Verification Conditions
CaD Project

Dawit Tirore

10. November 2020

Contents

1 Introduction

This report describes a Twelf-formalization of soundness and completeness proofs for Verification Conditions. The proofs rely on the soundnesss and completeness of Hoare logic. We start by defining Verification Conditions and Hoare logic. Then non-trivial areas of the Twelf-formalization are described. The report finishes by showcasing the Verification Conditions for a simple example program and their validity.

2 Hoare logic

Hoare logic defines a relation between a precondition, a command and a postcondition. This relation is called a Hoare-triple. We write a Hoare-triple as $\{A\}\ c\ \{B\}$.
c is a command in IMP, whose syntax is given by

$$\begin{array}{lcl} n & \in & \mathbf{Z} = \{\ldots, -1, 0, 1, 2, \ldots\} \\ X & \in & \mathbf{Loc} = \{x, y, z, sum, count, \ldots\} \\ a & ::= & \overline{n} \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1 \\ t & ::= & \mathbf{true} \mid \mathbf{false} \\ b & ::= & t \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b_0 \mid b_0 \wedge b_1 \\ c & ::= & \mathbf{skip} \mid X := a \mid c_0; c_1 \mid \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \mid \mathbf{while}\ b\ \mathbf{do}\ c_0 \end{array}$$ [1]

One deviation from IMP is that instead of expressing disjunction as syntactic sugar, $b_0 \vee b_1 := \neg(\neg b_0 \wedge \neg b_1)$, we allow disjunction in boolean expressions. Ie. $b_0 \vee b_1$ is a legal boolean expresion.

Preconditions and postconditions are written in the language of assertions with the syntax:

$$A ::= \mathbf{true} \mid \mathbf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg A \mid A_0 \wedge A_1 \mid A_0 \vee A_1 \mid A_0 \Rightarrow A_1 \mid \forall X.A \mid \exists X.A$$ [2]

The logic itself is given by:

$$\textsc{H-Skip}: \frac{}{\{A\}\,\mathbf{skip}\,\{A\}} \quad \textsc{H-Assign}: \frac{}{\{B[a/X]\}\,X := a\,\{B\}} \quad \textsc{H-Seq}: \frac{\{A\}\,c_0\,\{C\} \qquad \{C\}\,c_1\,\{B\}}{\{A\}\,c_0; c_1\,\{B\}}$$

$$\textsc{H-If}: \frac{\{A \wedge b\}\,c_0\,\{B\} \quad \{A \wedge \neg b\}\,c_1\,\{B\}}{\{A\}\,\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1\,\{B\}} \qquad \textsc{H-While}: \frac{\{A \wedge b\}\,c_0\,\{A\}}{\{A\}\,\mathbf{while}\ b\ \mathbf{do}\ c_0\,\{A \wedge \neg b\}}$$

$$\textsc{H-Conseq}: \frac{\models A \Rightarrow A' \qquad \{A'\}\,c\,\{B'\} \qquad \models B' \Rightarrow B}{\{A\}\,c\,\{B\}}$$

A mathematically sound assertion is denoted by $\models A$. To be more precise, considering the variables in A as mathematical variables, if A is a tautology, then $\models A$. We'll refer to premises of this form as semantic premises. Note that semantic premises only are used in H-CONSEQ.

[1] The syntax of IMP is taken from Ch. 2 of the Lecture notes given in the course Semantics and Types (SaT)

[2] The rules of Hoare Logic including the syntax of assertions, are taken from Ch. 3 of the Lecture Notes given in SaT

3 Verification Conditions

Verification Conditions is an alternative formalization of Hoare Logic, motivated by the fact that the only two parts of a proof in Hoare Logic that can't easily be automated, is determining the loop invariant and proving the semantic implications in H-CONSEQ. We give the syntax for the annotated command $\hat{c}$, only differing from c by including a loop-invariant in while-statements

$$\hat{c} \ ::= \ \textbf{skip} \mid X := a \mid \hat{c}_0; \hat{c}_1 \mid \textbf{if } b \textbf{ then } \hat{c}_0 \textbf{ else } \hat{c}_1 \mid \{I\}\, \textbf{while } b \textbf{ do } \hat{c}_0$$ [3]

The erasure of $\hat{c}$ is written $|\hat{c}| = c$.
We now define the functions vc and wp. The latter is a helper-function used by the former to compute verification conditions.

$$\begin{aligned}
wp(\textbf{skip}, Q) &= (Q, \emptyset)\\
wp(X := a, Q) &= (Q[a/X], \emptyset)\\
wp(\hat{c}_0; \hat{c}_1, Q) &= (P, \mathcal{V}_0 \cup \mathcal{V}_1), \text{where}\\
&\qquad (R, \mathcal{V}_1) = wp(\hat{c}_1, Q)\\
&\qquad (P, \mathcal{V}_0) = wp(\hat{c}_0, R)\\
wp(\textbf{if } b \textbf{ then } \hat{c}_0 \textbf{ else } \hat{c}_1, Q) &= ((b \Rightarrow P_0) \wedge (\neg b \Rightarrow P_1), \mathcal{V}_0 \cup \mathcal{V}_1), \text{where}\\
&\qquad (P_0, \mathcal{V}_0) = wp(\hat{c}_0, Q)\\
&\qquad (P_1, \mathcal{V}_1) = wp(\hat{c}_1, Q)\\
wp(\{I\}\, \textbf{while } b \textbf{ do } \hat{c}_0, Q) &= (I, \mathcal{V}_0 \cup \{I \wedge b \Rightarrow P_0, I \wedge \neg b \Rightarrow Q\}), \text{where}\\
&\qquad (P_0, \mathcal{V}_0) = wp(\hat{c}_0, I)
\end{aligned}$$

$$\begin{aligned}
vc(A, \hat{c}, B) &= \mathcal{V} \cup \{A \Rightarrow P\}, \text{where}\\
&\qquad (P, \mathcal{V}) = wp(\hat{c}, B)
\end{aligned}$$ [4]

[3]Taken from exam
[4]The definitions for *wp* and *vc* are taken from SaT's 2019 exam question about Verification Conditions

4 Soundness and Completeness of Verification Conditions

An operational semantic for IMP has not been shown because Verification Conditions only will be investigated in relation to Hoare-logic. Hoare-logic itself is proved to be both sound and complete with respect to an operational semantics [1]. Based on this fact, we wish to show that all derivations of verification conditions can equivalently be expressed as a derivation in Hoare-logic (soundness), and the other way around (completeness). More formally we wish to show the following

Soundness of *wp* *if $wp(\hat{c}, Q) = (P, V)$ and $\vDash \mathcal{V}$, then there exists a Hoare-logic derivation $\mathcal{H}$ of $\vdash \{P\}\ |\hat{c}|\ \{Q\}$*

Soundness of *vc* *if $vp(A, \hat{c}, B)$, then $\vdash \{A\}\ |\hat{c}|\ \{B\}$ (and hence, by soundness of Hoare logic, also $\vDash \{A\}\ |\hat{c}|\ \{B\}$)*

Completeness of *wp* *if $\vdash \{A\}\ c\ \{B\}$, then for any Q with $\vDash B \Rightarrow Q$, there exists an $\hat{c}$ (satisyfing $|\hat{c}| = c$) such that, taking $(P, V) = wp(\hat{c}, Q)$, we have $\vDash \mathcal{V}$ and $\vDash A \Rightarrow P$*

Completeness of *vc* *if $\vdash \{A\}\ c\ \{B\}$, (and hence, by completeness of Hoare logic, also if $\vDash \{A\}\ c\ \{B\}$), then there exists a $\hat{c}$ (satisfying $|\hat{c}| = c$), such that $\vDash vc(A, \hat{c}, B)$*

5 Formalising Verification Conditions

We formalize *wp* and *vc* in Twelf by the signatures

```
wp : comh -> asrt -> asrt -> list -> type.
%mode wp +Ch +Q1 -Q2 -V.

vc : asrt -> comh -> asrt -> list -> type.
%mode vc +A1 +Ch +A2 -L.
```

To differentiate the mathematical functions from their Twelf representations, we let *wp* and *vc* refer to the former, while `wp` and `vc` refer to the latter. For $wp(\hat{c}, Q) = (P, V)$ we refer to P as the precondition and V as the assertion-set. `wp` and `vc` represent total functions and the %total directive is used to ensure it's defined for all input-cases. Once rules for `wp` are given, `vc`'s rule is trivially constructed by using `wp` as a subgoal. We will therefore focus on giving the rules for `wp`. Giving these rules present three challenges.

