Technical Report

UCAM-CL-TR-324
ISSN 1476-2986

Number 324

Verifying modular programs in HOL

J. von Wright

January 1994

ISSN 1476-2986

Verifying Modular Programs in HOL

J. von Wright*

January 18, 1994

1 Introduction

This paper describes a methodology for verifying imperative programs that are modular, i.e., built using separately defined functions and procedures.

The verification methodology is based on a simple programming notation with a weakest precondition semantics. This notation has been semantically embedded in the HOL theorem prover [3] and a number of laws have been derived from the semantics.

These semantic laws are used to prove (in HOL) the correctness of functional procedures, by showing that a call to the procedure in question is equivalent to a call to the corresponding function as it is defined in the logic. This makes it possible to specify a program in an essentially functional style, but the functions are then implemented as imperative procedures (like user-defined functions in FORTRAN or Pascal).

We also show how to define non-functional procedures and calls to such procedures. Procedures may be recursive. Altogether, this gives us a basis for mechanical verification of modular imperative programs.

2 The programming notation

The programming notation that we use has a simple syntax. To make life simple, we use the syntax of the HOL-embedding directly. In a sense, this means that we blur the distinction between syntax and semantics.

2.1 Predicates

A *predicate* is an arbitrary boolean-valued function `p:*->bool`. In instances, the argument type `*` will be a product type and elements of this product type are called *states*.

A predicate always makes explicit what the state space is. For example if we want to model the predicate

$$x < y + 1$$

we have to know what the underlying state space is. Assuming that it is a triple (x,y,z) of variables ranging over the natural numbers, we would represent the predicate as

*Åbo Akademi University, Turku, Finland. E-mail: jwright@aton.abo.fi

```
λ(x,y,z). x < y + 1
```

This means that the state components (which model the program variables) are anonymous; they are identified by their position in the state tuple rather than by name.

We define functions corresponding to the ordinary logical operations on predicates:

```
⊢def false   = λu. F
⊢def true    = λu. T
⊢def not p   = λu. ¬p u
⊢def p and q = λu. p u ∧ q u
⊢def p or q  = λu. p u ∨ q u
⊢def p imp q = λu. p u ⇒ q u
```

We also introduce generalized conjunctions (greatest lower bounds) and disjunctions (least upper bounds) of sets of predicates:

```
⊢def glb P = λu. ∀p. P p ⇒ p u
⊢def lub P = λu. ∃p. P p ∧ p u
```

Note that sets are modelled by their characteristic functions.

Furthermore, we define the implication order on predicates:

```
⊢def p implies q =  ∀u. p u ⇒ q u
```

Thus `p implies q` means that `p` is stronger than `q`.

Our HOL-theory of predicates can be found in Appendix A.

2.2 Commands

A *command* is a function `c:(*2->bool)->(*1->bool)` from predicates to predicates. The idea behind this is to identify a command `c` with its weakest precondition predicate transformer. Predicate transformers map predicates over the final state space to predicates over the initial state space (note that these need not be the same state space). Intuitively, the predicate `c q` holds in an initial state `s` if execution of `c` from initial state `s` is guaranteed to terminate in a final state where the predicate `q` holds. What we call `c q` would in traditional weakest precondition methodology be called `wp(c,q)`.

In principle, we accept *any* function of type `(*2->bool)->(*1->bool)` as a command. However, we mainly want to reason about commands that have a specific syntax. For this, we introduce a command syntax, which essentially just consists of a number of abbreviations for commands. We start with a simple set of commands; later on we will extend them with a specification construct and recursion. The commands we want to reason about at this point have the following syntax:

```
c ::= skip
    | assert g
    | assign e
    | c1 seq c2
    | cond g c1 c2
    | do g c
```

where g is a predicate (for "guard") and e is a state-state function (of type *1->*2).

Intuitively, seq models sequential composition, cond models conditional composition (if-then-else) and do models iteration (a while-loop). Finally, assign e is a command which transforms a state s into a state e s. Ordinary assignment statements can always be written using this command.

The predicate transformer semantics of these commands is at the same time their definitions:

```
⊢def skip q          = q
⊢def assert g q      = g and q
⊢def assign e q      = λu. q (e u)
⊢def (c1 seq c2) q   = c1(c2 q)
⊢def cond g c1 c2 q = (g andd (c1 q)) or ((not g) andd (c2 q))
⊢def do g c q        = fix(λp. (g andd (c p)) or ((not g) andd q)))
```

Here fix is a least fixpoint operator for monotonic functions on predicates (see Appendix B).

As an example, consider the assignment $x := x + y$ in a three-component state space where x is the first and y the second component. This is represented by the command

```
assign λ(x,y,z). (x+y,y,z)
```

Note that we explicitly state that the values of the second and third component are unchanged. The following example shows how our definition of the assignment corresponds to the ordinary definition. Consider the weakest precondition for the assignment $x := x + y$ to establish the postcondition $x > z$. In traditional wp-semantics, we have the rule

$$\text{wp}(x := E, Q) \;=\; Q[E/x]$$

which means that

$$\text{wp}(x := x + y, x > z) \;=\; x + y > z$$

In our framework, the following calculation gives the same result:

```
(assign λ(x,y,z). (x+y,y,z))(λ(x,y,z).x>z)
= λ(x,y,z). (λ(x,y,z).x>z)(λ(x,y,z). (x+y,y,z))(x,y,z)
= λ(x,y,z). (λ(x,y,z).x>z)(x+y,y,z)
= λ(x,y,z). x+y>z
```

Our command notation is in a sense very arbitrary. We may add more constructs (as we do in Section 5). We also note that the skip command is redundant; it can be written as an assignment:

```
⊢ skip = assign λu.u
```

Healthiness conditions We identify four important criteria (often called "healthiness conditions") that commands with a weakest precondition semantics are usually assumed to satisfy. These criteria are monotonicity, strictness, conjunctivity and termination. Their definitions are as follows:

```
⊢def monotonic c   =  (∀p q. p implies q ⇒ (c p) implies (c q))
⊢def strict c      =  (c false = false)
⊢def conjunctive c =
       ∀P. (∃p. P p) ⇒ (c(glb P) = glb(. ∃p. P p ∧ (q = c p)))
⊢def terminating c =  (c true = true)
```

In fact, all the commands in our notation are monotonic, strict and conjunctive. The `assert` and `do` commands may be nonterminating.
Conjunctivity of `c` means that `c` distributes over arbitrary nonempty conjunctions of predicates. Intuitively, the conjunctivity assumption means that nondeterminism is assumed to be demonic. Note that since `true` is the empty conjunction, a command which is terminating and conjunctive distributes over all conjunctions.

