Doctoral Program
Computational Mathematics
Numerical Analysis and Symbolic Computation

On the Soundness of the Translation of MiniMaple to Why3ML

Muhammad Taimoor Khan

DK-Report No. 2014-03

02 2014

A–4040 LINZ, ALTENBERGERSTRASSE 69, AUSTRIA

Supported by

Austrian Science Fund (FWF)

Upper Austria

Editorial Board: Bruno Buchberger
Bert Jüttler
Ulrich Langer
Manuel Kauers
Esther Klann
Peter Paule
Clemens Pechstein
Veronika Pillwein
Silviu Radu
Ronny Ramlau
Josef Schicho
Wolfgang Schreiner
Franz Winkler
Walter Zulehner

Managing Editor: Silviu Radu

Communicated by: Wolfgang Schreiner
Franz Winkler

DK sponsors:

- **Johannes Kepler University Linz** (JKU)
- **Austrian Science Fund** (FWF)
- **Upper Austria**

On the Soundness of the Translation of *MiniMaple* to Why3ML*

Muhammad Taimoor Khan
Doktoratskolleg Computational Mathematics
and
Research Institute for Symbolic Computation
Johannes Kepler University
Linz, Austria
Muhammad.Taimoor.Khan@risc.jku.at

February 3, 2014

Abstract

In this paper, we first introduce the soundness statements for the various constructs of *MiniMaple* and then give the corresponding proofs for the soundness of the most interesting syntactic domains of *MiniMaple*, i.e. command sequences, assignment statements, conditionals and while-loops.

*The research was funded by the Austrian Science Fund (FWF): W1214-N15, project DK10.

Contents

1 Introduction

In order to show that the verification of the translated Why3ML program implies the correctness of the original *MiniMaple* program , we have to prove that the translation preserves the semantics of the program. In detail, we have to prove the equivalence of the denotational semantics of *MiniMaple* programs [4, 3, 2] and the operational semantics of Why3ML programs [1]. We have defined the denotational semantics of *MiniMaple* as a relationship between a pre and a post-state, e.g. the formal semantics of a *MiniMaple* command is defined as:

$$[\![C]\!](e)(s, s')$$

such that semantically, in a given type environment e, the execution of a command C in a pre-state s yields a post-state s'. In [1] a big-step operational semantics of Why3 is defined as a transition:

$$<s, e> \longrightarrow <s', v>$$

which says that in a pre-state s, the execution of a Why3 expression e yields a post-state s' and a value v. Based on these semantics, we have formulated and proved the soundness statements as discussed later in this document.

The rest of the paper is organized as follows: in Section 2, we discuss the overview of soundness of various *MiniMaple* constructs. Section 3 presents conclusions and future work. Appendix A introduces the semantic domains of *MiniMaple* and Why3 and Appendix B sketches the auxiliary functions and predicates that are later used in the proof of the soundness. Appendix C formulates the corresponding soundness statements while Appendix D gives the actual proof of the soundness statements for the selected constructs. The proof requires some additional lemmas and definitions which are defined in Appendices E and F respectively. The semantics of Why3 is defined in Appendix G while the derivations for the proof of the soundness of while-loop are discussed in Appendix H.

2 Overview of the Soundness

In this section, we describe the guidelines to read the different Appendices A, B, C and D with the help of some examples. Each of the following subsections presents the corresponding aforementioned appendix respectively.

2.1 Semantic Domains

This section gives the definition of various semantic domains of *MiniMaple* and Why3. We needed to extend some of the semantic domains for *MiniMaple*; while the definition of the corresponding semantic domains of Why3 are deduced from the operational semantics of Why3 as discussed in [1]. In the following, we introduce some critical (w.r.t. proof) semantic domains of *MiniMaple* and Why3, e.g. state and value. For the complete definition of all the semantic domains of Why3 and *MiniMaple*, please see Appendix A.

2.1.1 For Why3

The state values of Why3 are defined as a mapping of variables to their corresponding Why3 semantic values.

$State_w := Variable \rightarrow Value_w$

where the semantic values is a disjoint domain consists of

$Value_w = c + Exception_w + Function_w + Void$

Why3 constants c, an exception object $Exception_w$, a function value $Function_w$ and $Void$. Here the constant c models all the other values, e.g. booleans, integers, reals, tuples and lists.

2.1.2 For MiniMaple

The state values of *MiniMaple* are defined as a tuple of store and data values:

$State := Store \times Data$

where the corresponding store and data values are:

$Store := Variable \rightarrow Value$
$Data := Flag \times Exception \times Return$

The domain of semantic values of *MiniMaple* is also a disjoint domain as:

$Value = Procedure + List + Tuple + Boolean + Integer + \ldots + Symbol$

In order to make the various proof steps handy, based on the above definitions we have introduced a new semantic domain

$InfoData = Value + Data + Void$

which corresponds to the values domain $Value_m$ of Why3.

2.2 Auxiliary Functions and Predicates

This section gives the declaration and (partial) definitions of various critical auxiliary predicates which are very important w.r.t. the proof.

- **equals** $\subseteq$ **State** $\times$ $\mathbf{State_w}$: returns *true* only if the given *MiniMaple* state equals the given Why3 state as defined:

  $$equals(s,t) \Leftrightarrow \forall i : Identifier, v_m \in Value : i \in dom(s) \wedge <i, v_m> \in store(s) \Rightarrow \exists v_w \in Value_w : <i, v_w> \in t \wedge equals(v_m, v_w)$$

- **equals** $\subseteq$ **Value** $\times$ $\mathbf{Value_w}$: returns *true* only if the given *MiniMaple* value equals the given Why3 value as defined:

  $equals(v_m, v_w) \Leftrightarrow$
  cases v_m **of**
  [] $isInteger(int_m) \rightarrow$
  cases v_w **of**
  $isIntegerw(int_w) \rightarrow valueOf(int_m) = valueOf(inv_w)$

$[]\ \_ \rightarrow \mathit{false}$
end
$[]\ \mathit{isBoolean}(b_m) \rightarrow$
cases v_w **of**
$\mathit{isBooleanw}(b_w) \rightarrow \mathit{valueOf}(b_m) = \mathit{valueOf}(b_w)$
$[]\ \_ \rightarrow \mathit{false}$
end
$[] \ldots \rightarrow \ldots$
end

- **equals** $\subseteq$ **InfoData** $\times$ **Valuew**: returns *true* only if the given state information of *MiniMaple* equals the given Why3 value. This predicate is defined to make our proof handy and easier.