1. In the case of assignment, $wp(X := a, Q) = (Q[a/x], \emptyset)$. Substitution must be formalized.

2. In the case of command-sequencing and if-statements, the final assertion-set is a union of the sub-commands' assertion-sets.
3. In the case of if-statements, b is used as a boolean expression on the LHS and an assertion on the RHS.

5.1 Substitution

We formalize substitution by the signatures

```
asrt_context : asrt -> variable -> (arth -> asrt ) -> type.
%mode asrt_context +A +V -CON.
```

Given an assertion `A`, it abstracts away all occurrences of variable `V` and returns an assertion-context `CON`, which when applied to an arithmetic expression `a`, is instantiated as an assertion with all previous occurrences of `V`, replaced by `a`. Assertions can contain arithmetic expressions, so a similar signature must be given on the arithmetic level, called `arth_context`. We give the rules for constructing terms of type `arth_context` by induction over arithmetic expressions. The interesting case is when an arithmetic expression is a variable. The rule is

```
arth_context/arthvar : arth_context (arthvar (var I0)) (var I1) CON
                         <- compare I0 I1 R
                         <- do_con R (var I1) (arthvar ( var I0)) CON.
```

Variables are represented by natural numbers so `var I0` is a term of type variable and `I0` a term of type nat. The rule is implemented by two sub-goals. The first compares the two variables' natural numbers. The second takes the result of the comparison: If equal, the identity function is returned, otherwise the constant function (with the original variable as constant) is returned.
For constant terms of arithmetic type and assertion type the arithmetic- and assertion-contexts are defined as the constant function.
Most of the remaining rules in `arth_context` and `asrt_context` simply define how contexts are composed. These compositions are all similar. The two sub-contexts are given by using the induction hypothesis on the LHS and RHS of the binary operator along with the variable V. The context is then defined as a lambda abstraction over the binary operator applied to the two sub-contexts (that in turn have been applied to the lambda variable). One example is the case for the arithmetic expression $A_0 + A_1$

```
arth_context/+ : arth_context (A0 + A1) V  ([a] (CON0 a) + (CON1 a))
                   <- arth_context A0 V CON0
                   <- arth_context A1 V CON1.
```

The rules for quantifers are disinct because the quantified variable is represented by using HOAS. An example for the exists-quantifer is

```
asrt_context/exist : asrt_context (aexist  E) V ([a:arth] aexist [x] ((CONE x) a))
                       <- ({a:arth} ({V'} arth_context a V' ([x] a))
                                    -> asrt_context (E a) V (CONE a)).
```

We see that the rule is parametric in `a` an hypothetical in its' `arth_context` derivation. The rule can be read as, under the assumption that `a` is translated to the constant function, `E` translates to `CONE`. Note that a distinct `V'` has to be declared local to `arth_context a V' ([x] a)` to ensure the hypothesis covers substitution with all cases for variables.

We see `asrt_context` used in the rule `wp/assign`, given by:

```
wp/assign : wp (hassign V A) Q (CON A) nil
             <- asrt_context Q V CON.
```

We see that the sub-goal abstracts `Q` into the assertion-context `CON` and the main goal applies the arithmetic expression to CON.

5.2 Union of assertion-sets

The assertion-set is in Twelf represented as a list of assertions, with the usual nil and cons constructors. Union of assertion-sets is represented as the result of appending two assertion-lists, defined by the `append L0 L1 L2` type family.

5.3 Boolean expressions and assertions

Boolean expressions and assertions are distinct Twelf-types, the former named `bexp` and the latter `asrt`. They should be distinct because assertions are more expressive, as they can include implication and quantifiers. Each boolean term has a corresponding assertion term. The translation relation is defined by the type-family `bexp_asrt B AS`, which does as one would expect. Below the rule for if-statements can be seen using `bexp_asrt B AS` and `append L0 L1 L`.

```
wp/if : wp (hif B Ch0 Ch1) Q ((AS a=> P0) a& ((anot AS) a=> P1)) L
         <- bexp_asrt B AS
         <- wp Ch0 Q P0 L0
         <- wp Ch1 Q P1 L1
         <- append L0 L1 L.
```

6 Formalising Hoare logic

The Hoare-triple is represented as

```
h_triple : asrt -> com -> asrt -> type.
```

There is no accompanying mode declaration or total-directive because `h_triple` isn't total, unlike *vc*.
Similar to `wp`, H-ASSIGN relies on substitution and H-IF uses b both as a boolean expression and an assertion. These cases are handled as before. The only interesting case is H-CONSEQ that has two semantic premises. In a paper proof, it suffices to show that A is a sound mathematical statement. In Twelf,

we must formalize this. We do this with the type family `proof AS`, indexed by an assertion `AS`. We use the rules for Natural Deduction to build proofs. A term of type `proof AS` therefore represents a Natural Deduction proof of `AS`. Constants not directly given by Natural Deduction, are all derived. These will be mentioned later in the section Proving Soundness.
H-CONSEQ can now be represented in Twelf as

```
h_triple/conseq : h_triple A CO B
                   <- proof (A a=> A')
                   <- h_triple A' CO B'
                   <- proof (B' a=> B).
```

7 Proving Soundness

For proving soundness of wp and vc we have the following moded and total signatures

```
sound_wp : wp Ch Q P L -> list_proof L -> erasure Ch C -> h_triple P C Q -> type.
%mode sound_wp +WP +V +E -H.

sound_vc : vc A Ch B L -> list_proof L -> erasure Ch C -> h_triple A C B -> type.
%mode sound_vc +VCP +LP +EP -HP.
```

`sound_vc` is read the following way. We are given a derivation of verification conditions L for precondition A, command Ch, and postcondition B. We are also given proofs for L and an erasure derivation. From this we can construct a derivation $\mathcal{H}$ of $\{A\}\ c\ \{B\}$. Similar to `wp` and `vc`, it's also true that once rules are given for `sound_wp`, `sound_vc` can trivially be shown. We therefore focus on `sound_wp` where the following relevant aspects are shown:

1. We need a way to represent proofs of assertion-lists
2. Some rules for `sound_wp` rely on H-CONSEQ, ie. there is a need for natural deduction proofs.

7.1 Proofs for assertion-lists

Proofs for an assertion-list is given by the type-family `list_proof L`, where `L` is an assertion-list. Rules for the type-family do as expected. There's a nil case representing the proof-list for the empty assertion-list. Then there's the cons case, that adds a proof to a proof-list.
It should be possible to index into the proof-list and the reason why will be seen in a moment. This is represented by the signature

```
list_inj : list_proof L -> append L1 L2 L -> list_proof L1 ->
                    list_proof L2 -> type.
%mode list_inj +LP +AP -LP1 -LP2.
```

Given a proof-list, proving L, and a derivation showing that two assertion-lists appended yield L, then the list-proof is split into a proof-list for each of the sub-lists. It is clear that the relation must be total, so the %total directive is used.

