Decomposing the DSub Retrenchment

J.G.Hall
T.Gurukumba

2001

Department of Computing
Faculty of Mathematics and Computing
The Open University
Walton Hall,
Milton Keynes
MK7 6AA
United Kingdom

http://computing.open.ac.uk
Decomposing the $DSub$ Retrenchment

J.G. Hall†, T. Gurukumba†

3rd May 2001

†Computing Department, Faculty of Mathematics and Computing,
The Open University, Walton Hall,
Milton Keynes, MK7 6AA, UK

1 Introduction

The use of software has become prevalent in high integrity systems such as safety critical systems. The correctness of such software, i.e. the delivery of a proper service that adheres to specified requirements, is a fundamental issue.

Formal methods, which is a software development technique based on mathematics, addresses the issue of correctness of software. In the formal development of computer programs, a correctness-preserving transformation such as refinement [10, 11, 1] may be used. Functional correctness is preserved by means of data refinement as well as algorithmic refinement. ¹

One limitation of refinement is that it only works for operations of the same signature. In particular, in the concrete operation, some state variables may become input variables (e.g., when the variables denote values from some prior computations), some output variables may be added to the operations, (e.g., in order to totalise operations [10]), and some variables may be constrained to a particular type (e.g., due to the finiteness of computer representation). In that case the signature of the concrete operation would be different from that of the abstract operation. The above techniques above introduce the difference in signatures between abstract and concrete operations as a side effect but may be necessary in maintaining safe functionality in the concrete operation.

Retrenchment is a liberalized form of refinement [2], that can be used to reason about functional correctness-preservation where operations may have different signatures. The Operation Retrenchment Proof Obligation [3] is a predicate that characterizes the retrenchment of one operation by another. The proof obligation has slots for various characteristics of a particular retrenchment, e.g., invariants, preconditions, operations, of the two machines involved in the retrenchment. Proving this proof obligation may provide structured proofs in terms of the proof steps required to discharge the operation retrenchment proof obligation. In particular, patterns in proofs can be coded into proof tactics enabling proofs to be automated.

Automation enables the provision of software tools which may make the application of formal methods tractable, scalable and less prone to human error. Poppleton and Banach have worked on extending the B-Method to handle retrenchment [15]. The B-Toolkit is a software tool that

¹The B-method of [1] involves both data refinement (the abstract data types are refined to concrete data types available in programming languages) and algorithmic refinement (the abstract algorithms/operations are refined to executable code). Programs are viewed as machines – the abstract machine is the program to be refined, and the program that refines the abstract machine is called the concrete machine. The words program and machine will be used interchangeably in this document.
supports the B-Method. However the proof strategies for the B-Method have been reported to be poor [4].

This report explores retrenchment in PVS, using a numerical example of a B-machine that performs a subtraction algorithm. Section 2 examines the specification and nature of the retrenchment in the B-method. Section 3 looks at the specification in PVS – a specification and verification system with ample facilities for constructing proof tactics. Section 4 describes a tactic for proving operation retrenchment proof obligations in PVS. Section 5 concludes with some pointers to further work.

2 B Specification

Consider a divine machine with state variables $a$ and $b$, and an operation $DSub$ which, subtracts $b$ from $a$ and puts the result in $a$, provided $a \geq b$. The mundane machine has a state variable $a$, and a corresponding operation, $MSub$, which subtracts an input $bb$, from $aa$, and puts the result in $aa$, but does so only for $aa$ and $bb$ less than some threshold, $OF$, i.e., $MSub$ performs the subtraction and signals success when $(aa < OF \land aa \geq bb)$, otherwise $MSub$ signals that a subtraction is not possible.

The $DSub$ retrenchment is of the form shown in Figure 1.

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>DivineSub</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARIABLES</td>
<td>$a$, $b$</td>
</tr>
<tr>
<td>INVARIANT</td>
<td>$a \in \mathbb{N}$ $\land b \in \mathbb{N}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>MundaneSub</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARIABLES</td>
<td>$aa$</td>
</tr>
<tr>
<td>INVARIANT</td>
<td>$aa \in \mathbb{N}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INITIALISATION</th>
<th>$X(u)$</th>
<th>INITIALISATION</th>
<th>$Y(v)$</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>OPERATIONS</th>
<th>$DSub =$</th>
<th>OPERATIONS</th>
<th>$resp \leftarrow MSub(bb) =$</th>
</tr>
</thead>
</table>
| BEGIN | $a \geq b \mid a := a - b$ | BEGIN | $bb \in \mathbb{N} \mid$
| | | | $aa < OF \land aa \geq bb \Rightarrow$
| | | | $aa := aa - bb \mid resp := done$
| | | | $[]$
| | | | $aa = OF \lor aa < bb \Rightarrow$
| | | | $aa := aa \land resp := fail$
| | | | LVAR $A$
| | | | WITHIN
| | | | $A = a \land (bb < OF \Rightarrow b = bb)$
| | | | CONCEDES
| | | | $(resp = done \Rightarrow a = aa)$
| | | | $\land (resp = fail \Rightarrow a = A)$
| | | | END
| | | | END

Figure 1: One-off Retrenchment
Looking at the retrenching machine, \textit{MundaneSub}, we can see three components of two different forms of the retrenchment. The first is the architectural translation of a machine with no inputs, no outputs, that manipulates the state to one with one operation which has one input, no outputs, and manipulates state. We may factor out this architectural manipulation to give the retrenchment shown in Figure 2.

