of such proofs within our system. (An example is shown in Appendix B.)

6. Conclusion

We have presented a calculus that formalizes many of the activities that take place—often justified only pragmatically—in many transformation systems. The calculus is used as a kind of 'requirements specification' for the system CIP-S that is developed at the Technical University Munich under the guidance of F.L. Bauer; cf. [1]. A running prototype already realizes some of the principles outlined here; cf. [4]. Our experience has shown that without such a calculus it would have been virtually impossible to produce a 'correct' transformation system (in the sense of Section 0). The design and specification of such a system involves an abundance

of details—ranging from the concrete representation of 'terms', 'clauses', 'inferences', etc. to the forms of the rule catalogues and auxiliary service routines—such that the issues of soundness and adequacy (that are addressed in our calculus) would become almost intractable due to the overwhelming mass of programming technicalities.

A question that arises naturally in connection with formal calculi is that of completeness. Obviously, our calculus cannot be complete, since it is a true impoverishment of the usual Gentzen system. However, this question is not so relevant here, since our calculus is built on top of atomic predicates (such as *Defined*, *Determinate*, *Equivalent*, etc.) that are not axiomatizable in a complete way anyhow—at least for programming languages with repetition or recursion. Clearly, completeness relative to inherently incomplete atomic predicates is not very interesting. Analogously, the complexity of (relative) decision procedures is not a relevant issue in our context, since the calculus is built for an interactive system that is under strict user control.

The major motivation behind the calculus was to make the corresponding transformation system sound such that it guarantees correctness of all developments. To this end, we have thinned out the usual calculi from formal logic to a degree that exactly meets the needs of program transformation.

As a particular aspect, we have incorporated two prominent induction principles into the calculus, thus giving it enough power for deriving an abundance of new valuable rules from a small set of fundamental rules.

As has been seen, algebraic types play a major role in the whole process. And within algebraic types, particular emphasis lies on Decomposition Lemmas. Unfortunately, the verification of such lemmas in general is a non-trivial and time-consuming task. However, once this time has been invested upon definition of a type, it pays in each of its applications. So a predefined collection of fundamental types together with their appropriate Decomposition Lemmas is a most valuable tool in any programming environment.

Acknowledgment

The ideas presented in this paper have been conceived in the course of the design of the transformation system CIP-S that is under development at the Technical University Munich. My thanks go to all colleagues from this project, who have contributed to this work. I am particularly indebted to Berhard Möller, Helmut Partsch, Otto Paukner, Martin Wirsing, and Manfred Broy for a number of valuable discussions. A most valuable and constructive assessment of the paper has been given by Philippe de Groote.

Appendix A. Soundness proofs

We briefly comment here on the soundness of the calculus that has been presented in the paper.

Soundness of the logical rules

To see the soundness of the meta-inferences [I]-[V] from Section 1, let M be a structure in which the truth/falsity of atomic sentences is defined. By

$$M\text{-valid}\, A \quad (\text{short: } M \models A)$$

we express the fact that an atomic formula is *valid* in the structure M (that is, for all instantiations of the scheme variables and indeterminates in A by ground terms from $W[\text{PL}]$ the valuation of the resulting sentence in M yields *true*).

Then we have (see Section 1):

- A **clause**

$$A_1, \ldots, A_n \to B$$

is *M-valid* iff

$$(M \models A_1 \text{ and} \ldots \text{and } M \models A_n) \quad \text{implies} \quad (M \models B).$$

- An **inference**

$$\frac{\alpha_1, \ldots, \alpha_n}{\beta}$$

is *M-valid* iff

$$(M \models \alpha_1 \text{ and} \ldots \text{and } M \models \alpha_n) \quad \text{implies} \quad (M \models \beta).$$

Proposition. *The meta-inferences* [I]-[V] *from Section* 1 *are* M-valid *for arbitrary structures M.*

Proof. (We only consider the versions (ii) of the rules.)

ad[I]. Trivial.

