A partial correctness logic for procedures
(in an ALGOL-like language)

by

Kurt Sieber

A 84/13

Universität des Saarlandes
6600 Saarbrücken
West Germany

(submitted for publication)

Abstract:

We extend Hoare's logic by allowing quantifiers and other logical connectives to be used on the level of Hoare formulas. This leads to a logic in which partial correctness properties of procedures (and not only of statements) can be formulated adequately. In particular it is possible to argue about free procedures, i.e. procedures which are not bound by a declaration but only "specified" semantically. This property of our logic (and of the corresponding calculus) is important from both a practical and a theoretical point of view, namely:

- Formal proofs of programs can be written in the style of stepwise refinement.
- Procedures on parameter position can be handled adequately, so that some sophisticated programs can be verified, which are beyond the power of other calculi.

The logic as well as the calculus are similar to Reynolds' specification logic. But there are also some (essential) differences which will be pointed out in this paper.

1. Introduction

Developing a program by stepwise refinement means
- writing an abstract program that uses procedures which are not (yet) declared but only specified by semantical properties,
- independently writing declarations for these procedures, so that they satisfy their specifications.

If a program is developed in this style, then of course it should be proved in the same style. In a logic which allows such proofs, it must be possible
- to specify procedures, i.e. to describe them by purely semantical properties,
- to verify an abstract program starting from such specifications.

In order to understand the difficulties arising with this way of reasoning in Hoare's logic, consider the following example:
Assume we want to express that a (parameterless) procedure P increases the value of a (global) variable x by 1. Apparently this can be realized by the Hoare formula

$$\{x = y\}\ P(\ )\ \{x = y + 1\} \tag{1}$$

(apart from the fact that Hoare formulas only describe partial correctness). Assume further that P is used in an abstract program and we want to conclude from (1), say

$$\{x = 1\}\ P(\ )\ \{x = 2\} \tag{2}$$

which is usually written as a so-called proof line

$$\{x = y\}\ P(\ )\ \{x = y + 1\} \supset \{x = 1\}\ P(\ )\ \{x = 2\}. \tag{*}$$

Then the question arises how to define the semantics of formulas like (*) (which is "intuitively valid"). It is well known that the naive approach - where arbitrary procedures can be inserted for P - fails: If P is declared by

P( ) : x := y + 1 or

P( ) : <u>if</u> x = y <u>then</u> x := x + 1 <u>else</u> x := x + 2 <u>fi</u>,

then - in both cases - the premise (1) is satisfied but the conclusion (2) is not.

The usual way out is to forbid the variable y to be accessed by the procedure P. This can be achieved by inspecting a context which contains all variables accessed by P (cf. [dBa 80], [Old 81]) or by distinguishing between program variables (occuring in the programs) and algebraic variables (occuring in the assertions only, cf. [Sie 81], [CGH 83]).

Here we propose a more "natural" solution: We distinguish between value identifiers (a, b, c, ...), variable identifiers (x, y, z, ...) and procedure identifiers (P, Q, R, ...) of various types. Quantifiers for all these identifiers are allowed and explicit dereferencing from variables to values is expressed by a symbol "cont". A quantified value identifier can then be used to reformulate our example (*):

$$\forall c.\{cont(x) = c\}\ P(\ )\ \{cont(x) = c + 1\}$$
$$\supset \{cont(x) = 1\}\ P(\ )\ \{cont(x) = 2\} \qquad (**)$$

This formula is valid in the naive interpretation. The premise now really means that "for all values c" partial correctness holds with respect to cont(x) = c and cont(x) = c + 1, hence it holds in particular for the value 1. Note that the procedures declared by

P( ) : x := c + 1 or

P( ) : <u>if</u> x = c <u>then</u> x := cont(x) + 1 <u>else</u> x := cont(x) + 2 <u>fi</u>

don't satisfy the premise of (**).

(Of course this last argumentation was rather informal; it will be justified by the formal definition of our logic in section 3.)

So far, the use of quantifiers can be regarded as a matter of taste, because they can be replaced by variables which are not accessed by certain procedures. But it will turn out that - in connection with procedures on parameter position - quantifiers even increase the power of the calculus.

Semantically procedures with parameters are usually considered as functions of their parameters. In order to describe a function it is necessary to define its values <u>for all</u> arguments. From this point of view quantifiers are useful for all kinds of identifiers which may occur as parameters.
As an example consider the formula

$$\forall c,x.\{cont(x) = c\}\ Q(x)\ \{cont(x) = c + 1\}$$

which means that for each parameter x the procedure call Q(x) increases the value of x by 1. Hence it is satisfied, if Q is declared by:

$$Q(y) : \quad y := cont(y) + 1 \quad ,$$

but it fails for:

$$Q(y) : \quad y := cont(x) + 1 \quad .$$

Finally the use of quantifiers for procedure identifiers can be illustrated with the aid of the formula

$$\forall a,R.(\forall c.\{\underline{true}\}\ R(c)\ \{cont(x) = c + a\}$$
$$\supset \forall b.\{cont(x) = b\}\ P(a,R)\ \{cont(x) = 2 * b + 1\})$$

which will be used in section 4 to prove a (slight variant of a) program constructed by E. Olderog (cf. [Old 81]). The precise meaning of the formula is not important here, but note that it describes the effect of the procedure P under a certain <u>semantical</u> condition for the procedure parameter R. This property of the formula is essential, because in E. Olderog's program in each recursion a new procedure is declared and inserted on parameter position. "Classical" Hoare-like calculi which - like ours - use a first order oracle, fail in proving this program, because they require some <u>syntactical</u> "similarity" for the different procedures occuring on a certain parameter position (cf. [Old 81], [Old 83]). Hence it was only proved in calculi based on a higher order assertion language like those of [Old 84], [DaJ 83], in which semantical properties of procedure parameters are implicitely expressed with the aid of predicate variables. But even these calculi fail, when the complexity of the program is increased by additionally declaring a new variable in each recursion (cf. section 4), because they can deal with "simple side effects" only (a notion introduced in [Lan 83]).

We conclude with some remarks about related approaches (in which also quantifiers are used on the level of Hoare formulas).

(a) In [CGH 83] a calculus is presented, which <u>only</u> works for procedures without global variables. Nevertheless this paper contains some ideas about a completeness proof, which can possibly be applied to our calculus.

(b) The proof system of [Hal 83] again differs from ours by using a higher order assertion language (which is not even precisely defined!).