\begin{verbatim}
MACHINE DivineSub
VARIABLES a, b
INVARIANT a ∈ N ∧ b ∈ N
INITIALISATION X(u)
OPERATIONS
DSub =
BEGIN
  a ≥ b | a := a - b 
END

MACHINE ArchSub
RETRENCHES DivineSub
VARIABLES aa
INVARIANT aa ∈ N
INITIALISATION Y(v)
OPERATIONS
AMS Sub(bb) =
BEGIN
  bb ∈ N ∧ aa ≥ bb |
  aa := aa - bb
LVAR A WITHIN
  b = bb
END

END
\end{verbatim}

Figure 2: First Architectural Translation

This leaves a representation change, retrenching from divine natural numbers, \(\mathbb{N}\), to mundane/finite natural numbers, \(\text{FIN}\), as shown in Figure 3. This brings in with it a requirement to express how felicitous the retrenchment is.

The felicity of the retrenchment is expressed via the WITHIN and CONCEDES clauses. In particular we have a logical variable \(A\) that holds those values of the abstract machine that are not feasible in the concrete machine. This is essential since the mechanics of refinement are that the RETRIEVES relation holds in both the before-state and the after-state. Since the concrete machine is only finite whereas the abstract machine is not, there are values that can be represented in the abstract machine that are not representable in the concrete machine. These values of the abstract machine are held in the logical variable \(A\) in the retrenchment. The CONCEDES clause denotes where the RETRIEVE relation holds:

\[(aaa < OF ⇒ aa = aaa)\]

and where it does not:

\[(aaa ≥ OF ⇒ aa = A)\]
Figure 3: Representational Translation

A second architectural translation, Figure 4, then adds the response variable.
MACHINE              \text{RepArchSub} \quad \text{MACHINE}              \text{ArchRepArchSub} \\
VARIABLES             aaa \quad \text{RETRENCHES} \text{RepArchSub} \\
INVARIANT             aaa \in \mathbb{FN} \quad \text{VARIABLES}            aaaa \\
\text{INITIALISATION} X(u) \quad \text{INVARIANT}            aaaa \in \mathbb{FN} \\
\text{OPERATIONS}     \text{RETIRES} \quad (aaa < OF \Rightarrow aaa = aaaa) \quad \wedge (aaa = OF \Rightarrow aaa \geq OF) \\
AMSub = \quad \begin{array}{l}
\text{BEGIN} \\
bbb \in \mathbb{FN} | \\
\quad aaa < OF \land aaa \geq bbb \Rightarrow \\
\quad \quad aaa := aaa - bbb \\
\quad \left[ \right] \\
\quad aaa = OF \lor aaa < bbb \Rightarrow \\
\quad \quad aaa := aaa \\
\text{END}
\end{array} \\
MSub(bb) = \quad \begin{array}{l}
\text{BEGIN} \\
\quad bbbb \in \mathbb{FN} | \\
\quad aaaa < OF \land aaaa \geq bbbb \Rightarrow \\
\quad \quad aaaa := aaaa - bbbb \parallel \text{resp} = done \\
\quad \left[ \right] \\
\quad aaaa = OF \lor aaaa < bbbb \Rightarrow \\
\quad \quad aaaa := aaaa \parallel \text{resp} := \text{fail} \\
\text{LVAR} A \\
\text{WITHIN} \\
\quad A = aaaa \land (bbb < OF \Rightarrow bbb = bbbb) \\
\quad \wedge (bbb = OF \Rightarrow bbb \geq OF) \\
\text{CONCEDES} \\
\quad (aaa < OF \Rightarrow aaa = aaaa \wedge \text{resp} = done) \\
\quad \wedge (aaa \geq OF \Rightarrow aaa = A \wedge \text{resp} = \text{fail}) \\
\text{END}
\end{array}

Figure 4: Second Architectural Translation

Finally the CONCEDES clause can be manipulated to give:

\text{CONCEDES} \\
\quad (\text{resp} = \text{done} \Rightarrow aaa = aaaa) \land (\text{resp} = \text{fail} \Rightarrow aaa = A)

2.1 The nature of an architectural change

An architectural change in general will not be a refinement; the obvious instance is when state variables are moved to input, and/or output variables are introduced to indicate exceptional behaviour. For an architectural change to occur, we:

- need to characterize the architectural change;
- need to alter the RETRIEVE, LVAR, WITHIN, and CONCEDES clause to reflect this.

From the above, we see that the first architectural change (Figure 2) is:

- alter the signature to change state variable $b$ to input $bb$. The architecture (signature) of the function changes (as a result of function currying) from:
  \[ DSub \equiv DSub :: (N, \mathbb{N}) \to \mathbb{N} \]
to:
\[ AMSub(bb) \equiv AMSub : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \]

- change the operation body to:
  \[ bb \in \mathbb{N} \land aa \geq bb \mid aa := aa - bb \]

- The changes to the individual clauses are:
  - RETRIEVES becomes \( aa = a \), i.e. identity
  - LVAR remains empty
  - WITHIN becomes \( bb = b \), i.e. constraining the input to the state variable.
  - CONCEDES becomes false.