ad[II]. If $(M \models \Psi)$ suffices to imply $(M \models \alpha)$, then of course $(M \models \Psi$ and $M \models \Phi)$ also imply $(M \models \alpha)$.

ad[III]. If $(M \models \Phi)$ implies $(M \models \alpha)$, then $(M \models \Phi$ and $M \models \Psi)$ imply $(M \models \alpha$ and $M \models \Psi)$, which in turn implies $(M \models \gamma)$.

ad[IVa]. By assumption we have that $(M \models \Phi$ and $M \models A)$ imply $(M \models \mathbb{B}$ implies $M \models C)$. Now let $(M \models \Phi)$ hold. Then we have that $(M \models A)$ implies that $(M \models \mathbb{B}$ implies $M \models C)$. But this means that $(M \models A$ and $M \models \mathbb{B})$ together imply $(M \models C)$.

ad[IVb]. By assumption we have that $(M \models \Phi)$ implies that $((M \models A$ and $M \models \mathbb{B})$ imply $M \models C)$. Now let $(M \models \Phi)$ as well as $(M \models A)$ hold. Then $(M \models \mathbb{B}$ implies $M \models C)$.

ad[V]. Since a formula A is *M-valid* iff its valuation yields true for all instantiations of variables and indeterminates by (ground) *terms*, $(M \models A)$ implies $(M \models A\theta)$. □

When written as inferences, the versions (i) of rules [I]-[III] can be derived from the remainder of the calculus.

Proof of [I(i)]: By [I(ii)] and [IVa] we obtain

$$\frac{A}{A} \quad \text{and thus} \quad \frac{}{A \to A}.$$

Proof of [II(i)]: By [I(ii)], [II(ii)], and [IVa] we obtain successively

$$\frac{\mathbb{B} \to C}{\mathbb{B} \to C}, \frac{\begin{matrix} A \\ \mathbb{B} \to C \end{matrix}}{\mathbb{B} \to C}, \quad \text{and thus} \quad \frac{\mathbb{B} \to C}{A, \mathbb{B} \to C}.$$

Proof of [III(i)]: By [I(ii)] and [IVb] we obtain

$$[*] \qquad \frac{A \to B}{A \to B} \quad \text{and thus} \quad \frac{\begin{matrix} A \\ A \to B \end{matrix}}{\to B}.$$

Analogously we obtain again by [I(ii)] and [IVb]

$$[**] \qquad \frac{B, \mathbb{B} \to C}{B, \mathbb{B} \to C} \quad \text{and thus} \quad \frac{\begin{matrix} B \\ B, \mathbb{B} \to C \end{matrix}}{\mathbb{B} \to C}.$$

By the Cut Rule [III(ii)] we obtain from [*] and [**]

$$\frac{\begin{matrix} A \\ A \to B \\ B, \mathbb{B} \to C \end{matrix}}{\mathbb{B} \to C}.$$

Finally we use [IVa] to obtain

$$\frac{\begin{matrix} A \to B \\ B, \mathbb{B} \to C \end{matrix}}{A, \mathbb{B} \to C}.$$

Proofs of derived meta-inferences

ad[II*]. By the Cut Rule [III(i)] the following inference is valid:

$$\frac{\begin{matrix} \mathbb{A}, C \to A \\ C \end{matrix}}{\mathbb{A} \to A}.$$

From this inference and the given inference

$$\frac{\begin{matrix} \Phi \\ \mathbb{A} \to A \end{matrix}}{\mathbb{B} \to B}$$

we can infer again by the Cut Rule [III(ii)]

$$\frac{\begin{array}{c}\Phi\\ \mathbb{A}, C \to A\end{array}}{\begin{array}{c}C\\ \hline \mathbb{B} \to B\end{array}}.$$

Now importation [IVa] yields the desired result.

ad[III*]. By [II*] we obtain from the given inference

$$\frac{\Phi}{\mathbb{A} \to A}$$

the new one

$$\frac{\Phi}{\mathbb{A}, \mathbb{B} \to A}.$$

Then the Cut Rule [III(ii)] yields the desired result.

