3rd *International Conference on Applied Informatics*
Eger–Noszvaj, Hungary, August 24–28, 1997.

A Relational Model of Transformation in Programming*

Ákos Fóthi, Zoltán Horváth, Judit Nyéky Gaizler[1]

[1] Dept. of General Computer Science, Eötvös Loránd University, Budapest, Hungary
e-mail: fa@lngsc2.inf.elte.hu, hz@ludens.elte.hu, nyeky@ludens.elte.hu

Abstract

The notion of transformation is used in several senses in programming, such as program transformation, problem transformation, coordinate transformation or state space transformation. In this paper we define different kinds of transformations on a relational basis. We investigate the applicability and expressiveness of the transformations and show concrete example demonstrating their usefulness.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Program Verification - *Correctness proofs*; D.3.1 [Programming Languages]: Formal Definitions and Theory - *semantics*; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs - *assertions, invariants*

Key Words and Phrases: Programming methodology, transformation, relational semantics, program transformation

1 Introduction

It is a well known aspiration that aims to use those methods exclusively which guarantee the correctness of the developed program with respect to the posed problem. That is what made it essential to find abstract mathematical definition of program, problem, solution. We build a relational model of programming, where the most important fundamental concepts of programming - problem, state space, variable, program, abstract data type, object, etc. are treated in a uniform and consistent way. This approach makes possible to investigate the concept and correctness of several kind of problem and program transformations.

In our approach the two most important concepts are the *problem* and the *program*. Their abstract definitions are given in terms of *state space*. For the definition of *solution* (when a program is said to solve a problem) establishing the main connection between the problem and program we use the notion of *effect relation*.

The concept of the state space has already been used in several senses. For Mills the state space is a model of a von Neumann type computer. Others, e.g. Dijkstra, associate

* Supported by the Hungarian National Science Research Grant (OTKA), Grant Nr. T017800

this notion with the problem to be solved and, in this way, the elements of the state space are the possible states of the characteristics of the problem. So the program is "outside" of the state space operating on it

In this paper the notation of the state space is used in the second meaning [2].

First we concentrate on transformation of problems, in which case the transformation of the state space of the problem is usually necessary.

We may extend the state space by introducing new components, i.e., new variables and/or tighten the state space by omitting variables. The transformed problem is defined and solved over the new state space, finally the program is projected (respectively extended) for the original state space.

We have to substitute some of the components of the state space by new ones in more complex cases. This way we associate the points of the original state space to the points of the new state space.

Simple program transformations need an extension of the state space by introducing new auxiliary variables usually. Complex program transformations are performed compositionally based on the compositionality of the programming model. Since a compound problem is solved using the appropriate program constructs, we can easily find the corresponding construct of problems. We transform the component problems and their solutions. The overall transformed program is build up from the transformed components using the same construction methods as originally.

Complex problems are solved in two steps usually. Ffirst we transform the problem to a new state space, over which state space the problem is easy to formulate and easy to solve. The program, which is the solution of the transformed problem is transformed back to the original state space compositionally.

2 Preliminary definitions and notions

At first we shall refer to some fundamental notions of the mathematical model of programming [6].

Denote by A^* the set of all finite and by A^∞ the set of all infinite sequences of elements of A. Put $A^{**} ::= A^\infty \cup A^*$.

Let A_1, A_2, $\dots A_n$ be arbitrary sets.In this case the set $\mathop{\times}\limits_{i=1}^{n} A_i = \{\ (a_1, \dots, a_n) \mid a_i \in A_i, i \in [1..n]\ \}$ is called the *direct product* of the above sets.

Let $A = \mathop{\times}\limits_{i=1}^{n} A_i$, $m \leq n$, $\{\ i_j \mid j \in [1..m]\ \} \subset [1..n]$, and $B = \mathop{\times}\limits_{i=1}^{m} A_{i_j}$ be arbitrary direct products.

Any subset of any direct product is called a *relation.* Further we deal with relations which are subsets of the direct product of two sets (binary relations).

Let $R \subseteq A \times B$ be an arbitrary relation.

The *domain* of the relation R is defined by: $\mathcal{D}_R = \{a \in A | \exists b \in B : (a, b) \in R\}$

The *range* of the relation R is defined by: $\mathcal{R}_R = \{b \in B | \exists a \in A : (a, b) \in R\}$

The *value* of the relation R at $a \in A$ is: $R(a) = \{b \in B | (a, b) \in R\}$

The *value* of the relation R at a set $H \subseteq A$ is: $R(H) = \{b \in B | \exists a \in H : (a, b) \in R\}$.

A relation is said to be *deterministic*, or to be a *partial function*, if $\forall a \in A : |R(a)| \leq 1$.

A relation is said to be a *function* if $\forall a \in A : |R(a)| = 1$.