From the above we see that the second architectural change (Figure 4) is:

- Alter the signature to introduce output resp, i.e. change the operation to:
  \[ resp \leftarrow MSub(bb) \equiv MSub : \mathbb{F}\mathbb{N} \rightarrow \mathbb{F}\mathbb{N} \rightarrow \mathbb{F}\mathbb{N} \rightarrow Response \]

- The changes to the individual clauses are:
  - RETRIEVES becomes \( aaaa = aaa \), which remains the same identity relation.
  - LVAR remains A
  - WITHIN
    \[ A = aaa \land (bbbb < OF \Rightarrow bbb = bbbb) \land bbbb = OF \Rightarrow bbb \geq OF \]
  - CONCEDES introduces the \( resp \) variable:
    \( \langle resp = done \Rightarrow aaa = aaaa \rangle \land \langle resp = fail \Rightarrow aaa = A \rangle \)

### 2.2 The nature of a type change

A type change, in general, will not be a refinement: the obvious instance is when an infinite type is implemented as a finite type – in the above \( \mathbb{N} \) becomes \( \mathbb{F}\mathbb{N} \) (Figure 3). For a type change to occur we:

- Need to identify variables whose types may change.
- For each, we need to characterize the type change(s), i.e. to define the relationship between abstract and concrete variables – RETRIEVES relation.
- Need to alter the RETRIEVE, LVAR, WITHIN, and CONCEDES clauses to reflect this.

From the above we see that the change from \( \mathbb{N} \) to \( \mathbb{F}\mathbb{N} \) is:

- variables which change are \( aa : \mathbb{N} \) and \( bb : \mathbb{N} \), which change to \( aaa : \mathbb{F}\mathbb{N} \) and \( bbb : \mathbb{F}\mathbb{N} \).
- described by the relation:
  \[ aa R aaa \Rightarrow aaaa < OF \Rightarrow aaa = aa \land aaaa = OF \Rightarrow aa \geq OF \]

The changes to the individual clauses are:
• RETRIEVES becomes \( aa R aaa \), i.e.
\[
aaa < OF \Rightarrow aaa = aa \land aaa = OF \Rightarrow aa \geq OF
\]

• LVAR becomes \( A \).

• WITHIN becomes:
\[
A = aa \land (bbb < OF \Rightarrow bb = bbb) \\
\land bbb = OF \Rightarrow bb \geq OF
\]

• CONCEDES indicates the fidelity of the retrenchment:
\[
(aaa < OF \Rightarrow aa = aaa) \land (aaa \geq OF \Rightarrow aa = A)
\]
i.e. the type change introduces an exception condition which is when the types condition is not exact.

The change of representation (of data types) also entails that the algorithm used in the abstract machine may be different from that used in the concrete machine because the two operations return different types. In this regard, note that the requirement \( aa \geq bb \) is moved from the precondition to a guard. In fact all of the precondition can be moved to the guard, with appropriate accompanying architectural transformations. This alleviates the user from having some prior knowledge about the operation of the program – the precondition is a predicate detailing what the user must ensure in order that the program terminates successfully, i.e. with the correct output.

2.3 Sequential Composition of Retrenchments

Sequential Composition of Retrenchments:

\[
\begin{array}{ccc}
DSub & \overset{\text{oneoff\_ret}}{\longrightarrow} & ArchRepArchSub \\
\downarrow_{\text{arch\_transl1}} & & \uparrow_{\text{arch\_transl2}} \\
ArchSub & \overset{\text{rep\_transl}}{\longrightarrow} & ArchRepSub
\end{array}
\]

The diagram above depicts the fact that an architectural translation, \( \text{arch\_transl1} \), followed by a representational translation, \( \text{rep\_transl} \), which in turn is followed by another architectural translation, \( \text{arch\_transl2} \) should have the same overall effect as a one-off retrenchment, \( \text{oneoff\_ret} \). This effect is characterised by the retrenchment proof obligation, i.e., for every step the retrenching machine takes, there is a step that the abstract machine takes such that the retrieve relation or the CONCEDES clause is established.

3 PVS specification and proof

The style of PVS specification is adapted from that of PBS [12]. PBS uses the B-Method model of specification – B’s Abstract Machine Notation, as well as PVS statements are used to specify programs. Like the B-Method, PBS is based on Set Theory. On the other hand PVS is based on Lambda Calculus. Sets in PVS are defined as functions of type \( (\text{TYPE} \rightarrow \text{bool}) \), where \( \text{TYPE} \) is the type of the elements of the set. PBS uses subtyping to generate sets that can be used to reason about whether a machine is a refinement of another – for any two total relations \( R \) and \( S \), \( R \) refines \( S \) if \( R \) is a subset of \( S \) [10]:

\[
R \sqsubseteq S \iff R \subseteq S
\]
Thus refinement proof obligations using the PBS-Method can be automatically generated as type correctness conditions (TCCs) that the type of refining machine (the refiner) is a subset of the machine refined (the refiner), by using the PVS typechecker. The PVS tactic for proving TCCs, tcps, can be used to automatically prove those TCCs that can be proven with it. Munoz is working on tactics to prove those refinement TCCs that cannot be proved using tcp [12]. Refinement in the B-Method, and thus PBS, is only defined for machines with operations with the same signature [9]. Thus as is, PBS cannot be used to reason about refinement. Work has been done to extend the B-Method to reason about refinement [2]. However, the B-Toolkit is not easily extensible in terms of proof tactics which enable automation of proofs. PVS has its own tactic language, enabling proof tactics to be coded.

