ON RELATIONS BETWEEN PROGRAMS*)

M. Broy, P. Pepper, M. Wirsing [(1)]

Abstract

Equivalence relations between programs are strongly connected to the formal definition of the semantics of programming languages. In addition they provide a basis for the formal justification of the development of programs by transformations. Besides equivalences there are various other relations on programs and computational structures, which help to get a better understanding of both programming languages and the programming activity. In particular, the study of relations between nondeterministic programs allows to compare different concepts of nondeterminism.

---

*) This research was partially sponsored by the Sonderforschungsbereich 49 - Programmiertechnik - Munich .

(1) Technische Universität München, Institut für Informatik, Postfach 20 24 20, D-8000 München 2, Germany

1. Introduction

The equivalence of programs is an important question in computer science. Equivalence relations between programs are extensively studied in the theory of program schemata (cf. for instance /Luckham et al. 70/). But surprisingly, practical computer science so far has not paid much attention to this fundamental question; programmers still understand their programs - and consequently the equivalences between them - intuitively. They do all modifications, optimizations etc. only relying on this intuitive understanding. If programs are, however, developed according to formal rules - generally called "transformations" - one also has to formalize the notion of the equivalence of programs in order to give the approach a sound basis.

Of course, the notion of equivalence of programs is strongly connected to the semantics of the programming language in question. Any formal definition of the semantics immediately induces an equivalence relation: Two programs may be considered equivalent if their images under the semantical function are equal. On the other hand, establishing an equivalence relation between programs can be used as a formal method for defining the semantics of (parts of) a programming language.

Apart from these more fundamental applications, the study of various relations between programs leads to a considerably better understanding of both programming languages and the programming activity. This is in particular so for nondeterministic and for parallel programs. For instance, the examination of equivalence classes of nondeterministic programs can clarify the different concepts of nondeterminacy that are found in computer science today. Finally, the consideration of relations between data structures is useful to understand and to describe the process of finding implementations for abstract specifications.

2. Fundamental Requirements

In formal program development (cf. /CIP 80 b/) one starts from a given program P and tries to come up with a final program P' such that $P \sim P'$ holds for some interesting relation "$\sim$" (mostly an equivalence).

Relations for program development therefore have to meet the following requirements:

(I) Every relation between programs must be reflexive: $\forall P : P \sim P$

During the development one generally procedes through a series of intermediate versions $P = P_0, P_1, \ldots, P_n = P'$. In this case, it should be possible to conclude $P \sim P'$ from the fact that $P_i \sim P_{i+1}$ for all i. Therefore, we have to require:

(II) Every relation between programs must be transitive :

$$\forall\, P, Q, R : P \sim Q \wedge Q \sim R \Rightarrow P \sim R$$

Thus $\sim$ has to be a quasi-ordering. Usually we are not considering complete programs but only parts of programs (e.g. a single assignment, the body of a loop etc.). Therefore the "local" validity of a relation $\sim$ has to ensure also the "global" validity of that relation :

(III) Any complete program $P[Q]$ containing the program part Q must fullfil the substitution-property :

$$\forall\, Q, Q' : Q \sim Q' \Rightarrow P[Q] \sim P[Q']$$

Therefore, equivalence relations actually have to be congruence relations with respect to all language constructs, and

for partial orderings all language constructs have to be monotonic.

Remark

In connection with fixed points of functionals often the continuity of the language constructs with respect to a partial ordering is required:

A program (scheme) $P[x]$ is $\sim$-continuous iff for all $\sim$-chains $(E_i)_{i \in \mathbb{N}}$ with $E_i \sim E_{i+1}$ $\mathrm{lub}(P[E_i])$ exists if $\mathrm{lub}(E_i)$ exists, and $\mathrm{lub}(P[E_i]) = P[\mathrm{lub}(E_i)]$.

Continuity plays an essential role for instance in fixed point semantics or for induction proofs. (end of remark)

In the following sections we are going to discuss a number of relations which fulfil these three fundamental requirements of reflexivity, transitivity and monotonicity .