We shall define also *operations* between relations. Suppose $P \subseteq A \times B$ and $Q \subseteq B \times C$. The relation $R \subseteq A \times C$ is said to be the *composition* of the relations P and Q if $R = Q \circ P = \{(a, c) \in A \times C | \exists b \in B : (a, b) \in P \wedge (b, c) \in Q\}$.
Let $R \subseteq A \times B$. The relation $R^{(-1)}$ is said to be the *inverse* of the relation R if $R^{(-1)} = \{(b, a) \in B \times A | (a, b) \in R\}$.
Let $H \subseteq B$ be an arbitrary set. The set $R^{(-1)}(H) = \{a \in A | R(a) \cap H \neq \emptyset\}$ is said to be the *inverse image* of set H with respect to relation R.
Let us perceive that the concept of the inverse image is the same as the concept of the inverse image with respect to relation R. We shall call the set $R^{-1}(H) = \{a \in D_R | R(a) \subseteq H\}$ to be the *pre-image* of the set H with respect to relation R.
Let be $R \subseteq A \times \mathcal{L}$, where A is an arbitrary set and L is the set of the logical values. R is called a logical relation. The *truth set* of the relation R is $\lceil R \rceil ::= R^{-1}(\{ \mathit{true} \})$, while the *weak truth set* of the relation R is $\lfloor R \rfloor ::= R^{(-1)}(\{ \mathit{true} \})$.
Let $A_1, A_2, \ldots, A_n$ be arbitrary finite or numerable sets. The set $A = \mathop{\times}\limits_{i=1}^{n} A_i$ will be called a *state space*, while the sets A_i will be called type-value sets. This name refers to the fact that each component of the state space is the range of a type ("representing function").
We usually have to handle the components of the state space separately. To this end we use variables. The projections $v_i : A \to A_i$ of the state space $A = A_1 \times \ldots \times A_n$ are called *variables.*
The notion of the state space makes it possible to define the concept of a problem independently of any program. Any relation $F \subseteq A \times A$ is called a *problem.*
A program is defined in terms of sequences; the working of the program is characterized by a series of the elements of the state space. A relation $S \subseteq A \times A^{**}$ is called a *program* in the state space if

$$\forall a \in A : \forall \alpha \in S(a) : \alpha_1 = a, \quad \mathcal{D}_S = A, \quad \forall \alpha \in \mathcal{R}_S : \alpha = red(\alpha).$$

The second part of the definition means that the program during its normal functioning always gets into a new state. No change of state means an abnormal functioning of the program, in other words, in the case of normal functioning the program will always remain within the system.
A statement over the state space A is called *empty* and termed $SKIP$, if $\forall a \in A : SKIP(a) = \{(a)\}$. Let $A = A_1 \times ... \times A_n$, $F = (F_1, ..., F_n)$, where $F_i \subseteq A \times A_i$. The statement $s \subseteq A \times A^{**}$ is a *general assignment* defined by F, if $S ::= \{(a, red(a, b)) \mid a, b \in A \wedge a \in \bigcap_{i \in [1,n]} \mathcal{D}_{F_i} \wedge b \in F(a)\} \cup \{(a, (aaa....)) \mid a \in A \wedge a \notin \bigcap_{i \in [1,n]} \mathcal{D}_{F_i}\}$.
We introduce the notion of effect relation in order to express the result of the functioning. Marks: Let $\tau : A^* \to A$ be a function which, with each sequence associates its last element. $\tau(\alpha) ::= \alpha_{|\alpha|}$.
The *effect relation* of a program S is a relation $p(S) \subseteq A \times A$ defined as follows:

$$\mathcal{D}_{p(S)} = \{a \in A \mid S(a) \subseteq A^*\},$$
$$\forall a \in \mathcal{D}_{p(S)} : p(S)(a) = \{b \in A \mid \exists \alpha \in S(a) : \tau(\alpha) = b\}.$$

We use the notion of solution instead of the notion of program correctness because the problem is defined independently of the program. The program S is said to *solve* the

problem F iff $\mathcal{D}_F \subseteq \mathcal{D}_{p(S)}$, and $\forall a \in \mathcal{D}_F : p(S)(a) \subseteq F(a)$.

The concept of solution is similar to one used by Hoare for representation of types.

Based on Dijkstra's weakest precondition [1] the theorem of specification [6] gives an alternative way to solve a problem specified with pre- and postconditions.

Let R be a logical function over the state space A. $R : A \to \mathcal{L}$ and $S \subseteq A \times A^{**}$ be a program over the same state space. The *weakest precondition* of S with respect to R, $wp(S, R)$ is a logical function over the state space A. $(wp(S, R) : A \to L)$.

$\lceil wp(S, R) \rceil ::= \{a \in A | a \in \mathcal{D}_{p(S)} \wedge p(S)(a) \subseteq \lceil R \rceil\}$.

This means, that the image of a point $a \in A$ by the function $wp(S, R)$ is the set $\{true\}$, if the program S terminates surely starting from this point and all the sequences which are associated to a by S are finite and the program terminates in a state for which R holds, i.e. R holds for all the end points of the sequences associated to a.

Let $F \subseteq A \times A$ be a problem, $F_1 \subseteq A \times B$, $F_2 \subseteq B \times A$ be relations and $F = F_2 \ o \ F_1$. In this case B is called a *parameter space* of the problem. Let us define Q_b and R_b, the *pre- and postconditions* as logical functions over the state space on the following manner: $\forall b \in B$:

$\lceil Q_b \rceil ::= \{a \in A \mid (a, b) \in F_1\} = F_1^{(-1)}(b)$,

$\lceil R_b \rceil ::= \{a \in A \mid (b, a) \in F_2\} = F_2(b)$.

Theorem 2.1 *(The theorem of the specification)*

Let $F \subseteq A \times A$ be a problem, B a parameter space of the problem. Let $S \subseteq A \times A^{**}$ be a program over the state space A. If $\forall b \in B : Q_b \Rightarrow lf(S, R_b)$ then the program S is a solution of the problem F.

Giving the specification of the problem, first we determine the components of the state space, that is the types (classes) of the objects playing role in the problem. Then on the basis of these can we determine the real pre- and postconditions of the problem.

2.1 Data Types

The definition of a data type can fundamentally be divided into two parts. The first part is the specification of the data type: the abstract description of the objects of the type and the operations (methods) of the type. In the second part an appropriate representation for the set of values of the specified type is given and the solutions (effect relations) to the operations of the type. This second part will be called a "type".

