Hoare Logics for Recursive Procedures and Unbounded Nondeterminism

Tobias Nipkow

Fakultät für Informatik, Technische Universität München
`http://www.in.tum.de/~nipkow/`

Abstract. This paper presents sound and complete Hoare logics for partial and total correctness of recursive parameterless procedures in the context of unbounded nondeterminism. For total correctness, the literature so far has either restricted recursive procedures to be deterministic or has studied unbounded nondeterminism only in conjunction with loops rather than procedures. We consider both single procedures and systems of mutually recursive procedures. All proofs have been checked with the theorem prover Isabelle/HOL.

1 Introduction

Hoare logic has been studied extensively since its inception [8], both for its theoretical interest and its practical relevance. Strangely enough, procedures have not been treated with adequate attention: many proof systems involving procedures are unsound, incomplete, or ignore completeness altogether (see [2, 21] for details). In particular the combination of procedures with (unbounded) nondeterminism was an open issue altogether.

Let us briefly review the history of Hoare logics for deterministic languages with procedures. The system proposed by Hoare [9] was later shown to be sound and complete by Olderog [21]. Apt [2] presents sound and complete systems both for partial correctness (following Gorelick [6]) and for total correctness (following and completing Sokołowski [25]). The one for total correctness is later found to be unsound by America and de Boer [1], who modify the system and give new soundness and completeness proofs. A new twist is added by Kleymann (né Schreiber) [24] who uses a little known consequence rule due to Morris [14] to subsume the three adaption rules by America and de Boer. In particular, he formalizes his work in the theorem prover LEGO [23]: this is the first time that a new Hoare logic is first proved sound and complete in a theorem prover.

We continue our earlier work on Hoare logic in Isabelle/HOL [17, 18] while taking advantage of Kleymann's technical advances. The main contribution of our paper is to simplify some aspects of Kleymann's proof system and, more importantly, to provide the first Hoare logics for partial and for total correctness of recursive procedures in the context of unbounded nondeterminism, both for single procedures and mutually recursive procedures. At this point we connect with the work by Apt [3] and Apt and Plotkin [4] on unbounded nondeterminism.

The main differences are that they use ordinals and we use well-founded relations, and that they do not consider procedures, thus avoiding the difficulties explained below which are at the heart of many unsound and incorrect proof systems in the literature.

1.1 The problem with procedures

Consider the following parameterless procedure which calls itself recursively:

```
proc = if i=0 then skip else i := i-1; CALL; i := i+1
```

A classic example of the subtle problems associated with reasoning about procedures is the proof that `i` is invariant: {`i=N`} `CALL` {`i=N`}. This is done by induction: we assume {`i=N`} `CALL` {`i=N`} and have to prove {`i=N`} `body` {`i=N`}, where `body` is the body of the procedure. The case `i=0` is trivial. Otherwise we have to show {`i=N`}`i:=i-1;CALL;i:=i+1`{`i=N`}, which can be reduced to {`i=N-1`} `CALL` {`i=N-1`}. But how can we deduce{`i=N-1`} `CALL` {`i=N-1`} from the induction hypothesis {`i=N`} `CALL` {`i=N`}? Clearly, we have to instantiate `N` in the induction hypothesis — after all `N` is arbitrary as it does not occur in the program. The problems with procedures are largely due to unsound or incomplete adaption rules. We follow the solution of Morris and Kleymann and adjust the value of auxiliary variables like `N` with the help of the consequence rule. In §4.4 we show how this example is handled with our rules.

1.2 The extensional approach

In modelling the assertion language, we follow the *extensional* approach where assertions are identified with functions from states to propositions. That is, we model only the semantics but not the syntax of assertions. This is common practice in the theorem proving literature (with the exception of [11], but they do not consider completeness) and can also be found in standard sematics texts [16]. Because our underlying logic is higher order, *expressiveness*, i.e. whether the assertion language is strong enough to express all intermediate predicates that may arise in a proof, is not much of an issue. Thus our completeness results do not automatically carry over to other logical systems, say first order arithmetic. The advantage of the extensional approach is that it separates reasoning about programs from expressiveness considerations — the latter can then be conducted in isolation for each assertion language. We discuss this further in §6.

1.3 Isabelle/HOL

Isabelle/HOL [19] is an interactive theorem prover for HOL, higher-order logic. The whole paper is generated directly from the Isabelle input files, which include the text as comments. That is, if you see a lemma or theorem, you can be sure its proof has been checked by Isabelle. Most of the syntax of HOL will be familiar to anybody with some background in functional programming and logic. We just highlight some of the nonstandard notation.

The space of total functions is denoted by the infix $\Rightarrow$. Other type constructors, e.g. *set*, are written postfix, i.e. follow their argument as in *state set*.

The syntax $[\![P;\ Q]\!] \Longrightarrow R$ should be read as an inference rule with the two premises P and Q and the conclusion R. Logically it is just a shorthand for $P \Longrightarrow Q \Longrightarrow R$. Note that semicolon will also denote sequential composition of programs, which should cause no major confusion. There are actually two implications $\longrightarrow$ and $\Longrightarrow$. The two mean the same thing, except that $\longrightarrow$ is HOL's "real" implication, whereas $\Longrightarrow$ comes from Isabelle's meta-logic and expresses inference rules. Thus $\Longrightarrow$ cannot appear inside a HOL formula. For the purpose of this paper the two may be identified. However, beware that $\longrightarrow$ binds more tightly than $\Longrightarrow$: in $\forall\, x.\ P \longrightarrow Q$ the $\forall\, x$ covers $P \longrightarrow Q$, whereas in $\forall\, x.\ P \Longrightarrow Q$ it covers only P.

Set comprehension is written $\{x.\ P\}$ rather than $\{x \mid P\}$ and is also available for tuples, e.g. $\{(x,\ y,\ z).\ P\}$.

2 Syntax and operational semantics

Everything is based on an unspecified type *state* of states. This could be a mapping from variables to values, but to keep things abstract we leave this open. The type *bexp* of boolean expressions is defined as an abbreviation:

types $bexp = state \Rightarrow bool$

This model of boolean expressions requires a few words of explanation. Type *bool* is HOL's predefined type of propositions. Thus all the usual logical connectives like $\wedge$ and $\vee$ are available. Instead of modelling the syntax of boolean expressions, we model their semantics. For example, if states are mappings from variables to values, the programming language expression `x != y` becomes $\lambda s.\ s\ x \neq s\ y$.