Soundness of the language-dependent axioms

Let M be the underlying semantic model of the language PL, and let $\mathbb{M}: W[\text{PL}] \to M$ be the morphism 'from syntax to semantics' (cf. [16]). Then the semantic predicates are defined, e.g., as

$$\begin{array}{lll} \mathit{Equivalent}[t_1, t_2] & \Leftrightarrow & \mathbb{M}[\![t_1]\!] = \mathbb{M}[\![t_2]\!], \\ \mathit{Descendant}[t_1, t_2] & \Leftrightarrow & \mathbb{M}[\![t_1]\!] \supseteq \mathbb{M}[\![t_2]\!], \\ \mathit{Defined}[t] & \Leftrightarrow & \bot \notin \mathbb{M}[\![t]\!]. \end{array}$$

Consequently, the axioms such as *transitivity* and *reflexivity* (see Section 1) simply follow from the corresponding properties of the equality relation or of the subset relation.

The axiom of *monotonicity*, which semantically means, e.g., for the descendant relation

$$\mathbb{M}[\![t]\!] \supseteq \mathbb{M}[\![t']\!] \Rightarrow \mathbb{M}[\![cn[t]]\!] \supseteq \mathbb{M}[\![cn[t']]\!] \quad \text{for any context } cn[\cdot],$$

needs particular care during the language definition (see [1, 9]).

For the predicate *Continuous*[·] and for the induction rules, we have to be more specific about the semantics of the language—which we presuppose (for reasons of generality) to be nondeterministic. (The following treatment is strongly influenced by [6, 13, 22, 31].)

Let the semantics of PL have the following properties:

- The basic object sets are *flat domains Dom*, the order relation of which is denoted as $a \sqsubseteq b$, and the bottom element of which is denoted by $\bot$. In order to deal with relations below, we extend $\sqsubseteq$ to pairs from $Dom \times Dom$ by

$$\langle a, b\rangle \sqsubseteq \langle a', b'\rangle \Leftrightarrow_{\text{def}} a = a' \wedge b \sqsubseteq b'.$$

- The (nondeterministic) 'functions' of PL are associated to *relations* $R \sqsubseteq Dom \times Dom$, which obey the constraints
  (i) $\langle \perp, a \rangle \in R \Rightarrow a = \perp$ ('strict'),
  (ii) $\forall a \in Dom\ \exists b \in Dom : \langle a, b \rangle \in R$ ('left-total').
  (Note: b may be $\perp$.)
  We call a relation *finitary*, if [1]
  (iii) $card(xR) = \infty \Rightarrow \langle x, \perp \rangle \in R$.
  (Finitary relations are called 'program relations'; non-finitary relations constitute the so-called 'unbounded nondeterminism'.)
- On relations we have the *Egli-Milner ordering*

$$R \sqsubseteq S \Leftrightarrow \forall r \in R \exists s \in S: r \sqsubseteq s$$
$$\wedge \forall s \in S \exists r \in R: r \sqsubseteq s.$$

  With this ordering, our relations form again a cpo, with bottom element $\perp = \{\langle d, \perp \rangle \mid d \in Dom\}$.
- All constructs of PL are *monotonic* with respect to the orderings $\sqsubseteq$ and $\sqsupseteq$.
- The 'finite' language constructs (in the sense of [6]) are *continuous* (for directed sets of arbitrary cardinality—see [29]).

The last point, namely the issue of continuity, deserves further illustration: Finite constructs are, e.g., 'application of finitary relations', 'conditional', 'finite choice' etc., whereas infinite constructs are 'universal quantification', 'infinite choice', 'application of infinitary relations'. As an example, let us consider:

Example. *Continuity of the applications of finitary relations.* For a relation R and a set $X \subseteq Dom$ of values we have the definition

$$apply(R, X) = \{y \in Dom \mid \langle x, y \rangle \in R, x \in X\}.$$

For simplicity we only show continuity of *apply* in its first argument, with a single, fixed value x as second argument. We have to prove for any directed set

$$\Delta = \{R_1, R_2, \ldots, R_\alpha, \ldots\}$$

of finitary relations that the following property holds:

$$apply\left(\bigsqcup_{R \in \Delta} R, x\right) = \bigsqcup_{R \in \Delta} apply(R, x).$$