The main proof obligation is the refinement proof obligation [2]:

\[
\begin{align*}
\text{Operation Refinement Proof Obligation:} \\
& \frac{(I(u) \land G(u, v) \land J(v) \land \text{trm}(T)(v, j) \land P(i, j, u, v, A))}{\Rightarrow (\text{trms}(S)(u, i) \land \text{trm}(T)(v, j)} \\
& \quad \Rightarrow (\forall \bar{v}, \bar{p} \cdot \text{stp}(T)(v, j, \bar{v}, \bar{p})} \\
& \quad \Rightarrow (\exists \bar{u}, \bar{o} \cdot \text{stp}(S)(u, i, \bar{u}, \bar{o})} \\
& \quad \land (G(\bar{u}, \bar{v}) \lor C(\bar{u}, \bar{v}, \bar{o}, \bar{p}, A))))
\end{align*}
\]

Where \(u, v\) are state variables, \(o, p\) are output variables and \(i, j\) are input variables; \(\bar{u}, \bar{o}, \bar{v}, \bar{p}\) are the expressions or values to which state variables \(u, o, v, p\) are assigned to in the operation, respectively. In other words, \(\bar{u} = u', \bar{o} = o', \bar{v} = v', \bar{p} = p'\) where \(u', o', v', p'\) are the variables after the operation has completed, i.e. \(u', o', v', p'\) are the after-state variables; \(u, o, v, p\) are the before-state variables. For the abstract operation \(\text{stp}(S)(u, i, \bar{u}, \bar{o})\), \(I(u)\) is the invariant, \(\text{trm}(S)(u, i)\) is the terminating condition or precondition. For the concrete operation \(\text{stp}(T)(v, j, \bar{v}, \bar{p})\), \(J(v)\) is the invariant, \(\text{trm}(T)(v, j)\) is the precondition. For the fidelity of the refinement, \(P(i, j, u, v, A)\) is the conjunctive \text{WITHIN} clause which relates state variables, input, and logical variables \(A\), in the before-state thus strengthening the precondition, \(G(u, v)\) is the retrieve relation for state variables, \(C(\bar{u}, \bar{v}, \bar{o}, \bar{p}, A)\) is the disjunctive \text{CONCEDES} clause which relates state and logical variables in the after-state thus weakening the postcondition.

The specification of operations in PBS returns record types of variables. For the refinement proof obligation above to be type-correct, the operations should return boolean values. This is achieved by the equality [3]:

\[
\text{Operations as predicates:} \\
\text{stp}(S)(u, i, u', o) = \text{trm}(S)(u, i) \land \text{prd}(S)(u, i, u', o)
\]

Where \(\text{trm}(S)(u, i)\) is the precondition, and \(\text{prd}(S)(u, i, u', o)\) is the postcondition. The postcondition, is constructed by asserting that the after-state of the machine is equal to the state returned by the operation on its completion. The operations themselves are specified using an imperative, i.e. destructive, style of programming by using assignment. PVS provides the following imperative constructs:

- **assignment** – :=

- **choice** – IF-THEN-ELSE, CASES
• *iteration* which can be simulated by recursion – (\texttt{<operation>}: \texttt{RECURSIVE}: \texttt{<TYPE>}). The \texttt{MEASURE} function provides a counter utility which can be exploited to terminate the recursion appropriately.

• *sequence* which can be simulated by functional composition or function application. This is in the case that the results of a prior operation are used in a subsequent operation. Explicit control of execution cannot be handled by PVS.

Thus this may provide a mechanism for checking the logical correctness of the imperative-like code – the operation specified using imperative constructs is carried out, and it is the after-state of the operation which is used in the operation retrenchment proof obligation. Starting with a correct divine specification, we can thus proceed towards a correct mundane specification that preserves the functional correctness of the divine specification, by proving that the two specifications satisfy the retrenchment proof obligation. This may help eradicate logical errors in program development, which have been found to account for the greatest percentage of faults in a software development process [8].

### 3.1 Specification

The PVS specification corresponding to the first architectural translation shown in Figure 2, with its accompanying retrenchment proof obligation is shown in Figure 5.

### 3.2 Proof

PVS provides a typechecker for ensuring that the types of expressions are consistent throughout the specification. The type-correctness-conditions (TCCs) generated by typechecking the above PVS specification, Figure 2, are shown in Figure 6. The PVS tactic for proving TCCs, \texttt{tcp}, was used to prove the TCCs.

```plaintext
% Subtype TCC generated (at line 17, column 15) for  a(u) - b(u) % proved - complete stps_TCC1: OBLIGATION FORALL (u: Dvars | trmS(u)): u' a - u' b >= 0;

% Subtype TCC generated (at line 19, column 36) for u % unproved Qs_TCC1: OBLIGATION FORALL (u): trmS(u);

% Subtype TCC generated (at line 35, column 16) for  aa(v) - bb % proved - complete stpt_TCC1: OBLIGATION FORALL (v: Avars, bb: nat | trmT(v, bb)): v' aa - bb >= 0;

% Subtype TCC generated (at line 37, column 53) for bb % unfinished Qt_TCC1: OBLIGATION FORALL (bb: nat, v: Avars): trmT(v, bb);
```

Figure 6: TCCs generated for the first architectural translation
ArchSub: THEORY BEGIN

%%%====================================================================
%%%DivineSub
%%%------

%%%General Type
Dvars: TYPE = [\# a: nat, b: nat \#]

I(u: Dvars): bool = member(a(u), \{x:nat\true\}) AND member(a(u), \{x:nat\true\})

trmS(u: Dvars): bool = (a(u) >= b(u))

stpS(u: Dvars | trmS(u)): Dvars =
  LET u = u WITH
  [ a := a(u) - b(u)] IN u

QS(u, up: Dvars): bool = (up = stpS(u))

