ON IANOV SCHEMAS WITH ONE MEMORY LOCATION

K. INDERMARK

Introduction

One possibility of investigating programming languages is given by the fact that they possess data structures as well as control structures. A program schema can be understood roughly as a program without data structures. Thus, the theory of program schemas, sometimes calles schematology, models just control structures of programming languages. Their relations can be expressed in form of translatability theorems which are based on the notion of strong equivalence.

Ianov schemas (I-schemas) are very simple schemas allowing only one variable and no recursion. They have been generalized in several ways, e.g. by recursion or adding new variables, counters, stacks, etc.. In this paper we give two relatively small extensions of the class of I-schemas: a) IB-schemas - these are I-schemas where instead of a simple predicate symbol arbitrary I-schemas with two exits (boolean subschemas) may occur. Microprogramming gives examples for this situation. b) IM-schemas are I-schemas with a second variable that can only be used as a memory, namely for storing or fetching a value, but not to perform computations. Finally, by IR-schemas we mean I-schemas with recursion as introduced by de Bakker and Scott [BS 69].

It was shown in [IN 72] that the class of IB-schemas is not translatable into the class of IR-schemas, and vice versa. Here, we compare IB-schemas with IM-schemas. Our principal aim is to prove their intertranslatability. This theorem together with a theorem of Glushkov [GL 65] and a theorem of Ito [IT 71] yields the conjecture that it is effectively decidable whether two IM-schemas (IB-schemas) are strongly equivalent or not. This result would be interesting in so far as for monadic program schemas with two variables the strong equivalence problem is undecidable. Moreover, one knows only few classes of schemas where this problem is decidable. In the case of IR-schemas the solution of this problem is even unknown [PA 72].

This research was conducted at the Institut für Theorie der Automaten und Schaltnetzwerke (GMD) and supported by the Gesellschaft für Mathematik und Datenverarbeitung mbH., Bonn.

Basic definitions

In this section we define syntax and semantics of IB-schemas and IM-schemas.

a) Syntax

Let Φ, π, $\mathbb{N}$, {STO, FET, STA, END, RE1, RE2} be disjoint sets.

1) The set of instructions over these sets is the disjoint union of the following sets:

$\{STA\} \times \mathbb{N}$	start instructions	(I)
$\mathbb{N} \times \{END\}$	stop instructions	(II)
$\mathbb{N} \times \Phi \times \mathbb{N}$	assignment instructions	(III)
$\mathbb{N} \times \pi \times \mathbb{N} \times \mathbb{N}$	test instructions	(IV)
$\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$	boolean subroutine calls	(V)
$\mathbb{N} \times \{RE1,RE2\}$	return instructions	(VI)
$\mathbb{N} \times \{STO\} \times \mathbb{N}$	store instructions	(VII)
$\mathbb{N} \times \{FET\} \times \mathbb{N}$	fetch instructions	(VIII)

2) The non-negative integers occurring in these instructions are called labels. They are used for describing the flow of control. A label n of an instruction i is called L-label of i if it occurs in the first component, jump label of i if i is a subroutine call with n in the second component, and R-label of i in all other cases.

3) A non-empty, finite set S of instructions is called a schema iff S has the following properties:

(i) S has exactly one start instruction and one or more stop instructions.

(ii) Different instructions have different L-labels.

A schema S is called

(i) I-schema iff its instructions are of type I, II, III or IV,

(ii) IM-schema iff it has neither boolean subroutine calls nor return instructions,

(iii) IB-schema iff it has neither store nor fetch instructions but the following property: S can be decomposed into S_0, S_1, ..., S_r such that all start instructions, stop instructions and boolean subroutine calls are in S_0, the sets of L-, R-labels of different components are disjoint and the jump labels of S_0 occur as L-labels in S_1, ..., S_r.

b) Semantics

1) An interpretation of Φ and π is a pair $I = (D, \Gamma)$ where D is a set and Γ a mapping that gives for each $f \in \Phi$ a function $f^I : D \to D$ and for each $p \in \pi$ a predicate $p^I : D \to \{0, 1\}$. A schema S whose function and test symbols (elements of Φ and π) are interpreted by I is called a program P, formally $P = (S, I)$.