Proof. Let

$$R^* \overset{\text{def}}{=} \bigsqcup_{R \in \Delta} R, \qquad Y_R \overset{\text{def}}{=} apply(R, x), \qquad Y^* \overset{\text{def}}{=} \bigsqcup_{R \in \Delta} Y_R.$$

(i) By the monotonicity of *apply* we have $Y^* \sqsubseteq apply(R^*, x)$, because

$$R \sqsubseteq R^* \Rightarrow Y_R = apply(R, x) \sqsubseteq apply(R^*, x)$$

[1] As usual xR stands for $\{y \mid \langle x, y \rangle \in R\}$, and analogously Ry for $\{x \mid \langle x, y \rangle \in R\}$.

and hence

$$Y^* = \bigsqcup_{R \in \Delta} Y_R \sqsubseteq apply(R^*, x).$$

(ii) The converse direction $apply(R^*, x) \sqsubseteq Y^*$ is shown as follows:

Let $y \in apply(R^*, x)$. Since y lives in a flat domain, there must be some $R \in \Delta$ such that $y \in apply(R, x)$ and consequently also $y \in apply(R', x)$ for all R' with $R \sqsubseteq R'$. Hence

$$y \in \bigsqcup_{R \in \Delta} apply(R, x) = Y^*.$$

Thus we have the set-inclusion

$$apply(R^*, x) \subseteq Y^*,$$

which provides one-half of the Egli-Milner ordering, namely

$$\forall y \in apply(R^*, x)\ \exists y^* \in Y^*: y \sqsubseteq y^*.$$

The other half, namely

$$\forall y^* \in Y^*\ \forall y \in apply(R^*, x): y \sqsubseteq y^*,$$

is trivial, if $\perp \in apply(R^*, x)$. Otherwise, consider some $y^* \in Y^*$. Since all relations are finitary and y^* lives in a flat domain, there must be some Y_R such that $y^* \in Y_R$, and consequently $y^* \in Y_{R'}$ for all $R \sqsubseteq R'$. Hence, $y^* \in apply(R^*, x)$. □

As is known from literature, the composition of monotonic language constructs is again monotonic, and the composition of continuous constructs is again continuous. This enables us to conclude the monotonicity of arbitrary program terms from the monotonicity of all individual constructs of the language (which, of course, has to be proven for the concrete language at hand). Similarly, we can define a simple *syntactic* predicate *Continuous* that is sufficient to guarantee semantic continuity.

ad[*Scott induction*]: We do not prove here explicitly that Scott induction is indeed a sound principle for the two relations $\equiv$ and $\sqsubseteq$, since this can be found at many places in the literature. We do, however, consider the less common descendant relation $\sqsupseteq$.

The relation $\sqsupseteq$ is only guaranteed to be continuous over finitary relations. We have to show that

$$\left(\bigsqcup_{R \in \Delta} R\right) \sqsubseteq S \;=\; \bigsqcup_{R \in \Delta} (R \sqsubseteq S)$$

(where the *lub* on the right-hand side is taken in the two-element $cpo\, ff \sqsubseteq tt$).

As in the above proof, this immediately follows if

$$y \in \bigsqcup_{R \in \Delta} R \;\Leftrightarrow\; y \in R' \text{ for all } R'' \sqsubseteq R' \text{ for some } R'',$$

which is the case for finitary relations over flat domains.