The specification of a type consists of a set of values of the type and a set of problems (the specifications of the operations). On the other hand we have a definition of a type - which consists of a representation function ϱ and a set of programs defined on a state space not containing the specified set of values of the type and they don't seem to be easy to compare. The theorem of specification of data types gives a sufficient condition of the solution of a problem via a composition γ of the representation relation and the identity relation.

Definition 2.1 *A* triple $\mathcal{T}_s = (T, I_s, \mathbb{F})$ is said to be a *type specification* if the following conditions are satisfied: T: is an arbitrary base set, I_s:$T \longrightarrow \mathbb{L}$ is the invariant of the specification, $T_s = \lceil I_s \rceil$ is the type-value set, $\mathbb{F} = \{F_1, F_2, \ldots, F_n\}$, where $\forall i \in [1..n]$: $F_i \subseteq A_i \times A_i$, for which $A_i = A_{i_1} \times \ldots \times A_{i_{n_i}}$, and $A_{i_j} = T \Rightarrow pr_{A_{i_j}}(F_i) \subseteq \lceil I_S \rceil \times \lceil I_S \rceil$,

and $\exists j \in [1..n_i] : A_{i_j} = T$.
$F_1, F_2, \ldots, F_n$ are the specifications of the operations (methods) of the type.
The triple $\mathcal{T} = (\varrho, I, \mathbb{S})$ will be called a *type* if the following conditions are satisfied: $\varrho \subseteq E^* \times T$, is the representation relation, where E is the set of values of an elementary type, $I : E^* \longrightarrow \mathbb{L}$, is the invariant of the type, $\mathbb{S} = \{S_1, S_2, \ldots, S_m\}$, where $\forall i \in [1..m] : S_i \subseteq B_i \times B_i^{**}$ is a program, satisfying $B_i = B_{i_1} \times \ldots \times B_{i_{m_i}}$, and $\exists j \in [1..m_i] : B_{i_j} = E^*$.
We say that the *type* $\mathcal{T} = (\varrho, I, \mathbb{S})$ is *adequate* to the *type-specification* $\mathcal{T}_s = (T, I_s, \mathbb{F})$, if $\varrho(\lceil I \rceil) = T_s$ and $\forall F \in \mathbb{F} : \exists S \in \mathbb{S} : S$ is a solution through ϱ of F [4].

Theorem 2.2 *(The theorem of specification of data types)*
Let $\mathcal{T}_s = (T, I_s, \mathbb{F})$ be a type specification, $\mathcal{T} = (\varrho, I, \mathbb{S})$ be a type, $F \in \mathbb{F}$, $S \in \mathbb{S}$, let A be the state space, B a parameter space of F, and the specification of F is given.Let $\forall b \in B$ Q_b^γ and R_b^γ logical functions, where $\lceil Q_b^\gamma \rceil = \lfloor Q_b \circ \gamma \rfloor$ and $\lceil R_b^\gamma \rceil = \lceil R_b \circ \gamma \rceil$. Then if $\forall b \in B : Q_b^\gamma \Rightarrow wp(S, R_b^\gamma)$, then the program S is a solution of the problem F via ϱ.

2.2 Program constructions

Definition 2.2 *Let* S_1, $S_2 \subseteq A \times A^{**}$ *be programs. The sequential construction (sequence) of* S_1 and S_2 is the relation $S \subseteq A \times A^{**}$ which is defined as follows:

$$\forall a \in A : \; S(a) ::= \{\alpha \in A^\infty \mid \alpha \in S_1(a) \cap A^\infty\} \cup$$
$$\{\chi_2(\alpha, \beta) \mid \alpha \in S_1(a) \cap A^* \wedge \beta \in S_2(\tau(\alpha))\}$$

Notation: $S = (S_1; S_2)$.

S_1
S_2

Definition 2.3 *Let* $\pi_1, \ldots, \pi_n$ be conditions and $S_1, \ldots, S_n$ be programs over A. The *alternative construction* of programs $S_1, \ldots, S_n$ is the relation $IF \subseteq A \times A^{**}$ which is defined as follows: $\forall a \in A : IF(a) ::= \left(\bigcup_{i=1}^{n} w_i(a) \right) \cup w_0(a)$, where $\forall a \in A$

$$w_i(a) ::= \begin{cases} S_i(a), & a \in \lceil \pi_i \rceil \\ \emptyset, & \text{otherwise} \end{cases}$$

$$w_0(a) ::= \begin{cases} \{< a, a, \ldots >\}, & a \notin \bigcup_{i=1}^{n} \lceil \pi_i \rceil \\ \emptyset, & \text{otherwise} \end{cases}$$

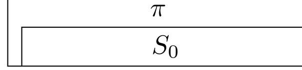
Notation: $IF = IF(\pi_1 : S_1; \ldots; \pi_n : S_n)$.

\ π_1	\ π_2	\ $\ldots$	\ π_n
S_1	S_2	$\ldots$	S_n

Definition 2.4 *Let* $\pi : A \to \mathbb{L}$ be a condition and S_0 a program over A. The *loop constructed from* π *and* S_0 is the relation $DO \subseteq A \times A^{**}$ which is defined as follows:
$\forall a \notin \lceil \pi \rceil : DO(a) ::= \{< a >\}$,

$$DO(a) ::= \big\{\alpha \in A^{**} \mid \exists \alpha_1, \dots, \alpha_n \in A^{**} :\ (\alpha = \chi_n(\alpha_1, \dots, \alpha_n) \land \\ \alpha_1 \in S_0(a) \land \forall i \in [1..n-1] :\ \alpha_i \in A^* \land \alpha_{i+1} \in S_0(\tau(\alpha_i)) \land \tau(\alpha_i) \in \lceil\pi\rceil) \land \\ ((\alpha_n \in A^* \land \tau(\alpha_n) \notin \lceil\pi\rceil) \lor \alpha_n \in A^{\infty})\ )\big\} \cup \\ \big\{\alpha \in A^{\infty} \mid \exists \alpha_1, \dots, \alpha_k, \dots \in A^* :\ (\alpha = \chi_{\infty}(\alpha_1, \dots, \alpha_k, \dots) \land \\ \alpha_1 \in S_0(a) \land \forall i \in \mathbb{N} :\ \alpha_i \in A^* \land \alpha_{i+1} \in S_0(\tau(\alpha_i)) \land \tau(\alpha_i) \in \lceil\pi\rceil)\big\}$$

Notation: $DO = DO(\pi : S_0)$.