The syntax of our programming language is defined by a recursive datatype (not shown). Statements in this language are called *commands*. Command $Do\ f$, where f is of type $state \Rightarrow state\ set$, represents an atomic command that leads in one step from some state s to a new state $t \in f\ s$, or blocks if $f\ s$ is empty. Thus *Do* can represent many well-known constructs such as `skip` ($Do\ (\lambda s.\ \{s\})$), `abort` ($Do\ (\lambda s.\ \{\})$), and (random) assignment. This is the only source of nondeterminism, but other constructs, like a binary choice between commands, are easily simulated.

In addition we have sequential composition ($c1;\ c2$), conditional ($IF\ b\ THEN\ c1\ ELSE\ c2$), iteration ($WHILE\ b\ DO\ c$) and the procedure call command *CALL*. There is only one parameterless procedure in the program. Hence *CALL* does not even need to mention the procedure name. There is no separate syntax for procedure declarations. Instead we introduce a new constant

consts $body :: com$

that represents the body of the one procedure in the program. Since *body* is unspecified, this is completely generic.

The semantics of commands is defined operationally, by the simplest possible scheme, a so-called *evaluation* or *big-step* semantics. Execution is defined via triples of the form $s\ -c\rightarrow\ t$ which should be read as "execution of c starting in state s may terminate in state t". This allows for different kinds of nondeterminism: there may be other terminating executions $s\ -c\rightarrow\ u$ with $t \neq u$, there may be nonterminating computations, and there may be blocking computations. Nontermination and blocking is only discussed in the context of total correctness. Execution of commands is defined inductively in the standard fashion and requires no comments. See §1.3 for the notation.

$t \in f\ s \Longrightarrow s\ -Do\ f\rightarrow\ t$
$[\![s0\ -c1\rightarrow\ s1;\ s1\ -c2\rightarrow\ s2]\!] \Longrightarrow s0\ -c1;\ c2\rightarrow\ s2$
$[\![b\ s;\ s\ -c1\rightarrow\ t]\!] \Longrightarrow s\ -IF\ b\ THEN\ c1\ ELSE\ c2\rightarrow\ t$
$[\![\neg\ b\ s;\ s\ -c2\rightarrow\ t]\!] \Longrightarrow s\ -IF\ b\ THEN\ c1\ ELSE\ c2\rightarrow\ t$
$\neg\ b\ s \Longrightarrow s\ -WHILE\ b\ DO\ c\rightarrow\ s$
$[\![b\ s;\ s\ -c\rightarrow\ t;\ t\ -WHILE\ b\ DO\ c\rightarrow\ u]\!] \Longrightarrow s\ -WHILE\ b\ DO\ c\rightarrow\ u$
$s\ -body\rightarrow\ t \Longrightarrow s\ -CALL\rightarrow\ t$

This semantics turns out not to be fine-grained enough. The soundness proof for the partial correctness Hoare logic below proceeds by induction on the call depth during execution. To make this work we define a second semantics $s\ -c-n\rightarrow\ t$ which expresses that the execution uses at most n nested procedure invocations, where n is a natural number. The rules are straightforward: n is just passed around, except for procedure calls, where it is decremented (*Suc* n is $n + 1$):

$t \in f\ s \Longrightarrow s\ -Do\ f-n\rightarrow\ t$
$[\![s0\ -c1-n\rightarrow\ s1;\ s1\ -c2-n\rightarrow\ s2]\!] \Longrightarrow s0\ -c1;\ c2-n\rightarrow\ s2$
$[\![b\ s;\ s\ -c1-n\rightarrow\ t]\!] \Longrightarrow s\ -IF\ b\ THEN\ c1\ ELSE\ c2-n\rightarrow\ t$
$[\![\neg\ b\ s;\ s\ -c2-n\rightarrow\ t]\!] \Longrightarrow s\ -IF\ b\ THEN\ c1\ ELSE\ c2-n\rightarrow\ t$
$\neg\ b\ s \Longrightarrow s\ -WHILE\ b\ DO\ c-n\rightarrow\ s$
$[\![b\ s;\ s\ -c-n\rightarrow\ t;\ t\ -WHILE\ b\ DO\ c-n\rightarrow\ u]\!] \Longrightarrow s\ -WHILE\ b\ DO\ c-n\rightarrow\ u$
$s\ -body-n\rightarrow\ t \Longrightarrow s\ -CALL-Suc\ n\rightarrow\ t$

By induction on $s\ -c-m\rightarrow\ t$ we show monotonicity w.r.t. the call depth:

lemma $s\ -c-m\rightarrow\ t \Longrightarrow \forall n.\ m \le n \longrightarrow s\ -c-n\rightarrow\ t$

With the help of this lemma we prove the expected relationship between the two semantics:

lemma *exec-iff-execn*: $(s\ -c\rightarrow\ t) = (\exists n.\ s\ -c-n\rightarrow\ t)$

Both directions are proved separately by induction on the operational semantics.

3 Hoare logic for partial correctness

As motivated in §1.1, auxiliary variables will be an integral part of our framework. This means that assertions must depend on them as well as on the state. Initially we do not fix the type of auxiliary variables but parameterize the type of assertions with a type variable $'a$:

types $'a\ assn = 'a \Rightarrow state \Rightarrow bool$

Reasoning about recursive procedures requires a context to store the induction hypothesis about recursive *CALL*s. This context is a set of Hoare triples:

types $'a\ cntxt = ('a\ assn \times com \times 'a\ assn)set$

In the presence of only a single procedure the context will always be empty or a singleton set. With multiple procedures, larger sets can arise. Contexts are denoted by C and D.

Validity (w.r.t. partial correctness) is defined as usual, except that we have to take auxiliary variables into account as well:

$$\models \{P\}c\{Q\} \;\equiv\; \forall s\ t.\ s -c\rightarrow t \longrightarrow (\forall z.\ P\ z\ s \longrightarrow Q\ z\ t)$$

The state of the auxiliary variables (*auxiliary state* for short) is always denoted by z.

Validity of a context and a Hoare triple in a context are defined as follows:

$$||\!\!\models C \;\equiv\; \forall (P,c,Q) \in C.\ \models \{P\}c\{Q\}$$
$$C \models \{P\}c\{Q\} \;\equiv\; ||\!\!\models C \longrightarrow \models \{P\}c\{Q\}$$

Note that $\{\} \models \{P\}\ c\ \{Q\}$ is equivalent to $\models \{P\}\ c\ \{Q\}$.