7.2 Natural Deduction proofs and H-CONSEQ

To motivate the need for `list_inj` and the use of natural deduction proofs in H-CONSEQ, I'll show the paper proof of soundness for *wp* in the case of if-statements. *wp* is here defined as:

$$\begin{aligned} wp(\textbf{if } b \textbf{ then } \hat{c}_0 \textbf{ else } \hat{c}_1, Q) \;&=\; ((b \Rightarrow P_0) \wedge (\neg b \Rightarrow P_1), \mathcal{V}_0 \cup \mathcal{V}_1), \text{where} \\ &\qquad (P_0, \mathcal{V}_0) = wp(\hat{c}_0, Q) \\ &\qquad (P_1, \mathcal{V}_1) = wp(\hat{c}_1, Q) \end{aligned}$$

We may assume $\models \mathcal{V}_0 \cup \mathcal{V}_1$ and must show a $\mathcal{H}$ deriving $\vdash \{(b \Rightarrow P_0) \wedge (\neg b \Rightarrow P_1)\}\ |\hat{c}|\ \{Q\}$, where $|\hat{c}|$ is the if-statement with erased annotations. For the assertion-set $\mathcal{V}$, $\models \mathcal{V}$ is defined as $\forall V \in \mathcal{V}. \models V$. Therefore $\models \mathcal{V}_0 \cup \mathcal{V}_1$ implies $\models \mathcal{V}_0$ and $\models \mathcal{V}_1$. Assuming the non-inductive cases have been showed, the induction hypothesis IH, can be applied to $|\hat{c_0}|$, Q and $\mathcal{V}_0$ to yield a $\mathcal{H}'_0$ deriving $\{P_0\}\ |\hat{c_0}|\ \{Q\}$. Applying the IH in the same fashion to the other sub-derivation of *wp* yields a $\mathcal{H}'_1$ deriving $\{P_1\}\ |\hat{c_1}|\ \{Q\}$. We now construct $\mathcal{H}$ by H-IF, requiring subderiviations $\mathcal{H}_0$, deriving
$\{((b \Rightarrow P_0) \wedge (\neg b \Rightarrow P_1)) \wedge b\}\ |\hat{c_0}|\ \{Q\}$ and $\mathcal{H}_1$ deriving
$\{((b \Rightarrow P_0) \wedge (\neg b \Rightarrow P_1) \wedge \neg b)\}\ |\hat{c_1}|\ \{Q\}$. We construct $\mathcal{H}_0$ with H-CONSEQ, using $\mathcal{H}'_0$ with the implications $((b \Rightarrow P_0) \wedge (\neg b \Rightarrow P_1)) \wedge b \Rightarrow P_0$ and $Q \Rightarrow Q$. Constructing $\mathcal{H}_1$ is analogous. This proves the existence of $\mathcal{H}$.

This can be seen done in Twelf as

```
sound_wp/if : sound_wp (wp/if AP WP1 WP0 BP)
                        LP
                       (erasure/if EP1 EP0)
                       (h_triple/if
                           (h_triple/conseq proof/impself HP1' proof/iff)
                           (h_triple/conseq proof/impself HP0' proof/ift)
                            BP)
                  <- list_inj LP AP LP0 LP1
                  <- sound_wp WP1 LP1 EP1 HP1'
                  <- sound_wp WP0 LP0 EP0 HP0'.
```

Our argument that $\models \mathcal{V}_0 \cup \mathcal{V}_1$ implies $\models \mathcal{V}_0$ and $\models \mathcal{V}_1$, is represented by `list_inj LP AP LP0 LP1`, where `LP` represents $\models \mathcal{V}_0 \cup \mathcal{V}_1$, `AP` is the partitioning of $\mathcal{V}_0 \cup \mathcal{V}_1$ into $\mathcal{V}_0$ and $\mathcal{V}_1$ and `LP0` and `LP1` respectively represent $\models \mathcal{V}_0$ and $\models \mathcal{V}_1$.

Appeals to the induction hypothesis is seen by the last two sub-goals.

The semantic premises of H-CONSEQ can be seen in the term `h_triple/if HP1 HP0 BP`. `HP0` and `HP1` represent $\mathcal{H}_0$ and $\mathcal{H}_1$. Considering $\mathcal{H}_0$ it contains the proof-terms and `proof/impself` and `proof/ift`. These are defined as derived rules. As an example, the derivation for `proof/ift` is given below.

```
proof/ift : proof ((((AS a=> P0) a& ((anot AS) a=> P1)) a& AS) a=> P0)
            = impi ([p] impe (andel (andel p)) (ander p) ).
```

8 Proving Completeness

For proving completeness we have the following moded and total signatures

```
comp_wp : h_triple A C B -> proof (B a=> Q) ->  erasure Ch C ->
                wp Ch Q P L -> list_proof L -> proof (A a=> P) -> type.
%mode comp_wp +HP +BQ -EP -WPP -LP -AP.

comp_vc : h_triple A C B -> erasure Ch C -> vc A Ch B L -> list_proof L -> type.
%mode comp_vc +H -E -V -LP.
```

`comp_vc H E V LP` is read the following. Given some $\mathcal{H}$ deriving $\{A\}c\{B\}$, the following must be constructed:

- An erasure deriving $|\hat{c}| = c$, for some $\hat{c}$.
- A set of verification conditions $vc(A, \hat{c}, B)$.
- A proof-list proving the verification conditions.

As always, the bulk of the work lies in the rules for `comp_wp HP BQ EP WPP LP AP`. The following interesting aspects are shown:

1. When we use the IH on sub-derivations to yield proof-lists, we need a way to append them.
2. The case of while needs a way to transfer the loop-invariant in the Hoare-triple to `Ch`.
3. The case of assignment presents several challenges which we'll see shortly.

8.1 Appending proof-lists

Just like assertion-lists can be appended, so can proof-lists. This is represented by the signature

```
list_append_lemma : list_proof L1 -> list_proof L2 ->
                            append L1 L2 L3 -> list_proof L3 -> type.
%mode list_append_lemma +LP1 +LP2 -APP -LP3.
```

The %total directive is applied to the signature to ensure that proof-lists indeed always can be appended.

8.2 Annotated while-statements and the loop-invariant

The erasure of a while-statement is represented by:

```
erasure/while : erasure (hwhile I B CH) (while B C)
                 <- erasure CH C.
```

Note that `I` and `B` aren't mentioned in the sub-goal. In `comp_wp/while` the erasure is build by `erasure/while EP0` where `EP0` is the erasure of the while-body. The boolean expression `B` is constrained by the signature and moding of `comp_wp`. The Twelf-representation of $\mathcal{H}$ is given as input, which constrains `B` in `Ch` and `C` to be the same term. No constraints are put on `I` however, as an annotation doesn't occur in `C`. The invariant is present in $\mathcal{H}$ so we simply add the type-information to the erasure

```
((erasure/while EP0) : erasure (hwhile A _ _) _ )
```

8.3 Case of assignment

To illustrate the challenges that occur in the case of assignment for proving completeness, the paper proof is given.
We are given a $\mathcal{H}$ deriving $\{A\}X := a\{B\}$ and a Q satisfying $\models B \Rightarrow Q$. We must construct its' corresponding annotated command which is simply $X := a$. When taking $(Q[a/X], \emptyset) = wp(X := a, Q)$, we must then show $\models \emptyset$ which trivially holds and $\models A \Rightarrow Q[a/X]$. Only H-ASSIGN derives a Hoare-triple with assignment so $\mathcal{H}$ must have ended in H-ASSIGN, making $A = B[a/X]$. We must show $\models B[a/X] \Rightarrow Q[a/x]$. We haven't specified the meaning of substitution on assertions, but take it to be the expected capture-avoiding congruent substitution. Then $(B \Rightarrow Q)[a/X] := B[a/X] \Rightarrow Q[a/x]$. We are given that $\models B \Rightarrow Q$, so it remains to show that for any assertion A and arithmetic expression a, the validity of A implies the validity of $A[a/X]$. A is a tautology and after substitution therefore remains a tautology. This informally proves $B[a/X] \Rightarrow Q[a/x]$.