2.3 Type constructions

Defining a new type using already existing (i.e. defined) types we have the choice of several possibilities: alter the methods of an existing type, render more strict the invariant of an existing type, build up a new set of values by the help of representation functions of existing types. This latter way traditionally means the construction of direct–product, union and iterated types [6].

Let $T_0, T_1, \dots, T_n,\ E_1, \dots, E_n$ denote non–empty, finite or numerable sets, $\varrho_i \subseteq E_i^* \times T_i$ representation functions, $\mathcal{T}_i = (\varrho_i, I_i, \mathbb{S}_i),\ \ (i = 0, \dots, n)$ types, $E ::= \cup_{i=1}^n E_i$.

Definition 2.5 *We say that* $\mathcal{T}$ is a *direct product* of $\mathcal{T}_i$ $(i = 0, \dots, n)$, if exists a representation function $\varrho \subseteq E^* \times T$, for that $\varrho = \psi_D \circ \varphi_D$, where $\psi_D \subseteq B \times T,\ B = T_1 \times \dots \times T_n$, $\varphi_D = \{(\varepsilon, b) \in E^* \times B \mid \exists \varepsilon_1, \dots, \varepsilon_n \in E^* : \forall i \in [1, n] : (\varepsilon_i, b_i) \in \varrho_i \land \varepsilon = \mathrm{con}(\varepsilon_1, \dots, \varepsilon_n)\}$. Notation: $\mathcal{T} = (\mathcal{T}_1, \dots, \mathcal{T}_n)$. If ψ_D is a one–to–one mapping, then $\mathcal{T}$ is a *record type.*

Definition 2.6 *We say that* $\mathcal{T}$ is a *union* of $\mathcal{T}_i$ $(i = 0, \dots, n)$, if exists a representation function $\varrho \subseteq E^* \times T$, for that $\varrho = \psi_U \circ \varphi_U$, where $\psi_U \subseteq B \times T,\ B = T_1 \cup \dots \cup T_n$,
$\varphi_U = \{(\varepsilon, b) \in E^* \times B \mid \exists i \in [1, n] :\ (\varepsilon, b) \in \varrho_i\}$. Notation: $\mathcal{T} = (\mathcal{T}_1; \dots; \mathcal{T}_n)$. If ψ_U is a one–to–one mapping, then $\mathcal{T}$ is a *conjunction type.*

Definition 2.7 *We say that* $\mathcal{T}$ is an *iteration* of $\mathcal{T}_0$, if exists a representation function $\varrho \subseteq E^* \times T$, for that $\varrho = \psi_I \circ \varphi_I$, where $\psi_I \subseteq B \times T,\ B = T_0^*$,
$\varphi_I = \{(\varepsilon, \beta) \in E^* \times T_0^* \mid \exists \alpha \in (E^*)^* : |\alpha| = |\beta| \land \forall i \in [1..|\alpha|] :\ (\alpha_i, \beta_i) \in \varrho_0 \land$
$\varepsilon = \mathrm{con}(\alpha_1, \dots, \alpha_{|\alpha|}\}$. Notation: $\mathcal{T} = \mathrm{it}(\mathcal{T}_0)$.

If ψ_I is a one–to–one mapping, then $\mathcal{T}$ is a *sequence type.* Notation: $\mathcal{T} = \mathrm{seq}(\mathcal{T}_0)$.

If $\mathcal{D}_{\psi_I} = T_0^*$ and $(\alpha, t), (\beta, t) \in \psi_I \iff \alpha \in \mathrm{perm}(\beta)$, then $\mathcal{T}$ is a *combination type* (or *bag*). Notation: $\mathcal{T} = \mathrm{com}(\mathcal{T}_0)$.

If $\mathcal{D}_{\psi_I} = T_0^*$ and $(\alpha, t), (\beta, t) \in \psi_I \iff \cup_{i=1}^{|\alpha|}\{\alpha_i\} = \cup_{i=1}^{|\beta|}\{\beta_i\}$, then $\mathcal{T}$ is a *set type.* Notation: $\mathcal{T} = \mathrm{set}(\mathcal{T}_0)$.

3 Extensions and projections of problems

Let us denote by pr_B the projection from A to B.

Definition 3.1 *Let the state space B be a subspace of the state space A. The relation $F' \subseteq A \times A$ is called the extension of the problem $F \subseteq B \times B$, if* $F' ::= \{(x, y) \in A \times A \mid (pr_B(x), pr_B(y)) \in F\}$.

In other words, the extension of a problem means that new variables are introduced without any restriction on them.