Unfortunately, this is not the end of it. As we have two semantics, $-c\rightarrow$ and $-c-n\rightarrow$, we also need a second notion of validity parameterized with the recursion depth n:

$$\models n\ \{P\}c\{Q\} \;\equiv\; \forall s\ t.\ s -c-n\rightarrow t \longrightarrow (\forall z.\ P\ z\ s \longrightarrow Q\ z\ t)$$
$$||\!\!\models n\ C \;\equiv\; \forall (P,c,Q) \in C.\ \models n\ \{P\}c\{Q\}$$
$$C \models n\ \{P\}c\{Q\} \;\equiv\; ||\!\!\models n\ C \longrightarrow \models n\ \{P\}c\{Q\}$$

Finally we come to the proof system for deriving triples in a context:

$C \vdash \{\lambda z\ s.\ \forall t \in f\ s.\ P\ z\ t\}\ Do\ f\ \{P\}$
$[\![C \vdash \{P\}\ c1\ \{Q\};\ C \vdash \{Q\}\ c2\ \{R\}]\!] \Longrightarrow C \vdash \{P\}\ c1;\ c2\ \{R\}$
$[\![C \vdash \{\lambda z\ s.\ P\ z\ s \wedge b\ s\}\ c1\ \{Q\};\ C \vdash \{\lambda z\ s.\ P\ z\ s \wedge \neg\ b\ s\}\ c2\ \{Q\}]\!]$ $\Longrightarrow C \vdash \{P\}\ IF\ b\ THEN\ c1\ ELSE\ c2\ \{Q\}$
$C \vdash \{\lambda z\ s.\ P\ z\ s \wedge b\ s\}\ c\ \{P\} \Longrightarrow C \vdash \{P\}\ WHILE\ b\ DO\ c\ \{\lambda z\ s.\ P\ z\ s \wedge \neg\ b\ s\}$
Consequence: $[\![C \vdash \{P'\}\ c\ \{Q'\};\ \forall s\ t.\ (\forall z.\ P'\ z\ s \longrightarrow Q'\ z\ t) \longrightarrow (\forall z.\ P\ z\ s \longrightarrow Q\ z\ t)]\!]$ $\Longrightarrow C \vdash \{P\}\ c\ \{Q\}$
CALL: $\{(P,\ CALL,\ Q)\} \vdash \{P\}\ body\ \{Q\} \Longrightarrow \{\} \vdash \{P\}\ CALL\ \{Q\}$
Assumption: $\{(P,\ CALL,\ Q)\} \vdash \{P\}\ CALL\ \{Q\}$

Note that Hoare triples to the left of $\vdash$ are written as real triples, whereas to the right of $\vdash$ both the customary $\{P\}\ c\ \{Q\}$ syntax and ordinary triples are permitted.

The rule for *Do* is the generalization of the assignment axiom to an arbitrary nondeterministic state transformation. The next 3 rules are familiar, except for their adaptation to auxiliary variables. The *CALL* rule embodies induction and has already been motivated in §1.1. Note that it is only applicable if the context is empty. This shows that we never need nested induction. For the same reason the assumption rule is stated with just a singleton context.

The consequence rule is unusual but not completely new. Modulo notation it is identical to a slight reformulation by Olderog [21] of a rule by Cartwright and Oppen [5]. A different reformulation of the rule seems to have appeared for the first time in the work by Morris [14]. A more recent reinvention and reformulation is due to Hofmann [10]:

$$\forall s\ t\ z.\ P\ z\ s \longrightarrow Q\ z\ t \vee (\exists z'.\ P'\ z'\ s \wedge (Q'\ z'\ t \longrightarrow Q\ z\ t))$$

Although logically equivalent to our side condition, the symmetry of our version appeals not just for aesthetic reasons but because one can actually remember it!

Our system differs from earlier Hoare logics for partial correctness because we have followed Kleymann [24] who realized that a rule like the above consequence rule subsumes the normal consequence rule — thus the latter has become superfluous.

The proof of the soundness theorem

theorem $C \vdash \{P\}c\{Q\} \implies C \models \{P\}c\{Q\}$

requires a generalization: $\forall n.\ C \models n\ \{P\}\ c\ \{Q\}$ is proved instead, from which the actual theorem follows directly via lemma *exec-iff-execn*. The generalization is proved by induction on $C \vdash \{P\}\ c\ \{Q\}$.

The completeness proof follows the *most general triple* approach [6]:

$MGT :: com \Rightarrow state\ assn \times com \times state\ assn$
$MGT\ c \equiv (\lambda z\ s.\ z = s,\ c,\ \lambda z\ t.\ z\ {-c\rightarrow}\ t)$

There are a number of points worth noting. For a start, the most general triple equates the type of the auxiliary state z with type *state*. The precondition equates the auxiliary state with the initial state, so to speak making a copy of it. Therefore the postcondition can refer to this copy and thus the initial state. Finally, the postcondition is the strongest postcondition w.r.t. the given precondition and command.

It is easy to see that $\{\} \vdash MGT\ c$ implies completeness:

lemma *MGT-implies-complete*:
$\{\} \vdash MGT\ c \implies \{\} \models \{P\}c\{Q\} \implies \{\} \vdash \{P\}c\{Q{::}state\ assn\}$

Simply apply the consequence rule to $\{\} \vdash MGT\ c$ to obtain $\{\} \vdash \{P\}\ c\ \{Q\}$ — the side condition is discharged with the help of $\{\} \models \{P\}\ c\ \{Q\}$ and a little predicate calculus reasoning. The type constraint $Q{::}state\ assn$ is required because pre and postconditions in $MGT\ c$ are of type *state assn*, not *'a assn*.

In order to discharge $\{\} \vdash MGT\ c$ one proves

lemma *MGT-lemma*: $C \vdash MGT\ CALL \implies C \vdash MGT\ c$

The proof is by induction on c. In the *WHILE*-case it is easy to show that $\lambda z\ t.\ (z,\ t) \in \{(s,\ t).\ b\ s \wedge s\ -c\rightarrow\ t\}^*$ is invariant. The precondition $\lambda z\ s.\ z{=}s$ establishes the invariant and a reflexive transitive closure induction shows that the invariant conjoined with $\neg\ b\ t$ implies the postcondition $\lambda z\ t.\ z\ -WHILE\ b\ DO\ c\rightarrow\ t$. The remaining cases are trivial.