$\mathit{equals}(d, v_w) \Leftrightarrow$
cases d of
$[]\ \mathit{isValue}(v_m) \rightarrow \mathit{equals}(v_m, v_w)$
$[]\ \mathit{isData}(d_m) \rightarrow$
IF $\mathit{exceptions}(d_m)$ THEN
cases v_w of
$\mathit{isExceptionw}(e_w) \rightarrow$
$\mathit{equals}(\mathit{getId}(d_m), \mathit{getId}(e_w)) \wedge \mathit{equals}(\mathit{getValue}(d_m), \mathit{getValue}(e_w))$
$[]\ \_ \rightarrow \mathit{false}$
end
ELSE ... END
$[]\ \mathit{isVoid}(mv) \rightarrow$
cases vw of
$\mathit{isVoid}(wv) \rightarrow \mathit{true}$
$[]\ \_ \rightarrow \mathit{false}$
end
end

- **extendsEnv** $\subseteq$ $\mathbf{Environment_w} \times \mathbf{Expression_w} \times \mathbf{Environment_w}$: returns *true* if the former environment extends the latter environment with the identifiers appearing in the given expression.

$\mathit{extendsEnv}(e_1, c, e_2) \Leftrightarrow$
$\forall I : \mathit{Identifier}, v \in \mathit{Value}, \mathit{Iseq} \in \mathit{Identifier\_Sequence}, \mathit{vseq} \in \mathit{Value\_Sequence}:$
$<I, v> \in e_2 \wedge \mathit{Iseq} = \mathit{extractIdentifiers}(c) \wedge \mathit{vseq} = \mathit{getValues}(\mathit{Iseq}, c)$
$\Rightarrow <I, v> \in e_1 \wedge e_1 = e_2 \cup \mathit{IVSeqtoSet}(\mathit{Iseq}, \mathit{vseq})$

The definitions of the corresponding predicates $extendsDecl$ and $extendsTheory$ are the same as of *extendsEnv* defined above. For the definitions of the complete list of functions and predicates, please see Appendix B.

2.3 Soundness Statements

In this section, we discuss the formulation of the soundness statements for the translation of *MiniMaple* to Why3. The general goal here is proof:

$\forall Cseq \in Command\_Sequence, C \in Command, E \in Expression :$
$Soundness\_cseq(Cseq) \wedge Soundness\_c(C) \wedge Soundness\_e(E)$

where

- **Soundness_cseq $\subseteq$ Command_Sequence**: defines the soundness statement for a *MiniMaple* command sequence as below:

  $Soundness\_cseq(Cseq) \Leftrightarrow$
  $\forall\ em \in Environment, cw \in Exprression_w, ew, ew' \in Environment_w,$
  $dw, dw' \in Decl_w, tw, tw' \in Theory_w :$
  $wellTyped(em, Cseq) \wedge consistent(em, ew, dw, tw) \wedge$
  $<cw, ew', dw', tw'> = \mathrm{T}[\![Cseq]\!](em, ew, dw, tw)$
  $\Rightarrow$
  $wellTyped(cw, ew', dw', tw') \wedge extendsEnv(ew', cw, ew) \wedge$
  $extendsDecl(dw', cw, dw) \wedge extendsTheory(tw', cw, tw) \wedge$
  $\forall t, t' \in State_w, vw \in Value_w : <t', cw> \longrightarrow <t', vw>$
  $\Rightarrow$
  $\exists s, s' \in State_m : equals(s, t) \wedge [\![Cseq]\!](e)(s, s') \wedge$
  $\forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge$
  $[\![Cseq]\!](e)(s, s') \wedge dm = infoData(s')$
  $\Rightarrow equals(s', t') \wedge equals(dm, vw)$

  In detail, the soundness statement for the command sequence $Cseq$ states that

  - if a command sequence $Cseq$ translates to Why3 expression *cw* such that various predicates holds for $Cseq$, e.g. well-typeness then,
  - various predicates also hold for the corresponding translated expression *cw*, e.g. extension of the declarations *extendsDecl* and theory $extendsTheory$ and
  - if for arbitrary Why3 states t and t', execution of the translated expression *cw* in state t yields to a post-state t' and a value *vw* then,
  - there are corresponding *MiniMaple* states s and s' such that states s and t are equal and execution of a command sequence $Cseq$ in this state s yields to a state s' and
  - if for arbitrary *MiniMaple* states s and s', corresponding states s and t are equal; moreover, with a given environment e execution of $Cseq$ in a pre-state s yields a post-state s' and *dm* is the state information of s' then,
  - the corresponding post-states s' and t' are equals and also the corresponding values *dm* and *vw* are equal.

- *Soundness_c $\subseteq$ Command*: defines the soundness statement for a *MiniMaple* command as below:

  $Soundness\_c(C) \Leftrightarrow$
  $\forall\ em \in Environment, cw \in Exprression_w, ew, ew' \in Environment_w,$
  $dw, dw' \in Decl_w, tw, tw' \in Theory_w :$
  $wellTyped(em, C) \wedge consistent(em, ew, dw, tw) \wedge$

$$
\begin{array}{l}
{<}cw, ew', dw', tw'{>} = \mathrm{T}[\![C]\!](em, ew, dw, tw) \\
\Rightarrow \\
\quad \mathit{wellTyped}(cw, ew', dw', tw') \wedge \mathit{extendsEnv}(ew', cw, ew) \wedge \\
\quad \mathit{extendsDecl}(dw', cw, dw) \wedge \mathit{extendsTheory}(tw', cw, tw) \wedge \\
\quad \forall t, t' \in \mathit{State}_w, vw \in \mathit{Value}_w, : {<}t', cw{>} \longrightarrow {<}t', vw{>} \\
\quad\; \Rightarrow \\
\quad\quad \exists s, s' \in \mathit{State}_m : \mathit{equals}(s, t) \wedge [\![C]\!](e)(s, s') \wedge \\
\quad\quad \forall s, s' \in \mathit{State}_m, dm \in \mathit{InfoData} : \mathit{equals}(s, t) \wedge \\
\quad\quad [\![C]\!](e)(s, s') \wedge dm = \mathit{infoData}(s') \\
\quad\quad\; \Rightarrow \mathit{equals}(s', t') \wedge \mathit{equals}(dm, vw)
\end{array}
$$

The formulation of the soundness statement for a command C is very similar to the soundness of command sequence $Cseq$ as stated above.