2) The semantics of a program will be given operationally ("by its effect on the state vector").
Let $Z := \mathbb{N} \times D \cup \mathbb{N} \times D \times D \cup \mathbb{N} \times (D \cup \{0,1\}) \times D \times \mathbb{N}$ be the set of states. Each instruction i not being a start or stop instruction evokes a state transition function $\Delta_i : Z \dashrightarrow Z$ [+)] which is determined by the interpretation I. They are listed up according to the instruction type:

(i) For an assignment instruction $i = (m, f, m')$ we have Δ_i by:

$(m, d) \mapsto (m', f^I(d))$
$(m, d, d') \mapsto (m', f^I(d), d')$
$(m, d, d', m'') \mapsto (m', f^I(d), d', m'')$

(ii) For a test instruction $i = (m, p, m_0, m_1)$ we have Δ_i by:

$(m, d) \mapsto (m_{p^I(d)}, d)$
$(m, d, d') \mapsto (m_{p^I(d)}, d, d')$
$(m, d, d', m') \mapsto (m_{p^I(d)}, d, d', m')$

(iii) For a store instruction $i = (m, STO, m')$ we have Δ_i by:

$(m, d) \mapsto (m', d, d)$
$(m, d, d') \mapsto (m', d, d)$

(iv) For a fetch instruction $i = (m, FET, m')$ we have Δ_i by:

$(m, d, d') \mapsto (m', d', d')$

(v) For a boolean subroutine call $i = (m, m', m_0, m_1)$ we have Δ_i by:

$(m, d) \mapsto (m', d, d, m)$
$(m, t, d) \mapsto (m_t, d) \quad t \in \{0,1\}$

(vi) For a return instruction $i = (m, RE t)$ we have Δ_i by:
$(m, d, d', m') \mapsto (m', t, d') \quad t \in \{0,1\}$

+) $f : A \dashrightarrow B$ denotes a partial function from A to B.

In all other cases Δ_i is undefined.
A remark to the boolean subroutine technique as given by (v) and (vi): at a subroutine call its L-label and the computed value are stored; when the subroutine is finished by a return instruction, the boolean result t is stored and control goes back to the subroutine call to look for the next instruction determined by t.

3) Now, let P = (S, I) be a program. As different instructions i_1 and i_2 of S have different L-labels it follows that dom $(\Delta_{i_1}) \cap$ dom $(\Delta_{i_2}) = \emptyset$. Thus, P produces a transition function Δ_P: $Z \dashrightarrow Z$ defined by $\Delta_P = \bigcup_{i \in S} \Delta_i$ representing functions by their graphs. Δ_P describes the meaning of P: for an input value $d \in D$ the program P generates a state sequence in Z by iteration of Δ_P on (m_{STA}, d) where $(STA, m_{STA}) \in S$. The state sequence of P at d contains the execution sequence as the sequence of corresponding L-values and the computation sequence as the sequence of corresponding values of D and $(D \cup \{o,1\}) \times D$.
This shows that P computes a function $F_P : D \dashrightarrow D$: we have $F_P(d) = d'$ iff the state sequence of P at d is finite ending with some (m, d') or (m, d', d'') $\in Z$ such that (m, END) $\in S$.

Equivalence

Definition Let S_1 and S_2 be schemas and C_1 and C_2 classes of schemas.

(i) S_1 is strongly equivalent to S_2 iff for all interpretations I we have $F_{(S_1, I)} = F_{(S_2, I)}$.

(ii) C_1 is translatable into C_2 iff for each $S \in C_1$ there is $S' \in C_2$ such that S is strongly equivalent to S'.