%%%====================================================================
%%%ArchSub
%%%------

Avars: TYPE = [\# aa: nat \#]

J(v: Avars): bool = member(aa(v), \{x:nat\true\})

G(u: Dvars, v: Avars): bool = (a(u) = aa(v))

trmT(v: Avars, bb: nat): bool = (aa(v) >= bb AND member(bb, \{x:nat\true\}))

stpT(v: Avars)(bb: nat | trmT(v, bb)): Avars =
  LET v = v WITH
  [ aa := aa(v) - bb] IN v

QT(v:Avars, bb: nat, vp: Avars): bool = (vp = stpT(v)(bb))

P(u: Dvars, v: Avars, bb:nat): bool = (b(u) = bb)

C(up: Dvars, vp: Avars): bool = (FALSE)

%====================================================================

%OPERATION RETRENCHEMENT PROOF OBLIGATION
%------------------------------------------

ArchSub: THEOREM

FORALL (u: Dvars, v: Avars, bb: nat):
  ((I(u) AND J(v) AND G(u, v) AND trmT(v, bb) AND P(u, v, bb))
  => (trmS(u) AND trmT(v, bb)
    AND (trmS(u)
    => (FORALL (vp: Avars): trmT(v, bb) AND QT(v, bb, vp)
    => (EXISTS (up: Dvars): trmS(u) AND QS(u, up)
      AND (G(up, vp) OR C(up, vp))))))

END ArchSub
• \textit{stpS\_TCC1} entails that the divine operation \textit{stpS} satisfies its invariant \textit{I}, i.e., after the operation, \(a \in \mathbb{N}\). This proves, therefore \(Dsub\) satisfies its invariant.

• \textit{QS\_TCC1} is the proof that the divine machine terminates. The proof remains unfinished because there is no information in the specification to the effect that \(a(u) \geq b(u)\). Such information may be provided by performing an initialisation step, and then passing the initial state to the operation, \(\text{stpS}(u: \text{Dvars} \mid \text{trm}(S(u)))\), as the record variable \(u\). This would conform to the meaning of a program specification – if the initial state satisfies the precondition then change the frame so that the resulting final state satisfies the precondition [11]. However the initial state only corresponds to one such initialisation. In a proof we are interested in all the possible initialisations that may occur. There is no proof that for any \(a\) and \(b\), \(a \geq b\).

• \textit{stpT\_TCC1} entails that the mundane operation, \textit{stpT}, satisfies its invariant, \(J\). This proves and so \(AMS\_sub\) satisfies its invariant.

• \textit{QT\_TCC1} is the proof that the mundane machine terminates. This fails for the same reason as \textit{QS\_TCC1}.

Note that the retrenchment proof obligation only requires that the after-state of the retrenching and retrenched operations satisfy the retrieve relation or concedes clause. The refinement proof obligation requires that the after state also satisfies the invariant of the refining machine:

\[
\begin{align*}
\text{Operation Refinement Proof Obligation:} \\
(\mathit{f}(u) \wedge \mathit{G}(u, v) \wedge \mathit{J}(v) \wedge \text{trm}(S)(u, i)) \\
\Rightarrow (\text{trm}(T)(u, j) \\
\wedge (\text{trm}(S)(u, i)) \\
\Rightarrow (\exists \, u', p' \cdot \text{stp}(T)(u, j, u', p') \\
\Rightarrow (\exists \, u', o' \cdot \text{stp}(S)(u, i, u', o') \\
\wedge (\mathit{G}(u', o') \wedge \mathit{J}(u')))))
\end{align*}
\]

The retrenchment proof obligation in Figure 5 does not prove with even PVS’s most powerful tactic \textit{grind}. This is because \textit{grind} uses automatic instantiation, and the instantiation it chooses is not what is required. In particular, the mechanics of the retrenchment proof obligation says that for every concrete operation step there is an abstract operation step which establishes the retrieve relation and the concedes clause. This signifies that both machines have done some computation and it is required that the after states of the two machines establish the retrieve relation. Now instead of automatically instantiating the after-state of the abstract machine, automatic instantiation instantiates with the before-state of the abstract machine. This means that whereas the concrete machine has progressed, the abstract has not, and as such the after-state of the two machines will not establish the required retrieve relation.

The manual proof shown in Figure 7 instantiates correctly. The instantiation is by the variables that are returned by the operation of the retrenched machine. This is achieved by invoking the operation to be retrenched using the corresponding skolemised parameters, i.e. \(\text{stpS}(u!1)\).
Figure 7: Proof of the first Architectural Translation

The representational translation, Figure 3, was specified in a similar manner as in Figure 5 and proved by the same proof steps as above, albeit with a modified instantiation, where u!1 are the skolem constants of the before-state variables of ArchSub, i.e. u!1 = [# aa!1: nat #] :

(INstantiate + "stpS(u!1)(bb!1)")

The second architectural translation was also specified and proved by the same basic steps as in Figure 7 above, with a modification to the instantiation, where u!1 = [# aaa!1: nat #] :

(INstantiate + "stpS(u!1)(bbb!1)")

The one-off retrenchment as depicted in Figure 1, proved with the same proof as shown in Figure 7.

Thus the same proof pattern is used throughout to prove the four retrenchments. If such a proof pattern can be encoded into a tactic, then the proof would be automated. Proving a similar retrenchment would then involve just calling the tactic.

4 A tactic for proving the operation retrenchment proof obligation

We see that the proof pattern follows the same basic format up to the instantiation. The instantiation step for each proof is different – it is determined by the nature of the operation that is being retrenched. After the instantiation, the proof steps are the same again.