We can now derive $\{\} \vdash MGT\ c$ as follows. By the assumption rule we have $\{MGT\ CALL\} \vdash MGT\ CALL$, which implies $\{MGT\ CALL\} \vdash MGT\ body$ by the *MGT-lemma*. From the *CALL* rule it follows that $\{\} \vdash MGT\ CALL$. Applying the *MGT-lemma* once more we obtain the desired $\{\} \vdash MGT\ c$ and hence completeness:

theorem $\{\} \models \{P\}c\{Q\} \implies \{\} \vdash \{P\}c\{Q{::}state\ assn\}$

This is the first proof of completeness in the presence of (unbounded) nondeterminism. Earlier papers, if they considered completeness at all, restricted themselves to deterministic languages. However, our completeness proof follows the one by Apt [2] quite closely. This will no longer be the case for total correctness.

4 Hoare logic for total correctness

4.1 Termination

To express total correctness, we need to talk about guaranteed termination of commands. Due to nondeterminism, the existence of a terminating computation in the big-step semantics does not guarantee that all computations from some state terminate. Hence we inductively define a new judgement $c \downarrow s$ that expresses guaranteed termination of c started in state s:

$f\ s \neq \{\} \implies Do\ f \downarrow s$
$[\![c1 \downarrow s0;\ \forall s1.\ s0\ -c1\rightarrow\ s1 \longrightarrow c2 \downarrow s1]\!] \implies (c1;\ c2) \downarrow s0$
$[\![b\ s;\ c1 \downarrow s]\!] \implies IF\ b\ THEN\ c1\ ELSE\ c2 \downarrow s$
$[\![\neg\ b\ s;\ c2 \downarrow s]\!] \implies IF\ b\ THEN\ c1\ ELSE\ c2 \downarrow s$
$\neg\ b\ s \implies WHILE\ b\ DO\ c \downarrow s$
$[\![b\ s;\ c \downarrow s;\ \forall t.\ s\ -c\rightarrow\ t \longrightarrow WHILE\ b\ DO\ c \downarrow t]\!] \implies WHILE\ b\ DO\ c \downarrow s$
$body \downarrow s \implies CALL \downarrow s$

The first rule expresses that if *Do f* blocks, i.e. there is no next state in $f\ s$, we do not consider this a normal termination. Thus $\downarrow$ rules out both infinite and blocking computations. The remaining rules are self-explanatory.

By induction on $\downarrow$ it is easily shown that if *WHILE* terminates in the sense of $\downarrow$ then one must eventually reach a state where the loop test becomes false:

lemma $[\![\ (WHILE\ b\ DO\ c) \downarrow f\ k;\ \ \forall i.\ f\ i\ -c\rightarrow\ f(Suc\ i)\ ]\!] \implies \exists i.\ \neg b(f\ i)$

The inductive proof requires $f\ k$ rather than the more intuitive $f\ 0$.

It follows that the executions of the body of a terminating *WHILE*-loop form a well-founded relation (for *wf* see below):

lemma *wf-WHILE*: $wf\ \{(t,s).\ WHILE\ b\ DO\ c \downarrow s \wedge b\ s \wedge s\ {-c\rightarrow}\ t\}$

Now that we have termination, we can define total validity, $\models_t$, as partial validity and guaranteed termination:

$$\models_t \{P\}c\{Q\} \ \equiv\ \models \{P\}c\{Q\} \wedge (\forall z\ s.\ P\ z\ s \longrightarrow c{\downarrow}s)$$

For validity of a context and validity of a Hoare triple in a context we follow the corresponding definitions for partial correctness:

$$||\!\models_t C \ \equiv\ \forall (P,c,Q) \in C.\ \models_t \{P\}c\{Q\}$$
$$C \models_t \{P\}c\{Q\} \ \equiv\ ||\!\models_t C \longrightarrow \models_t \{P\}c\{Q\}$$

4.2 Hoare logic

To distinguish the proofs of partial and total correctness the latter use the symbol $\vdash_t$. The rules for $\vdash_t$ differ from the ones for $\vdash$ only in the two places where nontermination can arise (loops and recursion) and in the consequence rule:

$[\![wf\ r;\ \forall s'.\ C \vdash_t \{\lambda z\ s.\ P\ z\ s \wedge b\ s \wedge s' = s\}\ c\ \{\lambda z\ s.\ P\ z\ s \wedge (s,\ s') \in r\}]\!]$ $\Longrightarrow C \vdash_t \{P\}\ WHILE\ b\ DO\ c\ \{\lambda z\ s.\ P\ z\ s \wedge \neg\ b\ s\}$
$[\![wf\ r;\ \forall s'.\ \{(\lambda z\ s.\ P\ z\ s \wedge (s,\ s') \in r,\ CALL,\ Q)\} \vdash_t$ $\{\lambda z\ s.\ P\ z\ s \wedge s = s'\}\ body\ \{Q\}]\!]$ $\Longrightarrow \{\} \vdash_t \{P\}\ CALL\ \{Q\}$
$[\![C \vdash_t \{P'\}\ c\ \{Q'\};$ $(\forall s\ t.\ (\forall z.\ P'\ z\ s \longrightarrow Q'\ z\ t) \longrightarrow (\forall z.\ P\ z\ s \longrightarrow Q\ z\ t)) \wedge$ $(\forall s.\ (\exists z.\ P\ z\ s) \longrightarrow (\exists z.\ P'\ z\ s))]\!]$ $\Longrightarrow C \vdash_t \{P\}\ c\ \{Q\}$

Before we discuss these rules in turn, a note on *wf*, which means well-founded: a relation r is well-founded iff there is no infinite descending chain

$$\ldots, (s_3, s_2), (s_2, s_1), (s_1, s_0) \in r.$$

The *WHILE*-rule is fairly standard: in addition to invariance one must also show that the state goes down w.r.t. some well-founded relation r. The only notable feature is the universal quantifier ($\forall s'$) that allows the postcondition to refer to the initial state. If you are used to more syntactic presentations of Hoare logic you may prefer a side condition that s' is a new variable. But since we embed Hoare logic in a language with quantifiers, why not use them to good effect?

The *CALL*-rule is like the one for partial correctness except that use of the induction hypothesis is restricted to those cases where the state has become smaller w.r.t. r. The $\forall s'$ fulfills a similar function as in the *WHILE*-rule. See §4.4 for an application of this rule which elucidates how $\forall s'$ is handled.

The consequence rule is like its cousin for partial correctness but with a version of precondition strengthening conjoined that takes care of the auxiliary state z: $\forall s.\ (\exists z.\ P\ z\ s) \longrightarrow (\exists z.\ P'\ z\ s)$.