Appendix B. The verification of Decomposition Lemmas

With every newly defined algebraic type we trivially have one constructor set—viz. the set of all operations with range **m** (where **m** is the newly introduced sort). So we have a first induction rule. From this one, we may then be able to deduce some other Decomposition Lemmas. Consider the example SET with the operations $\emptyset$, $\{x\}$, and $u \cup v$. So we have the initial rule

$$\frac{A[\![\emptyset]\!],\quad A[\![\{\hat{x}\}]\!],\quad A[\![\hat{u}]\!], A[\![\hat{v}]\!] \to A[\![\hat{u} \cup \hat{v}]\!]}{A[\![s]\!]}.$$

With this rule we can now prove the validity of a modified Decomposition Lemma, where s is to be split into subsets of approximately equal size:

$$\begin{aligned}\forall s \in \textbf{set}:\ & s \equiv \emptyset \ \vee \\ & \exists y \in \textbf{elem}:\ s \equiv \{y\} \ \vee \\ & \exists a, b \in \textbf{set}:\ s \equiv a \cup b \wedge card(a) \dot{-} card(b) \leqslant 1.\end{aligned}$$

Let us denote this lemma as $\forall s \in \textbf{set}$: DL$[\![s]\!]$. By applying the above induction rule to this predicate we obtain the new, derived rule:

$$\frac{\text{DL}[\![\emptyset]\!],\quad \text{DL}[\![\{\hat{x}\}]\!],\quad \text{DL}[\![\hat{u}]\!], \text{DL}[\![\hat{v}]\!] \to \text{DL}[\![\hat{u} \cup \hat{v}]\!]}{\text{DL}[\![s]\!]}.$$

The first two premises are trivially reduced to true. So let us consider the third one. It reads in full detail

$$\begin{aligned}&\hat{u} \equiv \emptyset \\ &\vee (\exists y \in \textbf{elem}:\ \hat{u} \equiv \{y\}) \\ &\vee (\exists a, b \in \textbf{set}:\ \hat{u} \equiv a \cup b \wedge card(a) \dot{-} card(b) \leqslant 1),\end{aligned}$$

$$\begin{aligned}&\hat{v} \equiv \emptyset \\ &\vee (\exists y \in \textbf{elem}:\ \hat{v} \equiv \{y\}) \\ &\vee (\exists a, b \in \textbf{set}:\ \hat{v} \equiv a \cup b \wedge card(a) \dot{-} card(b) \leqslant 1)\end{aligned}$$

$$\to$$

$$\begin{aligned}&\hat{u} \cup \hat{v} \equiv \emptyset \\ &\vee (\exists y \in \textbf{elem}:\ \hat{u} \cup \hat{v} \equiv \{y\}) \\ &\vee (\exists a, b \in \textbf{set}:\ \hat{u} \cup \hat{v} \equiv a \cup b \wedge card(a) \dot{-} card(b) \leqslant 1).\end{aligned}$$

From the combinatorial manifold of cases to be verified, we consider only the most

interesting one (where renaming of bound variables is used to resolve name clases)

$$(\hat{u} \equiv \hat{a} \cup \hat{b} \wedge card(\hat{a}) \dot{-} card(\hat{b}) \leqslant 1),$$
$$(\hat{v} \equiv \hat{a}' \cup \hat{b}' \wedge card(\hat{a}') \dot{-} card(\hat{b}') \leqslant 1)$$
$$\rightarrow \exists \hat{a}'', \hat{b}'' \in \textbf{set}: \hat{u} \cup \hat{v} \equiv \hat{a}'' \cup \hat{b}'' \wedge card(\hat{a}'') \dot{-} card(\hat{b}'') \leqslant 1.$$

Here we merely have to choose $\hat{a}'' =_{\text{def}} min(\hat{a}, \hat{b}) \cup max(\hat{a}', \hat{b}')$ and $\hat{b}'' =_{\text{def}} max(\hat{a}, \hat{b}) \cup min(\hat{a}', \hat{b}')$—where *min* and *max* are taken with respect to cardinality—in order to demonstrate the existence constructively.