4.1 Reducing to instantiation phase

The tactic shown in Figure 8 simplifies the retrenchment proof obligation to the point where instantiation can be evoked.
(defstep reduce-po
( )
(then
(skolem!)
(prop)
(try
(try
(grind)
(fail)
(skip))
(skip)
(then
(skolem!)
(flatten)))))

"Reduces the retrenchment proof obligation to the instantiation stage."
"Reducing RetPU to instantiation stage"

Figure 8: The tactic reduce-po

The tactic is defined in terms of predefined PVS strategies and basic rules. The predefined strategies used are [13]:

- **then**, a sequencing strategy. It takes a list of steps consisting of step1 and the remaining steps rest-steps. The first step is applied to the current goal. If any subgoals are generated, then (then : steps) rest-steps is applied to each of those subgoals. If step1 has no effect, then (then : steps) rest-steps is applied to the original goal.

- **try**, a basic control strategy for subgoaling and backtracking. It takes three arguments, step1, step2, step3. If step1 succeeds and generates subgoals, then step2 is applied to those subgoals. Otherwise step3 is applied to the current goal.

The predefined basic rules are:

- **skolem!**, which automatically eliminates quantifiers by providing skolem constants.
- **prop** is used for propositional simplification
- **fail** is used to trigger backtracking from a subgoal that has failed to prove.
- **skip** does nothing
- **flatten** applies disjunctive simplification.

The predefined PVS tactic **grind** is the most powerful of the predefined proof steps. It expands definitions, and tries repeated skolemisation, instantiation and if-lifting.

Thus the tactic applies **skolem!**, then **prop**, and then **grind** tries to prove all those subgoals generated. Because grind uses automatic instantiation, it fails, thus control is passed to the second then where **skolem!**, then **flatten** is applied to the subgoal that grind failed to prove.

### 4.2 Completing the proof

After the instantiation step, the proof completes with the predefined PVS tactic, **grind**.
4.3 Related Work

The above overall proof strategy was found to work for an addition example using the same algorithm, i.e. with the addition retrenched to one on a finite machine [3, 9]. The strategy is a repeated iteration of:

- **Quantifier elimination**: using skolem!, instantiate, inst?.
- **Unfolding definitions**: using grind.
- **Case Analysis**: using prop, flatten, assert, propax.

This strategy as mentioned in [5], was found to work for most hardware proofs [6, 7].

5 Discussion and Conclusions

The divine machine, DivineSub leaves unproven the fact that the operation DSub terminates. This is due to the precondition \( a \geq b \), which could not be proven since no specific values of \( a \) and \( b \) were given [subsection 3.2, Figure 6]. As mentioned in subsection 2.2, this precondition may be taken as a guard instead. In that case DivineSub can be seen as a retrenchment, i.e. a representational translation, of a maximally abstract machine, MASub, with a precondition which is true. A precondition which is true means that the operation will terminate successfully, and no termination TCCs (stpS_TCC1, stpT_TCC1) will be generated. Instead, \( a \geq b \) will be taken as a fact used to prove that the retrenching operation, DSub, establishes the retrenched operation, MASub, as well as, the retrieve relation or concedes clause – stpT is part of the antecedent when the retrenchment proof obligation is reduced to the instantiation stage by reduce-po – rather than as a proposition to be proved.

On the form of the derived tactic, the proof strategy for the retrenchment was:

**Operation Retrenchment Proof Pattern:**

Proof tactic = CommonSteps1 \( \frac{}{} \) UniqueSteps \( \frac{}{} \) CommonSteps2

Where CommonSteps1 = reduce-po, CommonSteps2 = grind, and UniqueSteps was the instantiation step. The fact that the OneOff-retrenchment, the architectural translations, the representational translation, and the addition example proved with the same proof strategy may establish the strategy as a theory for proving the operation retrenchment proof obligation. This may not be surprising since the proof strategy is based on the mechanics of retrenchment/refinement. Nevertheless the examples validate the tactic. As with the Church-Thesis, if no counterexample can be found, the proof strategy may be an acceptable theory for proving the operation retrenchment proof obligation.

Further work is needed to automate the instantiation step. In this instantiation step, the abstract operation takes the skolemised before-state and the skolemised input, that result after the first skolem step in reduce-po. Owe's inst-by-skolem-constants tactic instantiates existential variables by the skolem constants of universal variables that have been skolemised earlier, provided the existential and universal variables have the same identifier/name [14]. The ideas used therein may be helpful in our case, in particular, how to get skolem constants from a proof state.

The general proof strategy in the form of a tactic, can be used as a vehicle to derive the nature of retrenchments, i.e. what needs to be added to particular clauses in the retrenchment proof obligation in order for the proof obligation to be discharged. This may culminate in a general form of the retrenchment proof obligation PO\(_D\) for a particular domain \( D \). For example, a proof obligation might look like
$$PO_D = F_D \land C_D(S_i)$$

Where $F_D$ is a preproved common property of the specifications concerning the domain, and $C_D(S_i)$ is a property pertinent to the specification $S_i$ being proved.

References


The proof of ArchSub

Trying the proof with grind

Trying to prove the operation retrenchment proof obligation with just grind uses an unfavourable automatic instantiation where 0 is instantiated by u! 1 instead of stpS(u! 1). This results in two subgoals which cannot be proved, ArchSub1 and ArchSub2.