Our rules for total correctness are similar to those by Kleymann [13]. The difference in the *WHILE*-rule is that he has a well-founded relation on some arbitrary type α together with a function from *state* to α, which we have collapsed to a well-founded relation on *state*. This is equivalent but avoids the additional type α. The same holds for the *CALL*-rule. As a consequence our *CALL*-rule is much simpler than the one by Kleymann (and ultimately Sokolowski [25]) because we avoid the additional existential quantifiers over values of type α. Finally, the side condition in our rule of consequence looks quite different from the one by Kleymann, although the two are in fact equivalent:

$$\begin{aligned}&\textbf{lemma}\ ((\forall s\ t.\ (\forall z.\ P'\ z\ s \longrightarrow Q'\ z\ t) \longrightarrow (\forall z.\ P\ z\ s \longrightarrow Q\ z\ t)) \land\\ &\qquad (\forall s.\ (\exists z.\ P\ z\ s) \longrightarrow (\exists z.\ P'\ z\ s)))\\ &\quad = (\forall z\ s.\ P\ z\ s \longrightarrow (\forall t. \exists z'.\ P'\ z'\ s \land (Q'\ z'\ t \longrightarrow Q\ z\ t)))\end{aligned}$$

Kleymann's version (the proposition to the right of the =) is easier to use because it is more compact, whereas our new version clearly shows that it is a conjunction of the side condition for partial correctness with precondition strengthening, which is not obvious in Kleymann's formulation. Further equivalent formulations are explored by Naumann [15].

As usual, soundness is proved by induction on $C \vdash_t \{P\}\ c\ \{Q\}$:

$$\textbf{theorem}\ C \vdash_t \{P\}c\{Q\} \implies C \models_t \{P\}c\{Q\}$$

The *WHILE* and *CALL*-cases require well-founded induction along the given well-founded relation.

The key difference to previous work in the literature (Kleymann, America and de Boer, Apt, etc) emerges in the completeness proof. For total correctness, the most general triple used to be turned around: $\lambda z\ t.\ z\ {-c\rightarrow}\ t$ becomes the weakest precondition of $\lambda z\ s.\ z = s$. However, this only works if the programming language is deterministic. Hence we leave the most general triple as it is and merely add the termination requirement to the precondition:

$$MGT_t\ c\ \equiv\ (\lambda z\ s.\ z = s \land c{\downarrow}s,\ c,\ \lambda z\ t.\ z\ {-c\rightarrow}\ t)$$

The first two lemmas on the way to the completeness proof are unchanged:

$$\begin{aligned}&\textbf{lemma}\ \{\} \vdash_t MGT_t\ c \Longrightarrow \{\} \models_t \{P\}c\{Q\} \Longrightarrow \{\} \vdash_t \{P\}c\{Q{::}\mathit{state\ assn}\}\\ &\textbf{lemma}\ C \vdash_t MGT_t\ CALL \Longrightarrow C \vdash_t MGT_t\ c\end{aligned}$$

However, if we now try to continue following the proof at the end of §3 to derive $\{\} \vdash_t MGT_t\ c$ we can no longer do so directly because the *CALL*-rule has changed. What we would need is the following lemma:

$$\begin{aligned}&\textbf{lemma}\ \mathit{CALL\text{-}lemma}:\\ &\{(\lambda z\ s.\ (z{=}s \land \mathit{body}{\downarrow}s) \land (s,s') \in \mathit{rcall},\ CALL,\ \lambda z\ s.\ z\ {-\mathit{body}\rightarrow}\ s)\} \vdash_t\\ &\{\lambda z\ s.\ (z{=}s \land \mathit{body}{\downarrow}s) \land s = s'\}\ \mathit{body}\ \{\lambda z\ s.\ z\ {-\mathit{body}\rightarrow}\ s\}\end{aligned}$$

where *rcall* is some suitable well-founded relation. From that lemma the *CALL*-rule infers $\{\} \vdash_t \{\lambda z\ s.\ z = s \land CALL \downarrow s\}\ CALL\ \{\lambda z\ s.\ z\ {-CALL\rightarrow}\ s\}$ which is exactly $\{\} \vdash_t MGT_t\ CALL$. Completness follows trivially via the two lemmas further up. However, before we can even start to prove the hypothetical *CALL-lemma*, we need to provide the well-founded relation *rcall*, which turns out to be the major complicating factor.

Given a terminating *WHILE*, the iterated executions of the body directly yield the well-founded relation that proves termination. In contrast, given a terminating *CALL*, the big-step semantics does not yield a well-founded relation on states that decreases between the beginning of the execution of the body and a recursive call. The reason is that the recursive call is embedded in the body and thus the big-step semantics is too coarse. Informally what we want is the relation $\{(s', s) \mid$ starting the body in state s leads to a recursive *CALL* in state $s'\}$.

4.3 The termination ordering

In order to formalize the above informal description of the termination ordering we define a very fine-grained small-step semantics that one can view as an abstract machine operating on a command stack. Each step $(cs,\ s) \rightarrow (cs',\ s')$ (partially) executes the topmost element of the command stack *cs*, possibly replacing it with a list of new commands. Note that $x\ \#\ xs$ is the list with head x and tail xs.

$t \in f\ s \Longrightarrow (Do\ f\ \#\ cs,\ s) \rightarrow (cs,\ t)$
$((c1;\ c2)\ \#\ cs,\ s) \rightarrow (c1\ \#\ c2\ \#\ cs,\ s)$
$b\ s \Longrightarrow ((IF\ b\ THEN\ c1\ ELSE\ c2)\ \#\ cs,\ s) \rightarrow (c1\ \#\ cs,\ s)$
$\neg\ b\ s \Longrightarrow ((IF\ b\ THEN\ c1\ ELSE\ c2)\ \#\ cs,\ s) \rightarrow (c2\ \#\ cs,\ s)$
$\neg\ b\ s \Longrightarrow ((WHILE\ b\ DO\ c)\ \#\ cs,\ s) \rightarrow (cs,\ s)$
$b\ s \Longrightarrow ((WHILE\ b\ DO\ c)\ \#\ cs,\ s) \rightarrow (c\ \#\ (WHILE\ b\ DO\ c)\ \#\ cs,\ s)$
$(CALL\ \#\ cs,\ s) \rightarrow (body\ \#\ cs,\ s)$

Note that a separate *SKIP* command would obviate the need for lists: simply replace [] by *SKIP* and # by ;.