It is also common to define a notion of *continuity*. A restriction to continuous commands corresponds to an assumption that nondeterminism is never unbounded. However, we will not make use of continuity in this paper.

2.3 Other commands

In later sections, we will extend our programming notation with three important constructs. One is a specification construct (a nondeterministic assignment) `nondass` and another is a recursion construct `mu`. The third construct is `block` which lets us handle local variables.

In order to prove important facts in predicate transformer semantics, we have also defined a number of other commands. Since they will not be used in the programs we want to reason about, we will not show their definitions or consider them in detail. The interested reader can find their definitions in Appendix B. The following is a brief description of these commands:

`guard`	dual to `assert`
`dch`	demonic choice of two commands
`Dch`	demonic choice of a set of commands
`ach`	angelic choice of two commands
`Ach`	angelic choice of a set of commands
`dolib`	a variant of iteration with a greatest fixpoint semantics

3 Total correctness

A command `c` is said to be totally correct with respect to precondition `p` and postcondition `q` if `c` is guaranteed to establish `q` whenever it is executed from an initial state where `p` holds. Our definition is simple:

```
⊢ correct p c q  =  p implies (c q)
```

We have derived (i.e., proved in HOL) a number of rules for proving total correctness of commands. For a complete listing, see Appendix D.

For assignments, the rule is as follows:

```
⊢ (∀v. p v ⇒ q(e v)) ⇒ correct p (assign e) q
```

This corresponds closely to the ordinary Hoare rule (of total or partial correctness) for assignment:

$$\frac{P \;\Rightarrow\; Q[E/x]}{\{P\}\; x := E\; \{Q\}}$$

For sequential composition, we have a simple sequencing rule, familiar from Hoare logic:

```
⊢ (∃q. monotonic c1 ∧ correct p c1 q ∧ correct q c2 r)
    ⇒ correct p (c1 seq c2) r
```

When applying this rule, the user must supply a suitable intermediate predicate q. The monotonicity condition can be proved automatically for commands that are built using the basic command notation.

For the conditional commmand, we also have a simple Hoare-style rule:

```
⊢ correct (g and p) c1 q ∧ correct ((not g) and p) c2 q
    ⇒ correct p (cond g c1 c2) q
```

The rule for iteration is the well-known invariant-bound rule:

```
⊢ (∃inv t. monotonic c ∧
           p implies inv ∧
           ((not g) and inv) implies q ∧
           (∀x. correct (inv and (g and (λu. t u = x)))
                        c
                        (inv and (λu. (t u) < x))))
   ⇒ correct p (do g c) q
```

When this rule is used, the user has to supply the invariant inv and the bound function t. In fact, this is a special case of the more general rule where the bound function has values in an arbitrary well-founded set. The HOL-theory of well-founded relations that we have built is in Appendix C and the general rule for iteration in Appendix D.

We do not give a separate correctness rule for the assert command. This is because this command is not interesting as such. Instead it is used to makes reasoning about conditional correctness possible (see Section 4.4).

4 Functional abstraction

We now propose a method for using HOL to verify imperative programs that are built using functional procedures. The HOL theory of this section is listed in Appendix E.

4.1 A function call operator

We assume that a program c is supposed to compute a function f. We take the view that functions are uncurried and they always have at least one argument. This means that we can without loss of generality assume that f has type *1->*2. In an instantiation, the initial state *1 contains one component for each argument of the function, while the final state *s2 has only one component (the result returned by the function). Since this component may contain subcomponents, a multi-valued function can be treated as a single-valued function whose value is a tuple.

Now assume that we have written a program fragment c of which we can prove the following:

```
⊢ c = assign f
```

Then it is fair to say that c is an implementation of the function f, and that any occurrence of f in a program can be replaced by a call to c.

This idea has been formalized in the following way. We define an operator fcall, which extracts the function f from a command c in the case when c = assign f holds.

The definition of the fcall operator is not very illuminating:

```
⊢def fcall c = (λu. εv. glb(λq.c q u)v)
```

However, the important thing is that we can prove the crucial theorem:

```
⊢ fcall (assign f) = f
```

In fact, we can prove a theorem which is easily used as a proof rule:

```
⊢ conjunctive c ∧ strict c ∧
   (∀u0. correct (λu.u = u0) c (λv.v = f u0))
    ⇒ (fcall c = f)
```

Using this theorem, we can prove that a given program c implements a function f. Once this is done, we can replace any occurrence of f by fcall c, which models the call to the functional procedure c.

When the above theorem is used as a proof rule, the conjunctivity and strictness conditions can be proved automatically (we have implemented HOL tactics for this), so we are left with a total correctness condition. This can be proved using the rules in Section 3.

4.2 Example: list membership

As a small example, we implement a list membership function. List membership can be defined in HOL as follows:

```
⊢def (∀x. MEMBER(x,[]) = F) ∧
   (∀x h t. MEMBER(x,(CONS h t)) = (h = x) ∨ MEMBER(x,t))
```

(note that this is an uncurried version of the membership function).

The program that we propose to implement this function with is defined as follows:

```
⊢def fMEMBER =
    (assign λ(x:*,l:(*)list).(F,x,l)) seq
    (do (λ(r,x,l).¬(l=[]) ∧ ¬r)
        (assign λ(r,x,l).((x = HD l),x,TL l)) ) seq
    (assign λ(r,x,l).r)
```

Note how this program is built up. The global state components (i.e., the reference parameters) are `x` and `l`; the element and the list. The first **assign** command initializes the local boolean variable `r` to the boolean value `F`. The main body of the program is the iteration, which checks the elements of the list, one by one, until either the list is empty or until a match is found, in which case `r` is set to the boolean value `T`. The final **assign** command returns `r` as the result. In a Pascal-style syntax this function would be something like

```
function fMEMBER(x:*,l:(*)list):bool;
  var r:bool; begin r := F;
  while (not(l=[]) and not r) do begin
    r,l := (x = HD l),TL l
  end;
  fMEMBER := r
end;
```

(in fact, it should be possible to do such a translation automatically). The correctness of the `fMEMBER` procedure is proved in the theorem

```
⊢ fcall fMEMBER = MEMBER
```

which shows that any occurrence of the `MEMBER` function can be correctly replaced by a call to `fMEMBER`.