ArchSub :

\[\begin{array}{l}
\text{-------} \\
\{1\} \quad \text{FORALL } (u : \text{Dvars}, v : \text{Avars}, bb : \text{nat}) : \\
\quad \left( (I(u) \land J(v) \land G(u, v) \land \text{trmT}(v, bb) \land P(u, v, bb)) \implies \\
\quad \quad \text{trmT}(v, bb) \land \\
\quad \quad \text{trmS}(u) \implies \\
\quad \quad \text{(FORALL } (vp : \text{Avars}) : \\
\quad \quad \quad \text{trmT}(v, bb) \land \text{QT}(v, bb, vp) \implies \\
\quad \quad \quad \quad \text{(EXISTS } (up : \text{Dvars}) : \\
\quad \quad \quad \quad \quad \text{trmS}(u) \land \\
\quad \quad \quad \quad \quad \quad \text{QS}(u, up) \land (G(up, vp) \lor C(up, vp)))) \right) \\
\end{array}\]

Rule? (grind) member rewrites member(a(u), \{x : \text{nat} \mid \text{TRUE}\}) \\
to TRUE
I rewrites I(u) \\
to TRUE
member rewrites member(aa(v), \{x : \text{nat} \mid \text{TRUE}\}) \\
to TRUE
J rewrites J(v) \\
to TRUE
G rewrites G(u, v) \\
to (a(u) = aa(v))
member rewrites member(bb, \{x : \text{nat} \mid \text{TRUE}\}) \\
to TRUE
\text{trmT} rewrites \text{trmT}(v, bb) \\
to aa(v) \geq bb
P rewrites P(u, v, bb) \\
to (b(u) = bb)
\text{trmS} rewrites \text{trmS}(u) \\
to (a(u) \geq b(u))
\text{stpT} rewrites \text{stpT}(v)(bb) \\
to v \text{ WITH } [aa := aa(v) - bb]
\text{QT} rewrites \text{QT}(v, bb, vp) \\
to (vp = v \text{ WITH } [aa := aa(v) - bb])
\text{stpS} rewrites \text{stpS}(u) \\
to u \text{ WITH } [a := a(u) - b(u)]
\text{QS} rewrites \text{QS}(u, up) \\
to (up = u \text{ WITH } [a := a(u) - b(u)])
\]
G rewrites \( G(\text{up}, \text{vp}) \)
to \( (\text{a(\text{up})} = \text{aa(\text{vp})}) \)
C rewrites \( G(\text{up}, \text{vp}) \)
to \( \text{(FALSE)} \)

Trying repeated skolemization, instantiation, and if-lifting, this yields 2 subgoals: ArchSub.1:

\[
\{\text{-1} \} \quad \text{bb!1} \geq 0 \quad \{\text{-2} \} \quad (\text{a(\text{u!1})} = \text{aa(\text{v!1})}) \quad \{\text{-3} \} \quad \text{aa(\text{v!1})} \geq \text{bb!1} \\
\{\text{-4} \} \quad (\text{b(\text{u!1})} = \text{bb!1}) \quad \{\text{-5} \} \quad (\text{vp!1} = \text{v!1} \text{ WITH } [\text{aa} := \text{aa(\text{v!1}) - bb!1}])
\]

\]

\[
\{\text{1} \} \quad 0 = -1 \times \text{bb!1}
\]

Rule?

\[
\text{\texttt{Formulating the tactic \texttt{reduce-po}}}\]

The tactic \texttt{reduce-po} corresponds to the following proof steps in sequence:

\[
\begin{align*}
\texttt{(skolem!)} & \quad \texttt{(prop)} & \quad \texttt{(grind)} & \quad \texttt{(skolem!)} & \quad \texttt{(flatten)}
\end{align*}
\]

But coding this as the tactic

\[
\texttt{(then (skolem!)} & \quad \texttt{(prop)} & \quad \texttt{(grind)} & \quad \texttt{(skolem!)} & \quad \texttt{(flatten))}
\]

does not work. This is because the step \texttt{grind} tries to prove both subgoals, ArchSub.1 and ArchSub.2, generated by the step prop. ArchSub.1 is successfully proved, but with ArchSub.2, grind uses the unfavourable automatic instantiation mentioned above, and this results in two unprovable subgoals, ArchSub.2.1 and ArchSub.2.2. At that point \texttt{grind} stops and the proof transfers into manual mode.
ArchSub :

\[\begin{align*}
| & ------ \\
\{1\} & \text{FORALL (u: Dvars, v: Avars, bb: nat):} \\
& \quad ((\text{I}(u) \text{ AND } \text{J}(v) \text{ AND } G(u, v) \text{ AND } \text{trmT}(v, bb) \text{ AND } P(u, v, bb))) \Rightarrow \\
& \quad \text{trmS}(u) \text{ AND} \\
& \quad \text{trmT}(v, bb) \text{ AND} \\
& \quad \text{trmS}(u) \Rightarrow \\
& \quad \text{(FORALL (vp: Avars):} \\
& \quad \text{trmT}(v, bb) \text{ AND } \text{QT}(v, bb, vp) \Rightarrow \\
& \quad \text{(EXISTS (up: Dvars):} \\
& \quad \text{trmS}(u) \text{ AND} \\
& \quad \text{QS}(u, up) \text{ AND } (G(up, vp) \text{ OR } C(up, vp))))))
\end{align*}\]