The above semantics is intentionally different from the customary structural operational semantics. The latter features the following rule:

$$(c1,s) \rightarrow (c1',s') \Longrightarrow (c1;c2,s) \rightarrow (c1';c2,s')$$

In case *c1* is a nest of semicolons, it is not flattened as above, and hence one cannot easily see what the next atomic command is. Which we need to see, to define *rcall*, the well-founded ordering required for the application of the *CALL* rule in the completeness proof in §4.2 above:

$$rcall\ \equiv\ \{(t,s).\ body{\downarrow}s \wedge (\exists\, cs.\ ([body],\ s) \rightarrow^{*} (CALL\ \#\ cs,\ t))\}$$

theorem *wf rcall*

The amount of work to prove this theorem is significant and should not be underestimated, but for lack of space we cannot discuss the details. The complexity of the proof is due to the two notions of (non)termination, $\downarrow$ and infinite $\rightarrow$ reductions, which need to be related. However, abolishing $\downarrow$ would help very little:

the lengthy proofs are those about $\rightarrow$, and one would then need to replace a few slick proofs via $\downarrow$ by more involved ones via $\rightarrow$.

To finish the completeness proof in §4.2 it remains to prove *CALL-lemma*. It cannot be proved directly but needs to be generalized first:

lemma $\{(\lambda z\ s.\ (z{=}s \wedge body{\downarrow}s) \wedge (s,t) \in rcall,\ CALL,\ \lambda z\ s.\ z\ {-}body{\rightarrow}\ s)\} \vdash_t$
$\{\lambda z\ s.\ (z{=}s \wedge body{\downarrow}t) \wedge (\exists\, cs.\ ([body],t) \rightarrow^* (c\#cs,s))\}\ c\ \{\lambda z\ s.\ z\ {-}c{\rightarrow}\ s\}$

This lemma is proved by induction on c. The *WHILE*-case is a little involved and requires a local reflexive transitive closure induction.

The actual *CALL-lemma* follows easily, as does completeness:

theorem $\{\} \models_t \{P\}c\{Q\} \implies \{\} \vdash_t \{P\}c\{Q{::}state\ assn\}$

4.4 Example

To elucidate the use of our very semantic-looking proof rules we will now verify the example from §1.1, showing only the key steps and minimizing Isabelle-specific detail. We start by declaring a type *variables* and defining *state* to be *variables* $\Rightarrow$ *nat* — the variables in the example program range only over natural numbers. The program variable `i` is represented by a constant i of type *variables*. The body of the recursive procedure is defined by translating tests and assignments into functions on states. Updating a function s at point x with value e is a predefined operation written $s(x := e)$.

$body \equiv IF\ \lambda s.\ s\ i = 0\ THEN\ Do(\lambda s.\{s\})$
$\quad ELSE\ (Do(\lambda s.\ \{s(i := s\ i - 1)\});\ CALL;\ Do(\lambda s.\ \{s(i := s\ i + 1)\}))$

We will now prove the desired correctness statement:

lemma $\{\} \vdash_t \{\lambda z\ s.\ s\ i = z\ N\}\ CALL\ \{\lambda z\ s.\ s\ i = z\ N\}$

As a first step we apply the *CALL*-rule where we instantiate r to $\{(t,\ s).\ t\ i < s\ i\}$ — well-foundedness of this relation is proved automatically. This leaves us with the following goal:

1. $\forall s'.\ \{(\lambda z\ s.\ s\ i = z\ N \wedge s\ i < s'\ i,\ CALL,\ \lambda z\ s.\ s\ i = z\ N)\} \vdash_t$
$\{\lambda z\ s.\ s\ i = z\ N \wedge s = s'\}\ body\ \{\lambda z\ s.\ s\ i = z\ N\}$

Isabelle always numbers goals. In this case there is only one. We get rid of the leading $\forall s'$ via HOL's $\forall$-introduction rule which turns it into $\bigwedge s'$, the universal quantifier of Isabelle's meta-logic. Roughly speaking this means that s' is now considered an arbitrary but fixed value.

After unfolding the *body* we apply the *IF*-rule and are left with two subgoals:

1. $\bigwedge s'.\ \{(\lambda z\ s.\ s\ i = z\ N \wedge s\ i < s'\ i,\ CALL,\ \lambda z\ s.\ s\ i = z\ N)\} \vdash_t$
$\{\lambda z\ s.\ (s\ i = z\ N \wedge s = s') \wedge s\ i = 0\}\ Do\ (\lambda s.\ \{s\})\ \{\lambda z\ s.\ s\ i = z\ N\}$
2. $\bigwedge s'.\ \{(\lambda z\ s.\ s\ i = z\ N \wedge s\ i < s'\ i,\ CALL,\ \lambda z\ s.\ s\ i = z\ N)\} \vdash_t$
$\{\lambda z\ s.\ (s\ i = z\ N \wedge s = s') \wedge s\ i \neq 0\}$
$Do\ (\lambda s.\ \{s(i := s\ i - 1)\});\ CALL;\ Do\ (\lambda s.\ \{s(i := s\ i + 1)\})$
$\{\lambda z\ s.\ s\ i = z\ N\}$

Both are easy to prove. During the proof of the second one we provide the intermediate assertions $\lambda z\ s.\ 0 < z\ N \wedge s\ i = z\ N - 1 \wedge s\ i < s'\ i$ and $\lambda z\ s.\ 0 < z\ N \wedge s\ i = z\ N - 1$. This leads to the following subgoal for the *CALL*:

$$
\begin{aligned}
1.\ \textstyle\bigwedge s'.\ & \{(\lambda z\ s.\ s\ i = z\ N \wedge s\ i < s'\ i,\ CALL,\ \lambda z\ s.\ s\ i = z\ N)\} \vdash_t \\
& \{\lambda z\ s.\ 0 < z\ N \wedge s\ i = z\ N - 1 \wedge s\ i < s'\ i\}\ CALL \\
& \{\lambda z\ s.\ 0 < z\ N \wedge s\ i = z\ N - 1\}
\end{aligned}
$$

Applying consequence and assumption rules we are left with

$$
\begin{aligned}
1.\ \textstyle\bigwedge s'.\ & (\forall s\ t.\ (\forall z.\ s\ i = z\ N \wedge s\ i < s'\ i \longrightarrow t\ i = z\ N) \longrightarrow \\
& \qquad (\forall z.\ 0 < z\ N \wedge s\ i = z\ N - 1 \wedge s\ i < s'\ i \longrightarrow \\
& \qquad\qquad 0 < z\ N \wedge t\ i = z\ N - 1)) \wedge \\
& (\forall s.\ (\exists z.\ 0 < z\ N \wedge s\ i = z\ N - 1 \wedge s\ i < s'\ i) \longrightarrow \\
& \qquad (\exists z.\ s\ i = z\ N \wedge s\ i < s'\ i))
\end{aligned}
$$

which is proved automatically. This concludes the sketch of the proof.