3. Semantical Interpretations

Formally, the semantics of a programming language is defined by a mapping V into a given mathematical structure (e.g. Scott's continuous lattices). V should be a homomorphism, i.e. V induces a congruence relation between programs and therefore satisfies our conditions (I), (II), (III).

For applicative languages a particular semantical function V may be defined for every (deterministic) expression E by

$$V(E) = \begin{cases} e & \text{if } e \text{ is the result of the evaluation of } E \\ \omega & \text{if the evaluation of } E \text{ does not lead to a defined value.} \end{cases}$$

For instance, if V maps expressions of mode <u>integer</u> to the structure of the integral numbers, then the congruence relation induced by V is the so-called "mathematical equivalence".

But of course we can choose other semantical functions, too. When V gives for every program its "traces", it induces an operational equivalence (cf. /Hoare 78/, /Broy 80/). We even may restrict our attention to just one special property of programs and use e.g. a predicate that indicates whether a given program terminates or not. The resulting equivalence relation partitions the set of all programs into the two classes of all terminating and all non-terminating ones. Another example can be found in the approach of /Blikle 78/ : His equivalence relation is based on the predicate "P is correct with respect to its assertions". Hence, the two programs

$$\{i = N\}\ \underline{do}\ \ i > 0 \rightarrow i := i - 1\ \ \underline{od}\ \{i = 0\}$$

and

$$\{i = N \wedge s = 0\}\ \underline{do}\ \ i > 0 \rightarrow s, i := s + 1, i - 1\ \ \underline{od}\ \{i = 0 \wedge s = \sum_{i=0}^{N} i\}$$

are equivalent under this relation since both are assertion-correct.

The "axiomatic semantics" characterizes the semantics of procedural programming languages by giving proof rules, thus also inducing congruence relations on statements. Using Dijkstra's approach we may define (S, S' statements):

$$S \sim_{wp} S' \Leftrightarrow_{def} wp(S, R) = wp(S', R) \text{ for all postconditions } R$$

A comprehensive study of the impacts of different semantical functions is done in the field that has become known under the catchword "algebraic semantics" (cf. e.g. /Courcelle, Nivat 78/). Programs are viewed here as "program schemata" since the meaning of their base-functions is left open. This gives the possibility not only to study one particular "interpretation" (i.e.one particular semantical function V but whole classes of interpretations (/Courcelle, Guessarian 78/). Every such class C then induces an equivalence relation between expressions ("terms of the magma"):

$$E_1 \sim E_2 \Leftrightarrow E_1^{\,I} = E_2^{\,I} \quad \text{for all interpretations } I \in C \,.$$

This study leads to a number of important results, for instance to the validation of certain proof principles (such as computational induction for all algebraic interpretations).

4. Applicative versus Procedural Style

In programs and programming languages we have to distinguish two fundamentally different styles:

- In the applicative style programs are expressions the evaluation of which yields values.

- In the procedural style programs are statements that change the "state" - i.e. the values of program variables - or even change the "control state" by means of goto's. The semantical function for a statement S thus is a mapping from "states" to "states".

To cope with these two cases in a uniform way we can make use of the close relationship between statements and expressions: Let S be a statement without goto's and let x be the only global variable (of mode m) occurring in S (of course, x may stand here for a whole collection of variables). Then the statement S can be rewritten into

$$x := \lceil \underline{\text{var}}\ \underline{m}\ y := x\ ;\ S^y_x\ ;\ y \rfloor$$

where S^y_x denotes the statement S with all occurrences of x replaced by y. By introducing the local variable y the block on the right-hand side of the assignment now is a proper expression (without side-effects). By means of a few simple rewrite rules this block even can be converted into a purely applicative expression (cf./Pepper 78/ ; in the case of iterative statements this of course means the introduction of an explicit fixpoint operator). This expression then semantically characterizes the statement S .

<u>Def. 1</u> : Let S be a statement (without goto's) and let x be its only global variable. Let E_S be the purely applicative expression derived from the block

$$\lceil \underline{var}\ \underline{m}\ y := x \ ;\ S_x^y \ ;\ y \rfloor$$

Then E_S is called the <u>associated expression</u> of S .

Of course, the execution of S and E_S may be operationally different.