However, the same idea does not apply to the verification of, say, the simplified Decomposition Lemma

$$\forall n \in \textbf{nat}: n \equiv zero \vee \exists x \in \textbf{nat}: n \equiv succ(x)$$

from the initial rule (belonging to the full constructor set $\{zero, succ, pred\}$)

$$\frac{A[\![zero]\!],\ A[\![\hat{x}]\!] \rightarrow A[\![succ(\hat{x})]\!],\ \hat{x} \neq zero,\ A[\![\hat{x}]\!] \rightarrow A[\![pred(\hat{x})]\!]}{A[\![x]\!]}.$$

For here the verification of the third premise leads to the need to prove

$$\forall y \in \textbf{nat}: y \equiv zero \vee \exists z \in \textbf{nat}: y \equiv succ(z)$$

which is exactly the conclusion we are aiming at.

So we have to retreat to a very special proof technique that was employed in [18] in order to show the 'sufficient completeness' of algebraic types. Here we split the full constructor set into two subsets, the 'minimal' constructor set

$$\text{MC} = \{c_1, \ldots, c_m\}$$

and the 'extended constructor set'

$$\text{EC} \equiv \{e_1, \ldots, e_n\}.$$

Then we have to prove for every pair $e_i \in \text{EC}$ and $c_j \in \text{MC}$

$$e_i(\ldots, c_j(x_1, \ldots, x_n), \ldots) \equiv rhs$$

where the right-hand side *rhs* must be a term, in which e_i either does not occur at all or is applied only to variables.

For example, in NAT we have

$$pred(zero) \equiv undefined, \qquad pred(succ(n)) \equiv n,$$

which obviously meets the above requirements. Similarly, in the type QUEUE we typically have

$$rest(emptyqueue) \equiv emptyqueue, \qquad rest(append(q, x)) \equiv append(rest(q), x).$$

With this technique, we are able to (successively) take out functions from the full constructor set.

Remark. Since 'sufficient completeness' is a very important and fundamental property of algebraic types, most type specifications are already axiomatized in the way required above.

So we see that our calculus cannot find minimal constructor sets (and the associated induction rules) directly, but it allows us to derive refined Decomposition Lemmas from given constructor sets.

References

[1] F.L. Bauer et al., *The Munich Project CIP. Vol. I: The Wide Spectrum Language CIP-L*, Lecture Notes in Computer Science **183** (Springer, Berlin, 1985); *Vol. II: The Program Transformation System CIP-S*, to appear.

[2] F.L. Bauer and H. Wössner, *Algorithmic Language and Program Development* (Springer, Berlin, 1982).

[3] G. Birkhoff, *Lattice Theory* (American Mathematical Society, Providence, RI, 1973).

[4] B. Brass, F. Erhard, A. Horsch, H.-O. Riethmayer and R. Steinbrüggen: CIP-S: An instrument for program transformation and rule generation, Technische Universität München, Institut für Informatik, TUM-I8211, 1982.

[5] M. Broy, Transformation parallel ablaufender Prozesse, Dissertation, Technische Universität München, 1980.

[6] M. Broy, R. Gnatz and M. Wirsing, Semantics of nondeterministic and noncontinuous constructs, in: F.L. Bauer and M. Broy, Eds., *Program Construction*, Lecture Notes in Computer Science **69** (Springer, Berlin, 1979) 553–592.

[7] M. Broy, H. Partsch, P. Pepper and M. Wirsing, Semantic relations in programming languages, in: S.H. Lavington, Ed., *Information Processing 80* (North-Holland, Amsterdam, 1980) 101–106.

[8] M. Broy, P. Pepper and M. Wirsing, On relations between programs, in: B. Robinet, Ed., *Proc. 4th International Symposium on Programming*, Paris, Lectures Notes in Computer Science **83** (Springer, Berlin, 1980) 59–78.

[9] M. Broy, P. Pepper and M. Wirsing, On design principles for programming languages: An algebraic approach, in: J.W. de Bakker and J.C. van Vliet, Eds., *Proc. International Symposium on Algorithmic Languages*, Amsterdam, Netherlands (North-Holland, Amsterdam, 1981) 203–219.

[10] M. Broy, P. Pepper and M. Wirsing, On the algebraic definition of programming languages, *ACM TOPLAS* **9** (1987) 54–99.