5 More procedures

We now generalize from a single procedure to a whole set of procedures following the ideas of von Oheimb [20]. The basic setup of §2 is modified only in a few places:

- We introduce a new basic type *pname* of procedure names.
- Constant *body* is now of type *pname* $\Rightarrow$ *com*.
- The *CALL* command now has an argument of type *pname*, the name of the procedure that is to be called.
- The call rule of the operational semantics now says

$$s\ -body\ p\rightarrow\ t \Longrightarrow s\ -CALL\ p\rightarrow\ t$$

Note that this setup assumes that we have a procedure body for each procedure name. In particular, *pname* may be infinite.

5.1 Hoare logic for partial correctness

Types *assn* and and *cntxt* are defined as in §3, as are $\models \{P\}\ c\ \{Q\}$, $|\models C$, $\models n\ \{P\}\ c\ \{Q\}$ and $|\models n\ C$. However, we now need an additional notion of validity $C\ |\models D$ where D is a set as well. The reason is that we can now have mutually recursive procedures whose correctness needs to be established by simultaneous induction. Instead of sets of Hoare triples we may think of conjunctions. We define both $C\ |\models D$ and its relativized version:

$C\ |\models D \equiv |\models C \longrightarrow |\models D$
$C\ |\models n\ D \equiv |\models n\ C \longrightarrow |\models n\ D$

Our Hoare logic defines judgements of the form $C \Vdash D$ where both C and D are (potentially infinite) sets of Hoare triples; $C \vdash \{P\}\ c\ \{Q\}$ is simply an abbreviation for $C \Vdash \{(P,c,Q)\}$. With this abbreviation the rules for ";", *IF*, *WHILE* and consequence are exactly the same as in §3. The remaining rules are

$\bigcup p.\ \{(P\ p,\ CALL\ p,\ Q\ p)\} \Vdash \bigcup p.\ \{(P\ p,\ body\ p,\ Q\ p)\} \Longrightarrow$ $\{\} \Vdash \bigcup p.\ \{(P\ p,\ CALL\ p,\ Q\ p)\}$
$(P,\ CALL\ p,\ Q) \in C \Longrightarrow C \vdash \{P\}\ CALL\ p\ \{Q\}$
$\forall (P,\ c,\ Q) {\in} D.\ C \vdash \{P\}\ c\ \{Q\} \Longrightarrow C \Vdash D$
$[\![C \Vdash D;\ (P,\ c,\ Q) \in D]\!] \Longrightarrow C \vdash \{P\}\ c\ \{Q\}$

Note that $\bigcup p.$ is the indexed union $\bigcup_p$.

The *CALL* and the assumption rule are straightforward generalizations of their counterparts in §3. The fact that *CALL*-rule reasons about *all* procedures simultaneously merely simplifies notation: arbitrary subsets of procedures work just as well. The final two rules are structural rules and could be called conjunction introduction and elimination, because they put together and take apart sets of triples.

Soundness is proved as before, by induction on $C \Vdash D$:

theorem $C \Vdash D \implies C \models\!\models D$

But first we generalize from $C \models\!\models D$ to $\forall n.\ C \models\!\models n\ D$. Now the *CALL*-case can be proved by induction on n.

The completeness proof also resembles the one in §3 closely: the most general triple *MGT* is defined exactly as before, and the lemmas leading up to completness are simple generalizations:

lemma $\{\} \vdash MGT\ c \Longrightarrow \models \{P\}c\{Q\} \Longrightarrow \{\} \vdash \{P\}c\{Q{::}state\ assn\}$
lemma $\forall p.\ C \vdash MGT(CALL\ p) \implies C \vdash MGT\ c$
lemma $\{\} \Vdash \bigcup p.\ \{MGT(CALL\ p)\}$
theorem $\models \{P\}c\{Q\} \implies \{\} \vdash \{P\}c\{Q{::}state\ assn\}$

5.2 Hoare logic for total correctness

Hoare logic for total correctness of mutually recursive procedures has not received much attention in the literature. Sokolowski's system [25], the only one that comes with a completness proof, is seriously incomplete, as it lacks rules of adaption to deal with the problem described in §1.1.

Our basic setup of termination and validity is as in §4 but extended by one more notion of validity:

$C \models\!\models_t D \ \equiv\ \models\!\models_t C \longrightarrow \models\!\models_t D$

The rules for *Do*, ";", *IF*, *WHILE* and consequence are exactly the same as in §4.2. In addition we have the two structural rules called conjunction introduction and elimination from §5.1 above (but with $\vdash_t$ instead of $\vdash$). Only the *CALL*-rule changes substantially and becomes

$$
\begin{array}{l}
[\![\ wf\ r;\\
\quad \forall q\ pre.\ (\bigcup p.\ \{(\lambda z\ s.\ P\ p\ z\ s \wedge ((p,s),(q,pre)) \in r, CALL\ p, Q\ p)\})\\
\qquad\qquad \vdash_t \{\lambda z\ s.\ P\ q\ z\ s \wedge s = pre\}\ body\ q\ \{Q\ q\}\ ]\!]\\
\Longrightarrow \{\} \Vdash_t \bigcup p.\ \{(P\ p,\ CALL\ p,\ Q\ p)\}
\end{array}
$$

This rule appears to be genuinely novel. To understand it, imagine how you would simulate mutually recursive procedures by a single procedure: you combine all procedure bodies into one procedure and select the correct one dynamically with the help of a new program variable which holds the name of the currently called procedure. The well-founded relation in the above rule is of type $((pname \times state) \times (pname \times state))set$ thus simulating the additional program variable by making *pname* a component of the termination relation.

We consider an example from [12] which the authors claim is difficult to treat with previous approaches [25, 22].

```
proc pedal = if n=0 ∨ m=0 then skip
             else if n < m then (n:=n-1; m:=m-1; CALL coast)
             else (n:=n-1; CALL pedal)
proc coast = if n<m then (m:=m-1; CALL coast) else CALL pedal
```

One possible termination ordering (which is all we are interested in) is the reverse lexicographic product of the relation $\{(\texttt{pedal}, \texttt{coast})\}$ on *pname* with the lexicographic ordering on $(\texttt{n}, \texttt{m})$. If `coast` calls `pedal`, $(\texttt{n}, \texttt{m})$ is unchanged and the relation on *pname* decreases. In all other cases either `n` decreases or `n` is unchanged and `m` decreases.