Obviously we can procede analogously from an expression E to its associated statement by means of an assignment $x := E$ to a suitably chosen program variable x . In this way, any (equivalence) relation on expressions induces an (equivalence) relation on statements, and vice versa.

<u>Def. 2</u> : Let ρ be a relation on expressions. We then get an induced relation $\tilde{\rho}$ on statements by

$$S \ \tilde{\rho}\ S' \underset{def}{\Leftrightarrow} E_S \ \rho\ E_{S'}$$

where E_S and $E_{S'}$ are the expressions associated to S and S' resp.

Note, however, that E_S in general depends on the value of x, i.e. contains x as a free identifier. Hence, this expression has to be considered as a function $(\underline{m}\ x)\ \underline{m} : E_S$. We therefore have to use an (equivalence) relation on functions that also is induced by that on expressions according to

$$f \ \rho\ g \underset{def}{\Leftrightarrow} f(E) \ \rho\ g(E) \quad \text{for all expressions } E .$$

Now we can restrict ourselves to considering relations on expressions only. This way of proceeding is analogous to the formal definition of a programming language by "transformational semantics" - cf./Pepper 78/ - where most constructs of the language are reduced by "definitional transformation rules" to a small language kernel.

Such techniques apply in particular to languages comprising different styles of programming. As an example one may consider the Wide Spectrum Language which is developed in the course of the project CIP at the Technical University Munich.

5. Relations for Deterministic Programs

Now we assume some fixed semantical function V. Extending notions of /McCarthy 62/ we may define two different relations on (deterministic) expressions:

Def. 3: (strong equivalence)

Two expressions E and E' are called strongly equivalent, iff
$V(E) = V(E')$.

Hence two strongly equivalent expressions have the same "course-of-values" (cf. / CIP 80 b/).

Def. 4: (weak "equivalence")

Two expressions E and E' are called weakly "equivalent", iff
$V(E) = \omega \vee V(E') = \omega \vee V(E) = V(E')$

Example: In most programming languages the equation

$$\underline{\text{if}}\ B\ \underline{\text{then}}\ E_1\ \underline{\text{else}}\ E_2\ \underline{\text{fi}} = \underline{\text{if}}\ \neg B\ \underline{\text{then}}\ E_2\ \underline{\text{else}}\ E_1\ \underline{\text{fi}}$$

denotes a strong equivalence while

$$\underline{\text{if}}\ B\ \underline{\text{then}}\ E\ \underline{\text{else}}\ E\ \underline{\text{fi}} = E$$

only is a weak "equivalence" (the evaluation of B may not terminate).

Unfortunately, weak "equivalence" is not transitive and hence no equivalence relation. For instance, let us consider the three expressions E_1, E_2, E_3 such that $V(E_2) = \omega$, whereas E_1 and E_3 are defined but have different values. Then E_1 and E_2 as well as E_2 and E_3 are weakly "equivalent" but E_1 and E_3 are not. This means that in a sequence $p_o, \ldots, p_n$ of programs where any two successive programs p_i and p_{i+1} are weakly "equivalent", no correspondence can be guaranteed between p_o and P_n . Therefore, the notion of weak "equivalence" is in no way appropriate for the development of programs, since it violates the transitivity condition.

Much more appropriate is the following well-known notion of "less definedness":

Def. 5 : (definedness preservation)

$$E \sqsubseteq E' \Leftrightarrow V(E) = \omega \vee V(E) = V(E')$$

Example: Consider the following function

funct f ≡ (m x, n y) r :
    if B(x) then G(x) else H(x, y) fi

A call $f(E_1, E_2)$ is - according to call-by-value semantics - undefined if E_2 is undefined. Replacing this call by its body (the respective transformation rule has become known under the name "UNFOLD" , cf./ Burstall, Darlington 75/) leads to

if $B(E_1)$ then $G(E_1)$ else $H(E_1, E_2)$ fi

which is defined if $B(E_1)$ is true. Therefore the rule "UNFOLD" only guarantees definedness preservation (cf. /Kott 78/).