Definition 3.2 *Let the state space B be a subspace of the state space A, namely $A ::= \times_{i\in I} A_i$, $B ::= \times_{i\in J} A_i$ and $J \subseteq I \subset \mathbf{N}$. Denote by K the set $I \setminus J$, let $B' ::= \times_{i\in K} A_i$ and let S be a program on the state space B. The program $S' \subseteq A \times A$ is called the extension of the program S onto the state space A if* $\forall a \in A : \ S'(a) ::= \{\alpha \in A^{**} \mid \ pr_B(\alpha) \in S(pr_B(a)) \wedge \ \forall i \in \mathcal{D}_\alpha : pr_{B'}(\alpha_i) = pr_{B'}(a)\}$.

The extension of a program defined on a subspace gives rise to a program which operates on the subspace in the same way as the original program does and it does not change the rest of the components of the state space.

The extensions of problems and programs make it possible for us to understand two practical things theoretically. One of them is, that during program writing, we often need to introduce new variables, that is extend the state space. In this case the program will not solve the original problem, but its extension. The other one is it in order, to solve a subproblem we use a procedure, a program defined on a state space has common components with the state space of the problem. In this case the program /possibly the problem too/ has to be extended.

Definition 3.3 *Let B be a common subspace of A_1 and A_2. S_1 is a program over A_1, S_2 is a program over A_2. S_1 and S_2 are called equivalent on B, if $pr_B(p(S_1)) = pr_B(p(S_2))$* .

We use the following notations: B is a subspace of A and B' is the complement subspace of B in respect of A; S is a program over B, $F \subseteq B \times B$. S' is the extension of S and F' is the extension of F onto A. $\overline{S}$ is a program over A, $\overline{F} \subseteq A \times A$, $F = pr_{B\times B}(\overline{F})$, and the program $\overline{S}$ and S are equivalent over B. Then:

Definition 3.4 *The problem $\overline{F}$ is projection-invariant over B, iff $\forall\ a_1, a_2 \in \mathcal{D}_{\overline{F}}$: $(pr_B(a_1) = pr_B(a_2) \ \Rightarrow \ pr_B(\overline{F}(a_1)) = pr_B(\overline{F}(a_2)))$.*

Definition 3.5 *The problem $\overline{F}$ is an extended identity over the subspace B', iff $\forall\ (a_1, a_3) \in \overline{F} :\ \exists (a_1, a_4) \in \overline{F} : (pr_{B'}(a_4) = pr_{B'}(a_1) \ \wedge \ pr_B(a_3) = pr_B(a_4))$.*

Definition 3.6 *Let B be a subspace of A and B' is the complement subspace of B in respect of A and $H \subseteq B$. A relation $R \subseteq A \times A$ is called a semi-extension over H, iff $\forall h \in H : pr_B^{-1}(h) \subseteq \mathcal{D}_R$.*

The following theorems are related to these topics.

Theorem 3.1

1. S solves F if and only if S' solves F'.
2. if S solves F and $\overline{F}$ is an extended identity over B' and projection invariant over B then S' solves $\overline{F}$.
3. if S' solves $\overline{F}$ then S solves F.
4. if S solves F and $p(\overline{S})$ is a semi-extension over $\mathcal{D}_F$, then $\overline{S}$ solves F'.

5. if $\overline{S}$ solves F', then S solves F (theorem of introducing a new variable).
6. if $\overline{S}$ solves $\overline{F}$ and
   $\overline{F}$ is a semi-extension over $\mathcal{D}_F$ or $p(\overline{S})$ is a projection invariant over B, then S solves F.

   Proof. of 1. and 2. is given in [2], the remaining proofs are similar.

4 Coordinate transformations

We have to substitute some of the components of the state space by new ones in the case of more complex transformations. This way we associate the points of the original state space to the points of the new state space.

We may represent a given data type of the state space for example. A method for the implementation of abstract data types is presented in [4]. We exchange the type of one component by a related type often. The simplest type substitution methods are called type transformations and presented in this paper.

This kind of transformation may be generalized by introducing mappings between corresponding components of two state spaces. We call the state space transformation component-wise, if the association is determined by a mapping between a subspace of a state space and one single component of the other subspace. The state space transformation is called separable, if it is a composition of mappings from subspaces to subspaces.

4.1 Type transformations

Using the model we can derive, starting from an initial specification, a program which solves the specified problem. We can generalize some frequent problems thus we get a class of problems, and we can derive programs that solve these general problems [6]. These general programs are called *programming theorems*. The problems and their solutions, i.e., the programming theorems are usually formulated over an interval of integers. The specification of the problem and the solution can be easily transformed for several types using the following theorems.

Most of our programming theorems are constructed from elementary operations as like as choosing an element of an interval, set or sequence, taking the next element, etc. First we give define the semantics of the operations of the different type constructions using the notion of the weakest precondition.

Let $\mathcal{T} = \text{it}(\mathcal{T}_0)$ be an iterated type, $(\alpha, t) \in \psi_I$. $dom : T \to \mathbb{N}_0$ is defined as follows

$$dom(t) ::= \begin{cases} |\alpha|, & \text{if } \mathcal{T} \text{ is a sequence or a combination type} \\ \left| \bigcup_{i=1}^{|\alpha|} \{\alpha_i\} \right|, & \text{if } \mathcal{T} \text{ is a set type} \\ \text{undefined}, & \text{otherwise} \end{cases}$$

Let $\mathcal{T} = \text{seq}(\mathcal{T}_0)$ be a sequence type and let $(\alpha, t) \in \psi_I$

$lov : T \to T_0,\ lov(t) ::= \alpha_1$, if $dom(t) \neq 0$.

$hiext : T \times T_0 \to T,\ hiext(t, t_0) ::= \psi_I(con(\psi_I^{(-1)}(t), t_0))$.