Note that in the program `fMEMBER` we use other function that HOL can recognize: `HD` and `TL`. These can either be assumed to be built-in, or else we can write separate programs that implement them.

4.3 Conditional correctness

In some cases we only want to be sure that a functional procedure correctly implements a function in the case when a given precondition holds. We call this *conditional correctness* (not to be confused with partial correctness, which does not require termination).

We have derived the following rule for proving conditional correctness assertions:

```
⊢ conjunctive c ∧ strict c ∧
   (∀u0. p u0 ⇒ correct(λu. u = u0)c(λv. v = f u0))
   ⇒ (∀u. p u ⇒ (fcall c u = f u))
```

Here `p` is the precondition that the implementation is proved under.

For example, we may want to implement a function which retrieves a value from an association list, provided that the given argument occurs in the list.

Such a function `ASSOC` can be specified as follows:

```
⊢ ASSOC(x0,(CONS(x,y)l)) = ((x = x0) →  y | ASSOC(x0,l))
```

(note that this is a partial specification; we do not specify what happens in the case when the value is not found in the list).

If `x0` occurs as a first component of some pair in a list `l` then `ASSOC(x0,l)` returns the corresponding second component. Otherwise, its behaviour is unspecified.

The procedure `fASSOC` implements this function, under the precondition that we can find the given element in the association list.

```
⊢ fASSOC =
    (assign(λ(x,l).((εr.T),x,l))) seq
    (do (λ(r,x,l). ¬(l = []))
        (cond (λ(r,x,l).x = FST(HD l))
              (assign λ(r,x,l).(SND(HD l),x,[]))
              (assign λ(r,x,l).(r,x,TL l)) ) ) seq
    (assign(λ(r,x,l). r))
```

The first `assign` command initializes the local component `r` to an arbitrary value (since the type of `r` is polymorphic, the only way to choose an arbitrary value is by using the Hilbert operator `@`). Then the list is searched until it is empty. If a match is found then the corresponding value is stored in `r` and the list is nulled. The final `assign` command then returns the value of `r`. In Pascal-style syntax, the function can be written in the following way:

```
function fASSOC(x:*,l:(*#**)list):bool;
  var r:**; begin
  while (not(l=[])) do begin
    if x = FST(HD l) then
      r,l := SND(HD l),[]
    else begin
      l := TL l
    end
  end;
  fASSOC := r
end;
```

The initialisation of the local variable is not modelled here (for a comment on this, see Section 5.2). The correctness theorem that we have proved for this procedure is the following:

```
⊢ MEMBER(x,(MAP FST l)) ⇒ (fcall fASSOC(x,l) = ASSOC(x,l))
```

This means that an occurrence of `ASSOC` can be replaced by a call to `fASSOC` provided that we have a guarantee that the membership condition holds.

4.4 Using assertions

We now show how the `assert` command can be used to introduce calls to conditionally correct procedures.

First of all, we note the following theorem which shows how assertions can be introduced in a program:

```
⊢ conjunctive c ∧ correct true c g ⇒ (c = c1 seq (assert g))
```

We use the above example to show how this works. For simplicity, we assume that we are working in a three-component state space of type `**#*#((*#**))list`. We consider the following program `P`:

```
⊢ P = c seq (assign λ(y,x,l).(ASSOC(x,l),x,l))
```

We want to replace the occurrence of `ASSOC` by a call to the procedure `fASSOC`. To do this, we first prove the correctness of the following assertion introduction:

```
⊢ c = c seq (assert λ(y,x,l).MEMBER(x,l))
```

Together with the associativity of sequential composition this theorem permits us to rewrite the original program as

```
⊢ P = c seq
       ((assert λ(y,x,l).MEMBER(x,l)) seq
        (assign λ(y,x,l).(ASSOC(x,l),x,l)) )
```

The conditional correctness theorem for the `fASSOC` procedure states that a call to `fASSOC` is equivalent to `ASSOC` whenever the membership predicate holds. This theorem can now be used to prove the following equivalence:

```
⊢ (assert λ(y,x,l).MEMBER(x,l)) seq (assign λ(y,x,l).(ASSOC(x,l),x,l))
   = (assert λ(y,x,l).MEMBER(x,l)) seq
     (assign λ(y,x,l).(fcall fASSOC(x,l),x,l))
```

Now this fact is used to rewrite the original program:

```
⊢ P = c seq
       ((assert λ(y,x,l).MEMBER(x,l)) seq
        (assign λ(y,x,l).(fcall fASSOC(x,l),x,l)) )
```

Finally we rewrite using the associativity of `seq` and the theorem that introduced the assertion again to get

```
⊢ P = c seq (assign λ(y,x,l).(fcall fASSOC(x,l),x,l))
```

This method for introducing calls to conditionally correct procedures cannot be formalized in a single theorem, since the functions can have an arbitrary number of parameters and they may occur in arbitrary places in the state tuple. Hoever, it is possible to write a standard proof strategy for proving the correctness of such call introductions, leaving the user only to prove the assertion introduction theorem.

5 Nondeterminism and refinement

So far, we have not consider any nondeterminism in our notation. We have also considered only equivalences between programs. We now show how nondeterminism can be used to give more natural specifications of functional procedures in some situations.

5.1 A nondeterministic assignment

We extend our notation with the nondeterministic assignment construct `nondass`. Let `P:*s1->*s2->bool` be a relation between the initial and the final state space. Intuitively, `nondass P` is a command which for the initial state `v` chooses an arbitrary final state `v'` such that `P v v'` holds. If there is no such state, then the command is *miraculous*, i.e., its behaviour is not specified. This means that the `nondass` command is not necessarily strict. It is always monotonic, conjunctive and terminating, though. For example, the following command increases the first component of a two-component state space arbitrarily:

```
nondass λ(x,y)(x',y'). (x'>x) ∧ (y'=y)
```

Formally, the definition of the nondeterministic assignment is the following:

```
⊢def nondass P q = (λv. ∀v'. P v v' ⇒ q v')
```

The nondeterministic assignment is useful in a situation where an ordinary assignment is over-specific. For example, we often do not care how a local variable is initialized. If we use the `assign` construct to introduce the variable, we are forced to supply an initial value. On the other hand, the `nondass` construct permits us to introduce the variable without restricting its initial value.