Although not being an equivalence relation, the definedness preservation is useful in program developments since it is reflexive, transitive and monotonic (for the usual language constructs). In particular when starting from a (w.r.t. a given specification) totally correct program p_0 , the definedness preservation guarantees that the final program p_n is totally correct, too. On the other hand, when starting from an undefined program $V(p_0) = \omega$ nothing can be said about the final program p_n .

If we consider the reverse relation of "ω-preservation", i.e. $E \sqsupseteq E'$, then again the partial correctness of p_0 implies that of p_n (and vice versa) . The transformation "FOLD" being the inverse of "UNFOLD" provides an obvious example. In contrast to the definedness preservation now the total correctness of the final programm p_n ensures that also the original program p_0 had been totally correct. This may be interesting in those cases where the termination proof for the resulting program - e.g. a loop - is simpler than that for the original one - e.g. a complex nested recursion.

Obviously, two expressions E and E' are strongly equivalent iff both $E \sqsubseteq E'$ and $E \sqsupseteq E'$ hold. In this way, the $\sqsubseteq$-relation also is used as a basis for the fixpoint-theory underlying the technique of denotational semantics.

By virtue of the definition in section 4 every equivalence relation on expressions induces an equivalence relation on statements and vice versa. Thus e.g. strong equivalence for statements is defined by the strong equivalence of their associated expressions.

For instance, using Dijstra's "predicate transformers" we have

$$wp(S,\ R) = wp(x := E_S,\ R) = \begin{cases} R_x^e & \text{if } e = V(E_S) \text{ is defined} \\ \text{false} & \text{otherwise} \end{cases}$$

Therefore strong equivalence and $\sim_{wp}$ (cf. section 3) coincide for statements.

Analogously the definedness preservation leads to (let S and S' be deterministic statements) :

$$S \sqsubseteq S' \Leftrightarrow wp(S,\ R) \Rightarrow wp(S',\ R) \quad \text{for all postconditions } R,$$

since only undefined situations (e.g. "abort") yield false for every postcondition R.

6. Relations for Nondeterministic Programs

If a program allows some freedom of choice during the evaluation it is called nondeterministic . Since the definitions of section 4 are valid for both deterministic and nondeterministic cases, we again can restrict our attention solely to expressions. Nondeterminism here means that there may be different results for the same expression. Therefore we now have to extend the semantical function $V(E)$ to a function B, called "breadth", that gives the set $B(E)$ of all possible values of evaluations of E (cf./CIP 78/).

An expression is called determinate if $|B(E)| = 1$, i.e. if all possible evaluations yield the same result or are all undefined . If there exists a non-terminating evaluation then $\omega \in B(E)$ holds. An expression is called totally defined iff $\omega \notin B(E)$, and totally undefined iff $B(E) = \{\omega\}$. If there exist infinitely many possible results, i.e. if $|B(E)| = \infty$, we speak of unbounded nondeterminism . Note that $B(E)$ is never empty .

Example: Consider the following ambiguous function where "$E_1 [] E_2$" denotes an arbitrary choice between the expressions E_1 and E_2, i.e.
$B(E_1 [] E_2) = B(E_1) \cup B(E_2)$

```
funct f ≡ (nat x) nat :  if x = 0  then  0
                         [] x = 1  then  f(0) [] 1
                         [] x > 1  then  f(x-1) + 1 [] f(x+1)  fi.
```

The expression f(0) is determinate while f(1) is not determinate but defined and f(2) is unboundedly nondeterministic.

Remarkably, there are two major areas in computer science using the notion of "non-determinism" with different (although related) meanings. In the theory of automata, in artificial intelligence and in particular in complexity theory one has to decide for a given nondeterminitic expression E and a given value x whether $x \in B(E)$, i.e. whether x may result from evaluating E (cf. /Floyd 67/). To answer this question, all possible evaluations of E may have to be explored. This notion will be called here <u>nondeterminstic exhaustion</u>.

The other understanding of nondeterminism can be found in the theory of parallel programs and in formal program development. Here one accepts any arbitrary value of B(E) as a result of evaluating the expression E (cf. /Dijkstra 76/). This notion will be called <u>nondeterministic choice</u> here.

<u>Example</u>: To elucidate this distinction we consider the so-called "knapsack problem" which is known to be np-complete: Given a sequence of integers, is there a subsequence that sums exactly to a specific value k ?