$lorem : T \to T,\ loext(lorem(t), lov(t)) = t$, if $dom(t) \neq 0$.

a) $t : hiext(t_0) \ ::= \ t := hiext(t, t_0)$

b) $t : lorem \ ::= \ t := lorem(t)$

Let us suppose, that $\mathcal{T} = seqInp(\mathcal{T}_0)$, i.e. a sequential input file type. If the last element of the file has a special value: ext, then the operation $dx, x : lopop$ is defined to read the next element of the sequential input file, if it is not empty

a) $dx, x : lopop \ ::= \ x, dx := lorem(x), x.lov,$

otherwise we use the *read* operation:

b)

$$st, dt, t : read \ ::= \begin{cases} st, dt, t := norm, lov(t), lorem(t), & \text{if } dom(t) \neq 0 \\ st := abnorm, & \text{otherwise} \end{cases}$$

We define operations for sequential output files too:

A $\quad SeqOut(T_0) = (seq(T_0); create, t : write(t_0))$

a) $t : write(t_0) ::= t : hiext(t_0).$

Let us suppose, that $\mathcal{T} = Set(\mathcal{T}_0)$, i.e. a set type. Let us denote the union of the sets p and $\{e\}$ by $p\tilde{\cup}\{e\}$, if $e \notin p$. Similarly the $p\tilde{-}\{e\}$ denotes the set $p \setminus \{e\}$, if $e \in p$. The operation $e :\in \ p$ is a nondeterministic assignment to e choosing an arbitrary element of p if it is not an emptyset.

For arrays we use the usual operations. The formal definition of the operations is given in [6].

Theorem 4.1 *(Transformation of the programming theorems for other types)*

The schemes below define the substitution method of operations occurring in programming theorems. The operation of the first type has to be replaced everywhere of the operation of the second type printed in the same line.

a) From sequences (a) to **files** (lopop,ext) (x) : (the technique of reading in forward)

	$sx, dx, x : read$
$a.lov$	dx
$a : lorem$	$sx, dx, x : read$
$a.dom \neq 0$	$sx = abnorm$

The transformed program starts with an extra *lopop* operation. The transformation is similar, if we use a *lopop* operation instead of *read* .

Proof: Sequence a is mapped to $concat(< dx >, x)$. This association is preserved by the operations, so the transformation is correct.

b) From sets (x, y, z) to strict monotonic ordered **sequence** (a, b, c) (in the case of elementwise processing.)

$z := \emptyset$	$c := ()$
$x \neq \emptyset \vee y \neq \emptyset$	$a.dom \neq 0 \vee b.dom \neq 0$
$e :\in (x \cup y)$	$e := min(a.lov, b.lov)$
$z \tilde{\cup}$	$c : hiext$
$x \tilde{-}$	$a : lorem$
$e \in x \ , \ e \notin x$	$e = a.lov \ , \ e < a.lov$!!

Proof: see [8].

c) From function defined over an interval ($[m, n]$, $i \in [m, n]$ and function argument is $i + 1$) to **sequence** (x)

$i = n$	$x.dom = 0$
$f(i + 1)$	$x.lov$
$i := i + 1$	$x : lorem$

Proof: The interval $[i+1, n]$ is mapped to the sequence x. This association is preserved by the operations, so the transformation is correct.

d) From sequences (a, b) to **arrays** (v_1, v_2 , $j \in [v_1.lob, v_1.hib]$, $j \in [v_2.lob, v_2.hib]$) (to function type, if $b.dom \leq v_2.dom$)

$a.dom = 0$	$i = v_1.hib + 1$
$a.lov$	$v_1(i)$
$a : lorem$	$i := i + 1$
$b : hiext(e)$	$v_2(j + 1), j := e, j + 1$
$b :=<>$	$j := v_2.lob - 1$

Proof: The sequences a, b are mapped to the arrays $[v_1(i)..v_1(v_1.hib)]$ and $[v_2(v_2.lob)..j]$ respectively. This association is preserved by the operations, so the transformation is correct.

Example 4.1

Let us suppose we have to count the number of the words of a text, of which length is at least k.

$$A = \mathcal{F} \times \mathcal{N} \times \mathcal{N}_0, \quad x : \mathcal{F}, k : \mathcal{N}, s : \mathcal{N}_0. \quad \mathcal{F} = Seq(char).$$
$$B = \mathcal{F} \times \mathcal{N}, \quad x' : \mathcal{F}, \ k' : \mathcal{N}.$$
$$Q_{x',k'} = (x = x' \wedge k = k'),$$
$$R_{x',k'} = (s = \text{words}(x', k) \ \wedge \ k = k'),$$
$$\text{where } \text{words}(t, k) = \sum_{i=1}^{t.dom-k+1} (\beta(t, i, k)),$$

$$\beta(t,i,k) = \begin{cases} 1, & \text{if } \forall j \in [i, i+k-1] : \\ & \quad \text{letter}(t_j) \wedge (i = 1 \vee \neg\ \text{letter}(x_{i-1})) \\ 0, & \text{otherwise.} \end{cases}$$

We can solve the problem by a loop using the derivation rule of loop [6]. The loop invariant is

$$\begin{aligned} P_{x',k'} = (x' = \ &\text{kon}(x'', x) \wedge s = \ \text{words}(x'', k) \wedge \\ &(x''.dom \neq 0 \wedge x.dom \neq 0) \rightarrow \\ &(\neg\ \text{letter}(x''.hiv) \wedge\ \text{letter}(x.lov))) \end{aligned}$$

The loop body is a sequence of a special version of counting and of a linear searching [7]. Both theorems are transformed for sequences.