Rule? (then (skolem!) (prop) (grind)) Skolemizing, this simplifies to: ArchSub :

\[\begin{align*}
| & ------ \\
\{1\} & ((\text{I}(u!1) \text{ AND} \\
& \quad \text{J}(v!1) \text{ AND } G(u!1, v!1) \text{ AND } \text{trmT}(v!1, bb!1) \text{ AND } P(u!1, v!1, bb!1)) \Rightarrow \\
& \quad \text{trmS}(u!1) \text{ AND} \\
& \quad \text{trmT}(v!1, bb!1) \text{ AND} \\
& \quad \text{trmS}(u!1) \Rightarrow \\
& \quad \text{(FORALL (vp: Avars):} \\
& \quad \text{trmT}(v!1, bb!1) \text{ AND } \text{QT}(v!1, bb!1, vp) \Rightarrow \\
& \quad \text{(EXISTS (up: Dvars):} \\
& \quad \text{trmS}(u!1) \text{ AND} \\
& \quad \text{QS}(u!1, up) \text{ AND } (G(up, vp) \text{ OR } C(up, vp))))))
\end{align*}\]

Applying propositional simplification, this yields 2 subgoals:
ArchSub.1 :

\[\begin{align*}
\{1\} & \text{I}(u!1) \{-2\} \text{ J}(v!1) \{-3\} \text{ G}(u!1, v!1) \{-4\} \text{ trmT}(v!1, bb!1) \\
\{-5\} & \text{ P}(u!1, v!1, bb!1) \\
| & ------ \\
\{1\} & \text{trmS}(u!1)
\end{align*}\]

trmS rewrites \text{trmS}(u!1)
to (a(u!1) \geq b(u!1))

\[\text{member rewrites member(a(u!1), \{x: nat ! TRUE\}) to TRUE}\]

\[\text{I rewrites I(u!1) to TRUE}\]

\[\text{member rewrites member(aa(v!1), \{x: nat | TRUE\}) to TRUE}\]

\[\text{J rewrites J(v!1)}\]
to TRUE
G rewrites G(u1, v1)
to (a(u1) = aa(v1))
member rewrites member(bb1, \{x: nat | TRUE\})
to TRUE
trmT rewrites trmT(v1, bb1)
to aa(v1) >= bb1
P rewrites P(u1, v1, bb1)
to FALSE
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of ArchSub.1.

ArchSub.2:

{-1} trmS(u1) {-2} I(u1) {-3} J(v1) {-4} G(u1, v1) {-5}
trmT(v1, bb1) {-6} P(u1, v1, bb1)

quires

{1} FORALL (vp: Avars):

trmT(v1, bb1) AND QT(v1, bb1, vp) =>
(EXISTS (up: Dvars):

trmS(u1) AND QS(u1, up) AND (G(up, vp) OR C(up, vp)))

trmS rewrites trmS(u1)
to (a(u1) >= b(u1))
member rewrites member(bb1, \{x: nat | TRUE\})
to TRUE
trmT rewrites trmT(v1, bb1)
to aa(v1) >= bb1
stpT rewrites stpT(v1)(bb1)
to v1 WITH [aa := aa(v1) - bb1]
QT rewrites QT(v1, bb1, vp)
to (vp = v1 WITH [aa := aa(v1) - bb1])
stpS rewrites stpS(u1)
to u1 WITH [a := a(u1) - b(u1)]
QS rewrites QS(u1, up)
to (up = u1 WITH [a := a(u1) - b(u1)])
G rewrites G(up, vp)
to (a(up) = aa(vp))
C rewrites C(up, vp)
to (FALSE)
member rewrites member(a(u1), \{x: nat | TRUE\})
to TRUE
I rewrites I(u1)
to TRUE
member rewrites member(aa(v1), \{x: nat | TRUE\})
to TRUE
J rewrites \( J(v!1) \)
  to TRUE
G rewrites \( G(u!1, v!1) \)
  to \( (a(u!1) = aa(v!1)) \)
P rewrites \( P(u!1, v!1, bb!1) \)
  to \( (b(u!1) = bb!1) \)

Trying repeated skolemization, instantiation, and if-lifting, this yields 2 subgoals: ArchSub.2.1:

\[
\begin{align*}
\{-1\} & \quad (aa(v!1) \geq bb!1) \quad \{-2\} \quad (vp!1 = v!1 \text{ WITH } [aa := aa(v!1) - bb!1]) \quad \{-3\} \quad I(u!1) \quad \{-4\} \quad J(v!1) \quad \{-5\} \quad (a(u!1) = aa(v!1)) \quad \{-6\} \\
(b(u!1) = bb!1) \\
\end{align*}
\]

\{1\} \quad 0 = -1 * bb!1

Postponing ArchSub.2.1.

ArchSub.2.2:

\[
\begin{align*}
\{-1\} & \quad (aa(v!1) \geq bb!1) \quad \{-2\} \quad (vp!1 = v!1 \text{ WITH } [aa := aa(v!1) - bb!1]) \quad \{-3\} \quad I(u!1) \quad \{-4\} \quad J(v!1) \quad \{-5\} \quad (a(u!1) = aa(v!1)) \quad \{-6\} \\
(b(u!1) = bb!1) \\
\end{align*}
\]

\{1\} \quad (u!1 = u!1 \text{ WITH } [a := aa(v!1) - bb!1])

Rule?

The tactic reduce-po uses fail to backtrack from this failed proof path to the subgoal which generated that proof path, i.e. ArchSub.2. ArchSub.2 can then be simplified to the manual instantiation step by skolem!, and then flatten. The strategy try is used to explore alternative proof paths.