(iii) C_1 and C_2 are intertranslatable iff C_1 is translatable into C_2 and vice versa.

Translatability between IB-schemas, IM-schemas and IR-schemas

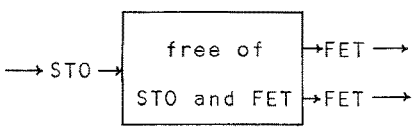
The above definitions provide a suitable framework for a comparison of the introduced extensions of I-schemas. We review first the relationship between IB-schemas and IR-schemas (de Bakker-Scott-schemas) as presented in [IN 72]. Subsequently, we compare IB-schemas with IM-schemas.

Theorem 1 The class of IB-schemas is not translatable into the class of IR-schemas and vice versa.
For the proof we use a special interpretation called gsm. It has the following property: it interprets I-schemas as deterministic generalized sequential machines, IR-schemas as deterministic pushdown transducers and IB-schemas as gsm's with a look-ahead on the input tape. The induced classes of "pushdown" - and "look-ahead"-transductions turn out to be orthogonal with respect to inclusion, thus proving the theorem.

The semantical definition of IB-schemas shows that boolean subroutines are evaluated in the main store while keeping the current main program value on a memory location. This makes clear that an IB-schema is effectively translatable into an equivalent IM-schema.

Theorem 2 The class of IB-schemas is translatable into the class of IM-schemas.
Proof Let $S = S_0 \cup S_1 \cup \ldots \cup S_r$ be an IB-schema with main schema S_0 and subschemas $S_1, \ldots, S_r$. We have to change only the subroutine technique. Therefore, we simulate entries to and exits from subschemas by store and fetch instructions. We assume S to have the following property: different subroutine calls address different subschemas. This can be achieved by introducing new label sets to make copies of subschemas. Now, let (m, j, m_0, m_1) be a boolean subroutine call of S_0 and $(n_1, RE\uparrow_1), \ldots, (n_s, RE\uparrow_s)$ all return instructions of the corresponding subschema. We replace these instructions by (m, STO, j), $(n_1, FET, m_{\uparrow_1}), \ldots, (n_s, FET, m_{\uparrow_s})$. These changes affect the computation sequences only on return from a subschema to S_0: one state is omitted, the boolean value $\uparrow$ does not appear again. Thus, the resulting schema is strongly equivalent to S. Finally, we drop those subschemas which are not addressed by subroutine calls.

The following remark illustrates why the converse of Theorem 2 is not so obvious. IM-schemas that correspond to IB-schemas by the above proof technique have a special structure: store and fetch instructions occur only in certain constructs as given by the picture.



Yet, in an arbitrary IM-schema store and fetch instructions are distributed without any restriction. E.g., a stored value can be fetched many times, before a new value is stored.

We shall demonstrate now how an arbitrary IM-schema is effectively translatable into an equivalent IB-schema. The translation procedure can be characterized as "cutting and pasting" [GL 73]. It consists of two parts. At first, we translate an IM-schema into an unmixed IM-schema, and after that into an IB-schema.

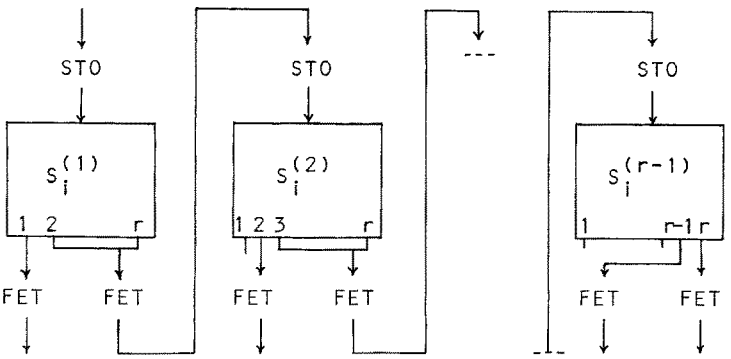
Definition Let S be an IM-schema.