In formalizing this proof, the first problem we run into is how to construct a term of type `wp Ch Q Q' L`, where `Q'` represents $Q[a/X]$. As `Ch` is an assignment, the derivation ends in the rule `wp/assign`. Refreshing our memory, it was defined as

```
wp/assign : wp (hassign V A) Q (CON A) nil
             <- asrt_context Q V CON.
```

An assertion-context, for which `Q` is an instance of, must be provided as a sub-goal. In the paper proof we are given a $\mathcal{H}$ of $\{B[a/X]\}X := a\{B\}$, which represented in Twelf is the term `h_triple/assign CONB`. `CONB` is an assertion-context that B is an instance of. We however need a `CONQ`. We solve this by noting that an assertion-context can be constructed if the assertion and to-be-abstracted variable is given.
This is represented in Twelf by

```
asrt_context_exist : {A} {V} asrt_context A V CON -> type.
%mode asrt_context_exist +A +V -CON.
```

In the paper proof above we made several informal arguments that must be formalized. We assumed substitution to be capture-avoiding and defined inductively on the subterms of an implication. In Twelf, capture-avoidance is implicitly given by our HOAS representation of assertion-quantifiers. For implication (and the other connectives), `sub_asrt` has been defined inductively on its' subterms.
The final and largest challenge is how to formalize the argument that a tautology remains a tautology after substitution. We formalize this as a theorem we wish to prove in Twelf

```
proof_sub : {V} proof A -> asrt_context A V ([a] CON a) -> ({a} proof (CON a)) -> type.
%mode proof_sub +V +P +CON -PO.
```

It is read as, given a variable `V`, an assertion `A` and an assertion-context `CON`, which `A` is an instance of, then a proof can be constructed for `CON`, parametric in the arithmetic expression `a`. I did not have the time to prove this lemma. `proof_sub` is needed in `comp_wp/assign` so the fix I used was to extend `proof` with the axiom `proof/sub` which is the lemma I wanted to prove. `proof_sub` then has a single proof case that appeals to this axiom.

Proving the assignment case of completeness is then represented in Twelf by

```
comp_wp/assign : comp_wp (h_triple/assign CONB  : h_triple _ (assign V A) _)
                          (BQ : proof (B a=> Q))
                          erasure/assign
                          (wp/assign CONQ)
                          list_proof/nil
                          (PA A)
                   <- asrt_context_exist Q V CONQ
                   <- proof_sub V BQ (asrt_context/=> CONQ CONB) PA .
```

We see the first sub-goal that constructs `CONQ`. We also see the second sub-goal constructing a parametric proof `PA`, instantiated to a proof by the application `PA A`.

9 Verification Conditions for a simple program

In this section we test the type-family `vc P Ch Q L`. Testing was done by querying the type-family with fixed `P`, `Ch` and `Q` and ensuring only 1 solution is given. Proofs for the assertions in the reconstructed `L` have then manually been derived from the axioms. The rules of natural deduction were extended with the ring axioms to prove some properties about integers.
We test two programs. The first is a while-loop where neither precondition nor postcondition use a quantifier. The second is a simple if-statment that uses the existence quantifier.

While-loop The program is given by

$$\begin{aligned}&y := 0;\\&\{z * (y+1) = x\}\textbf{while } \neg(y = i);\\&x := x + z;\\&y := y + 1\end{aligned}$$

We use the precondition

$$x = z$$

And the postcondition

$$x = (i+1) * z$$

Twelf generates three assertions

$$x = z \Rightarrow z * (0+1) = x \tag{1}$$
$$z * (y+1) = x \wedge \neg(y == i) \Rightarrow z * (y+1+1) = x + z \tag{2}$$
$$z * (y+1) == x \wedge (\neg(\neg(y = i)) \Rightarrow x = (I+1) * z \tag{3}$$

Proofs are given for the three assertions in the Twelf-code by the terms `proof/ass1`, `proof/ass2` and `proof/ass3`.

If-statement The program is given by

$$\textbf{if } x \leq y \textbf{ then skip else } x := x * (1+1)$$

The precondition and postcondition are the same. They are:

$$\exists y.\ x = y * (1+1) \tag{4}$$

Twelf generates one assertion

$$\exists y.\ x = y * (1+1) \Rightarrow$$
$$((x \leq z \Rightarrow \exists y'.\ x = y' * (1+1))\ \wedge (\neg(x \leq z) \Rightarrow \exists y'.\ x * (1+1) == y' * (1+1)))$$

A proof is given for this assertion in the Twelf code by the term `proof/ass4`

10 Conclusion

In this report we looked at the the challenges in formalizing Verification Conditions and proofs for its' soundness and completeness by relying on the soundness and completeness of Hoare Logic. A challenge before even beginning the soundness and completeness proofs, was how to express substitution perfomed on an assertion.
In the proof of soundness, the semantic implications in H-CONSEQ, motivated

the use of natural deduction proofs. A paper proof for the case of if-statements was given and compared to its' Twelf representation.
In the proof of completeness the case of assignment presented some challenges. In the construction of `wp/assign CON` when `h_triple/assign CON'` was given, the substitution-deriviation `CON` had to be constructed from the given assertion, variable and arithmetic expression. This was done by the `asrt_context_exist`. As `asrt_context` relied on several signatures as sub-goals, similar exist-signatures were also defined for them. Finally, the generation of Verification Conditions for two simple programs was shown by querying the `vc` signature.

For the scope of this project, the formalization turned out to be more challenging than I expected. It was hard to predict which areas would consume the most time. For example, the assignment case of the completeness proofs requried both an additional lemma about substitutions respecting the soundness of a proof, but also all the exist-related signatures. On the other hand, extending the formalization to support infinitely many variable names, rather than three predefined names, took 10 minutes.

This process of struggling with formalizing something that seems to intuitively be true has made it more clear to me, how many underlying assumptions that are needed for an informal proof to be sound. I also noticed that summarizing the soundness and completeness lemmas as moded type-signatures, made them easier to prove by hand. The signature made it easier to keep a glance over what I was allowed to assume and what needed to be shown. Overall this project has been a valuable exercise for me in mathematical precision.

References

[1] Glynn Winskel. 1993. The formal semantics of programming languages: an introduction. MIT Press, Cambridge, MA, USA.