(c) The most similar approach is Reynolds' specification logic ([Rey 81], [Rey 82]).

But there is one essential difference: Reynolds considers call-by-name as the basic mechanism for parameter passing. As a consequence he only uses identifiers which stand for integer <u>expressions</u> or variable <u>expressions</u> (and not for integers or variables). If now e is such an identifier, then a formula like

$$\forall e.\ \{cont(x) = e\}\ P(\ )\ \{cont(x) = e + 1\}$$

does not have the "desired" meaning, because in particular e can be replaced by cont(x), which leads to

$$\{\underline{true}\}\ P(\ )\ \{\underline{false}\}.$$

Hence it is again necessary (like in usual Hoare logic) to impose a restriction on e, e.g. that the value of e does not depend on the value of x (called "non-interference" in specification logic). This means that one important advantage of using quantifiers - which was illustrated above with the formula (**) - has been lost in specification logic, and that difficulties with procedure parameters reappear (cf. section 11 in [Rey 82]).

On the other hand it should be mentioned that specification logic is of course much more general than our approach, e.g. type coercion is considered and higher order predicates are used.

2. The programming language

We consider a fully typed ALGOL-like language in which arbitrary nesting of blocks and (mutually recursive) procedure declarations is possible. Procedures may have value-parameters (call-by-value), var-parameters (call-by-reference) and procedure parameters of all types. But note that self application is impossible, because every procedure has its (finite) type. Sharing (between parameters or between parameters and global variables) is allowed without restrictions. Global variables and procedures are handled by the static scope rule.

We begin with syntactical definitions.

First a set Type of *types* τ is defined by:

$\tau ::= \underline{\text{value}} \mid \underline{\text{var}} \mid \underline{\text{bool}} \mid \underline{\text{stat}} \mid (\tau_1 \to \tau_2) \mid \tau_1 \times \ldots \times \tau_n$.

stat is intended to be the type of statements and parameterless procedures; hence the subsets Proctype of *procedure types* and Partype of *parameter types* are defined by:

proctype $::= \underline{\text{stat}} \mid (\text{partype}_1 \times \ldots \times \text{partype}_n \to \underline{\text{stat}})$

partype $::= \underline{\text{value}} \mid \underline{\text{var}} \mid$ proctype

For every parameter type τ a set Id_τ of *identifiers* id of type τ is given, in particular:

- $\text{Id}_{\underline{\text{value}}}$ is the set of *value identifiers* a, b, c ...,
- $\text{Id}_{\underline{\text{var}}}$ is the set of *variable identifiers* x, y, z ...,
- $\bigcup_{\tau \in \text{Proctype}} \text{Id}_\tau$ is the set of *procedure identifiers* P, Q, R...

Additionally a signature Ω is needed, containing *function symbols* f and *predicate symbols* pr of various arities.

On this basis the constructs C of the programming language (and the type of each construct C, denoted type(C)) can be defined as follows.

Terms t:

$t ::= c \mid \text{cont}(x) \mid f(t_1,\ldots,t_n)$ where n is the arity of f.
(Remember the meaning of "cont".)
$\text{type}(t) = ((\underline{\text{var}} \to \underline{\text{value}}) \to \underline{\text{value}})$ for all terms t.

Formulas p(q,r,s):

$p ::= x = y \mid t_1 = t_2 \mid \neg p' \mid (p_1 \wedge p_2) \mid \forall c.p$
$\quad \mid pr(t_1,\ldots,t_n)$ where n is the arity of pr.

(x = y is intended to express sharing in contrast to cont(x) = cont(y); this is necessary, because formulas will be also used as assertions in Hoare formulas. The same argument applies to quantifiers, but note that they are restricted to value identifiers c.)
$\text{type}(p) = ((\underline{\text{var}} \to \underline{\text{value}}) \to \underline{\text{bool}})$ for all formulas p.

Statements St:

$St ::= x := t \mid \underline{\text{if}}\ p\ \underline{\text{then}}\ St_1\ \underline{\text{else}}\ St_2\ \underline{\text{fi}} \mid (St_1;St_2)$
$\quad \mid \underline{\text{begin}}\ \underline{\text{var}}\ x;St\ \underline{\text{end}} \mid \underline{\text{begin}}\ E;St\ \underline{\text{end}}$
$\quad \mid \text{Proc}(par_1,\ldots,par_n)$ where $par_1,\ldots,par_n$ must be of "adequate" type.

(Of course the parantheses in $(St_1;St_2)$ are omitted if possible, otherwise they are replaced by "<u>begin</u>...<u>end</u>".)
$\text{type}(St) = \underline{\text{stat}}$ for all statements St.

Procedures Proc:

Proc ::= P | Pb
type(Proc) is inherited from P or Pb.

Parameters par:

par ::= x | t | Proc
Again the type is inherited.

Procedure bodies Pb:

$Pb ::= St \mid \underline{\lambda}\ id_1,\ldots,id_n.St$ where $id_1,\ldots,id_n$ are different.
In the first case type(Pb) = <u>stat</u>, in the second case
$\text{type}(Pb) = (\tau_1 \times \ldots \times \tau_n \to \underline{\text{stat}})$ where τ_i is the type of id_i.

Procedure declarations E:

$E ::= P_1 \Leftarrow Pb_1;\ldots;P_m \Leftarrow Pb_m$ where $P_1,\ldots,P_m$ are different and $\mathrm{type}(P_i) = \mathrm{type}(Pb_i)$.

$\mathrm{type}(E) = \mathrm{type}(P_1) \times\ldots\times \mathrm{type}(P_m)$; $P_1,\ldots,P_m$ are called the procedure identifiers *declared* by E, the set (or the tuple) of these identifiers is denoted decl(E).

This concludes the syntactic definitions.
Note in particular that

- complete procedure bodies may occur in procedure calls (on "call position" as well as on parameter position),
- procedure declarations have the form $P \Leftarrow \underline{\lambda}\, id_1,\ldots,id_n.St$ instead of $P(id_1,\ldots,id_n) : St$.

These conventions are necessary for defining a substitution of procedure identifiers by procedure bodies.

The semantics of our language is defined in a purely denotational style, i.e. without any operational concepts like - say - the copy rules of [Old 81].