S is called unmixed iff S allows a partition $S = S_0 \cup S_1 \cup \ldots \cup S_r$ with the following properties: (i) S_0 has all start and stop instructions of S but neither store nor fetch instructions. (ii) Each S_i ($i = 1, \ldots, r$) has exactly one store instruction. (iii) The label sets of the components are pairwise disjoint except that L-labels of store instructions occur as R-labels only in S_0 or in fetch instructions, and R-labels of fetch instructions occur as L-labels only in S_0 or in store instructions.

Intuitively, S is unmixed iff S decomposes in main schema and boolean subschemas. The latter ones are however more general than those used for the definition of IB-schemas as they permit more than two exits. The next theorem shows that this restriction in defining IB-schemas was only a syntactical one.

Theorem 3 The class of unmixed IM-schemas is translatable into the class of IB-schemas.

Proof A boolean subschema S_i of an unmixed IM-schema S represents with respect to an interpretation $I = (D, \Gamma)$ a partial decision function $g: D \dashrightarrow \{1, 2, \ldots, r\}$, r being the number of fetch instructions of S_i. It can be assumed that $r \geq 2$: if $r = 0$, S_i can be dropped, and if $r = 1$, we add a second fetch instruction which is never accessed. Now, it is easy to see how g can be simulated by a net of r-1 boolean subschemas with two exits (fetch instructions). One takes copies of S_i where all but one exits are matched and passes them successively. The corresponding schema is given by the picture, details are left to the reader.



The difficulty of translating an IM-schema S into an unmixed one lies in the fact that some parts of S can be entered both from store and from fetch instructions. Such parts operate therefore as main schema and as boolean subschema. S appears to be a superposition of several other schemas. The translation technique consists in separating these layers and connecting them afterwards appropriately.

Theorem 4 The class of IM-schemas is translatable into the class of unmixed IM-schemas.

Proof Let S be an IM-schema with store instructions $i_{s1}, \ldots, i_{sp}$ and fetch instructions $i_{f1}, \ldots, i_{fq}$. Corresponding L- and R-labels are denoted by L_{s1}, R_{s1}, etc. We shall construct a strongly equivalent unmixed IM-schema by successive effective transformations. The proof is illustrated below for the case p = q = 1.

1. From S without start instruction we produce p copies $S^{(s1)}, \ldots, S^{(sp)}$. A label m of S has corresponding labels $m^{(sj)}$ in $S^{(sj)}$. Now, we replace in S all occurrences of L_{sj} as R-labels by $L_{sj}^{(sj)}$ (j=1,...,p). The effect is to leave S when storing a value. Furthermore, we replace in $S^{(sj)}$ the R-labels of all fetch instructions by their corresponding labels of S, except that an eventually occurring $L_{sk}^{(sj)}$ is not replaced by L_{sk} but by $L_{sk}^{(sk)}$ (j, k = 1, ..., p). Thus, after fetching a value control returns to S or to the "store entry" of some $S^{(sk)}$. In the resulting schema S_1 the store instructions of S are never accessed and can therefore be dropped.