- *Soundness_e* $\subseteq$ *Expression*: defines the soundness statement for a *Mini-Maple* expression as below:

$$
\begin{array}{l}
\mathit{Soundness\_e}(E) \Leftrightarrow \\
\forall\ em \in \mathit{Environment}, expw \in \mathit{Exprression}_w, ew, ew' \in \mathit{Environment}_w, \\
\quad dw, dw' \in \mathit{Decl}_w, tw, tw' \in \mathit{Theory}_w : \\
\quad \mathit{wellTyped}(em, E) \wedge \mathit{consistent}(em, ew, dw, tw) \wedge \\
\quad {<}expw, ew', dw', tw'{>} = \mathrm{T}[\![E]\!](em, ew, dw, tw) \\
\quad \Rightarrow \\
\quad\quad \mathit{wellTyped}(expw, ew', dw', tw') \wedge \mathit{extendsEnv}(ew', expw, ew) \wedge \\
\quad\quad \mathit{extendsDecl}(dw', expw, dw) \wedge \mathit{extendsTheory}(tw', expw, tw) \wedge \\
\quad\quad \forall t, t' \in \mathit{State}_w, vw \in \mathit{Value}_w, : {<}t', cw{>} \longrightarrow {<}t', vw{>} \\
\quad\quad\; \Rightarrow \\
\quad\quad\quad \exists s, s' \in \mathit{State}_m, vm \in \mathit{Value} : \mathit{equals}(s, t) \wedge [\![E]\!](e)(s, s', vm) \wedge \\
\quad\quad\quad \forall s, s' \in \mathit{State}_m, vm \in \mathit{Value} : \mathit{equals}(s, t) \wedge [\![E]\!](e)(s, s', vm) \\
\quad\quad\quad\; \Rightarrow \mathit{equals}(s', t') \wedge \mathit{equals}(dm, vw)
\end{array}
$$

In detail, the soundness statement for the expression E states that

  - if an expression E translates to Why3 expression *expw* such that various predicates holds for E, e.g. well-typeness then,
  - various predicates also hold for the corresponding translated expression *expw*, e.g. extension of the declarations *extendsDecl* and theory *extendsTheory* and
  - if for arbitrary Why3 states t and t', execution of the translated expression *expw* in state t yields to a post-state t' and a value vw then,
  - there are corresponding *MiniMaple* states (s and s') and a value vm such that the states s and t are equal and evaluation of the expression E in this state s yields to a state s' and a value vm and
  - if for arbitrary *MiniMaple* states (s and s') and value vm, corresponding states s and t are equal; and with a given environment e evaluation of E in a pre-state s yields a post-state s' and a value vm then,
  - the corresponding post-states s' and t' are equals and also the corresponding values vm and vw are equal.

For further technical details and definitions of other predicates used in the soundness statements, please see Appendix B.

2.4 Proof of Soundness

In this section, we sketch the structure and strategy for the proof of the soundness of the selected *MiniMaple* constructs, i.e. command sequence and conditional, assignment and while-loop commands. In order to carry the proof, we have slightly modified the grammar for *MiniMaple* as shown below:

Cseq := C | C;Cseq // originally was EMPTY | C;Cseq
C := ... | **if** E **then** Cseq **else** Cseq **end if** | **while** E **do** Cseq **end do** | ...
E := ... | E **and** E | E **or** E | E = E | E <E | E $\leq$ E | E >E | E $\geq$ E | **not** E | ...
Eseq := E | E;Eseq // originally was EMPTY | E;Eseq

We prove the goal (as formulated in Section 2.3) by structural induction on $Cseq$, C and E whose formal grammar rules are defined. Also the rules for the questioned semantics of Why3 are defined by "$\_ \longrightarrow \_$" notation as introduced in Section 1 and in [1]. Hence, the goal splits into the following subgoals:

1. $Soundness\_cseq(Cseq)$
2. $Soundness\_c(C)$
3. $Soundness\_e(E)$

In the following subsection, we give the sketch of the proof of some of the structural cases of $Cseq$ and C. Based on our proof strategy, the corresponding proof for the rest of the constructs is an easy exercise to rehearse.

2.4.1 Command Sequence

As per the grammar for command sequence $Cseq$ above, there are two cases. In this section, we discuss the proof of the complex case, i.e. when $Cseq$ is $C; Cseq$. In order to prove, first we expand the definition of the goal $Soundness\_cseq(Cseq)$, where $Cseq = C; Cseq$ and get

$$
\begin{array}{l}
\forall\ em \in Environment, cw \in Exprression_w, ew, ew' \in Environment_w,\\
\quad dw, dw' \in Decl_w, tw, tw' \in Theory_w :\\
\quad wellTyped(em, C; Cseq) \wedge consistent(em, ew, dw, tw) \wedge\\
\quad <cw, ew', dw', tw'> = \mathrm{T}[\![C; Cseq]\!](em, ew, dw, tw)\\
\quad \Rightarrow\\
\qquad wellTyped(cw, ew', dw', tw') \wedge extendsEnv(ew', cw, ew) \wedge\\
\qquad extendsDecl(dw', cw, dw) \wedge extendsTheory(tw', cw, tw) \wedge\\
\qquad \forall t, t' \in State_w, vw \in Value_w : <t', cw> \longrightarrow <t', vw>\\
\qquad\ \Rightarrow\\
\qquad\quad \exists s, s' \in State_m : equals(s, t) \wedge [\![C; Cseq]\!](e)(s, s') \wedge\\
\qquad\quad \forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge\\
\qquad\quad [\![C; Cseq]\!](e)(s, s') \wedge dm = infoData(s')\\
\qquad\qquad \Rightarrow equals(s', t') \wedge equals(dm, vw)
\end{array}
$$

Let $em, cw, em, ew', dw, dw', tw, tw'$, be arbitrary but fixed.