As usual the basis of the semantics definition is an interpretation $I = (D, I_o)$ where D is a nonempty set of *datas* ($\delta \in D$) and I_o assigns functions and predicates on D to the symbols of Ω. Additionally an infinite set Adr($\alpha \in$ Adr) of *addresses* (storage locations) is assumed to be available. A (total) function σ : Adr $\to$ D is then called a (storage) *state*, the set of all states is denoted Σ, and relations $\rho \subseteq \Sigma \times \Sigma$ are called (nondeterministic) *state transformations*.

The semantical domains are partial orders (po's) which do not necessarily contain bottom elements. Those which have one (denoted $\perp$) are called *strict* (due to [GTW^2 77]); those which are ordered by the equality are called *trivial*. If D and E are partial orders, then $D \times E$ denotes their cartesian product with the componentwise defined ordering and if additionally M is a set, then $(M \to E)$ denotes the set of functions from M to E with the argumentwise defined ordering.

With these notations a semantical domain D_τ is defined for each type τ, such that:

(i) $D_{\underline{value}} = D$,

(ii) $D_{\underline{var}} = Adr$,

(iii) $D_{\underline{bool}} = Bool = \{true, false\}$,

(each made into a trivial po by the equality);

(iv) $D_{\underline{stat}}$ is an "adequate" set of state transformations, containing in particular the empty set $\emptyset \subseteq \Sigma \times \Sigma$, (made into a strict po by the subset relation "$\subseteq$");

(v) $D_{\tau_1 \times \ldots \times \tau_n} = D_{\tau_1} \times \ldots \times D_{\tau_n}$,

(vi) $D_{(\tau_1 \to \tau_2)}$ is an "adequate" subset of $(D_{\tau_1} \to D_{\tau_2})$,

(each made into a po by the induced ordering).

Note that D_τ is strict for all procedure types τ, because $D_{\underline{stat}}$ is strict.

The precise definition of $D_{\underline{stat}}$ and $D_{(\tau_1 \to \tau_2)}$ is left open here. We only give some informal remarks:
The crucial point of the semantics definition is the connection of local variables with global (i.e. free) procedures. Consider e.g. the block

<u>begin</u> <u>var</u> x; P(x) <u>end</u>.

We want to define the semantics of the variable declaration by allocating to the identifier x a "new" address, which is not "global" for P, i.e. which is "not accessed by P itself". But in our purely denotational framework P is interpreted as a function $f \in D_{(\underline{var} \to \underline{stat})}$, hence it is necessary to define the set of addresses "accessed" by such a function, i.e. to define the notion of "access" on the <u>semantical</u> level. A precise solution of this problem can be found in [HMT 83], here we only want to present the main idea.
There are three kinds of access (of a procedure to a variable):

- access by <u>writing</u>: the contents of the variable is (possibly) changed by the (nondeterministic) procedure;
- access by <u>reading</u>: the initial contents of the variable has an influence on the output of the procedure;
- access by a <u>sharing effect</u>: This can be illustrated with the aid of an example. Let P be declared by

$$P \Leftarrow \underline{\lambda} y.\underline{\text{begin}}\ y := \text{cont}(x) + 1;\ \underline{\text{if}}\ \text{cont}(x) = \text{cont}(y)\ \underline{\text{then}}\ y := 1\ \underline{\text{else}}\ y := 2\ \underline{\text{fi}}\ \underline{\text{end}}.$$

Then there is neither a writing nor a reading access of P to x, but nevertheless x has a certain (semantical) influence on P: While the call P(x) sets its parameter x to 1, each other call P(y) sets its parameter y to 2.

It is not necessary to distinguish exactly between these three kinds of access, but we are only interested in the following notions and facts:
The set Glob(f) of *global addresses* of f contains all addresses which are accessed by f (by writing, reading or a sharing effect). The set Out(f) of *output addresses* of f only contains those elements of Glob(f) which are accessed by writing. The definition of each semantical domain D_τ ($\tau \in$ Proctype) guarantees that Glob(f) (and hence Out(f)) is finite for every $f \in D_\tau$. This makes it possible to select a "new" address in the semantics definition of variable declarations.

We want to present the most interesting clauses of this semantics definition. For this purpose we first need the following definition.
The set Env of *environments* ε is defined by

$$\text{Env} = \prod_{\tau \in \text{Partype}} (\text{Id}_\tau \to D_\tau),$$

i.e. every environment ε is a family of mappings

$$\varepsilon_\tau : \text{Id}_\tau \to D_\tau.$$

The *meaning* of each syntactical construct C of type τ is then defined as a function $M(C): \text{Env} \to D_\tau$.

Note in particular that (due to our definition of types):

- $M(t)(\varepsilon) : \Sigma \to D$ for every term t,
- $M(p)(\varepsilon) : \Sigma \to Bool$ for every formula p,
- $M(St)(\varepsilon) \subseteq \Sigma \times \Sigma$ for every statement St.

The most interesting clauses of the definition of M are:

(i) <u>block with variable declaration:</u>

$M(\underline{begin}\ \underline{var}\ x;St\ \underline{end})(\varepsilon)$

$= \{(\sigma[\alpha/\delta], \sigma'[\alpha/\delta]) \mid (\sigma,\sigma') \in M(\underline{\lambda}x.St)(\varepsilon)(\alpha)\}$ [1]

where α is an arbitrary address, not occuring in $Glob(M(\underline{\lambda}x.St)(\varepsilon))$.

Intuitively this definition means that α is a "new" address and that the block statement is executed in three steps:

- the initial contents δ of α is replaced by a random value $\sigma(\alpha)$,
- $M(\underline{\lambda}x.St)(\varepsilon)(\alpha) = M(St)(\varepsilon[x/\alpha])$ [1] is executed (transforming σ to σ'),
- the initial contents δ of α is restaured.

This careful definition of the variable declaration semantics guarantees that

- the semantics is indeed independent of the particular choice of α,
- α is not accessed (in the sense explained above) by the state transformation $M(\underline{begin}\ \underline{var}\ x;St\ \underline{end})(\varepsilon)$.

(ii) <u>block with procedure declaration:</u>

$M(\underline{begin}\ E;St\ \underline{end})(\varepsilon) = M(St)(\varepsilon[\bar{P}/M(E)(\varepsilon)])$
where $\bar{P} = decl(E)$

---

[1] As usual the variant $f[m/n]$ of a function $f : M \to N$ is defined by $f[m/n](m) = n$
$f[m/n](m') = f(m')$ for all $m' \neq m$.