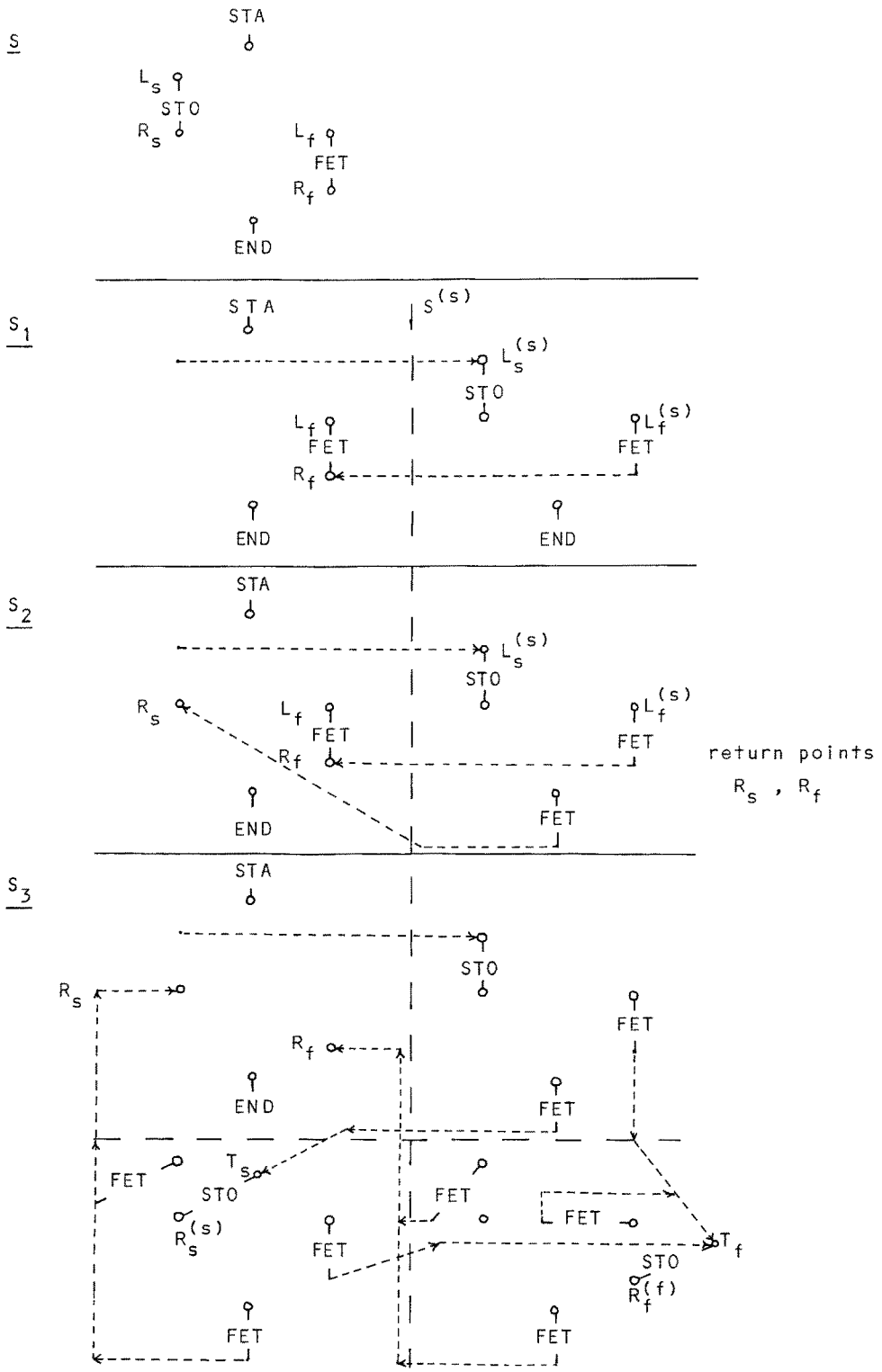
2. Our next aim is to convert the subschemas $S^{(sj)}$ into boolean subschemas with several exits. $S^{(sj)}$ has already the property that it can be entered only by the store instruction with L-label $L_{sj}^{(sj)}$, and it is left just after a fetch instruction. So, we have only

to remove further store and stop instructions. But, if we reach a store or stop instruction in $S^{(sj)}$, the storage on entry was unnecessary: we could have done this computation in the main schema S. Therefore, we replace (m, STO, m') resp. (m, END) in $S^{(sj)}$ by (m, FET, R_{sj}) provided that $m \neq L_{sj}^{(sj)}$, thus getting S_2 with the desired boolean subschemas.