The following elementary program yields the sum of an arbitrary subsequence:

```
funct subsum ≡ (sequ int s) int :
      if empty(s) then  0
                  else  subsum(rest(s))  [] first(s) + subsum(rest(s)) fi
```

We can base the nondeterministic predicate on it :

```
funct knapsack ≡ (sequ int s, int k) bool :  k = subsum(s).
```

Note that this only partly solves our problem. If a call of the function knapsack yields false, then there still may be another subsequence which sums to k. The result true, however, provides a definite answer.

By UNFOLD and FOLD techniques the function can be developed into

```
funct knapsack ≡ (sequ int s, int k) bool :
     if empty(s)  then  k = 0
                  else  knapsack(rest(s), k) [] knapsack(rest(s), k-first(s)) fi
```

This program represents a nondeterministic choice and it is said to "nondeterministcally solve the given problem in linear time". This means that the answer whether true $\in B$(knapsack(s, k)) is given by the above program if in each recursive call a "lucky choice" is taken. Since such a benevolent oracle, however, does not exist in reality, one has to trace out the tree of all possible computations in order to get the corret answer in any case. Hence, one has to replace the above nondeterministic choice by an exhaustive computation. In our specific case this simply can be done by replacing the choice-operator "[]" by the logical disjunction-operator "v" The resulting deterministc program, however, is not linear recursive and needs exponential time to compute the result.

Note : Using the program

```
funct iknapsack ≡ (sequ int s, int k) bool :

     if  k = 0   then true
                 else iknapsack(rest(s), k) [] iknapsack(rest(s), k-first(s)) fi
```

we get a semi-decision procedure which even works with sequences of infinite length. (end of note)

These different notions of nondeterminism, of course, lead to different "useful" relations for program development. Following the approach of /CIP 78/ we concentrate on the nondeterministic choice and do not consider the nondeterministic exhaustion.

First we look at relations corresponding to the strong equivalence of the deterministic cases, i.e. at relations that do not treat the undefined element ω separately :

Def. 6 : (strong equivalence)

Two (nondeterministic) expressions E and E' are called strongly equivalent, iff B(E) = B(E').

Since the breadth-function gives a whole set of values it is quite natural to consider not only set equality but also set inclusion. This leads to the transitive relation (cf./McCarthy 62/) :

Def. 7 : (strong descendant)

An expression E' is called a strong descendant of E , iff $B(E') \subseteq B(E)$.

The strong equivalence is applicable both for nondeterministic exhaustion and for nondeterministic choice. The relation of strong descendants, however, is only useful for the latter case. It is very important for program developments since a restriction of the choice usually represents a major design decision.

Example: The following nondeterministic program searches for the position of a given element x in an ordered array a (under the assertion that x indeed is in a, and that initially m is the lower and n the upper bound of a) :

```
funct search ≡ (m x, array a, nat m, n) nat:
    if m = n  then m
              else nat r = some nat i : m ≤ i ≤ n ;
                   if  a[r] =  x then r
                   []  a[r] <  x then search(x, a, r+1, n)
                   []  a[r] <  x then search(x, a, m, r-1)  fi fi
```

Where the expression "Some nat i : m ≤ i ≤ n" has the breadth B ={m, ..., n} .

Hence, both "m" and "(m + n) ÷ 2" are strong descendant of it and may be substituted for it. (The choice "nat r ≡ m" leads to linear search, "nat r ≡ (m + n) ÷ 2" leads to binary search.)

(end of example)

If we try to carry the notion of weak "equivalence" over to nondeterministic constructs, we get

Def. 8 : (weak "equivalence")

Two expressions E and E' are called weakly "equivalent", iff
$\omega \in B(E) \vee \omega \in B(E') \vee B(E) = B(E')$

Again, this relation is not transitive and therefore not suited for the development of programs. The same were true if we would define a similar notion of a weak descendant. We get an equivalence relation, however, by requiring that only the defined values of the expressions coincide:

Def. 9 : (weak equivalence)

Two expressions E and E' are called weakly equivalent, iff
$B(E) \setminus \{\omega\} = B(E') \setminus \{\omega'\}$

Note that for totally defined/undefined expressions - and hence in particular for deterministic ones - this notion coincides with that of strong equivalence. Analogously we get a weak descendant by the definition

Def. 10 : (weak descendant)

An expression E' is called a weak descendant of an expression E, iff
$B(E') \setminus \{\omega\} \subseteq B(E)$

Example: Both E and E' are strong descendants of (E [] E') . A totally undefined expression E' is a weak descendant of any expression.