This definition says that the procedure declaration E is evaluated (to the least fixpoint of a functional, cf. next clause) "at declaration time" and that the resulting objects (of procedure type) are bound to the procedure identifiers $\bar{P}$ (in order to use them in a procedure call). This shows that our programming language works with static scope of variable and procedure identifiers.

(iii) procedure declaration:

Let E be the declaration $P_1 \Leftarrow Pb_1;\ldots;P_m \Leftarrow Pb_m$ and let τ_i be the type of P_i. Then $M(E)(\varepsilon)$ is defined as the least fixpoint of the function

$$\Phi_{E,\varepsilon} : D_{\tau_1} \times\ldots\times D_{\tau_m} \to D_{\tau_1} \times\ldots\times D_{\tau_m}$$

$$(\eta_1,\ldots,\eta_m) \to (M(Pb_1)(\varepsilon[P_1/\eta_1] \ldots [P_m/\eta_m]),$$
$$\vdots$$
$$M(Pb_m)(\varepsilon[P_1/\eta_1] \ldots [P_m/\eta_m])).$$

Unfortunately our semantics does not fit into the classical framework of Scott's theory: A sequence of elements $f_n \in D_\tau$ ($\tau \in$ Proctype) for which every set $Glob(f_n)$ is finite, may have a least upper bound f with infinitely many global addresses. Hence the partial orders D_τ are not complete and - moreover - the semantics of the variable declaration leads to functions which are not continuous. These (technical but difficult) problems are discussed and solved in [HMT 83] by using a "refined" version of Scott's theory.

The connection between [HMT 83] and our approach is as follows: Our set Glob(f) corresponds to their "support of f", apart from one (small but serious) difference. We have separately defined a set Out(f) of output addesses, e.g. for each $f \in D_{(\underline{stat}\to\underline{stat})}$: $Out(f) = \bigcup_{\rho\in D_{\underline{stat}}} (Out(f(\rho))\setminus Out(\rho))$

(i.e. α is called an output address of f if its contents is changed by a call $f(\rho)$ without the aid of ρ). In [HMT 83] this set is not defined and its finiteness is not required.

On the other hand this set plays an important role in our proof system, so that this difference of the two semantical approaches has serious consequences (cf. example (iv) of section 3, the definition of the formula strange(x, P), the variable declaration axiom and example (ii) of section 4).

3. The logic

The basic objects of our partial correctness logic are (classical) Hoare formulas. Other formulas are constructed from them by two kinds of operators, namely

- the usual logical operators $\neg$, $\wedge$, $\forall$ ($\vee$, $\supset$, $\equiv$, $\exists$ are considered as abbreviations);
- a *substitution operator* <E> for every procedure declaration E.

More precisely we define the set of *generalized Hoare formulas* h by:

$h ::= \{p\}\ St\ \{q\} \mid \neg h' \mid (h_1 \wedge h_2) \mid \forall id.h' \mid \langle E\rangle h'.$

In order to define the semantics of these formulas, a formal definition of partial correctness is needed:
A state transformation $\rho \subseteq \Sigma \times \Sigma$ is called *partially correct* with respect to the predicates $\pi, \pi' : \Sigma \rightarrow$ Bool, if $\pi(\sigma) =$ true implies $\pi'(\sigma') =$ true for all pairs $(\sigma,\sigma') \in \rho$.

Now a meaning $M(h) : Env \rightarrow$ Bool can be assigned to every generalized Hoare formula h by:

(i) $M(\{p\}\ St\ \{q\})(\varepsilon) =$ true
$\Leftrightarrow M(St)(\varepsilon)$ is partially correct with respect to $M(p)(\varepsilon)$ and $M(q)(\varepsilon)$
(remember that $M(p)(\varepsilon)$ and $M(q)(\varepsilon)$ are predicates);

(ii) $M(\neg h)(\varepsilon) =$ true $\Leftrightarrow M(h)(\varepsilon) =$ false;

(iii) $M(h_1 \wedge h_2)(\varepsilon) =$ true $\Leftrightarrow M(h_1)(\varepsilon) =$ true and $M(h_2)(\varepsilon) =$ true;

(iv) $M(\forall id.h)(\varepsilon) =$ true $\Leftrightarrow M(h)(\varepsilon[\![id/\eta]\!]) =$ true for all $\eta \in D_\tau$,
where τ is the type of id;

(v) $M(\langle E\rangle h)(\varepsilon) =$ true $\Leftrightarrow M(h)(\varepsilon[\bar{P}/M(E)(\varepsilon)]) =$ true,
where $\bar{P} = decl(E)$
(i.e. <E>h means that h is true "after" the declaration E).

Some comments are necessary:

- Note that in our logic the Hoare formulas are evaluated and combined (with the aid of logical operators) on the level of environments, and not (like in dynamic logic, cf. [Har 79]) on the level of states.

- Recall that $M(h)$ (like all semantical definitions) depends on the underlying interpretation I of the signature Ω, i.e. (in a more precise notation) it is a function $M_I(h) : Env_I \rightarrow Bool$.

With this last notation we define: h is *valid* in I (notation: $\models_I h$) if $M_I(h)(\varepsilon) = true$ for all $\varepsilon \in Env_I$.

We now present some examples illustrating the use of our logic. It is assumed that the interpretation I assigns the usual meaning to the symbols 0, 1, ..., +, *, ... In all examples a detailed interpretation of the formulas is left to the reader.

(i) Let E be the declaration

$P \Leftarrow$ if $cont(x) = c$ then $x := cont(x)+1$ else $x := cont(x)+2$ fi

and let h be the formula

$\{cont(x) = c\}\ P(\ )\ \{cont(x) = c + 1\}$.

Then $\models_I \langle E\rangle h$ holds, but $\langle E\rangle \forall c.h$ is not valid in I (as already indicated in section 1).

On the other hand the declaration E' defined by

$P \Leftarrow x := cont(x) + 1$ yields $\models_I \langle E'\rangle \forall c.h$.

(ii) Let h be as in (i) and let h' be the formula

$\{cont(x) = 1\}\ P(\ )\ \{cont(x) = 2\}$.

Then $\forall c.h \supset h'$ is valid in I, but $h \supset h'$ is not. This possibility of substituting (only) quantified identifiers indicates the predicate logical character of our formulas, which will also come out in the axioms and rules of the calculus (in section 4). Moreover - together with (i) - we get a first hint how to accomplish (formal) stepwise refinement proofs: $\langle E'\rangle \forall c.h$ means that the procedure P declared by E' satisfies the semantical property $\forall c.h$. The formula $\forall c.h \supset h'$ says that this (general) property implies the (special) instance h', which is possibly needed for the proof of an "abstract program" calling the procedure P. Hence it must be possible to conclude

<E'>h' from these two formulas; this will be accomplished by the "stepwise refinement axiom" of section 4.