[11] R.M. Burstall and J. Darlington, Some transformations for developing recursive programs, *J. ACM* **24**(1) (1977) 44–67.

[12] D.C. Cooper, The equivalence of certain computations, *Comput. J.* **9** (1966) 45–52.

[13] J.W. de Bakker and W.P. de Roever, A calculus for recursive program schemes, *Proc. International Conference on Automata, Languages and Programming* (1973) 167–196.

[14] H.B. Enderton, *A Mathematical Introduction to Logic* (Academic Press, New York, 1972).

[15] G. Gentzen, Untersuchungen über das logische Schließen, *Math. Z.* **39** (1934) 176–210 (Teil I), 405–431 (Teil II).

[16] J.A. Goguen, J.W. Thatcher and E.G. Wagner, Initial algebra semantics and continuous algebras, *J. ACM* **24** (1977) 68–95.

[17] D. Gries, *The Science of Programming* (Springer, Berlin, 1981).

[18] J.V. Guttag, The specification and application to programming of abstract data types, Ph.D. Thesis, University of Toronto, Department of Computer Science, Report CSRG-59, 1975.

[19] A. Heyting, *Intuitionism: An Introduction* (North-Holland, Amsterdam, 1966).

[20] D. Hilbert and W. Ackermann, *Grundzüge der theoretischen Logik* (Springer, Berlin, 1967).

[21] D. Hilbert and P. Bernays, *Grundlagen der Mathematik I* (Springer, Berlin, 1968).

[22] P. Hitchock and D. Park, Induction rules and termination proofs, *Proc. International Conference on Automata, Languages and Programming* (1973) 225-251.
[23] S.C. Kleene, *Introduction to Metamathematics* (North-Holland, Amsterdam, 1952).
[24] R. Kowalski, Algorithm = Logic + Control, *Comm. ACM* **22**(7) (1979) 424-436.
[25] P. Lorenzen, *Einführung in die operative Logik und Mathematik* (Springer, Berlin, 1955).
[26] Z. Manna, *Mathematical Theory of Computation* (McGraw-Hill, New York, 1974).
[27] Z. Manna and R. Waldinger, A deductive approach to program synthesis, *ACM TOPLAS* **2**(1) (1980) 90-121.
[28] Z. Manna and R. Waldinger, Deductive synthesis of the unification algorithm, *Sci. Comput. Programming* **1** (1981) 5-48.
[29] G. Markowsky, Chain-complete posets and directed sets with applications, *Algebra Univ.* **6** (1976) 53-68.
[30] J. McCarthy, A basis for a mathematical theory of computation, in: P. Braffort and D. Hirschberg, Eds., *Computer Programming and Formal Systems* (North-Holland, Amsterdam, 1963).
[31] D.M.R. Park, Fixpoint induction and proofs of program properties, *Machine Intelligence 5* (Edinburgh University Press, Edinburgh, 1969) 59-78.
[32] P. Pepper, Ed., *Program Transformation and Programming Environments*, NATO ASI-Series in Computer Science **F8** (Springer, Berlin, 1984).
[33] P. Pepper and H. Partsch, Algebraic types as a framework for program transformation, Technische Universität München, Institut für Informatik, TUM-I8408, 1984; Revised version: Program transformations expressed by algebraic type manipulations, *Technique et Science Informatiques* **5**(3) (1986) 197-212.
[34] J.A. Robinson, a machine-oriented logic based on the resolution principle, *J.ACM* **12**(1) (1965) 23-41.
[35] D. Scott, Outline of a mathematical theory of computation, *Proc. 4th Annual Princeton Conference on Information Sciences and Systems* (1970) 169-176.
[36] J.R. Shoenfield, *Mathematical Logic* (Addison-Wesley, Reading, MA, 1967).
[37] M. Wirsing, P. Pepper, H. Partsch, W. Dosch and M. Broy, On hierarchies of abstract data types, *Acta Informat.* **20** (1983) 1-33.