We assume:

$$wellTyped(em, C; Cseq) \tag{2.4.1.1}$$

$$consistent(em, ew, dw, tw) \tag{2.4.1.2}$$

$$<cw, ew', dw', tw'> = T[C; Cseq](em, ew, dw, tw) \tag{2.4.1.3}$$

We show:

- $wellTyped(cw, ew', dw', tw')$ (a)
- $extendsEnv(ew', cw, ew)$ (b)
- $extendsDecl(dw', cw, dw)$ (c)
- $extendsTheory(tw', cw, tw)$ (d)
- $\forall t, t' \in State_w, vw \in Value_w : <t', cw> \longrightarrow <t', vw>$
  $\Rightarrow$
  $\exists s, s' \in State_m : equals(s, t) \wedge [\![C; Cseq]\!](e)(s, s') \wedge$
  $\forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge$
  $[\![C; Cseq]\!](e)(s, s') \wedge dm = infoData(s')$
  $\Rightarrow equals(s', t') \wedge equals(dm, vw)$ (e)

In the following, we prove each of the above five goals.

Goal (a)

We instantiate lemma $(L - cseq1)$ with

$cseq$ as $C; Cseq$, em as em, e as cw, ew as ew, ew' as ew', dw as dw, dw' as dw', tw as tw, tw' as tw' and get

$wellTyped(em, C; Cseq) \wedge <cw, ew', dw', tw'> = \mathrm{T}[\![C; Cseq]\!](em, ew, dw, tw)$
$\Rightarrow wellTyped(cw, ew', dw', tw')$

This goal follows from assumptions (2.4.1.1) and (2.4.1.3).

Goal (b)

By the definition of the translation function $(D2)$ of $\mathrm{T}[\![C; Cseq]\!]$, there are $e1, e2, ew'', dw'', tw''$ for which

$$<cw, ew', dw', tw'> = T[C; Cseq](em, ew, dw, tw) \tag{2.4.1.4}$$

where

$$cw = e1; e2 \tag{2.4.1.5}$$

$$<e1, ew'', dw''', tw''> = T[C](em, ew, dw, tw) \tag{2.4.1.6}$$

$$em' = Env(em, C) \tag{2.4.1.7}$$

$$<e2, ew', dw', tw'> = T[Cseq](em', ew'', dw'', tw'') \tag{2.4.1.8}$$

Here $e1; e2$ is a syntactic sugar for the Why3 semantic construct **let** $\_ = e1$ **in** $e2$.

We instantiate lemma $(L - cseq3)$ with
em as em, em' as em', C as C and $Cseq$ as $Cseq$
from which the following holds

$$wellTyped(em, C) \tag{2.4.1.9}$$

$$em' = Env(em, C) \tag{2.4.1.10}$$

$$wellTyped(em', Cseq) \tag{2.4.1.11}$$

We instantiate the soundness statement for C with
em as em, cw as e1, ew as ew, ew' as ew'', dw as dw, dw' as dw'', tw as tw, tw' as tw'' to get

$$
\begin{array}{l}
wellTyped(em, C) \wedge consistent(em, ew, dw, tw) \wedge \\
<e1, ew'', dw'', tw''> = \mathrm{T}[\![C]\!](em, ew, dw, tw) \\
\Rightarrow \\
\quad wellTyped(e1, ew'', dw'', tw'') \wedge extendsEnv(ew'', e1, ew) \wedge \\
\quad extendsDecl(dw'', e1, dw) \wedge extendsTheory(tw'', e1, tw) \wedge \\
\quad \forall t, t' \in State_w, vw \in Value_w, : <t', e1> \longrightarrow <t', vw> \\
\quad\ \Rightarrow \\
\qquad \exists s, s' \in State_m : equals(s, t) \wedge [\![C]\!](e)(s, s') \wedge \\
\qquad \forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge \\
\qquad [\![C]\!](e)(s, s') \wedge dm = infoData(s') \\
\qquad\ \Rightarrow equals(s', t') \wedge equals(dm, vw) \qquad\qquad \text{(A)}
\end{array}
$$

From (A) and assumptions (2.4.1.9), (2.4.1.2) and (2.4.1.6), it follows that

$$extendsEnv(ew'', e1, ew) \tag{2.4.1.12}$$

We instantiate lemma $(L - cseq4)$ with
em as em, em' as em', C as C, $Cseq$ as $Cseq$, ew as ew, ew' as ew', e1 as e1, e2 as $e2$, dw as dw, dw' as dw', tw as tw, tw' as tw', ew'' as ew'', dw'' as dw'', tw'' as tw'' to get

$$
\begin{array}{l}
<e1, ew'', dw'', tw''> = \mathrm{T}[\![C]\!](em, ew, dw, tw) \wedge em' = Env(em, C) \wedge \\
<e2, ew', dw', tw'> = \mathrm{T}[\![Cseq]\!](em', ew'', dw'', tw'') \wedge consistent(em, ew, dw, tw) \\
\Rightarrow consistent(em', dw'', dw'', tw'') \qquad\qquad \text{(B)}
\end{array}
$$

From (B) with assumptions (2.4.1.6), (2.4.1.6), (2.4.1.8) and (2.4.1.2), it follows that

$$consistent(em', ew'', dw'', tw'') \tag{2.4.1.13}$$

We instantiate the induction assumption for $Cseq$ with
em as em', cw as $e2$, ew as ew'', ew' as ew', dw as dw'', dw' as dw', tw as tw'', tw' as tw' to get

$$
\begin{array}{l}
wellTyped(em', Cseq) \wedge consistent(em', ew'', dw'', tw'') \wedge \\
<e2, ew', dw', tw'> = \mathrm{T}[\![Cseq]\!](em', ew'', dw'', tw'') \\
\Rightarrow \\
\quad wellTyped(e2, ew', dw', tw') \wedge extendsEnv(ew', e2, ew'') \wedge \\
\quad extendsDecl(dw', e2, dw'') \wedge extendsTheory(tw', e2, tw'') \wedge
\end{array}
$$

$$\forall t, t' \in State_w, vw \in Value_w, : <t', e2> \longrightarrow <t', vw>$$
$$\Rightarrow$$
$$\exists s, s' \in State_m : equals(s,t) \wedge [\![Cseq]\!](e)(s,s') \wedge$$
$$\forall s, s' \in State_m, dm \in InfoData : equals(s,t) \wedge$$
$$[\![Cseq]\!](e)(s,s') \wedge dm = infoData(s')$$
$$\Rightarrow equals(s',t') \wedge equals(dm, vw) \qquad \text{(C)}$$