(iii) The following example illustrates the use of quantified variable identifiers, which are particularly interesting in connection with sharing effects.
Let E be the declaration

```
P ⇐ λy. begin y := cont(x) + 1;
              if cont(x) = cont(y) then y := 1 else y := 2 fi
        end ,
```

let h be the formula {true} P(x) {cont(x) = 1}
and h' the formula {true} P(y) {cont(y) = 2}.
Then <E>h is valid in I (because of sharing), but <E>∀x.h ist not. Moreover <E>h' is not valid in I, because it fails in the case of sharing, nor is <E>∀y.h'.
A valid formula describing the non sharing case is:

$$<E>\ \forall y.\ \ \{\neg y = x\}\ P(y)\ \{cont(y) = 2\}.$$

Substituting x for y in this formula and applying the stepwise refinement lemma mentioned in example (ii) yields:

$$<E>\ \{\neg x = x\}\ \ P(x)\ \{cont(x) = 2\},$$

which is valid in I (but meaningless).
This treatment of variable identifiers (which again shows the predicate logical character of our formulas) is possible, because variable identifiers stand for addresses and in particular different identifiers can denote the same address. In many other Hoare-like svstems such an unrestricted substitution of variable identifiers is not allowed, because the underlying semantics is defined without the aid of addresses (cf. [Old 81]).

(iv) The last example illustrates the difficulties which can arise from the connection of local variables and global procedures.
Let St be the statement

```
begin var x; x := 1; begin E; P(R); R( ) end end
```

where E is the declaration R ⇐ y := cont(x).
We want to prove that $\models_I$ {true} St {cont(y) = 1}. The argumentation remains a bit informal because of our vague definition of the semantics.

First note that P is global w.r.t. St, hence it can be assumed that (the address assigned to) x is not accessed by (the function assigned to) P. As moreover R does not access x by writing (but only by reading), also P(R) cannot change the contents of x. Because x initially contains 1, this value is finally assigned to y by the call of R, and this proves the validity of the formula.

Note that the argumentation of example (iv) was only possible because the set of output addresses of P is defined seperately (as mentioned at the end of section 2). If this is not the case (like in [HMT 83], [Hal 83]) then the validity of the above-mentioned formula is (at least) questionable.
Note moreover that this validity is not only a matter of taste. If St occurs within a procedure Q where a new procedure P is created in each recursion and inserted on parameter position of Q (like in example (iii) of section 4) then the formula might be a step in the proof of a "complete program" (without global procedures).

We conclude this section with some preparations for the calculus:

(i) First we need a definition of free (resp. bound) occurence and of substitution for the constructs C of our programming language and the formulas h of our logic. The binding mechanisms are: quantifiers, $\underline{\lambda}$-abstraction (i.e. formal parameters), variable declarations and procedure declarations. The sets free (C) and free(h) of identifiers which are not bound by one of these mechanisms are defined inductively, e.g.:

$\text{free}(P_1 \Leftarrow Pb_1;\ldots;P_m \Leftarrow Pb_m) = \biguplus_{i=1}^{m} \text{free}(Pb_i) \setminus \{P_1,\ldots,P_m\}$,

$\text{free}(\underline{\text{begin}}\ E;St\ \underline{\text{end}}) = (\text{free}(E) \cup \text{free}(St)) \setminus \text{decl}(E)$.

Now it is possible to define a *substitution* of

- a value identifier c by a term t,
- a variable identifier x by a variable identifier y,

- a procedure identifier P by a procedure Proc of the same type.

The substitutions are defined "as usual", i.e. bound identifiers must be possibly renamed, in order to avoid new bindings.

The formulas which are obtained from h are denoted h_c^t, h_x^y and h_P^{Proc} respectively, similarly for constructs C.

As usual a *substitution theorem* holds:
- $M(h_c^t)(\varepsilon) = M(h)(\varepsilon[c/M(t)(\varepsilon)])$ if t is variable free,
- $M(h_x^y)(\varepsilon) = M(h)(\varepsilon[x/\varepsilon(y)])$,
- $M(h_P^{Proc})(\varepsilon) = M(h)(\varepsilon[P/M(Proc)(\varepsilon)])$.

Again everything generalizes to constructs C and to a simultaneous substitution of two or more identifiers.

The theorem shows that the substitution operator <E> can be considered as an abbreviation: If decl(E) = $\{P_1,\ldots,P_m\}$, then <E>h is obtained from h by substituting for each P_i the procedure body

$$\underline{\lambda} id_1,\ldots,id_n.\ \underline{\text{begin}}\ E;P_i(id_1,\ldots,id_n)\ \underline{\text{end}}.$$

This will be the point of view in section 4.

(ii) For our variable declaration axiom we need a formula strange(x,P) such that

$$M(\text{strange}(x,P))(\varepsilon) = \text{true} \Leftrightarrow \varepsilon(x) \notin Out(\varepsilon(P))$$

(cf. section 2)

It can be defined by induction on the type of P. For type (P) = $\underline{\text{stat}}$ it is the formula:

$$\forall c.\{\text{cont}(x) = c\}\ P(\ )\ \{\text{cont}(x) = c\}$$

and for type (P) = $(\tau_1 \times\ldots\times\tau_n \to \tau)$:

$$\forall id_1,\ldots,id_n.(\bigwedge_{\tau_i \in \text{Proctype}} \text{strange}(x,id_i)$$
$$\supset \forall c.\{\bigwedge_{\tau_i = \underline{\text{var}}} \neg x{=}id_i \wedge \text{cont}(x){=}c\}\ P(id_1,\ldots,id_n)\{\text{cont}(x){=}c\}).$$

Note that strange(x,P) is just an abbreviation for a generalized Hoare formula. The other authors who need a similar axiom or rule ([Hal 83], [Rey 82]) have introduced a <u>new</u> formula in order to express the (stronger) property $\varepsilon(x) \notin \mathrm{Glob}(\varepsilon(P))$ (or a similar condition). We conjucture that our variable declaration axiom is still strong enough in spite of the weaker assumption.