- $s := 0$
- loop while $x.dom \neq 0$:
  - $d := 0$
  - loop while $\text{letter}(x.lov) \wedge x.dom \neq 0$:
    - $d := d + 1$
    - $x : \text{lorem}$
  - loop while $\neg\text{letter}(x.lov) \wedge x.dom \neq 0$:
    - $x : \text{lorem}$
  - if $d \geq k$:
    - then $s := s + 1$
    - else $SKIP$

4.2 State space transformation

In this section we solve our example problem by the application of a separable state space transformation. We simplify the problem by introducing a new state space.

We associate a sequential file of natural numbers of the new state space to the sequence of characters of the old state space. The i-th element of the new sequence is the length of the i-th word of the original sequence.

Example 4.2

$$\begin{aligned} &A_1 = \mathcal{G} \times \mathcal{N} \times \mathcal{N}_0, \quad x : \mathcal{G}, k : \mathcal{N}, s : \mathcal{N}_0. \quad \mathcal{G} = InpSeq(\mathcal{N}) \\ &B = \mathcal{G} \times \mathcal{N}, \quad x' : \mathcal{G}, k' : \mathcal{N}. \\ &Q_{x',k'} = (x = x' \wedge k = k'), \\ &R_{x',k'} = (s = \ \text{long}(x', k) \ \wedge \ k = k'), \\ &\quad \text{where } \ \text{long}(t, k) = \sum_{i=1}^{t.dom} (\beta(t, i, k)), \end{aligned}$$

$$\beta(t,i,k) = \begin{cases} 1, & \text{if } t_i \geq k \\ 0, & \text{otherwise.} \end{cases}$$

We get the solution of the problem by a simple application of the theorem of counting transformed for sequential files:

Open(t)

st, t, dt : **read**

$s := 0$

$st = \text{norm}$

$dt \leq k$

$s := s + 1$	$SKIP$

st, t, dt : **read**

We have to solve the subproblems of implementing the abstract open and read operations to get a solution over the original state space. The solution is produced by an application of the programming theorems as like as before.

Open(t)

st, t, dt : read

$sx = \text{norm} \wedge \neg\ \text{letter}(dx)$

st, t, dt : read

st, dt, t : **read**

$sx = \text{norm}$

$st := \text{norm}$ $dt := 0$ $\text{letter}(dx) \wedge sx = \text{norm}$ $dt := dt + 1$ st, dt, t : read $\neg\ \text{letter}(dx) \wedge sx = \text{norm}$ st, dt, t : read	$st := \text{abnorm}$

5 Simple program transformations

Definition 5.1 *We* say that the effect relation $p(S)$ of the program S doesn't depend on the state space component A_i, if

$\forall a, b \in \mathcal{D}_{p(S)} : (\forall k \in ([1, i-1] \cup [i+1, n]) : a_k = b_k) \rightarrow p(S)(a) = p(S)(b)$

Definition 5.2 *We* say that the variable $a_i : A_i$ is a constant function in the program S, if $\forall a \in A : \forall \alpha \in S(a) : (\forall \alpha_k \in \alpha : \alpha_{k\,i} = a_i)$

Definition 5.3 *We* say that the execution of the program S does not change the variable $a_i : A_i$, if $\forall a \in \mathcal{D}_{p(S)} : p(S)(a)_i = a_i)$

Theorem 5.1 *(Substitution of not elementary conditions of an alternative construct)*
Let $S \subseteq A \times A^{**}$, where $S = IF(\pi_1 : S_1, ..., \pi_n : S_n)$, $A = A_1 \times ... \times A_m$. Let $S' \subseteq A' \times A'^{**}$, where $A' = A_1 \times ... \times A_m \times \mathcal{L}_1 \times ... \times \mathcal{L}_n$, $a_i : A_i, l_j : \mathcal{L}_j$.

$S' \subseteq A' \times A'^{**}$:

$l_1 \dots l_n := \pi_1(a_1, ..., a_m) \dots \pi_n(a_1, ..., a_m)$
$IF(l_1 : S_1, ..., l_n : S_n)$

The program S' is equivalent with the program S over the state space A.

Theorem 5.2 *(Substitution of the loop condition by a variable)*
Let $S \subseteq A \times A^{**}$, where $S = DO(\pi : S_0)$, $A = A_1 \times ... \times A_m$. Let $S_{DO} \subseteq A' \times A'^{**}$, where $A' = A_1 \times ... \times A_m \times \mathcal{L}$, $a_i : A_i, l : \mathcal{L}$.

$S_{DO} \subseteq A' \times A'^{**}$:

$l := \pi(a_1, ..., a_m)$
l
S_0
$l := \pi(a_1, ..., a_m)$

The program S_{DO} is equivalent with the program S over the state space A.

Theorem 5.3
(*The transformation of the simultaneous assignment into simple assignments)*
$S \subseteq A \times A^{**}$, where $A = A_1 \times ... \times A_m$, $S' \subseteq A' \times A'^{**}$, where
$A' = A_{11} \times ... \times A_{m1} \times A_{12} \times ... \times A_{m2}$, $a_{i1} : A_{i1}$, $a_{i2} : A_{i2}$.
$S ::= a_{11}, ..., a_{m1} := f_1(a_{11}, ..., a_{m1}), ..., f_m(a_{11}, ..., a_{m1})$.

$S' \subseteq A' \times A'^{**}$:

$a_{12} := f_1(a_{11}, ..., a_{m1}) \dots a_{m2} := f_m(a_{11}, ..., a_{m1})$
$a_{11} := a_{12} \dots a_{m1} := a_{m2}$

The program S' is equivalent with the program S over the state space A.

Note: A special case is when f_i does not depend on every component of A .

Theorem 5.4 *(The interchange of the two members of the sequence of programs)*
If $S_{12} = (S_1; S_2), S_{21} = (S_2; S_1)$ and every component of the state space on which S_1 depends on is not changed by the execution of S_2 and in reverse, then S_{12} is equivalent to S_{21} over A.