As an example, we consider a functional procedure for the `ASSOC` function, defined as follows:

```
⊢ fASSOC =
    (assign(λ(x,l).((εr.T),x,l))) seq
    (do (λ(r,x,l). ¬(l = []))
        (cond (λ(r,x,l).x = FST(HD l))
              (assign λ(r,x,l).(SND(HD l),x,[]))
              (assign λ(r,x,l).(r,x,TL l)) ) ) seq
    (assign(λ(r,x,l). r))
```

In this function, we can replace the initial assignment by the nondeterministic assignment

```
nondass λ(x,l)(r,x',l'). (x'=x) ∧ (l'=l)
```

which introduces the local component `r` without restricting its initial value. Thus our alternative procedure `fASSOC'` is defined as follows:

```
⊢ fASSOC' =
    (nondass λ(x,l)(r,x',l'). (x'=x) ∧ (l'=l)) seq
    (do (λ(r,x,l). ¬(l = []))
        (cond (λ(r,x,l).x = FST(HD l))
              (assign λ(r,x,l).(SND(HD l),x,[]))
              (assign λ(r,x,l).(r,x,TL l)) ) ) seq
    (assign(λ(r,x,l). r))
```

This procedure avoids the use of the arbitrary initialization to `@r.T` used in `fASSOC`. The two procedures are equivalent when the membership precondition holds: we can prove the same correctness theorem for `fASSOC'` as for `fASSOC` and thus deduce that any occurrence of `ASSOC` can be replaced by a call to `fASSOC'`.

5.2 Refinement

It can be argued that an occurrence of **nondass** makes a procedure difficult to implement, since most programming languages do not contain any nondeterministic constructs.

However, if we use **nondass** only to introduce an uninitialized local variable, then we can consider it to be implemented, e.g., by the **var**-initialization in a Pascal-program.

Alternatively, we can make *refinements* to a procedure before we implement it. The refinement relation `ref` on commands is defined as follows:

```
⊢def ∀c c'. c ref c' = (∀q. (c q) implies (c' q))
```

The refinement relation is reflexive, antisymmetric and transitive (i.e., it is a preorder). An equivalent characterization of refinement is the following:

```
⊢ ∀c c'. c ref c' = (∀p q. correct p c q ⇒ correct p c' q)
```

i.e., refinement means preserving total correctness. This means that if we have proved a total correctness theorem for a procedure c and we then refine c into `c'`, we know that the correctness theorem holds for `c'` also.

In particular, we can refine any implementation of a function and the result will still be an implementation of the same function, as long as the strictness and conjunctivity conditions are not violated. Since the nondeterministic assignment is not always strict, we have to be a bit careful. In particular, the command

```
nondass λu v. false
```

(the miracle command) is a correct refinement of any command.

Refining a nondeterministic assignment The following simple rule shows how we can refine away a nondeterministic assignment:

```
⊢ ∀P e. (∀u. P u (e u)) ⇒ (nondass P) ref (assign e)
```

Thus we can always replace a nondeterministic assignment with an ordinary assignment, provided that the value assigned is one of the values permitted in the nondeterministic assignment.

Furthermore, it is easy to prove that refinements of subcommands are always permitted. The following theorems show this:

```
⊢ c1 ref c1' ∧ c2 ref c2' = (c1 seq c2) ref (c1' seq  c2')
⊢ c1 ref c1' ∧ c2 ref c2' = (cond g c1 c2) ref (cond g c1' c2')
⊢ c ref c' = (do g c) ref (do g c')
```

Thus we can easily (in fact automatically) prove the theorem

```
⊢ fASSOC' ref fASSOC
```

from which follows

```
⊢ ((assert λ(x,l). MEMBER(x,(MAP FST l))) seq fASSOC′)
   ref ((assert λ(x,l). MEMBER(x,(MAP FST l))) seq fASSOC)
```

From this, and the correctness theorems for `fASSOC` and `fASSOC'`, it is not too difficult to prove that under the membership precondition, the procedures `fASSOC` and `fASSOC'` implement the same function:

```
⊢ MEMBER(x,(MAP FST l)) ⇒ (fcall fASSOC′(x,l) = fcall ASSOC(x,l))
```

We could have proved the same thing without the notion of refinement. However, the proof that `fASSOC'` and `fASSOC` are equivalent under the membership precondition would then have been much harder. This equivalence holds, but in order to prove it we have to consider both procedures as wholes. The proof of refinement is simple, because it only requires a trivial proof on the subcomponent level.

Removing assertions Another application of refinement is in the removal of assertions. Assertion commands are introduced using the rule

```
⊢ conjunctive c ∧ correct true c g ⇒ (c = c1 seq (assert g))
```

After they have been used in the reasoning, we usually want to remove assertions. If we work only with equivalence, removing an assertion may require a non-trivial proof, if it is permitted at all. However, removing assertions is always a correct refinement step, as the following two theorems show:

```
⊢ ∀g c. (c seq (assert g)) ref c
⊢ ∀g c. ((assert g) seq c) ref c
```

6 Procedures

We will take the view that a procedure is a program fragment which implements (refines) a specification. A specification, in turn, is generally a nondeterministic assignment.

In this section, we will show how procedures can be handled in our HOL notation. We define a procedure call operator `pcall` and we show how parameters are handled. The definitions and theorems of this theory can be found in Appendix F (procedures and recursion).

6.1 A construct with local variables

Procedures generally have local variables, so we extend our command notation with a new construct for handling them smoothly. The command `block p c` stands for a block with a local state component (added as the first component of the state). The first argument `p` is an *initialisation predicate* and `c` is the body of the block. Intuitively, the execution of a block can be described as follows. Initially, the local state component is given a value such that p is established. Then c is executed and finally, the local component is removed from the state.

The definition of the block construct is as follows:

```
⊢ block p c q = (λv. ∀x. p(x,v) ⇒ c(λ(x',v'). q v')(x,v))
```

The block does not add anything new to the language; we can always simulate it using assignments and sequential composition:

```
⊢ block p c =
     (nondass λv(x',v').p(x,v) ∧ (v'=v)) seq c seq (assign λ(x,v).v)
```

The advantage of using the block construct is that it corresponds closely to the handling of local variables in ordinary programming languages.

6.2 A procedure call with reference parameters

A procedure works on a state space that only involves the necessary state components. When we make a call to the procedure, we must indicate which state components are to be mapped into the state space of the procedure. The execution of a procedure call should work on the indicated state components and leave the rest of the state unchanged.