(iii) In order to formulate the fixpoint induction principle we must characterize a subset of generalized Hoare formulas, which express admissible predicates (cf. [Man 74]). More precisely for every finite set $\{P_1,\ldots,P_m\}$ of procedure identifiers a set of so-called *specifications* spec for $P_1,\ldots,P_m$ is defined syntactically which all have two semantical properties:

For every environment ε

- $M(\mathrm{spec})(\varepsilon[P_1/\bot_1]\ldots[P_m/\bot_m]) = \mathrm{true}$ and
- the predicate Φ_ε defined by
  $\Phi_\varepsilon(f_1,\ldots,f_m) = M(\mathrm{spec})(\varepsilon[P_1/f_1]\ldots[P_m/f_m]$
  is admissible.

The precise syntactical definition is omitted here; a similar restriction is imposed on the formulas in Reynolds' axiom of recursion ([Rey 82]).

4. The proof system

Our proof system consists of three groups of axioms and rules.

I. *Logical* axioms and rules:
They are needed for "purely logical reasoning" on the level of generalized Hoare formulas.

(a) Tautology-rule:

$$\frac{h_1,\ldots,h_n}{h} \quad \text{if } (h_1 \wedge \ldots \wedge h_n) \supset h \text{ is a tautology.}$$

(b) Substitution-axioms:

(i) $\forall c.h \supset h^t_c$ if t is variable free.

(ii) $\forall x.h \supset h^y_x$

(iii) $\forall P.h \supset h^{Proc}_P$

(c) $(\forall)$-rule:

$$\frac{h \supset h'}{h \supset \forall id.h'} \quad \text{if } id \notin free(h).$$

II. Axioms and rules for *partial correctness*:
They are needed for manipulating the assertions p and q of a Hoare formula $\{p\}$ St $\{q\}$ without referring to the special structure of St.

(a) Invariance-axiom:
$\{p\}$ St $\{p\}$ if p does not contain the symbol "cont".

(b) $(\wedge)$-axiom:
$(\{p\}\ St\ \{q\} \wedge \{r\}\ St\ \{s\}) \supset \{p \wedge r\}\ St\ \{q \wedge s\}$

(c) $(\vee)$-axiom:
$(\{p\}\ St\ \{q\} \wedge \{r\}\ St\ \{s\}) \supset \{p \vee r\}\ St\ \{q \vee s\}$

(d) $(\forall)$-axiom:
$\forall c.\ \{p\}\ St\ \{q\} \supset \{\forall c.p\}\ St\ \{\forall c.q\}$ if $c \notin free(St)$

(e) $(\exists)$-axiom:
$\forall c.\{p\}\ St\ \{q\} \supset \{\exists c.p\}\ St\ \{\exists c.q\}$ if $c \notin free(St)$

(f) ($\supset$)-rule:

$$\frac{r \supset p \quad , \quad q \supset s}{\{p\}\ St\ \{q\} \supset \{r\}\ St\ \{s\}}$$

III. *Language specific* axioms and rules:
They are used to deal with the "operations" of the programming language like composition or recursion.

(a) (:=)-axioms:

(i) $\forall x,c.\ \{\underline{true}\}\ x := c\ \{cont(x) = c\}$

(ii) $\forall y,c.\ \{\neg y{=}x \wedge cont(y){=}c\}\ x{:=}t\ \{cont(y){=}c\}$

(b) (;)-axiom:

$(\{p\}\ St_1\ \{q\} \wedge \{q\}\ St_2\ \{r\}) \supset \{p\}\ St_1;St_2\ \{r\}$

(c) ($\underline{if}$)-axiom:

$(\{p \wedge r\}\ St_1\ \{q\} \wedge \{p \wedge \neg r\}\ St_2\ \{q\})$

$\supset \{p\}\ \underline{if}\ r\ \underline{then}\ St_1\ \underline{else}\ St_2\ \underline{fi}\ \{q\}$

(d) (PD)-axiom (for procedure declarations):

$\langle E\rangle\ \{p\}\ St\ \{q\} \supset \{p\}\ \underline{begin}\ E;St\ \underline{end}\ \{q\}$

(e) (VD)-axiom (for variable declarations):

$\forall x.((\bigwedge_{P\in free(St)} strange(x,P)) \supset \{p \wedge \neg x{=}y_1 \wedge \ldots \wedge \neg x{=}y_n\}\ St\ \{q\})$

$\supset \{p\}\ \underline{begin}\ \underline{var}\ x;St\ \underline{end}\ \{q\}$

if $x \notin free(p) \cup free(q) \cup \{y_1,\ldots,y_n\}$

(f) (FPI)-axiom (for fixpoint induction):

$\forall P_1,\ldots,P_m.(spec \supset spec^{Pb_1,\ldots,Pb_m}_{P_1,\ldots,P_m}) \supset \langle P_1 \Leftarrow Pb_1;\ldots;P_m \Leftarrow Pb_m\rangle spec$

if spec is a specification for $P_1,\ldots,P_m$ (cf. section 3).

(g) ($\underline{\lambda}$)-axiom:

$\{p\}\ St^{par_1,\ldots,par_n}_{id_1,\ldots,id_n}\ \{q\} \supset \{p\}\ \underline{\lambda}\ id_1,\ldots,id_n.St(par_1,\ldots,par_n)\ \{q\}$

if no par_i is a term which contains variables.

(h) call-by-value-axiom:

$\forall c.\{p \wedge t = c\}\ St\ \{q\} \supset \{p\}\ St^t_c\ \{q\}$

if $c \notin free(p) \cup free(t) \cup free(q)$

and St is an assignment or a procedure call without procedure bodies.

Instead of giving comments on the axioms and rules we want to illustrate their use with the aid of some examples. For this purpose we first present a derived axiom:

(SR)-axiom:

$$(\langle E\rangle spec \wedge \forall P_1,\ldots,P_m.(spec \supset h)) \supset \langle E\rangle h \quad \text{if } decl(E) = \{P_1,\ldots,P_m\}.$$

This axiom reflects the idea of stepwise refinement: If the formula h expresses partial correctness of an "abstract program", using the free procedure identifiers $P_1,\ldots,P_m$, then E<h> can be proved in two steps:

- h is proved under the assumption spec, which expresses certain semantical properties of the procedures;
- spec is proved for the procedures declared by E.

The derivation of this axiom is easy: Recall that the operator <E> is considered as a syntactical substitution; now apply the substitution axiom (iii) and the tautology rule.