Theorem 5.5 *(The interchange of the two components of the sequence of programs which are embedded in a loop body)*
If $S = DO(\pi : S_0)$, $S_0 = (i := i + 1; S_{01})$, $S' = ({S_{01}}^{i \leftarrow i+1}; i := i + 1)$ and i is a constant function in S_{01}, then S is equivalent with S' over A.

Theorem 5.6 *(The substitution of a function by a variable)*
Let f be a function, which is defined over the subspace $A_{i_1} \times ... \times A_{i_k}$. Let the set H be the range of f. ($fo(a_1, ..., a_n)$ is defined over the state space A) If the variables $a_{i_1}, ..., a_{i_k}$ are constant functions in S, then the program S is equivalent with the program

$S' = (z := f(a_{i_1}, ..., a_{i_k}); S^{f(a_{i_1},...,a_{i_k}) \leftarrow z})$, over the sate space A, where $S' \subseteq A' \times A'^{**}$ and $A' = A_1 \times ... \times A_n \times H$, $a_i : A_i$, $z : H$.

Theorem 5.7 *The substitution of a recursive function*

Let H be an arbitrary set , $k > 0$ an integer, and let $f : \mathcal{Z} \rightarrow H$, $h : \mathcal{Z} \times H^k \rightarrow T$ be functions and let hold $f(0) = t_0$, $f(-1) = t_{-1}$,..., $f(-k+1) = t_{-k+1}$ and $\forall i > 0 :$ $f(i+1) = h(i+1, f(i), ..., f(i-k+1))$. $S ::=$

$i := 0$	
S_1	
π	
	S_0
	$i := i+1$

where i is a constant function in S_1 as well as in S_0, and there is a reference to $f(i+1)$ in S_0. S is equivalent with the following program over A:

$i, z, z_{-1}, .., z_{-k+1} := 0, t_0, t_{-1}, .., t_{-k+1}$	
S_1	
π	
	$z, z_{-1}, .., z_{-k+1} := h(i+1, f(i), ..., f(i-k+1)), z, .., z_{-k+2}$
	${S_0}^{f(i+1) \leftarrow z}$
	$i := i+1$

The proofs of the theorems 5.1-5.7 are based on the definition 3.3.

Theorem 5.8 *(Program inversion)*

The problem of program inversion, a well-known and widely used program transformation, - described informally by M.Jackson as a solution to the problem of structure clashes is adapted to this model. Using the notion of elementwise processing, the concepts of elementwise producer and elementwise consumer programs are introduced. With the help of these it is proved how one can eliminate the intermediate sequential files needed in the original solution of the problem while creating an "inverted" program as a result. This transformation is relatively complex, a detailed discussion is presented in [3].

Example 5.1 *We* show an example for the application of the transformation called "the substitution of a recursive function". We solve the same example problem as before. The values of the recursive function f are defined to take the actual number of letters of the current word. Our problem can be specified as counting how many times takes function f value k.

$$f_0 = 0$$

$$f_{i+1} = \begin{cases} f_i + 1, & \text{if letter}(x_{i+1}) \\ 0, & \text{otherwise.} \end{cases}$$

We apply the theorem of counting [6] for the interval $[1..x.dom]$ and finally we apply the theorem of "the substitution of a recursive function" and we transform the result for sequences (see previous section).

$s, i := 0, 0$	
$i \neq x.dom$	
$f_{i+1} = k$	
$s := s + 1$	$SKIP$
$i := i + 1$	

$s, z := 0, 0$	
$x.dom \neq 0$	
letter($x.lov$)	
$z := z + 1$	$z := 0$
$z = k$	
$s := s + 1$	$SKIP$
x : lorem	

References

[1] Dijkstra, E. W., Scholten, C. S.: *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990.

[2] Fóthi, Á.: A Mathematical Approach to Programming, *Annales Uni. Sci. Budapest. de R. Eötvös Nom. Sectio Computatorica*, Tom. IX. (1988), 105–114.

[3] Fóthi Á., Nyéky Gaizler, J.: Program Inversion. Varga, L., ed., *Proceedings of the Fourth Symposium on Programming Languages and Software Tools*, Visegrád, Hungary, June 8-14, 1995 (1995), to appear in: *Annales Uni. Sci. Budapest de R. Eötvös Nom. Sectio Computatorica* (1997).

[4] Fóthi, Á., Horváth, Z., Nyékyné Gaizler, J.: Data Types and Specifications. Paakki, J., ed., *Proceedings of the Fifth Symposium on Programming Languages and Software Tools*, Jyväskylä, Finland, June 7-8, 1997 (1997) 135-144.

[5] Malcolm, G.: Data Structures and Program Transformation, *Science of Computer Programming*, 14 (1990) 255-279.

[6] Workgroup on Relational Models of Programming - Fóthi Á. et al.: Some concepts of a Relational Model of Programming. Varga, L., ed., *Proceedings of the Fourth Symposium on Programming Languages and Software Tools*, Visegrád, Hungary, June 8-14, 1995 (1995) 434-446.

[7] Fóthi, Á.: Introduction into Programming (Bevezetés a programozáshoz, in Hungarian), Tankönyvkiadó, Budapest, 1983.

[8] Gregorics T., Sike S.: Type transformation for elementwise processing, Presentation on ICAI'97, Eger-Noszvaj, Hungary, 1997. Aug.

Postal Addresses

Ákos Fóthi, Zoltán Horváth, Judit Nyéky-Gaizler

Department of General Computer Science
Eötvös Loránd University
H-1088 Budapest, Múzeum krt. 6–8.
Hungary