From (C) with assumptions (2.4.1.11), (2.4.1.13) and (2.4.1.8), it follows that

$$extendsEnv(ew', e2, ew'') \qquad (2.4.1.14)$$

From (2.4.1.5), we can re-write the goal (b) as

$$extendsEnv(ew', e1; e2, ew)$$

In order to prove this goal, we instantiate lemma $(L - cseq2)$ with em as em, C as C, $Cseq$ as $Cseq$, ew as ew, ew' as ew', ew'' as ew'', $e1$ as $e1$, $e2$ as $e2$, dw as dw, dw' as dw', dw'' as dw'', tw as tw, tw' as tw', tw'' as tw'' to get

$$wellTyped(em, C; Cseq) \wedge <e1; e2, ew', dw', tw'> = \mathrm{T}[\![C; Cseq]\!](em, ew, dw, tw)$$
$$\Rightarrow$$
$$[extendsEnv(ew'', e1, ew) \wedge extendsEnv(ew', e2, ew'') \Rightarrow extendsEnv(ew', e1; e2, ew)] \wedge$$
$$[extendsDecl(dw'', e1, dw) \wedge extendsDecl(dw', e2, dw'') \Rightarrow extendsDecl(dw', e1; e2, dw)] \wedge$$
$$extendsTheory(tw'', e1, tw) \wedge extendsTheory(tw', e2, tw'') \Rightarrow$$
$$extendsTheory(tw', e1; e2, tw)] \qquad \text{(D)}$$

The goal (b) follows from (D) and assumptions (2.4.1.1), (2.4.1.4), (2.4.1.5), (2.4.1.12) and (2.4.1.14). Hence proved.

Goals (c) and (d)

The goals (c) and (d) are very similar to goal (b) and thus can be easily rehearsed based on the proof of goal (b).

Goal (e)

Let t, t', cw, vw be arbitrary but fixed.

We assume:

$$<t, cw> \longrightarrow <t', vw> \qquad (2.4.1.15)$$

From (2.4.1.5), and Why3 semantics, we know

$$cw = e1; e2 \sim \mathbf{let}\_ = e1 \mathbf{in} e2 \qquad (2.4.1.16)$$

From Why3 semantics $(com - s)$, we get

$$<t, \mathbf{let}\_ = e1 \mathbf{in} e2> \longrightarrow <t', vw> \qquad (2.4.1.17)$$
$$<t, e1> \longrightarrow <t'', vw'> \qquad (2.4.1.18)$$

for some t'', where vw' is not an exception

$$<t'', e2> \longrightarrow <t', vw> \qquad (2.4.1.19)$$

for some t''.

We show:

$\exists s, s' \in State : equals(s,t) \wedge [\![C; Cseq]\!](em)(s,s')$ (e.a)

$\forall s, s' \in State, dm \in InfoData : equals(s,t) \wedge [\![C; Cseq]\!](em)(s,s') \wedge dm = infoData(s')$
$\Rightarrow equals(s',t') \wedge equals(dm, vw)$ (e.b)

In the following, we prove these two sub-goals (e.a) and (e.b) of goal (e).

Sub-Goal (e.a)

To prove this goal, we define

$$s := constructs(t) \qquad (2.4.1.20)$$

We split the original goal (e.a) and show the following sub-goals:

$equals(s,t)$ (e.a.1)

$[\![C; Cseq]\!](em)(s,s')$ (e.a.2)

Now, we prove the following two further sub-goals (e.a.1) and (e.a.2) in order to prove the goal (e.a).

Sub-Goal (e.a.1)

We instantiate lemma $(L - cseq5)$ with s as s and t as t to get

$s = construct(t) \Rightarrow equals(s,t)$ (E)

The sub-goal (e.a.1) follows from (E) with assumption (2.4.1.20). Hence proved.

Sub-Goal (e.a.2)

We instantiate the soundness statement for C with
em as em, cw as $e1$, ew as ew, ew' as ew'', dw as dw, dw' as dw'', tw as tw, tw' as tw'' to get

$wellTyped(em, C) \wedge consistent(em, ew, dw, tw) \wedge$
$<e1, ew'', dw'', tw''> = \mathrm{T}[\![C]\!](em, ew, dw, tw)$
$\Rightarrow$
$wellTyped(e1, ew'', dw'', tw'') \wedge extendsEnv(ew'', e1, ew) \wedge$
$extendsDecl(dw'', e1, dw) \wedge extendsTheory(tw'', e1, tw) \wedge$
$\forall t, t' \in State_w, vw \in Value_w, : <t', e1> \longrightarrow <t', vw>$
$\Rightarrow$
$\exists s, s' \in State_m : equals(s,t) \wedge [\![C]\!](e)(s,s') \wedge$
$\forall s, s' \in State_m, dm \in InfoData : equals(s,t) \wedge$
$[\![C]\!](e)(s,s') \wedge dm = infoData(s')$
$\Rightarrow equals(s',t') \wedge equals(dm, vw)$ (F)

From (F) with assumptions (2.4.1.9), (2.4.1.2), (2.4.1.6), we get

$$\forall t, t' \in State_w, vw \in Value_w : <t', e1> \longrightarrow <t', vw>$$
$$\Rightarrow$$
$$\exists s, s' \in State_m : equals(s, t) \wedge [\![C]\!](e)(s, s') \wedge$$
$$\forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge$$
$$[\![C]\!](e)(s, s') \wedge dm = infoData(s')$$
$$\Rightarrow equals(s', t') \wedge equals(dm, vw) \quad \text{(F.1)}$$