In order to show how this works, we consider a simple example. The procedure (program fragment) pREVERSE is defined as follows:

```
⊢ pREVERSE =
     block (λ(rl,l). rl=[])
           ((do (λ(rl,l).¬(l=[]))
               (assign λ(rl,l).(CONS(HD l)rl,TL l))) seq
            (assign λ(rl,l).rl))");;
```

Internally, this procedure works on a state with two components. However, from the external point of view, the state space has only one global component (a list of arbitrary type). When the loop is exited, the local variable contains the reversed list. This reversed list is copied into the global state component before the local variable is removed. Note that the global state component is at the same time a *reference parameter* of the procedure.

Now we want a call to pREVERSE to give the following result: the state component that is passed as a parameter should be reversed while all other state components should be unchanged.

Imagine that we make the call from a four-component state space and the second component is a list which we want reversed. We indicate this by the *parameter expression*

```
λ(x,y,z,u). y
```

We also have to indicate which state components are unchanged:

```
λ(x,y,z,u). (x,z,u)
```

(it is not possible to define one of these in terms of the other inside HOL). The call is then

```
pcall pREVERSE (λ(x,y,z,u).y) (λ(x,y,z,u).(x,z,u))
```

6.3 Procedure correctness

The definition of the procedure call operator is as follows:

```
⊢  pcall c P R q =
     λu. (∃p. (∀u'. p(P u') ∧ (R u' = R u) ⇒ q u') ∧ c p(P u))
```

Here c is the procedure, P is the parameter expression and R indicates the unchanged part of the state (the Rest).

There is no single theorem that can convincingly show that this captures the idea of a procedure call (it is possible to give a theoretical argument, but it is quite elaborate). However, we have a small ML program (prove_pcall_ref) which proves the correctness of the call for each case separately (it also calculates the R parameter given the P parameter).

To show how this works, we consider the list reversal example again. we assume that we have proved the following theorem (which states that pREVERSE actually reverses a list):

```
⊢  (nondass λl l'. l' = REVERSE l) ref pREVERSE
```

We call this theorem pREVcorrect. We invoke our ML program by typing

```
#prove_pcall_ref pREVcorrect "λ(x,y,z,u).y";;
```

and the result is the following theorem:

```
⊢ (nondass λ(x,y,z,u)(x',y',z',u').
        (y' = REVERSE y) ∧ (x,z,u = x',z',u'))
   ref (pcall pREVERSE (λ(x,y,z,u).y) (λ(x,y,z,u).(x,z,u)))
```

This theorem shows that the procedure call is a command which refines a certain nondeterministic assignment specification. By considering what this specification says, we see that the procedure call in fact reverses the second component of the state space.

6.4 Procedures with value parameters

Reference parameters are handled elegantly by the method shown above. Value parameters can be handled by adding new local variables and storing the values that passed in these variables. However, for this to work, the calling program would have to store each value in a variable before making the call. This is not acceptable, since we want to use expressions as value parameters.

To make this work, we have implemented a separate call to procedures with value parameters. Values are passed using an expression similar to the parameter expression for the reference parameters. In this case, the procedure is a command with an argument, i.e. a function from some type (the type of the value parameter) to commands.

As an example, we consider a procedure which removes all occurrences of a given value from a list. The value is given as a value parameter and the list as a reference parameter. The procedure is defined as follows:

```
⊢ pREMOVE a =
     block (λ(x,l). x=[])
     ((do (λ(x,l).¬(l=[]))
          (cond (λ(x,l). HD l = a)
                (assign λ(x,l).(x,TL l))
                (assign λ(x,l).(SNOC(HD l)x,TL l)))) seq
      (assign λ(x,l).(x,x)))
```

which corresponds to something like

```
procedure pREMOVE(var l:(*)list; a:*);
  var x:(*)list; begin x := [];
  while not(l=[]) do begin
    if HD l = a then
      l := TL l
    else begin
      x,l := SNOC(HD l)x,TL l
    end
  end;
  l := x
end
```

Once we have defined the operation REMOVE in HOL we can prove the correctness theorem

```
⊢ ∀a. (nondass λl l'. l' = REMOVE a l) ref (pREMOVE a)
```

which we assume is called pREMcorrect. Note that pREMOVE can be seen as a family of procedures; one for each possible value of the value parameter a.

Now imagine a three-component state space where the second component is a list of numbers and we want to remove SUC z, where z is the third component. We get the correctness theorem for the procedure call by evaluating

```
#prove_pvcall_ref pREMcorrect "λ(x,y,z).y" "λ(x,y,z).SUC z";;
```

The last argument here is the *value parameter expression* which shows how the value parameter is calculated from the current state.

The correctness theorem is the following:

```
⊢ (nondass (λ(x,y,z)(x',y',z'). (y' = REMOVE(SUC z)y) ∧ (x,z=x',z')))
     ref (pvcall pREMOVE (λ(x,y,z).y) (λ(x,y,z).(x,z)) (λ(x,y,z).SUC z))
```

Here pvcall is the procedure-with-value-call operator. Its definitions is

```
⊢ pvcall c G R V q =
    λu. (∃p. (∀u'. p(G u') ∧ (R u' = R u) ⇒ q u') ∧ c (V u) p (G u))
```

which differs from the definition of pcall in that the first argument to c is the value of the value parameter expression V in the current state.

7 Recursion

Although iteration is generally sufficient for writing arbitrary procedures, a specification using recursion is often much more natural. This section describes how recursion is added into the command notation. We also show that using recursion in functional procedures is not problem-free. The results on recursion are part of Appendix F.

7.1 The recursion operator

It is well known that any monotonic function on a complete lattice has a least fixpoint. Since the commands (predicate transformers) of a given type are a complete lattice, we can define a least fixpoint operator mu:

```
⊢def mu f = Dch(λc. monotonic c ∧ (f c) ref c)
```

(Dch is demonic choice, a greatest lower bound operator on commands). We can now show that mu f is in fact the least fixpoint of the function f, provided that f is *regular*. Regularity is defined as follows:

```
⊢ regular f =
    (∀c. monotonic c ⇒ monotonic(f c)) ∧
    (∀c c'. monotonic c ∧ monotonic c' ∧ c ref c' ⇒ (f c) ref (f c'))
```

i.e., f is regular if it maps monotonic commands to monotonic commands and it is monotonic, viewed as a function on monotonic commands.