11 Twelf code

```
%%%%%%%%%%%%%%%Variables and arithmetic expressions%%%%%%%%%%%%%%%%%%%%
nat : type.
z : nat.
s : nat -> nat.

int  :  type.
pos : nat -> int.
neg : nat -> int.

variable : type.
var : nat -> variable.

arth : type.
arthvar : variable -> arth.
n : int -> arth.
+ : arth -> arth -> arth. %infix left 7 +.
- : arth -> arth -> arth. %infix left 7 -.
* : arth -> arth -> arth. %infix left 8 *.

0 : arth
     = n (pos z).
1 : arth
     = n (pos (s z)).


%%%%%%%Boolean expressions and assertions %%%%%%%%%%%%%%
bexp : type.
true : bexp.
false : bexp.
== : arth -> arth -> bexp. %infix none 6 ==.
leq : arth -> arth -> bexp. %infix none 6 leq.
& : bexp -> bexp -> bexp. %infix left 5 &.
or : bexp -> bexp -> bexp. %infix left 4 or.
not : bexp -> bexp.

asrt : type. %name asrt A.
atrue : asrt.
afalse : asrt.
a== : arth -> arth -> asrt. %infix none 6 a==.
aleq : arth -> arth -> asrt. %infix none 6 aleq.
a& : asrt -> asrt -> asrt. %infix left 5 a&.
aor : asrt -> asrt -> asrt. %infix left 4 aor.
a=> : asrt -> asrt -> asrt. %infix right 3 a=>.
% derived connective
```

```
anot : asrt -> asrt
       = [a] a a=> afalse.
aexist : (arth -> asrt) -> asrt.
aforall : (arth -> asrt) -> asrt.


bexp_asrt : bexp -> asrt -> type.
%mode bexp_asrt +B -AS.
bexp_asrt/true : bexp_asrt true atrue.
bexp_asrt/false : bexp_asrt false afalse.
bexp_asrt/== : bexp_asrt (A0 == A1) (A0 a== A1).
bexp_asrt/leq : bexp_asrt (A0 leq A1) (A0 aleq A1).
bexp_asrt/& : bexp_asrt (B0 & B1) (AS0 a& AS1)
       <- bexp_asrt B0 AS0
       <- bexp_asrt B1 AS1.
bexp_asrt/or : bexp_asrt (B0 or B1) (AS0 aor AS1)
       <- bexp_asrt B0 AS0
       <- bexp_asrt B1 AS1.
bexp_asrt/not : bexp_asrt (not B0) (anot A0)
 <- bexp_asrt B0 A0.

%worlds () (bexp_asrt _ _).
%total B (bexp_asrt B _).



% Annotated commands
comh : type. %name comh Ch.

hskip : comh.
hassign : variable -> arth -> comh.
h; : comh -> comh -> comh. %infix left 7 h;.
hif : bexp -> comh -> comh -> comh.
hwhile : asrt -> bexp -> comh -> comh.


% Non-annotated commands
com : type. %name com C.

skip : com.
assign : variable -> arth -> com.
; : com -> com -> com. %infix left 7 ;.
if : bexp -> com -> com -> com.
while : bexp -> com -> com.

% List of assertions.
```

```
asrtlist : type. %name asrtlist L.
nil : asrtlist.
cons : asrt -> asrtlist -> asrtlist. %infix right 7 cons.

append : asrtlist -> asrtlist -> asrtlist ->  type.
append/nil : append nil L L.
append/cons : append (AS cons L0) L (AS cons L')
       <- append L0 L L'.
%mode append +L1 +L2 -L3.
%worlds () (append _ _ _).
%total L1 (append L1 _ _).


erasure : comh -> com -> type.
%mode erasure +CH -C.

erasure/skip : erasure hskip skip.
erasure/assign : erasure (hassign V AS) (assign V AS).
erasure/; : erasure (CH0 h; CH1) (C0 ; C1)
     <- erasure CH0 C0
     <- erasure CH1 C1.
erasure/if : erasure (hif B CH0 CH1) (if B C0 C1)
     <- erasure CH0 C0
     <- erasure CH1 C1.

erasure/while : erasure (hwhile I B CH) (while B C)
 <- erasure CH C.

%worlds () (erasure _ _).
%total (CH) (erasure CH _).

%%%%%%%%%%%Substitution%%%%%%%%%%%%%%%%%%%
result : type.
yes : result.
no : result.


compare : nat -> nat -> result -> type.
%mode compare +I1 +I2 -R.
compare/zz : compare z z yes.
compare/zs : compare z (s I) no.
compare/sz : compare (s I) z no.
compare/ss : compare (s I1) (s I2) R
                <- compare I1 I2 R.
%worlds () (compare _ _ _).
%total I (compare  I _ _).
```

```
do_con : result -> variable -> arth -> (arth -> arth) -> type.
%mode do_con +R +V +A -CON.
do_con/yes : do_con yes V A ([a] a) .
do_con/no : do_con no V A ([a] A).

%block do_block : block  {a:arth}.

%worlds (do_block) (do_con _ _ _ _).
%total {} (do_con _ _ _ _).

arth_context : arth -> variable -> (arth -> arth ) -> type.
%mode arth_context +A +V -CON.
arth_context/arthvar : arth_context (arthvar (var I0)) (var I1) CON
<- compare I0 I1 R
<- do_con R (var I1) (arthvar ( var I0)) CON.
arth_context/n : arth_context (n I) V ([a] (n I)).
arth_context/+ : arth_context (A0 + A1) V  ([a] (CON0 a) + (CON1 a))
  <- arth_context A0 V CON0
  <- arth_context A1 V CON1.
arth_context/- : arth_context (A0 - A1) V  ([a] (CON0 a) - (CON1 a))
  <- arth_context A0 V CON0
  <- arth_context A1 V CON1.
arth_context/* : arth_context (A0 * A1) V  ([a] (CON0 a) * (CON1 a))
  <- arth_context A0 V CON0
  <- arth_context A1 V CON1.

%block a_con : block    {a:arth} {_:{V} arth_context a V ([x:arth] a)}.

%worlds (a_con) (arth_context _ _ _).
%total A   (arth_context A _ _).

asrt_context : asrt -> variable -> (arth -> asrt ) -> type.
%mode asrt_context +A +V -CON.
asrt_context/true : asrt_context atrue V ([a] atrue).
asrt_context/false : asrt_context afalse V ([a] afalse).
asrt_context/== : asrt_context (A a== B) V ([a] (CONA a) a== (CONB a))
   <- arth_context A V CONA
   <- arth_context B V CONB.
asrt_context/leq : asrt_context (A aleq B) V ([a] (CONA a) aleq (CONB a))
   <- arth_context A V CONA
   <- arth_context B V CONB.
asrt_context/& : asrt_context (A a& B) V ([a] (CONA a) a& (CONB a))
   <- asrt_context A V CONA
   <- asrt_context B V CONB.
asrt_context/or : asrt_context (A aor B) V ([a] (CONA a) aor (CONB a))
```

```
   <- asrt_context A V CONA
   <- asrt_context B V CONB.
asrt_context/=> : asrt_context (A a=> B) V ([a] (CONA a) a=> (CONB a))
   <- asrt_context A V CONA
   <- asrt_context B V CONB.
asrt_context/exist : asrt_context (aexist  E) V ([a:arth] aexist [x] ((CONE x) a))
      <- ({a:arth} ({V'} arth_context a V' ([x] a)) -> asrt_context (E a) V (CONE a)).
asrt_context/forall : asrt_context (aforall  E) V ([a:arth] aforall [x] ((CONE x) a))
      <- ({a:arth} ({V'} arth_context a V' ([x] a)) -> asrt_context (E a) V (CONE a)).


%worlds (a_con) (asrt_context _ _ _).
%total A   (asrt_context A _ _).


%%%%%%%%%%%Verfication Conditions%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

wp : comh -> asrt -> asrt -> asrtlist -> type. %name wp WPP.
%mode wp +Ch +Q1 -Q2 -V.