We now present three derivations in the form of "derivation trees". We always concentrate on the most difficult "branches" of the tree; e.g. proofs for assignments are ommited at all. (Indeed proving the partial correctness of an assignment is tedious with the "pure" calculus; another derived axiom would be needed for reasonable derivations.)

We start with a procedure computing the factorial function (as a warming up example):

(i) Let E be the declaration $P \Leftarrow \underline{\lambda} x,a.St$ where St is the following statement:
$\underline{if}$ $a = 0$ $\underline{then}$ $x := 1$ $\underline{else}$ $P(x,a-1);\ x := cont(x) * a$ $\underline{fi}$,
and let spec be the specification
$\forall x,a.\ \{\underline{true}\}\ P(x,a)\ \{cont(x) = a!\}$.
Then the (valid) formula <E>spec can be derived as follows

(1) <E>spec

---- (FPI)-axiom, ($\forall$)-rule

(2) spec $\supset \forall x,a.$ {true} $\lambda x,a.St(x,a)$ {cont(x)=a!}

---- ($\forall$)-rule, (λ)-axiom

(3) spec $\supset$ {true} St {cont(x) = a!}

---- (if)-axiom, (;)-axiom, tautology-rule

(4) {true $\wedge$ a = 0} x := 1 {cont(x) = a!}

(5) spec $\supset$ {true $\wedge \neg$ a=0} P(x,a-1) {cont(x)=(a-1)! $\wedge \neg$ a=0}

(6) {cont(x)=(a-1)! $\wedge \neg$ a=0} x := cont(x)$*$a {cont(x)=a!}

We restrict our attention to the procedure call (5):

(5)

---- ($\wedge$)-axiom, tautology-rule

(7) spec $\supset$ {true} P(x,a-1) {cont(x) = (a-1)!}

(8) {$\neg$ a=0} P(x,a-1) {$\neg$ a=0}

(7) is an instance of the substitution axiom (i), and (8) is an instance of the invariance axiom. □

The second example illustrates the connection between global procedures and local variables. It was already considered from the semantical point of view in section 3.

(ii) Let St be the statement

begin var x; x := 1; begin E; P(R); R( ) end

where E is the declaration R $\Leftarrow$ y := cont(x).

We want to prove {true} St {cont(y) = 1}.

For this purpose we need the following specification spec for R:

$\forall$c. {cont(x)=c} R( ) {cont(x)=c} $\wedge$ $\forall$b. {cont(x)=b} R( ) {cont(y)=b}

and the formula strange(x,**P**)

$\forall$ R. ($\forall$c. {cont(x)=c} R( ) {cont(x)=c} $\supset$ $\forall$c. {cont(x)=c} P(R) {cont(x)=c})

Then we get the following derivation

(1) {true} St {cont(y) = 1}

---- (VD)-axiom, ($\forall$)-rule

(2) strange(x,P) $\supset$ {true} x:=1; begin E; P(R); R( ) end {cont(y)=1}

---- (;)-axiom, tautology rule

(3) {true} x:=1 {cont(x) = 1}

(4) strange(x,P) $\supset$ {cont(x)=1} begin E; P(R); R() end {cont(y)=1}

(3) is trivial.

(4)

---- (PD)-axiom

(5) $\text{strange}(x,P) \supset \langle E\rangle \{\text{cont}(x)=1\}\ P(R);R(\ )\ \{\text{cont}(y)=1\}$

---- (SR)-lemma, ($\forall$)-rule, tautology rule

(6) $\langle E\rangle \text{spec}$

(7) $\text{strange}(x,P) \supset (\text{spec} \supset \{\text{cont}(x)=1\}\ P(R);R()\ \{\text{cont}(y)=1\})$

The derivation of (6) is routine, we concentrate on (7):

(7)

---- (;)-axiom, tautology rule

(8) $(\text{strange}(x,P) \wedge \text{spec}) \supset \{\text{cont}(x)=1\}\ P(R)\ \{\text{cont}(x)=1\}$

(9) $\text{spec} \supset \{\text{cont}(x) = 1\}\ R(\ )\ \{\text{cont}(y) = 1\}$

(8) and (9) can be derived with the aid of the tautology rule and the substitution axioms. □

The last example is a (slight variant) of a procedure constructed by E. Olderog in order to illustrate the limits of his own calculus (in [Old 81]).

(iii) Let E be the declaration $P \Leftarrow Pb$,
where Pb is the procedure body
$\underline{\lambda}\ a,R.\ \underline{\text{begin}}\ Q \Leftarrow \underline{\lambda} c.R(c+1);St\ \underline{\text{end}}$
and St is the statement
$\underline{\text{if}}\ a < \text{cont}(x)\ \underline{\text{then}}\ P(a+1,Q)\ \underline{\text{else}}\ R(a+1)\ \underline{\text{fi}}$.

We want to prove $\langle E\rangle h$, where h is the formula
$\forall b.\{\text{cont}(x)=b\}\ P(0,\underline{\lambda} c.x:=c)\ \{\text{cont}(x)=2*b+1\}$.
For this purpose we choose the following specification spec for P:
$\forall a,R.(\forall c.\{\underline{\text{true}}\}\ R(c)\ \{\text{cont}(x) = c + a\}$
$\supset \forall b.\ \{\text{cont}(x)=b \wedge a \leq b\}\ P(a,R)\ \{\text{cont}(x)=2*b+1\})$

Then we get the following derivation

(1) $\langle E\rangle h$

---- (SR)-axiom, ($\forall$)-rule

(2) $\langle E\rangle \text{spec}$

(3) $\text{spec} \supset h$

The derivation of (3) is relatively easy: The main step is the substitution of a by 0 and of R by $\underline{\lambda}c.x:=c$. We concentrate on (2):

(2) $\langle E\rangle \mathrm{spec}$

---- (FPI)-axiom, ($\forall$)-rule

(4) $\mathrm{spec} \supset (\forall c.\{\underline{\mathrm{true}}\}\ R(c)\ \{\mathrm{cont}(x) = c + a\}$
$\qquad \supset \forall b.\{\mathrm{cont}(x)=b \wedge a \leq b\}\ Pb(a,R)\ \{\mathrm{cont}(x) = 2*b+1\})$