The crucial fixpoint theorems are the following:

```
⊢ regular f ⇒ (f(mu f) = mu f)
⊢ ∀c. monotonic c ∧ (f c) ref c ⇒ (mu f) ref c
```

which show that mu is in fact a least fixpoint operator.

As an example, we show how iteration can be expressed in terms of recursion (this is in fact how we have defined iteration; the "definition" shown in Section 2.2 is derived as a theorem):

```
⊢ do g c = mu(λx. cond g (c seq x) skip)
```

7.2 Recursion introduction

The following theorem shows how a specification can be implemented by a recursive construct:

```
⊢ ∀c. regular f ∧ monotonic c ∧
    (∃t.∀i. ((assert(λu.t u=i)) seq c) ref (f((assert(λu.(t u)<i)) seq c)))
    ⇒  c ref (mu f)
```

Assume that we want to refine command c by introducing a recursion. We then start from

```
(assert(λu.t u=i)) seq c
```

where t is a termination function on the state and i is a free variable. Now we try, through a sequence of refinement steps, to end up with a command of the form

```
f((assert(λu.(t u)<i)) seq c))
```

If we succeed, then the above theorem tells us that mu f is in fact a refinement of c.

7.3 Recursive procedures

Our verification methodology works for recursive procedures as well as nonrecursive ones. As an example, we consider a recursive list reversal:

```
⊢ pREV = mu λX.
        block true
          (cond (λ(x,l).l=[]) skip
                ((assign λ(x,l).(HD l,TL l)) seq
                 (pcall X (λ(x,l).l) (λ(x,l).x)) seq
                 (assign λ(x,l).(x,SNOC x l))))
```

(this procedure could well be derived using the method described in Section 7.2, using general refinement laws [1, 5]).

The list reversal procedure declares a local variable to hold the head of the list while the procedure is called recursively to reverse the tail (it bottoms out when the empty list is reached). After the recursive call, the head is added to the end of the reversed tail. Note that the command variable X occurs inside a pcall; this is the recursive call.

In Pascal-style syntax, the procedure could be written as

```
procedure pREV(var l:(*)list);
  var x:*; begin
  if l=[] then
    skip
  else begin
     x,l := HD l,TL l;
     pREV(l);
     x := SNOC x l
  end
end
```

The correctness theorem for this procedure is exactly as for the previous one:

```
⊢  (nondass λl l'. l' = REVERSE l) ref pREV
```

and we get the automatically proved theorems that shows the correctness of the procedure call in the same way (here in a three-component state):

```
⊢ (nondass λ(x,y,z)(x',y',z').(y' = REVERSE y) ∧ (x,z = x',z'))
   ref (pcall pREV (λ(x,y,z).y) (λ(x,y,z).(x,z)))
```

8 Conclusion

We have shown how modular programs can be verified using the HOL system. The language used is a simple Dijkstra-style language with a weakest precondition semantics, extended with procedures and procedure calls.

The idea of embedding simple languages in HOL is not new. We build on earlier work embedding the *refinement calculus* [1, 5] in HOL [2, 4]. However, we are not aware of any work similar to the verification methodology that we propose.

We have assumed post hoc-verification, i.e., that the programmer starts the verification only after the code for a procedure is already written. However, as we indicated in Section 7.3, these procedures can well be developed using refinement methodology. In fact, there is a contribution (a kind of inofficial library) in the HOL system, specifically dedicated to program refinement in HOL.

In modern programming methodology, the importance of modular program structure is often stressed. Large programs are built from a number of smaller procedures that make calls to each other. Since our verification methodology works well for such modular programs, we hope to demonstrate in future case studies that it can be used to verify larger (more than just a few dozen lines of code), nontrivial programs.

References

[1] R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[2] R.J.R. Back and J. von Wright. Refinement concepts formalised in higher-order logic. *Formal Aspects of Computing*, 2:247–272, 1990.

[3] M.J.C. Gordon. Mechanizing programming logics in higher-order logic. In G. Birtwistle and P.A. Subrahmanyam (ed.), *Current Trends in Hardware Verification and Theorem Proving*. Springer-Verlag, 1989.

[4] J. von Wright, J. Hekanaho, P. Luostarinen, T. Långbacka. Mechanising some advanced refinement concepts. *Formal Methods in System Design*, 3:49–81, 1993.

[5] C.C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

Appendix A: A theory of predicates

```
The Theory RCpredicate

Definitions --
  false_DEF  ⊢ false = (λv. F)
  true_DEF  ⊢ true = (λv. T)
  not_DEF  ⊢ ∀p. not p = (λv. ¬p v)
  and_DEF  ⊢ ∀p q. p and q = (λv. p v ∧ q v)
  or_DEF  ⊢ ∀p q. p or q = (λv. p v ∨ q v)
  imp_DEF  ⊢ ∀p q. p imp q = (λv. p v ⇒ q v)
  implies_DEF  ⊢ ∀p q. p implies q = (∀v. p v ⇒ q v)
  glb_DEF  ⊢ ∀P. glb P = (λv. ∀p. P p ⇒ p v)
  lub_DEF  ⊢ ∀P. lub P = (λv. ∃p. P p ∧ p v)
  monotonic_DEF
    ⊢ ∀f. monotonic f = (∀p q. p implies q ⇒ (f p) implies (f q))
  fix_DEF  ⊢ ∀f. fix f = glb(λp. (f p) implies p)
  gfix_DEF  ⊢ ∀f. gfix f = lub(λp. p implies (f p))

Theorems --
  shunt  ⊢ (b and p) implies q = p implies ((not b) or q)
  implies_prop
    ⊢ (∀p. p implies p) ∧
       (∀p q. p implies q ∧ q implies p ⇒ (p = q)) ∧
       (∀p q r. p implies q ∧ q implies r ⇒ p implies r)
  and_glb  ⊢ p and q = glb(λp'. (p' = p) ∨ (p' = q))
  glb_bound  ⊢ ∀P p. P p ⇒ (glb P) implies p
  glb_greatest  ⊢ ∀P q. (∀p. P p ⇒ q implies p) ⇒ q implies (glb P)
  fix_least  ⊢ ∀p. (f p) implies p ⇒ (fix f) implies p
  fix_fp  ⊢ monotonic f ⇒ (f(fix f) = fix f)
  fix_char
    ⊢ monotonic f ∧ (f a) implies a ∧ (∀x. (f x = x) ⇒ a implies x) ⇒
       (a = fix f)
  gfix_greatest  ⊢ ∀p. p implies (f p) ⇒ p implies (gfix f)
  gfix_fp  ⊢ monotonic f ⇒ (f(gfix f) = gfix f)
  gfix_char
    ⊢ monotonic f ∧ a implies (f a) ∧ (∀x. (f x = x) ⇒ x implies a) ⇒
       (a = gfix f)
  or_into_glb  ⊢ q or (glb P) = glb(λp'. ∃p. P p ∧ (p' = q or p))
  glb_and
    ⊢ ∀f g. glb(. ∃p. P p ∧ (q = (f p) and (g p))) =
             (glb(. ∃p. P p ∧ (q = f p))) and
             (glb(. ∃p. P p ∧ (q = g p)))

******************** RCpredicate ********************
```