wp/skip : wp hskip Q Q nil.

wp/assign : wp (hassign V A) Q (CON A) nil
             <- asrt_context Q V CON.
wp/; : wp (Ch0 h; Ch1) Q P L
  <- wp Ch1 Q R L1
  <- wp Ch0 R P L0
  <- append L0 L1 L.

wp/if : wp (hif B Ch0 Ch1) Q ((AS a=> P0) a& ((anot AS) a=> P1)) L
 <- bexp_asrt B AS
 <- wp Ch0 Q P0 L0
 <- wp Ch1 Q P1 L1
 <- append L0 L1 L.

wp/while : wp (hwhile I B Ch)
              Q
              I
              (((I a& AS) a=> P) cons (I a& (anot AS) a=> Q) cons L0)
    <- bexp_asrt B AS
    <- wp Ch I P L0.

%worlds () (wp _ _ _ _).
%total Ch (wp Ch Q1 Q2 V).
```

```
vc : asrt -> comh -> asrt -> asrtlist -> type.
%mode vc +A1 +Ch +A2 -L.

vc/ : vc A Ch B ((A a=> P) cons V)
       <- wp Ch B P V.


%worlds () (vc _ _ _ _).
%total Ch (vc A1 Ch A2 V).


%%%%%%%%%%%%%%%%%%Proofs%%%%%%%%%%%%%%%%%%%%%%%%%%

proof : asrt -> type.

andi : proof A -> proof B -> proof (A a& B).
andel : proof (A a& B) -> proof A.
ander : proof (A a& B) -> proof B.

oril : proof A -> proof (A aor B).
orir : proof B -> proof (A aor B).
ore : proof (A aor B) -> (proof A -> proof C) -> (proof B -> proof C)
        -> proof C.

impi : (proof A -> proof B) -> proof (A a=> B).
impe : proof (A a=> B) -> proof A -> proof B.

truei : proof atrue.

falsee : proof afalse -> proof A.

foralli : ({a} proof (P a)) -> proof (aforall [x] P x).
foralle : proof (aforall [x] P x) -> {T} proof (P T).

existi : {T} proof (P T) -> proof (aexist [x] P x).
existe : proof (aexist [x] P x) -> ({a} proof (P a) -> proof C) -> proof C.

eqi : proof (T a== T).
eqe : {P} proof (T a== T') -> proof (P T) ->  proof (P T').

not_not : proof (anot (anot A)) -> proof A.

%% Derivations taken from peano.elf given during the course assignment 4
sym : proof (T1 a== T2) -> proof (T2 a== T1)
     = [e:proof (T1 a== T2)] eqe ([t] t a== T1) e eqi.
```

```
trans : proof (T1 a== T2) -> proof (T2 a== T3) -> proof (T1 a== T3)
     = [e12] [e23] eqe ([t] T1 a== t) e23 e12.

cong : {C} proof (T1 a== T2) -> proof (C T1 a== C T2)
     = [C] [e] eqe ([t] C T1 a== C t) e eqi.



%%%%%%%%%%%%%%Ring axioms%%%%%%%%%%%%%%%%%%%%%%
r_add_assoc : proof ((A + B) + C a== A + (B + C)).
r_add_comm : proof (A + B a== B + A).
r_add_id : proof ( 0 + A a== A).
r_mul_assoc : proof ((A * B) * C a== A * (B * C)).
r_mul_comm : proof (A * B a== B * A).
r_mul_dist_l : proof (A * (B + C) a== A * B + A * C).
r_mul_dist_r : proof ((A + B) * C a== A * C + B * C).
r_mul_0 : proof (A * 0 a== 0).
r_mul_1 : proof (A * 1 a== A).



proof/sub : {V} proof A -> asrt_context A V ([a] CON a) -> ({a} proof (CON a)).



%%%%%%%%%%%%%%%%%%%%% Hoare logic %%%%%%%%%%%%%%%%%%%%%%%%%%%%%

asrt_eq : asrt -> asrt -> type.
asrt_eq/ : asrt_eq A A.

% hoare triple. Imodsætning til wp er denne relation ikke total.
h_triple : asrt -> com -> asrt -> type.

h_triple/skip : h_triple P skip P.

h_triple/assign : h_triple (CON AS) (assign V AS) Q
                    <- asrt_context Q V CON.
h_triple/; : h_triple P (C0 ; C1) Q
      <- h_triple P C0 Q'
      <- h_triple Q' C1 Q.

h_triple/if : h_triple P (if B C0 C1) Q
       <- bexp_asrt B AS
       <- h_triple (P a& AS) C0 Q
       <- h_triple (P a& anot AS) C1 Q.
```

```
h_triple/while : h_triple A (while B C0) (A a& anot AS)
  <- bexp_asrt B AS
  <- h_triple (A a& AS) C0 A.

h_triple/conseq : h_triple A C0 B
   <- proof (A a=> A')
   <- h_triple A' C0 B'
   <- proof (B' a=> B).





%%%%%%%%%%%%%%%%%%%%%%%%%% Soundness %%%%%%%%%%%%%%%%%%%%%%

asrtlist_proof : asrtlist -> type.
asrtlist_proof/nil : asrtlist_proof nil.
asrtlist_proof/cons : asrtlist_proof (AS cons L0)
   <- proof AS
   <- asrtlist_proof L0.

asrtlist_inj : asrtlist_proof L -> append L1 L2 L -> asrtlist_proof L1 -> asrtlist_proof L2
%mode asrtlist_inj +LP +AP -LP1 -LP2.
asrtlist_inj/nil : asrtlist_inj LP append/nil asrtlist_proof/nil LP.
asrtlist_inj/cons : asrtlist_inj (asrtlist_proof/cons LP0 ASP0) (append/cons AP) (asrtlist_p
 <- asrtlist_inj LP0 AP LP1 LP2.

%worlds () (asrtlist_inj _ _ _ _).
%total AP (asrtlist_inj _ AP _ _).


proof/impself : proof(AS a=> AS)
      = impi ([p] p).
proof/ift : proof ((((AS a=> P0) a& ((anot AS) a=> P1)) a& AS) a=> P0)
             = impi ([p] impe (andel (andel p)) (ander p) ).
proof/iff : proof ((((AS a=> P0) a& ((anot AS) a=> P1)) a& (anot AS)) a=> P1)
     = impi ([p] impe (ander (andel p)) (ander p) ).



sound_wp : wp Ch Q P L -> asrtlist_proof L -> erasure Ch C -> h_triple P C Q -> type.
%mode sound_wp +WP +V +E -H.

sound_wp/skip : sound_wp wp/skip asrtlist_proof/nil erasure/skip h_triple/skip.
```

```
sound_wp/assign : sound_wp (wp/assign SUB)
                           asrtlist_proof/nil
                           erasure/assign
                           (h_triple/assign SUB).

sound_wp/; : sound_wp (wp/; AP WP0 WP1) LP (erasure/; EP1 EP0) (h_triple/; HP1 HP0)
      <- asrtlist_inj LP AP LP0 LP1
      <- sound_wp WP0 LP0 EP0 HP0
      <- sound_wp WP1 LP1 EP1 HP1.


sound_wp/if : sound_wp (wp/if AP WP1 WP0 BP)
                       LP
                       (erasure/if EP1 EP0)
                       (h_triple/if (h_triple/conseq proof/impself HP1 proof/iff)
                                    (h_triple/conseq proof/impself HP0 proof/ift)
                                     BP)
       <- asrtlist_inj LP AP LP0 LP1
       <- sound_wp WP1 LP1 EP1 HP1
       <- sound_wp WP0 LP0 EP0 HP0.

sound_wp/while : sound_wp (wp/while WP0 BA)
                          (asrtlist_proof/cons (asrtlist_proof/cons LP' InB) IB)
                          (erasure/while EP0)
                          (h_triple/conseq InB  (h_triple/while
                                                  (h_triple/conseq proof/impself HP0 IB)
                                                  BA)
                          proof/impself)
       <- sound_wp WP0 LP' EP0 HP0.


%worlds () (sound_wp _ _ _ _).
%total (WP) (sound_wp WP _ _ _).

% bevis for soundness af vc
sound_vc : vc A Ch B L -> asrtlist_proof L -> erasure Ch C -> h_triple A C B -> type.