---- tautology rule, ($\forall$)-rule, ($\underline{\lambda}$)-axiom

(5) $(\mathrm{spec} \wedge \forall c.\{\underline{\mathrm{true}}\}\ R(c)\ \{\mathrm{cont}(x) = c + a\})$
$\qquad \supset \{\mathrm{cont}(x) = b \wedge a \leq b\}\ \underline{\mathrm{begin}}\ Q \Leftarrow \underline{\lambda}c.R(c+1);St\ \underline{\mathrm{end}}$
$\qquad\qquad \{\mathrm{cont}(x) = 2 * b + 1\}$

---- (PD)-axiom, (SR)-axiom, ($\forall$)-rule

(6) $(\mathrm{spec} \wedge \forall c.\ \{\underline{\mathrm{true}}\}\ R(c)\ \{\mathrm{cont}(x) = c + a\})$
$\qquad \supset \langle Q \Leftarrow \underline{\lambda}c.\ R(c+1)\rangle\ \ \forall c.\ \{\underline{\mathrm{true}}\}\ Q(c)\ \{\mathrm{cont}(x) = c+a+1\}$

(7) $(\mathrm{spec} \wedge \forall c.\{\underline{\mathrm{true}}\}\ R(c)\ \{\mathrm{cont}(x) = c + a\})$
$\qquad \supset (\forall c.\ \{\underline{\mathrm{true}}\}\ Q(c)\ \{\mathrm{cont}(x) = c + a + 1\}$
$\qquad\qquad \supset \{\mathrm{cont}(x) = b \wedge a \leq b\}\ St\ \{\mathrm{cont}(x)=2 * b + 1\})$

(6) is routine. As far as (7) is concerned, note that:

- spec together with the specification of Q is sufficient to deal with the *then*-part of the statement St (where the main step is a substitution of R by Q and of a by a + 1)
- the specification of R is sufficient to deal with the *else*-part (because a = cont(x) = b in this case)

$\square$

As mentioned in the introduction, example (iii) has been proved with the aid of calculi using higher order oracles (in [Old 84], [DaJ 83]). In order to obtain a program which even exceeds the power of these calculi, just replace the declaration of Q by:

$\underline{\mathrm{var}}\ y;\ Q \Leftarrow \underline{\lambda}c.\underline{\mathrm{begin}}\ y:=c;R(\mathrm{cont}(y)+1)\ \underline{\mathrm{end}}$.

This does not change the semantics of P, and in our calculus the formula h of example (iii) can still be proved. (The variable declaration is just removed with the aid of the (VD)-axiom, the information strange(x,P) is not needed.)

But from a syntactical point of view the new program contains a "serious side effect", because y is global in the body of Q and local in the body of P (cf. [Lan 83]). Hence in each recursion a new procedure Q is generated (and inserted on parameter position), which has one additional global variable. This phenomenon does not fit into the framework of [Old 84] or [DaJ 83].

5. Conclusion

As usual the question arises now, if our proof system is sound and in some sense complete. The soundness can be proved without difficulties. Completeness - even relative completeness in the sense of [Coo 78] - cannot be expected for the partial correctness theory of the full programming language. This was proved in [Cla 79] by showing that for such a powerful language the divergence problem (i.e. the question if a program does not terminate for any input) is unsolvable even for finite interpretations. Hence we must look for less powerful sublanguages, in order to get completeness results.

Several adequate sublanguages can be found in [Old 81]: With our calculus we can simulate so-called standard proofs in E. Olderog's system $H(C_{60})$, provided that all procedures have finite mode (i.e. self application is not allowed). But of course this result does not exploit the power of our logic and calculus. A more interesting candidate for a sublanguage would be Clarke's L4 (cf. [Cla 79], [DaJ 83]), in which procedures with global variables are not allowed, a restriction which makes the divergence problem solvable for finite interpretations. A first hint how to obtain a completeness proof for a similar calculus can be found in [CGH 83], and we hope that their idea can be applied to our proof system.

In spite of this lack of reliable completeness results we hope that this paper has convinced the reader that our proof system is natural and powerful.

Acknowledgement: I am grateful to E. Olderog for his comments on an earlier version of this paper.

References:

[Cla 79] Clarke, E.M.: Programming language constructs for which it is impossible to obtain good Hoare-like axioms. JACM 26, 129 - 147, 1979

[CGH 83] Clarke, E.M., German, S.M. and Halpern, J.Y.: Reasoning about procedures as parameters. Proc. of the CMU Workshop on Logics of Programs, LNCS 164, 206-220, 1983

[Coo 78] Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM Journ. on Comp. 7, 70 - 90, 1978

[dBa 80] de Bakker, J.W.: Mathematical theory of program correctness. Prentice-Hall, 1980

[DaJ 83] Damm, W. and Josko, B.: A sound and relatively* complete Hoare-logic for a language with higher type procedures. Acta Informatica 20, 59 - 102, 1983

[$GTW^2$77] Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.B.: Initial algebra semantics and continuous algebras. JACM 24, 68 - 95, 1977

[Hal 83] Halpern, J.Y.: A good Hoare axiom system for an ALGOL-like language. Proc. 11th POPL Conf., 262 - 271, 1983

[Har 79] Harel, D.: First order dynamic logic, LNCS 68, Springer-Verlag, 1979

[HMT 83] Halpern, J.Y., Meyer, A.R. and Trakhtenbrot, B.A.: The semantics of local storage, or what makes the free-list free? Proc. 11th POPL Conf., 245 - 257, 1983

[Lan 83] Langmaack, H.: Aspects of programs with finite modes. Proc. of the FCT-Conference, LNCS 158, 241 - 254, 1983

[Man 74] Manna,Z.: Mathematical theory of computation. McGraw-Hill, 1974

[Old 81] Olderog, E.R.: Sound and complete Hoare-like calculi based on copy rules. Acta Informatica 16, 161 - 197, 1981

[Old 83] Olderog, E.R.: A characterization of Hoare's logic for programs with PASCAL-like procedures. Proc. 15th ACM Symp. on Theory of Computing, 320 - 329, 1983

[Old 84] Olderog, E.R.: Correctness of programs with PASCAL-like procedures without global variables. TCS 30, 49 - 90, 1984

[Rey 81] Reynolds,J.C.: The craft of programming. Prentice-Hall International Series in Comp. Sc. 1981

[Rey 82] Reynolds, J.C.: Idealized ALGOL. Tools and notions for program construction. D. Néel ed., Cambridge University Press, 121 - 161, 1982

[Sie 81] Sieber, K.: A new Hoare-calculus for programs with recursive parameterless procedures. Bericht A 81/02, Universität Saarbrücken, 1981