How do these notions correspond to the definedness preservation $E \sqsubseteq E'$ for deterministic expressions ? The idea there was that starting from a totally correct program one is guaranteed to arrive at a totally correct program, too. This means for nondeterministic programs that ω must not be in B(E') if it is not in B(E). In addition, one should require that in cases of unbounded nondeterminism the defined values persist.

Def. 11 : (definedness preservation)

$$E \sqsubseteq E' \underset{def}{\Leftrightarrow} (\omega \in B(E) \wedge B(E) \setminus \{\omega\} \subseteq B(E')) \vee (B(E) = B(E'))$$

This means that we require strong equivalence for totally defined expressions and otherwise content ourselves with a weak descendant. This relation coincides with the "Egli-Milner"-ordering, that is used to define the semantics of nondeterministic recursive functions (cf. /de Bakker 76/).

The analogous definition using a descendant instead of an equivalence relation in the totally defined case is not very meaningful, since definedness preservation implies that a number of defined values may be added in the place of ω , whereas descendant means that a number of values may be left out. The combination of these two notions would lead to a relation which were valid for nearly any two programs.

Again looking at Dijkstra's predicate transformers, we now have for a nondeterministic statement S the translation :

$$wp(S, R) = \begin{cases} \forall e \in B(E_S) : R_x^e & , \text{ if } \omega \notin B(E_S) \\ \text{false} & , \text{ otherwise} \end{cases}$$

where E_S is the expression associated to S .

In contrast to the deterministic case, the relation $\sim_{wp}$ does only coincide with the strong equivalence if the two statements are totally defined or totally undefined.

7. Relations on Data Structures

In the last sections we have studied relations on programs for some fixed semantical function. Transformations often hold for whole classes of semantical functions or equivalently they are valid over different data structures with similar properties.

To begin with, it seems more appropriate to speak of "computational structures" (cf. /CIP 80 b/) instead of "data structures", in order to stress the point that the basic operations are an integrated part of these structures. Such computational structures are considered equivalent, if they show the same behaviour. This behaviour can be described formally by means of (algebraic) "abstract data types" (cf. e.g./Liskov, Zilles 74/, /Guttag 75/, /ADJ 78/, /CIP 79/) which leads to the fundamental relation: "A computational structure D is of a type T". Informally, this means that there are fixed correspondences between the sorts of T and the carrier sets of D and between the function symbols of T and the operations of D and that the laws (axioms) of T hold in D . In order to have induction methods over types we require furthermore that every object of a computation structure is denoted by a "well-formed term" of T, i.e. by a term (without free identifiers) built up from the function symbolds of T respecting their functionalities (for a more detailed discussion cf. e.g. /ADJ 78/, /ĊIP 79/).

We are led in a straightforward manner to relations between computational structures :

Def. 12 : Two computational structures D and D' are strongly equivalent if for every type T the structure D is of type T if and only if also D' is of type T.

Since strongly equivalent structures are isomorphic, this notion is too restrictive for being applicable in program development. Consider for instance the abstract concept of (finite) sets. The strong equivalence does not allow to switch from a representation by "characteristic functions" to a representation by "linked lists", since these two structures have different properties. Therefore we should better work with the notion: "Two computational structures D and D' are equivalent with respect to a given type T if both D and D' are of that type T."

But when working with abstract types we immediately are led to the question of the "equivalence of types", thus being able to do a formal program development also on the very abstract level of specifications. Since the notion of a type is closely related to the notion of a theory in formal logic, we could adopt the respective definition, viz. that two theories are equivalent if they have the same language and the same theorems (cf./Shoenfield 67/). But this means that the two types have exactly the same computational structures as models, which makes the notion too restrictive for the needs of program development.