%mode sound_vc +VCP +LP +EP -HP.

sound_vc/ : sound_vc (vc/ WPP)
                     (asrtlist_proof/cons LP0 ASP)
                     EP
                     (h_triple/conseq proof/impself HP0 ASP)
     <- sound_wp WPP LP0 EP HP0.

%worlds () (sound_vc _ _ _ _).
%total (VCP) (sound_vc VCP _ _ _).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%% Existence proofs%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

compare_exist : {I1} {I2} compare I1 I2 R -> type.
%mode compare_exist +I1 +I2 -CMP.
compare_exist/zz : compare_exist z z compare/zz.
compare_exist/zs : compare_exist z (s I) compare/zs.
compare_exist/sz : compare_exist (s I) z compare/sz.
compare_exist/ss : compare_exist (s I1) (s I2) (compare/ss CMP0)
                <- compare_exist I1 I2 CMP0.
%worlds () (compare_exist _ _ _).
%total I (compare_exist  I _ _).



do_con_exist :{R} {V} {A} do_con R V A CON -> type.
%mode do_con_exist +R +V +A -DO.
do_con_exist/yes : do_con_exist yes V A do_con/yes.
do_con_exist/no : do_con_exist no V A do_con/no.

%block do_con_exist_block : block    {a:arth} {VCON: {V} arth_context a V ([x:arth] a)}.
%worlds (do_con_exist_block) (do_con_exist _ _ _ _).
%total {} (do_con_exist  _ _ _ _).

arth_context_exist : {A} {V} arth_context A V CON -> type.
%mode arth_context_exist +A +V -ACON.
arth_context_exist/arthvar : arth_context_exist (arthvar (var I0))
                                                (var I1)
                                                (arth_context/arthvar DOCON CMP)
      <- compare_exist I0 I1 (CMP : compare _ _ R)
      <- do_con_exist R (var I1) (arthvar (var I0)) DOCON.

arth_context_exist/n : arth_context_exist (n I) V arth_context/n.
arth_context_exist/+ : arth_context_exist (A + B) V (arth_context/+ ACON1 ACON0)
<- arth_context_exist A V ACON0
<- arth_context_exist B V ACON1.
arth_context_exist/- : arth_context_exist (A - B) V (arth_context/- ACON1 ACON0)
<- arth_context_exist A V ACON0
<- arth_context_exist B V ACON1.
arth_context_exist/* : arth_context_exist (A * B) V (arth_context/* ACON1 ACON0)
<- arth_context_exist A V ACON0
<- arth_context_exist B V ACON1.

%block arth_con_exist_block : block {a:arth}
                                    {VCON: {V} arth_context a V ([x:arth] a)}
```

```
                                   {_ : {V} arth_context_exist a V (VCON V)}.


%worlds (arth_con_exist_block ) (arth_context_exist _ _ _).
%total A (arth_context_exist  A _ _).


asrt_context_exist : {A} {V} asrt_context A V CON -> type.
%mode asrt_context_exist +A +V -CON.
asrt_context_exist/true : asrt_context_exist atrue V asrt_context/true.
asrt_context_exist/false : asrt_context_exist afalse V asrt_context/false.
asrt_context_exist/== : asrt_context_exist (A a== B) V (asrt_context/== ACON1 ACON0)
 <- arth_context_exist A V ACON0
 <- arth_context_exist B V ACON1.
asrt_context_exist/leq : asrt_context_exist (A aleq B) V (asrt_context/leq ACON1 ACON0)
 <- arth_context_exist A V ACON0
 <- arth_context_exist B V ACON1.
asrt_context_exist/& : asrt_context_exist (A a& B) V (asrt_context/& ACON1 ACON0)
 <- asrt_context_exist A V ACON0
 <- asrt_context_exist B V ACON1.
asrt_context_exist/or : asrt_context_exist (A aor B) V (asrt_context/or ACON1 ACON0)
 <- asrt_context_exist A V ACON0
 <- asrt_context_exist B V ACON1.
asrt_context_exist/=> : asrt_context_exist (A a=> B) V (asrt_context/=> ACON1 ACON0)
 <- asrt_context_exist A V ACON0
 <- asrt_context_exist B V ACON1.
asrt_context_exist/exist : asrt_context_exist (aexist E) V (asrt_context/exist ACONE)
             <- ({a:arth} ({VCON:{V'} arth_context a V' ([x] a)}
                     ({V} arth_context_exist a V (VCON V))
                          -> asrt_context_exist (E a) V (ACONE a VCON))).
asrt_context_exist/forall : asrt_context_exist (aforall E) V (asrt_context/forall ACONE)
             <- ({a:arth} ({VCON:{V'} arth_context a V' ([x] a)}
                     ({V} arth_context_exist a V (VCON V))
                          -> asrt_context_exist (E a) V (ACONE a VCON))).


%worlds (arth_con_exist_block ) (asrt_context_exist _ _ _).
%total A (asrt_context_exist  A _ _).

proof_sub : {V} proof A -> asrt_context A V ([a] CON a) -> ({a} proof (CON a)) -> type.
%mode proof_sub +V +P +CON -P0.

 - : proof_sub V (P:proof A) CON (proof/sub V P CON).

%worlds () (proof_sub _ _ _ _).
%total P (proof_sub _ P _ _).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Completeness %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

asrtlist_append_lemma : asrtlist_proof L1 -> asrtlist_proof L2 ->
                                  append L1 L2 L3 -> asrtlist_proof L3 -> type.
%mode asrtlist_append_lemma +LP1 +LP2 -APP -LP3.

asrtlist_append_lemma/nil : asrtlist_append_lemma asrtlist_proof/nil LP append/nil LP.
asrtlist_append_lemma/cons : asrtlist_append_lemma (asrtlist_proof/cons LP0 ASP)
                                                   LP2
                                                   (append/cons AP)
                                                   (asrtlist_proof/cons LP' ASP)
            <- asrtlist_append_lemma LP0 LP2 AP LP'.

%worlds () (asrtlist_append_lemma _ _ _ _).
%total LP1 (asrtlist_append_lemma LP1 _ _ _).


% only used in andimp2
proof/andimp : proof ((A a& B) a=> P0 ) -> proof (A a=> (B a=> P0))
     = [p0] (impi [a] (impi [b] (impe p0 (andi a b)))).
proof/andimp2 : proof ((A a& B) a=> P0 ) -> proof (A a& (anot B) a=> P1 )
                                  -> proof (A a=> (B a=> P0) a& ((anot B) a=> P1))
    = [p0][p1] impi [a] andi (impe (proof/andimp p0) a) (impe (proof/andimp p1) a).
proof/impimp : proof (A a=> B) -> proof (B a=> C) -> proof (A a=> C)
     = [ab][bc] impi ([a] impe bc (impe ab a)).
comp_wp : h_triple A C B -> proof (B a=> Q) ->  erasure Ch C -> wp Ch Q P L
                                  -> asrtlist_proof L -> proof (A a=> P) -> type.
%mode comp_wp +HP +BQ -EP -WPP -LP -AP.