3. The final task will be the elimination of fetch instructions in the main schema S. If a fetch instruction of S is met without having used a subschema $S^{(sj)}$, computation stops with undefined value. If it is reached after some $S^{(sj)}$, the return from $S^{(sj)}$ to S could have accessed directly its R-value. This would avoid superfluous computation. Therefore, we could drop the fetch instructions of S, if we had a device that gives us on return from some $S^{(sj)}$ the last fetch instruction of S encountered during computation. To this end, we insert a test procedure that computes the return label. For each return point of S (R-labels of store and fetch instructions) we create a new copy of S (without start instructions) $\tilde{S}^{(r)}$ with $r \in \{s1, \ldots, sp, f1, \ldots, fq\}$. Control enters these routines by the following changes: replace in each fetch instruction of boolean subschemas the R-label R_r by a new label T_r and add (T_r, STO, $\tilde{R}_r^{(r)}$) to $\tilde{S}^{(r)}$ ($r \in \{s1, \ldots, fq\}$). Now, we convert $\tilde{S}^{(r)}$ into a boolean subschema: replace ($m^{(r)}$, STO, $n^{(r)}$) or ($m^{(r)}$, END) by ($m^{(r)}$, FET, R_r) and ($m^{(r)}$, FET, $\tilde{R}_t^{(r)}$) by ($m^{(r)}$, FET, T_t). Thus, control moves between these subschemas until a store or stop instruction was originally encountered. The last subschema in this process gives the return label. Fetch instructions of S are no more accessed and can be dropped. The resulting schema S_3 consists of one main schema without store and fetch instructions and 2p + q boolean subschemas. It is unmixed and strongly equivalent to S.

Example p = q = 1

As the translation procedure changes only the labels of start, stop, store and fetch instructions, it is sufficient to represent S only by this label set.


S
STA
L_s
STO
R_s
L_f
FET
R_f
END
S_1
STA
$S^{(s)}$
$L_s^{(s)}$
STO
L_f
FET
R_f
$L_f^{(s)}$
FET
END
END
S_2
STA
$L_s^{(s)}$
STO
R_s
L_f
FET
R_f
$L_f^{(s)}$
FET
return points
R_s , R_f
END
FET
S_3
STA
STO
R_s
R_f
FET
END
FET
FET
T_s
STO
$R_s^{(s)}$
FET
FET
FET
T_f
STO
$R_f^{(f)}$
FET
FET

Combining the last three theorems we conclude our main result.

Theorem 5 The class of IM-schemas and the class of IB-schemas are intertranslatable.

Decidability of IM-schemas

The translation procedure from IM-schemas to IB-schemas is effective (see the proofs to Theorem 4 and 3). IB-schemas are closely related to Glushkov's regular microprograms [GL 65] . Ito claims to have a proof for the decidability of their equivalence [IT 71]. As we have not yet been able to fill the gaps we can only state a conjecture.

Conjecture The strong equivalence problem of IM-schemas is decidable.

References

[BS 69] de Bakker, J.W. / Scott, D.: A theory of programs. Unpublished memo (1969)

[GL 73] Garland, S.J. / Luckham, D.C.: Program schemas, recursion schemas, and formal languages. JCSS 7 (1973), 119 - 16o

[GL 65] Glushkov, V.: Automata theory and formal microprogram transformations. Kibernetika 1 (1965) 5, 1 - 9

[IN 72] Indermark, K.: Programmschemata mit booleschen Unterprogrammen. 2. GI-Jahrestagung (1972), 1o7 - 115

[IT 71] Ito, T.: A theory of formal microprograms. Summer-School, St. Raphael (1971)

[LPP 7o] Luckham, D. / Park, D. / Paterson, M.: On formalised computer programs. JCSS 4 (197o), 22o - 249

[PA 72] Paterson, M.: Decision problems in computational models. Proc. ACM Conference Las Cruces (1972), 74 - 82