Soundness and completeness are proved almost exactly as for a single procedure. We do not even need to show the theorems. Previous work on total correctness of mutually recursive procedures is either incomplete [25] or lacks completeness proofs [22, 12].

6 Expressiveness and relative completeness

In the literature, most completeness results for Hoare logics are qualified with the word *relative*, meaning relative to the completeness of the deductive system for the assertion language, which enters the picture in the consequence rule. This issue is absent in our formalization for the following reason: both $\models$ and $\vdash$ are specified in the same finite logical system, HOL. Thus they both inherit HOL's incompleteness. In particular, there must be valid Hoare triples whose validity is not provable in HOL. What the completeness theorem tells us is that both $\models$ and $\vdash$ are equally incomplete. This is important because it means we never need to resort to the operational semantics to prove some Hoare triple, we can always do it just as well in the Hoare logic.

The second important issues that we have ignored so far is expressiveness, i.e. the ability to express the intermediate predicates that may arise in a proof. In the following discussion we restrict attention to programs where the boolean expressions and the functions in the *Do*-commands are definable in the assertion language.

Clearly HOL is expressive as the completeness proofs can be formalized in it. We will narrow things down to weaker logical systems, although an analysis of the precise proof theoretic strength required is beyond the scope of this paper. For partial correctness the customary result that first-order arithmetic is expressive still holds, essentially because the most general triple can be expressed in it. The details are standard. For total correctness matters change. First-order arithmetic is still expressive for bounded nondeterminism (as shown by Apt [3] for Dijkstra's guarded commands) but fails to be so in the presence of unbounded nondeterminism [3, 4]. The reason is that we now have to formalize assertions about termination. Apt solves the problem by enriching the assertion language with a least fixedpoint operator, i.e. moving towards the μ-calculus. Essentially we have used the same trick: termination ($\downarrow$) is defined inductively, which can be expressed as a least fixedpoint (and this is in fact what Isabelle/HOL translates inductive definitions into internally). Therefore first-order arithmetic enriched with least fixedpoints is expressive in our setting, too.

However, there is one more complication: our proof rules for loops and procedure calls employ arbitrary well-founded orderings on the state space. Fortunately we can dispense with general well-founded orderings. Studying the completeness proof in §4, we find that two termination orderings suffice, namely the one in lemma *wf-WHILE* for loops (§4.1) and *rcall* for procedures (§4.3). Hence we could specialize the two rules with these most general termination orderings, thus removing the well-foundedness premise while retaining completeness. And if we examine the definition of these orderings, we find that they require the same ingredients as the most general triple, namely the transition semantics and the termination predicate ($\downarrow$). Thus the version of the μ-calculus used by Apt [3, 4], or any reasonable logic that can express most general triples, is expressive for procedures as well. In contrast, Apt and Plotkin require (recursive) ordinals on top of their μ-calculus. They are aware that the ordinals are strictly speaking not necessary (Hitchcock and Park [7] do without them) but leave it as an open question to find a *syntax directed* system without ordinals. Our proof system provides one answer.

Acknowledgments I am indebted to Thomas Kleymann and David von Oheimb for providing the logical foundations, to Krzysztof Apt and Kamal Lodaya for very helpful comments, and to Markus Wenzel for the Isabelle document preparation system.

References

1. Pierre America and Frank de Boer. Proving total correctness of recursive procedures. *Information and Computation*, 84:129–162, 1990.
2. Krzysztof Apt. Ten Years of Hoare's Logic: A Survey — Part I. *ACM Trans. Programming Languages and Systems*, 3(4):431–483, 1981.
3. Krzysztof Apt. Ten Years of Hoare's Logic: A Survey — Part II: Nondeterminism. *Theoretical Computer Science*, 28:83–109, 1984.
4. Krzysztof Apt and Gordon Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33:724–767, 1986.
5. Robert Cartwright and Derek Oppen. The logic of aliasing. *Acta Informatica*, 15:365–384, 1981.
6. Gerald Arthur Gorelick. A complete axiomatic system for proving assertions about recursive and non-recursive programs. Technical Report 75, Dept. of Computer Science, Univ. of Toronto, 1975.

7. Peter Hitchcock and David Park. Induction rules and termination proofs. In M. Nivat, editor, *Automata, languages, and programming*, pages 225–251. North Holland, 1973.
8. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:567–580,583, 1969.
9. C.A.R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Semantics of algorithmic languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer-Verlag, 1971.
10. Martin Hofmann. Semantik und Verifikation. Lecture notes, Universität Marburg. In German, 1997.
11. Peter V. Homeier and David F. Martin. Mechanical verification of mutually recursive procedures. In M.A. McRobbie and J.K. Slaney, editors, *Automated Deduction — CADE-13*, volume 1104 of *Lect. Notes in Comp. Sci.*, pages 201–215. Springer-Verlag, 1996.
12. Peter V. Homeier and David F. Martin. Mechanical verification of total correctness through diversion verification conditions. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics (TPHOLs'98)*, volume 1479 of *Lect. Notes in Comp. Sci.*, pages 189–206. Springer-Verlag, 1998.
13. Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11:541–566, 1999.
14. J.H. Morris. Comments on "procedures and parameters". Undated and unpublished.
15. David Naumann. Calculating sharp adaptation rules. *Information Processing Letters*, 77:201–208, 2000.
16. Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications*. Wiley, 1992.
17. Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lect. Notes in Comp. Sci.*, pages 180–192. Springer-Verlag, 1996.
18. Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.
19. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002.
20. David von Oheimb. Hoare logic for mutual recursion and local variables. In C. Pandu Rangan, V. Raman, and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, volume 1738 of *Lect. Notes in Comp. Sci.*, pages 168–180. Springer-Verlag, 1999.
21. Ernst-Rüdiger Olderog. On the notion of expressiveness and the rule of adaptation. *Theoretical Computer Science*, 24:337–347, 1983.
22. P. Pandya and M. Joseph. A structure-directed total correctness proof rule for recursive procedure calls. *The Computer Journal*, 29:531–537, 1986.
23. Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.
24. Thomas Schreiber. Auxiliary variables and recursive procedures. In *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *Lect. Notes in Comp. Sci.*, pages 697–711. Springer-Verlag, 1997.
25. Stefan Sokołowski. Total correctness for procedures. In *Mathematical Foundations of Computer Science (MFCS)*, volume 53 of *Lect. Notes in Comp. Sci.*, pages 475–483. Springer-Verlag, 1977.