We instantiate the above formula (F.1) with
t as t and t' as t'', vw as vw' to get

$$\forall t, t'' \in State_w, vw' \in Value_w : <t', e1> \longrightarrow <t'', vw'>$$
$$\Rightarrow$$
$$\exists s, s' \in State_m : equals(s, t) \wedge [\![C]\!](e)(s, s') \wedge$$
$$\forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge$$
$$[\![C]\!](e)(s, s') \wedge dm = infoData(s')$$
$$\Rightarrow equals(s', t'') \wedge equals(dm, vw') \quad \text{(F.2)}$$

From (F.2) with assumption (2.4.1.18), we know
$\exists s, s' \in State : equals(s, t) \wedge [\![C]\!](em)(s, s')$ (F.3)
By instantiating (F.3) with s as s, s' as s'', we know that
there is s, s'' s.t.

$$[C](em)(s, s'') \quad (2.4.1.21)$$

We instantiate the induction assumption for $Cseq$ with
em as em', cw as $e2$, ew as ew'', ew' as ew', dw as dw'', dw' as dw', tw as tw'', tw' as tw' to get

$$wellTyped(em', Cseq) \wedge consistent(em', ew'', dw'', tw'') \wedge$$
$$<e2, ew', dw', tw'> = \mathrm{T}[\![Cseq]\!](em', ew'', dw'', tw'')$$
$$\Rightarrow$$
$$wellTyped(e2, ew', dw', tw') \wedge extendsEnv(ew', e2, ew'') \wedge$$
$$extendsDecl(dw', e2, dw'') \wedge extendsTheory(tw', e2, tw'') \wedge$$
$$\forall t, t' \in State_w, vw \in Value_w, : <t', e2> \longrightarrow <t', vw>$$
$$\Rightarrow$$
$$\exists s, s' \in State_m : equals(s, t) \wedge [\![Cseq]\!](em')(s, s') \wedge$$
$$\forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge$$
$$[\![Cseq]\!](em')(s, s') \wedge dm = infoData(s')$$
$$\Rightarrow equals(s', t') \wedge equals(dm, vw) \quad \text{(G)}$$

From (G) with assumptions (2.4.1.11), (2.4.1.13) and (2.4.1.8), it follows that

$$\forall t, t' \in State_w, vw \in Value_w : <t, e2> \longrightarrow <t', vw>$$
$$\Rightarrow$$
$$\exists s, s' \in State_m : equals(s, t) \wedge [\![Cseq]\!](em')(s, s') \wedge$$
$$\forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge$$
$$[\![Cseq]\!](em')(s, s') \wedge dm = infoData(s')$$
$$\Rightarrow equals(s', t'') \wedge equals(dm, vw') \quad \text{(G.1)}$$

We instantiate the formula (G.1) with t as t'', t' as t', vw as vw to get

$\forall t'', t' \in State_w, vw \in Value_w : <t'', e2> \longrightarrow <t', vw>$
$\Rightarrow$
$\exists s, s' \in State_m : equals(s, t) \wedge [\![Cseq]\!](em')(s, s') \wedge$
$\forall s, s' \in State_m, dm \in InfoData : equals(s, t'') \wedge$
$[\![C]\!](em')(s, s') \wedge dm = infoData(s')$
$\Rightarrow equals(s', t') \wedge equals(dm, vw')$ (G.2)

From (G.2) and assumption (2.4.1.19), we get
$\exists s, s' \in State : equals(s, t'') \wedge [\![Cseq]\!](em')(s, s')$ (G.3)
By instantiating (G.3) with s as s'', s' as s', we know that there is s'', s' s.t.

$$[Cseq](em')(s'', s') \quad (2.4.1.22)$$

This sub-goal (e.a.2), which is a definition of the semantics of the command sequence $C; Cseq$ follows from the assumptions (2.4.1.21), (2.4.1.22) and (2.4.1.7).

Hence sub-goals (e.a.1) and (e.a.2) are proved thus the sub-goal (e.a) is proved.

Sub-Goal (e.b)

Let s, s', dm be arbitrary but fixed.

We assume:

$$equals(s, t) \quad (2.4.1.23)$$

$$[C; Cseq](em)(s, s') \quad (2.4.1.24)$$

$$dm = infoData(s') \quad (2.4.1.25)$$

We define:

$$s' := constructs(t') \quad (2.4.1.26)$$

$$vw := constructs(dm) \quad (2.4.1.27)$$

To prove this goal, we split the original goal (e.b) and show the following sub-goals:

$equals(s', t')$ (e.b.1)

$equals(dm, vw)$ (e.b.2)

In the following, we prove the sub-goals (e.b.1) and (e.b.2) in order to prove the original goal (e.b).

Sub-Goal (e.b.1)

We instantiate lemma $(L - cseq5)$ with
s as s' and t as t' to get

$s' = constructs(t') \Rightarrow equals(s', t')$ (I)

This sub-goal follows from (I) with assumption (2.4.1.26).

Sub-Goal (e.b.2)

We instantiate lemma $(L - cseq6)$ with v as vw, v' as dm to get

$vw = constructs(dm) \Rightarrow equals(dm, vw)$ (J)

This sub-goal follows from (J) with assumption (2.4.1.27).

Consequently, the goal (e.b) follows from (e.b.1) and (e.b.2); also the goal (e) follows from goals (e.a) and (e.b).

Thus the soundness statement for command sequence follows from sub-goals (a), (b), (c), (d) and (e).

2.4.2 Conditional and Assignment

The proof structure respective strategy for the soundness of a conditional command is the same as shown above for the command sequence. However, the proof of a conditional command later splits into two cases, when the conditional expression E evaluates to *true* or *false*. The soundness proof for the assignment command is also similar to the soundness proof for a command sequence thus can be easily rehearsed. The complete proof for the conditional and assignment command is shown in the Appendix D.