Appendix B: A theory of commands

```
The Theory RCcommand
Parents --  RCpredicate

Definitions --
  ref_DEF  ⊢ ∀c c'. c ref c' = (∀q. (c q) implies (c' q))
  guard_DEF  ⊢ ∀b q. guard b q = b imp q
  dch_DEF  ⊢ ∀c1 c2 q. (c1 dch c2)q = (c1 q) andd (c2 q)
  Dch_DEF  ⊢ ∀C q. Dch C q = glb(λp. ∃c. C c ∧ (p = c q))
  ach_DEF  ⊢ ∀c1 c2 q. (c1 ach c2)q = (c1 q) or (c2 q)
  Ach_DEF  ⊢ ∀C q. Ach C q = lub(λp. ∃c. C c ∧ (p = c q))
  dolib_DEF
    ⊢ ∀g c q. dolib g c q = gfix(λp. (g or q) andd ((not g) or (c p)))
  skip_DEF  ⊢ ∀q. skip q = q
  assert_DEF  ⊢ ∀g q. assert g q = g andd q
  assign_DEF  ⊢ ∀E q. assign E q = (λv. q(E v))
  nondass_DEF  ⊢ ∀P q. nondass P q = (λv. ∀v'. P v v' ⇒ q v')
  seq_DEF  ⊢ ∀c1 c2 q. (c1 seq c2)q = c1(c2 q)
  cond_DEF
    ⊢ ∀g c1 c2 q.
        cond g c1 c2 q = (g andd (c1 q)) or ((not g) andd (c2 q))
  mu_DEF  ⊢ ∀f. mu f = Dch(λc. monotonic c ∧ (f c) ref c)
  do_DEF  ⊢ ∀g c. do g c = mu(λx. cond g (c seq x) skip)
  strict_DEF  ⊢ ∀c. strict c = (c false = false)
  terminating_DEF  ⊢ ∀c. terminating c = (c true = true)
  biconjunctive_DEF
    ⊢ ∀c. biconjunctive c = (∀p q. c(p andd q) = (c p) andd (c q))
  uniconjunctive_DEF
    ⊢ ∀c.
        uniconjunctive c =
        (∀P. c(glb P) = glb(. ∃p. P p ∧ (q = c p)))
  conjunctive_DEF
    ⊢ ∀c.
        conjunctive c =
        (∀P. (∃p. P p) ⇒ (c(glb P) = glb(. ∃p. P p ∧ (q = c p))))
  pmonotonic_DEF
    ⊢ ∀f.
        pmonotonic f =
        (∀c c'.
          monotonic c ∧ monotonic c' ∧ c ref c' ⇒ (f c) ref (f c'))
  mono_mono_DEF
    ⊢ ∀f. mono_mono f = (∀c. monotonic c ⇒ monotonic(f c))
  regular_DEF  ⊢ ∀f. regular f = pmonotonic f ∧ mono_mono f
```

```
Theorems --
  ref_prop
    ⊢ (∀c. c ref c) ∧
       (∀c c'. c ref c' ∧ c' ref c ⇒ (c = c')) ∧
       (∀c c' c''. c ref c' ∧ c' ref c'' ⇒ c ref c'')
  mono_Dch  ⊢ (∀c. C c ⇒ monotonic c) ⇒ monotonic(Dch C)
  Dch_bound  ⊢ ∀C c. C c ⇒ (Dch C) ref c
  Dch_greatest  ⊢ ∀C c. (∀c'. C c' ⇒ c ref c') ⇒ c ref (Dch C)
  mono_mu  ⊢ ∀f. monotonic(mu f)
  mu_least  ⊢ ∀c. monotonic c ∧ (f c) ref c ⇒ (mu f) ref c
  mu_fp  ⊢ regular f ⇒ (f(mu f) = mu f)
  mu_char
    ⊢ regular f ∧
       monotonic a ∧
       (f a) ref a ∧
       (∀x. (f x = x) ⇒ a ref x) ⇒
       (a = mu f)
  do_thm
    ⊢ ∀c.
        monotonic c ⇒
        (do g c q = fix(λp. (g andd (c p)) or ((not g) andd q)))
  do_expand
    ⊢ monotonic c ⇒
       (do g c p = (g andd (c(do g c p))) or ((not g) andd p))
  do_implies
    ⊢ monotonic c ⇒
       ((g andd (c q)) or ((not g) andd p)) implies q ⇒
       (do g c p) implies q
  seq_assoc  ⊢ c1 seq (c2 seq c3) = (c1 seq c2) seq c3
  mono_subuniconj
    ⊢ monotonic c ⇒
       (c(glb P)) implies (glb(. ∃p. P p ∧ (q = c p)))
  conj_biconj  ⊢ conjunctive c ⇒ biconjunctive c
  conj_mono  ⊢ conjunctive c ⇒ monotonic c
  uniconj_conjterm  ⊢ uniconjunctive c = conjunctive c ∧ terminating c
  uniconj_conj  ⊢ uniconjunctive c ⇒ conjunctive c
  uniconj_mono  ⊢ uniconjunctive c ⇒ monotonic c
  mono_skip  ⊢ monotonic skip
  mono_assert  ⊢ ∀b. monotonic(assert b)
  mono_assign  ⊢ ∀e. monotonic(assign e)
  mono_nondass  ⊢ ∀p. monotonic(nondass p)
  mono_seq
    ⊢ ∀c c'. monotonic c ∧ monotonic c' ⇒ monotonic(c seq c')
  mono_cond
    ⊢ ∀g c1 c2.
        monotonic c1 ∧ monotonic c2 ⇒ monotonic(cond g c1 c2)
  mono_do  ⊢ ∀g c. monotonic c ⇒ monotonic(do g c)
  do_dolib
    biconjunctive c ⊢ do g c q = (dolib g c q) andd (do g c true)
  conj_skip  ⊢ conjunctive skip
  conj_seq
    ⊢ ∀c c'. conjunctive c ∧ conjunctive c' ⇒ conjunctive(c seq c')
  conj_do  ⊢ ∀g c. conjunctive c ⇒ conjunctive(do g c)
  nondass_complete
    ⊢ uniconjunctive c ⇒ (c = nondass(λu u'. glb(. c q u)u'))

******************** RCcommand ********************
```