Instead of considering (the provability of) all kinds of formulas, we therefore restrict our attention to (various) equivalences between the terms of the given type.

Two terms which represent the same object in every computation structure of the respective type obviously have to be considered equivalent:

Def. 13 : Two terms s and t of a type T are strongly equivalent if the equation $s = t$ is valid in T .

But this notion is quite restrictive, too. For instance, in a data base system we are not interested whether two internal representations are exactly the same, but rather whether they behave alike for every possible inquiry. Such a data base is to be described as a new structure that is built up from given "primitive" structures. The corresponding hierarchy of types suggests to distinguish terms of two kinds: those that represent objects of the newly defined type, and those that represent objects of the already existing "primitive" types. The latter are called terms of primitive sort (their outermost operation symbol has as its range one of the primitive structures). These terms of primitive sort determine the behaviour that a computational structure (viewed as a "black box") exhibits to the outer world.

Def. 14 : Two (arbitrary) terms s and t are called visibly equivalent, if for all terms u[x] of primitive sort (containing only one free variable x) u[s] and u[t] are equivalent (in the respective primitive type).

Note that for sufficiently complete types (cf. /Guttag 75/) both u[s] and u[t] can be reduced to terms only consisting of operation symbols of the primitive type in question. In the above definition we have left open which kind of equivalence is assumed for u[s] and u[t]. Actually, any suitable relation between the primitive terms induces a respective relation between the nonprimitive terms.

Any equivalence relation between terms induces an equivalence relation between types. Let T and T' be two types having the same signature up to renaming (homologous types, cf./ CIP 80 b/). Then T and T' are strongly/visibly equivalent if any two nonprimitive terms s, t are strongly/visibly equivalent in T iff they are it in T'. Of course, this requires that the same equivalence relation in the primitive types is taken as a basis.

Let us now consider two equationally defined types T and T' based on the same (or at least strongly equivalent) primitive types. T and T' are strongly equivalent iff their initial (cf. /ADJ 78/) computation structures are isomorphic. T and T' are visibly equivalent iff their terminal (cf. /CIP 79/) computation structures are isomorphic.

For a nontrivial example for these notions see /Pepper 78/ (cf. also section 4). There a language kernel serves as a primitive type, while procedural constructs are defined by axiomatic transformation rules (playing the role of conditional equations).

8. Concluding Remarks

Methods of program development which use several versions of the "same" program presuppose a notion of program equivalence; even assertion methods induce such equivalences. A justification of such development processes requires a sound formalization of these relations, thus leading to a "calculus" of transformations. In particular, these approaches turn out to be considerably more flexible if not only equivalences are employed but also partial orderings like the descendant relation. Note that, if one is interested in the basic concepts of a specific programming methodology, one should carefully study the underlying relations characterizing the approach.

Apart from these practical aspects of program development relations between programs also give valuable theoretical insights into the structure of programming languages. So far, however, relations mainly have been considered on semantical domains (fixed point theory, denotational semantics). Trivially, in this way also relations between programs are induced by the semantical mappings. And it is not surprising that we can procede the other way round: Using conditional equations (called transformation rules) to establish equivalences between programs one can specify the semantics (cf. /Pepper 78/, /CIP 80 a/). These techniques allow to explain basic properties of a programming language without constructing complex semantical domains. Moreover, one can define a language in a modularized way, both from the syntactic and from the semantic point of view, leading to a "stepwise development of the semantics". In particular, one gets design criteria ensuring the coherence and independence of the concepts of the language. An illustrative example is given in /Broy 80/ where (in connection with parallel programs) the incompatibility of certain fairness conditions with the continuity of the language constructs with respect to the Egli-Milner ordering is shown.

Acknowledgement

The results presented in this paper were strongly influenced by discussions we had with our colleagues of the CIP research group at the Technical University Munich, notably with Prof. F.L. Bauer and Prof. K. Samelson.