2.4.3 While-loop

The goal for the soundness of command can be re-stated for the while-loop command as:

$$\begin{aligned}
&\forall\ em \in Environment, e1, e2 \in Exprression_w, ew, ew' \in Environment_w,\\
&\quad dw, dw' \in Decl_w, tw, tw' \in Theory_w :\\
&\quad wellTyped(em, \textbf{while}\ E\ \textbf{do}\ Cseq\ \textbf{end})\ mathit \wedge consistent(em, ew, dw, tw) \wedge\\
&\quad < \textbf{while}\ e1\ \textbf{do}\ e2, \text{ew'}, \text{dw'}, \text{tw'}>= \text{T}[\![\textbf{while}\ e1\ \textbf{do}\ e2]\!](em, ew, dw, tw)\\
&\quad \Rightarrow\\
&\qquad wellTyped(\textbf{while}\ e1\ \textbf{do}\ e2, ew', dw', tw') \wedge extendsEnv(ew', \textbf{while}\ e1\ \textbf{do}\ e2, ew) \wedge\\
&\qquad extendsDecl(dw', \textbf{while}\ e1\ \textbf{do}\ e2, dw) \wedge extendsTheory(tw', \textbf{while}\ e1\ \textbf{do}\ e2, tw) \wedge\\
&\qquad \forall t, t' \in State_w, vw \in Value_w, : <t, \textbf{while}\ e1\ \textbf{do}\ e2> \longrightarrow <t', vw>\\
&\qquad\quad \Rightarrow\\
&\qquad\qquad \exists s, s' \in State_m : equals(s, t) \wedge [\![\textbf{while}\ E\ \textbf{do}\ Cseq\ \textbf{end}]\!](em)(s, s') \wedge\\
&\qquad\qquad \forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge\\
&\qquad\qquad [\![\textbf{while}\ E\ \textbf{do}\ Cseq\ \textbf{end}]\!](em)(s, s') \wedge dm = infoData(s')\\
&\qquad\qquad\quad \Rightarrow equals(s', t') \wedge equals(dm, vw)
\end{aligned}$$

Let $em, e1, e2, ew, ew', dw, dw', tw, tw', dm$ and vw be arbitrary but fixed.

We assume:

$$wellTyped(em, \textbf{while}\ E\ \textbf{do}\ Cseq\ \textbf{end}) \tag{2.4.3.1}$$

$$consistent(em, ew, dw, tw) \tag{2.4.3.2}$$

$$<\textbf{while}\ e1\ \textbf{do}\ e2, ew', dw', tw'> = T[\textbf{while}\ E\ \textbf{do}\ Cseq\ \textbf{end}](em, ew, dw, tw) \tag{2.4.3.3}$$

By expanding the definition of (2.4.3.3), we know

$$<e1, ew'', dw'', tw''> = T[E](em, ew, dw, tw) \tag{2.4.3.4}$$

$$em' = Env(em, E) \tag{2.4.3.5}$$

$$<e2, ew', dw', tw'> = T[Cseq](em', ew'', dw'', tw'') \tag{2.4.3.6}$$

We show:

- $wellTyped(\textbf{while } e1 \textbf{ do } e2, ew', dw', tw')$ (a)
- $extendsEnv(ew', \textbf{while } e1 \textbf{ do } e2, ew)$ (b)
- $extendsDecl(dw', \textbf{while } e1 \textbf{ do } e2, dw)$ (c)
- $extendsTheory(tw', \textbf{while } e1 \textbf{ do } e2, tw)$ (d)
- $\forall t, t' \in State_w, vw \in Value_w : <t', \textbf{while } e1 \textbf{ do } e2> \longrightarrow <t', vw>$
  $\Rightarrow$
  $\exists s, s' \in State_m : equals(s, t) \wedge [\![\textbf{while } E \textbf{ do } Cseq\textbf{end}]\!](em)(s, s') \wedge$
  $\forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge$
  $[\![\textbf{while } E \textbf{ do } Cseq \textbf{ end}]\!](e)(s, s') \wedge dm = infoData(s')$
  $\Rightarrow equals(s', t') \wedge equals(dm, vw)$ (e)

In the following, we prove each of the above five goals.

Goal (a)

We instantiate lemma $(L - c1)$ with
c as **while** E **do** $Cseq$ **end**, em as em, e as cw, ew as ew, ew' as ew', dw as dw, dw' as dw', tw as tw, tw' as tw' and get

$wellTyped(em, \textbf{while } E \textbf{ do } Cseq \textbf{ end}) \wedge$
$<cw, ew', dw', tw'> = \mathrm{T}[\![\textbf{while } E \textbf{ do } Cseq \textbf{ end}]\!](em, ew, dw, tw)$
$\Rightarrow wellTyped(\textbf{while } e1 \textbf{ do } e2, ew', dw', tw')$

This goal follows from assumptions (2.4.3.1) and (2.4.3.3).

Goal (b)

We instantiate lemma $(L - c9)$ with
em as em, em' as em', E as E, $Cseq$ as $Cseq$ to get

$wellTyped(em, \textbf{while } E \textbf{ do } Cseq \textbf{ end}) \Rightarrow$
$wellTyped(em, E) \wedge em' = Env(em, E) \wedge wellTyped(em', Cseq)$

From the above formula with assumptions (2.4.3.1), we know

$$wellTyped(em, E) \tag{2.4.3.7}$$

$$em' = Env(em, E) \tag{2.4.3.8}$$

$$wellTyped(em', Cseq) \tag{2.4.3.9}$$

We instantiate lemma $(L - c10)$ with
em as em, em' as em', E as E, $Cseq$ as $Cseq$, ew as ew, ew' as ew', ew'' as ew'', dw as dw, dw' as dw', dw'' as dw'', tw as tw, tw' as tw', tw'' as tw'' to get

$<e1, ew'', dw'', tw''> = \mathrm{T}[\![E]\!](em, ew, dw, tw) \wedge em' = Env(em, E) \wedge$
$<e2, ew', dw', tw'> = \mathrm{T}[\![Cseq]\!](em', ew'', dw'', tw'') \wedge consistent(em, ew, dw, tw)$
$\Rightarrow consistent(em', dw'', dw'', tw'')$