comp_wp/skip : comp_wp h_triple/skip BQ erasure/skip wp/skip asrtlist_proof/nil BQ.

comp_wp/assign : comp_wp (h_triple/assign CONB  : h_triple _ (assign V A) _)
                         (BQ : proof (B a=> Q))
                         erasure/assign
                         (wp/assign CONQ)
                         asrtlist_proof/nil
                         (PA A)
           <- asrt_context_exist Q V CONQ
           <- proof_sub V BQ (asrt_context/=> CONQ CONB) PA .



comp_wp/; : comp_wp ((h_triple/; H1 H0))
                    BQ
                    (erasure/; EP1 EP0)
                    (wp/; APP WP0 WP1)
                    LP
                    IMP2
     <- comp_wp H1 BQ EP1 WP1 LP1 IMP
     <- comp_wp H0 IMP EP0 WP0 LP0 IMP2
     <- asrtlist_append_lemma LP0 LP1 APP LP.

comp_wp/if : comp_wp (h_triple/if H1 H0 BA)
                     BQ
                     (erasure/if EP1 EP0)
                     (wp/if APP WP1 WP0 BA)
                     LP
                     (proof/andimp2 IMP0 IMP1)
     <- comp_wp H0 BQ EP0 WP0 LP0 IMP0
     <- comp_wp H1 BQ EP1 WP1 LP1 IMP1
     <- asrtlist_append_lemma LP0 LP1 APP LP.

comp_wp/while : comp_wp (h_triple/while H0 BA : h_triple A _ _)
                        BQ
                        ((erasure/while EP0) : erasure (hwhile A _ _) _ )
                        (wp/while WP0 BA)
                        (asrtlist_proof/cons (asrtlist_proof/cons LP BQ) IMP)
                        proof/impself
 <- comp_wp H0 proof/impself EP0 WP0 LP IMP.

comp_wp/conseq : comp_wp (h_triple/conseq P1 H P0) BQ EP WP LP (proof/impimp P0 IMP)
  <- comp_wp H (proof/impimp P1 BQ) EP WP LP IMP.

%worlds () (comp_wp  _ _ _ _ _ _).
%total (HP) (comp_wp HP _ _ _ _ _).

comp_vc : h_triple A C B -> erasure Ch C -> vc A Ch B L -> asrtlist_proof L -> type.
%mode comp_vc +H -E -V -LP.

comp_vc/ : comp_vc H E (vc/ WP) (asrtlist_proof/cons L IMP)
    <- comp_wp H proof/impself E WP L IMP.

%worlds () (comp_vc  _ _ _ _).
%total H (comp_vc H _ _ _).


%%%%%%%%%%%%%% Test %%%%%%%%%%%%%%%%%%

varx : variable
= var z.
```

```
vary : variable
= var (s z).
varz : variable
= var (s (s z)).
vari : variable
= var (s (s (s z))).


%%%%%%%%%%Test 1%%%%%%%%%%%%%%

test_prog1 : comh
     =   (hassign vary 0) h;
 (hwhile ((arthvar varz) * ((arthvar vary) + 1) a== (arthvar varx))
        (not ((arthvar vary) ==  (arthvar vari)))
     (
      (hassign varx (arthvar varx + (arthvar varz))) h;
      (hassign vary (arthvar vary + 1))
     )
       ).

%abbrev test_prog1_p : asrt
     = (arthvar varx) a== (arthvar varz).

%abbrev test_prog1_q : asrt
     = (arthvar varx) a== (arthvar vari + 1) * (arthvar varz).


%abbrev test_prog1_ass1 : asrt
  = X a== Z a=> Z * (0 + 1) a== X.
%abbrev test_prog1_ass2 : asrt
  =   Z * (Y + 1) a== X a&  (anot (Y a== I)) a=>  Z * (Y + 1 + 1) a== X + Z.
%abbrev test_prog1_ass3 : asrt
  =  Z * (Y + 1) a== X a& (anot (anot (Y a== I))) a=>  X a== (I + 1) * Z.




% Z * (0 + 1) = Z * 0 + Z * 1 = 0 + Z*1 = Z*1 = Z
proof/ass1 : proof (X a== Z a=> Z * (0 + 1) a== X)
      = impi [xz] trans (trans (trans (trans r_mul_dist_l
                                             (cong ([a] a + Z * 1) r_mul_0)
                                      )
                               r_add_id)
                        r_mul_1)
                 (sym xz).
```

```
% Z * (y + 1 + 1) == Z * (y + 1) + Z * 1 == Z * (y + 1) + Z = X + Z
proof/ass2 : proof (Z * (Y + 1) a== X a&  (anot (Y a== I))
                                        a=> Z * (Y + 1 + 1) a== X + Z)
      = impi [zyx] trans (trans r_mul_dist_l
                                (cong ([a] Z * (Y + 1) + a) r_mul_1)
                         )
                         (cong ([a] a + Z ) (andel zyx)).

% X = Z * (Y + 1) = Z * (I + 1) = (I + 1) * Z
proof/ass3 : proof (Z * (Y + 1) a== X a& (anot (anot (Y a== I)))
                                              a=> X a== (I + 1) * Z)
      =  impi [zyxAy] trans (trans (sym (andel zyxAy))
                                   (cong ([a] Z * (a + 1 )) (not_not (ander zyxAy)))
                            )
                            r_mul_comm.

ass_list_1 : asrtlist
 = test_prog1_ass1 cons test_prog1_ass2 cons test_prog1_ass3 cons nil.

proofs_of_ass_list_1 : asrtlist_proof ass_list_1
   = asrtlist_proof/cons (asrtlist_proof/cons (asrtlist_proof/cons
                                                    asrtlist_proof/nil
                                                    proof/ass3
                                              )
                                              proof/ass2) proof/ass1.


% |= vc(p,test_prog,q).
%query 1 5 D : vc test_prog1_p test_prog1 test_prog1_q  ass_list_1.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Test 2%%%%%%%%%%%%%%%%%%%

test_prog2 : comh
      = hif ((arthvar varx) leq (arthvar vary))
 hskip
 (hassign varx ((arthvar varx) * (1 + 1) )).

%abbrev test_prog2_pq : asrt
     = aexist [y] (arthvar varx) a== y * (1 + 1).


%abbrev test_prog2_ass4 : asrt
 = (aexist ([y:arth] X a== y * (1 + 1))
                          a=> (X aleq Z a=> aexist ([y:arth] X a== y * (1 + 1)))
                          a&
      (anot (X aleq Z) a=> aexist ([y:arth] X * (1 + 1) a== y * (1 + 1)))).
```

```
ass_list_2 : asrtlist
  = test_prog2_ass4 cons nil.

proof/ass4 : proof (aexist ([y:arth] X a== y * (1 + 1))
                              a=> (X aleq Z a=> aexist ([y:arth] X a== y * (1 + 1)))
                              a&
       (anot (X aleq Z) a=> aexist ([y:arth] X * (1 + 1) a== y * (1 + 1))))
 =  impi [ey] andi (impi [_] ey)
                   (impi [_] existe ey ([y'][py'] existi (y' * (1 + 1))
                                                         (cong ([i] i * (1 + 1)) py')
                                       )
                   ).

proofs_of_ass_list_2 : asrtlist_proof ass_list_2
    = asrtlist_proof/cons asrtlist_proof/nil proof/ass4.

%query 1 5 D : vc test_prog2_pq test_prog2 test_prog2_pq ass_list_2.
```