References:

/ADJ 78/

J. A. Goguen, J. W. Thatcher, E. G. Wagner: An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. In: R. T. Yeh (ed.): Current Trends in Programming Methodology, Vol. 3, Data Structuring, Englewood Cliffs: Prentice Hall 1978

/de Bakker 76/

J. W. de Bakker: Semantics and Termination of Nondeterministic Recursive Programs. 3rd Int. Symp. on Automata, Languages and Programming, Edinburgh 1976

/Blikle 78/

A. Blikle: Specified Programming. In: Mathematical Studies in Information Processing, Proc. Int. Conf., Kyoto, Aug. 1978

/Broy 80/

M. Broy: Transformation parallel ablaufender Programme. Technische Universität München, Dissertation an der Fakultät für Mathematik, Februar 1980

/Burstall, Darlington 75/

R. M. Burstall, J. Darlington: Some Transformations for Developing Recursive Programs. Proc. of 1975 Int. Conf. on Reliable Software, Los Angeles 1975, 465-472. Also: J. ACM 24, 1, 44-67 (1977)

/CIP 78/

M. Broy, R. Gnatz, M. Wirsing: Semantics of Nondeterministic and Noncontinuous Constructs. In: F. L. Bauer, M. Broy (eds.): Program Construction. (Proc. Int. Summer School, Marktoberdorf 1978) Lecture Notes in Computer Science 69, 553-592, Berlin: Springer 1979

/CIP 79/

M. Broy, W. Dosch, H. Partsch, P. Pepper, M. Wirsing: Existential Quantifiers in Abstract Data Types. In: H. A. Maurer (ed.): Proc. of the Sixth Colloquium on Automata, Languages and Programming, Graz, Lecture Notes in Computer Science 71, 73-87, Berlin: Springer 1979

/CIP 80 a/

M. Broy, M. Wirsing: Programming Languages as Abstract Data Types. To appear in the Proc. of the 5th Colloquium on "Arbres en Algèbre et en Programmation" Lille 1980

/CIP 80 b/

F. L. Bauer, H. Wössner: Algorithmic Language and Program Development. (to appear)

/Courcelle, Guessarian 78/

B. Courcelle, I. Guessarian: On Some Classes of Interpretations. JCSS 17 : 3, 388-413 (1978)

/Courcelle, Nivat 78/

B. Courcelle, M. Nivat: The Algebraic Semantics of Recursive Program Schemes. In: Proc. Math. Foundations of Comp. Sc., Zakopane 1978

/Dijkstra 76/

E. W. Dijkstra: A Discipline of Programming. Englewood Cliffs: Prentice Hall, 1976

/Floyd 67/

R. W. Floyd: Nondeterministic Algorithms. J. ACM 14, 636-644, (1967)

/Guttag 75/

J. V. Guttag: The Specification and Application to Programming of Abstract Data Types. Ph. D. Thesis, Univ. of Toronto, Dept. of Comp. Sc., Rep. CSRG-59, 1975

/Hoare 78/

C. A. R. Hoare: Some Properties of Predicate Transformers. J. ACM 25, 3, 461-480, (1978)

/Kott 78/

L. Kott: About a Transformation System: A Theorectical Study. In: B. Robinet (ed.): Program Transformations, Proc. 3rd Int. Symp. on Programming, Paris : Dunod 1978

/Liskov, Zilles 74/

B. Liskov, S. Zilles: Programming with Abstract Data Types. Proc. ACM SIG-PLAN Conf. on Very High Level Languages, SIGPLAN Notices 9, 4, 50-59 (1974)

/Luckham et al. 70/

D. Luckham, D. Park, M. Paterson: On Formalized Computer Programs. J. CSS 4 (1970)

/McCarthy 62/

J. McCarthy: A Basis for a Mathematical Theory of Computation. In: P. Braffort, D. Hirschberg (eds.): Computer Programming and Formal Systems. Amsterdam: North-Holland 1963

/Pepper 78/

P. Pepper: A Study on Transfomational Semantics. In: F. L. Bauer, M. Broy (eds.): Program Construction. (Proc. Int. Summer School, Marktoberdorf 1978) Lecture Notes in Computer Science 69, 322-405, Berlin: Springer 1979. (Also: Dissertation, Techn. Univ. München 1979)

/Shoenfield 67/

J. R. Shoenfield: Mathematical Logic. Reading: Addison-Wesley 1967