Appendix C: A theory of well-founded sets

```
The Theory RCwellf
Parents --  HOL

Definitions --
  order_DEF
    ⊢ ∀po.
        order po =
        (∀x y. po x y ⇒ ¬(x = y)) ∧
        (∀x y z. po x y ∧ po y z ⇒ po x z)
  minimal_DEF  ⊢ ∀x M po. minimal x M po = M x ∧ (∀y. M y ⇒ ¬po y x)
  wellf_DEF
    ⊢ ∀po.
        wellf po = order po ∧ (∀M. (∃x. M x) ⇒ (∃x. minimal x M po))

Theorems --
  wellf_INDUCT
    wellf po ⊢ (∀x. (∀y. po y x ⇒ P y) ⇒ P x) ⇒ (∀x. P x)
  strong_induct  ⊢ ∀P. (∀n. (∀m. m < n ⇒ P m) ⇒ P n) ⇒ (∀n. P n)
  num_wellf  ⊢ wellf $<

******************** RCwellf ********************
```

Appendix D: A theory of total correctness

```
The Theory RCcorrect
Parents --  RCcommand    RCwellf

Definitions --
  correct_DEF  ⊢ ∀p c q. correct p c q = p implies (c q)

Theorems --
  correct_assign  ⊢ (∀v. p v ⇒ q(e v)) ⇒ correct p(assign e)q
  correct_seq
    ⊢ ∀c1 c2 p q r. monotonic c1 ∧ correct p c1 q ∧ correct q c2 r ⇒
        correct p(c1 seq c2)r
  correct_cond
    ⊢ ∀g c1 c2 p q. correct(g and p)c1 q ∧ correct((not g) and p)c2 q ⇒
        correct p(cond g c1 c2)q
  wellf_do_inv_rule
    ⊢ monotonic c ∧
       (∃po inv t. wellf po ∧ p implies inv ∧
         ((not g) and inv) implies q ∧
         (∀x. correct (inv and (g and (λu. t u = x)))
                      c
                      (inv and (λu. po(t u)x)))) ⇒
       correct p(do g c)q
  num_do_inv_rule
    ⊢ monotonic c ∧
       (∃inv t. p implies inv ∧ ((not g) and inv) implies q ∧
         (∀x. correct (inv and (g and (λu. t u = x)))
                      c
                      (inv and (λu. (t u) < x)))) ⇒
       correct p(do g c)q
  impl_assign
    ⊢ monotonic c ∧ (∀u0. correct(λu. u = u0)c(λv. v = e u0)) ⇒
       (assign e) ref c
  assert_intro
    ⊢ ∀g c.
        conjunctive c ∧ correct true c g ⇒ (c = c seq (assert g))

******************** RCcorrect ********************
```

Appendix E: A theory of functional abstraction

```
The Theory RCfunction
Parents --  RCcorrect

Definitions --  fcall_DEF  ⊢ ∀c. fcall c = (λu. εv. glb(. c q u)v)

Theorems --
  assign_call  ⊢ fcall(assign e) = e
  fcall_thm
    ⊢ ∀c f.
        conjunctive c ∧
        strict c ∧
        (∀u0. correct(λu. u = u0)c(λv. v = f u0)) ⇒
        (fcall c = f)
  fcall_rule
    ⊢ ∀c f.
        conjunctive c ∧ strict c ∧ (assign f) ref c ⇒ (fcall c = f)
  fcall_thm_pre
    ⊢ ∀g c f.
        conjunctive c ∧
        strict c ∧
        (∀u0. g u0 ⇒ correct(λu. u = u0)c(λv. v = f u0)) ⇒
        (∀u. g u ⇒ (fcall c u = f u))
  fcall_rule_pre
    ⊢ ∀g c f.
        conjunctive c ∧ strict c ∧ ((assert g) seq (assign f)) ref c ⇒
        (∀u. g u ⇒ (fcall c u = f u))

******************** RCfunction ********************
```

Appendix F: A theory of procedures and recursion

```
The Theory RCprocrec
Parents --  RCcorrect

Definitions --
  pcall_DEF
    ⊢ ∀c G R q u.
        pcall c G R q u =
        (∃p. (∀u'. p(G u') ∧ (R u' = R u) ⇒ q u') ∧ c p(G u))
  pvcall_DEF
    ⊢ ∀c G R V q u.
        pvcall c G R V q u =
        (∃p. (∀u'. p(G u') ∧ (R u' = R u) ⇒ q u') ∧ c(V u)p(G u))

Theorems --
  mu_thm
    ⊢ ∀f c.
        regular f ∧
        monotonic c ∧
        (∃t. ∀i.
          ((assert(λu. t u = i)) seq c) ref
          (f((assert(λu. (t u) < i)) seq c))) ⇒
        c ref (mu f)
  regular_const  ⊢ ∀c. monotonic c ⇒ regular(λx. c)
  regular_id  ⊢ regular(λx. x)
  regular_seq
    ⊢ ∀f f'. regular f ∧ regular f' ⇒ regular(λx. (f x) seq (f' x))
  regular_cond
    ⊢ ∀g f f'.
        regular f ∧ regular f' ⇒ regular(λx. cond g(f x)(f' x))
  pcall_mono
    ⊢ ∀c c'. c ref c' ⇒ (∀G R. (pcall c G R) ref (pcall c' G R))
  regular_pcall  ⊢ ∀G R. regular(λx. pcall x G R)
  pvcall_mono
    ⊢ ∀c c'.
        (∀a. (c a) ref (c' a)) ⇒
        (∀G R V. (xpcall c G R V) ref (xpcall c' G R V))
******************** RCprocrec ********************
```