From the above formula with assumptions (2.4.3.4), (2.4.3.5), (2.4.3.6), (2.4.3.2), we know

$$consistent(em', ew'', dw'', tw'') \qquad (2.4.3.10)$$

We instantiate the soundness statement for E with
em as em, $expw$ as e1, ew as ew, ew' as ew'', dw as dw, dw' as dw'', tw as tw, tw' as tw'' to get

$$\begin{array}{l}
wellTyped(em, E) \wedge consistent(em, ew, dw, tw) \wedge \\
<e1, ew'', dw'', tw''> =\mathrm{T}[\![E]\!](em, ew, dw, tw) \\
\Rightarrow \\
\quad wellTyped(e1, ew'', dw'', tw'') \wedge extendsEnv(ew'', e1, ew) \wedge \\
\quad extendsDecl(dw'', e1, dw) \wedge extendsTheory(tw'', e1, tw) \wedge \\
\quad \forall t, t' \in State_w, vw \in Value_w, : <t', e1> \longrightarrow <t', vw> \\
\quad\; \Rightarrow \\
\quad\quad \exists s, s' \in State_m, vm \in Value : equals(s, t) \wedge [\![E]\!](em)(s, s', vm) \wedge \\
\quad\quad \forall s, s' \in State_m, vm \in Value : equals(s, t) \wedge \\
\quad\quad [\![E]\!](em)(s, s', wm) \\
\quad\quad\; \Rightarrow equals(s', t') \wedge equals(vm, vw) \qquad (A)
\end{array}$$

From (A) and assumptions (2.4.3.9), (2.4.3.2) and (2.4.3.6), it follows that

$$extendsEnv(ew'', e1, ew) \qquad (2.4.3.11)$$

We instantiate the soundness statement for $Cseq$ with
em as em', cw as $e2$, ew as ew'', ew' as ew', dw as dw'', dw' as dw', tw as tw'', tw' as tw' to get

$$\begin{array}{l}
wellTyped(em', Cseq) \wedge consistent(em', ew'', dw'', tw'') \wedge \\
<e2, ew', dw', tw'> =\mathrm{T}[\![Cseq]\!](em', ew'', dw'', tw'') \\
\Rightarrow \\
\quad wellTyped(e2, ew', dw', tw') \wedge extendsEnv(ew', e2, ew'') \wedge \\
\quad extendsDecl(dw', e2, dw'') \wedge extendsTheory(tw', e2, tw'') \wedge \\
\quad \forall t, t' \in State_w, vw \in Value_w, : <t, e2> \longrightarrow <t', vw> \\
\quad\; \Rightarrow \\
\quad\quad \exists s, s' \in State_m : equals(s, t) \wedge [\![Cseq]\!](em)(s, s') \wedge \\
\quad\quad \forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge \\
\quad\quad [\![Cseq]\!](em)(s, s') \wedge dm = infoData(s') \\
\quad\quad\; \Rightarrow equals(s', t') \wedge equals(dm, vw) \qquad (B)
\end{array}$$

From (B) and assumptions (2.4.3.9), (2.4.3.2) and (2.4.3.6), it follows that

$$extendsEnv(ew', e2, ew'') \qquad (2.4.3.12)$$

We instantiate lemma $(L - c11)$ with
em as em, E as E, $Cseq$ as $Cseq$, e1 as e1, e2 as e2, ew as ew, ew' as ew',ew'' as ew'', dw as dw, dw' as dw', dw'' as dw'', tw as tw, tw' as tw', tw'' as tw'' to get

$$\begin{array}{l}
wellTyped(em, \textbf{while } E \textbf{ do } Cseq \textbf{ end}) \wedge \\
<\textbf{while } e1 \textbf{ do } e2, ew', dw', tw'> = \mathrm{T}[\![\textbf{while } E \textbf{ do } Cseq \textbf{ end}]\!](em, ew, dw, tw) \wedge
\end{array}$$

$$
\begin{aligned}
&<e1, ew'', dw'', tw''> = \mathrm{T}[\![E]\!](em, ew, dw, tw)\wedge\\
&em' = Env(em, E)\wedge\\
&<e2, ew', dw', tw'> = \mathrm{T}[\![Cseq]\!](em', ew'', dw'', tw'')\\
&\Rightarrow\\
&\quad [extendsEnv(ew'', e1, ew)\wedge extendsEnv(ew', e2, ew'')\\
&\qquad\qquad \Rightarrow extendsEnv(ew', \textbf{while}\ e1\ \textbf{do}\ e2, ew)]\wedge\\
&\quad [extendsDecl(dw'', e1, dw) \wedge extendsDecl(dw', e2, dw'')\\
&\qquad\qquad \Rightarrow extendsDecl(dw', \textbf{while}\ e1\ \textbf{do}\ e2, dw)]\wedge\\
&\quad [extendsTheory(tw'', e1, tw) \wedge extendsTheory(tw', e2, tw'')\\
&\qquad\qquad \Rightarrow extendsTheory(tw', \textbf{while}\ e1\ \textbf{do}\ e2, tw)] \qquad \text{(C)}
\end{aligned}
$$

From (C) with assumptions (2.4.3.1), (2.4.3.3), (2.4.3.4), (2.4.3.5), (2.4.3.6), (2.4.3.11) and (2.4.3.12) , we know

$$extendsEnv(ew', whilee1doe2, ew) \qquad (2.4.3.13)$$

which is goal (b). Hence proved.

Goals (c) and (d)

The goals (c) and (d) are very similar to goal (b) above and thus can be easily rehearsed based on the proof of goal (b).

Goal (e)

Let t, t', cw, vw be arbitrary but fixed s.t.

We assume:

$$<t, \textbf{while}\ e1\ \textbf{do}\ e2> \longrightarrow <t', vw> \qquad (2.4.3.14)$$

We show:

$$\exists s, s' \in State : equals(s, t)\wedge [\![\textbf{while}\ E\ \textbf{do}\ Cseq\ \textbf{end}]\!](em)(s, s') \qquad \text{(e.a)}$$

$$
\begin{aligned}
&\forall s, s' \in State, dm \in InfoData : equals(s, t)\wedge\\
&[\![\textbf{while}\ E\ \textbf{do}\ Cseq\ \textbf{end}]\!](em)(s, s') \wedge dm = infoData(s')\\
&\Rightarrow equals(s', t') \wedge equals(dm, vw) \qquad \text{(e.b)}
\end{aligned}
$$

The semantics of the classical Why3 while-loop is defined by a complex exception-handling mechanism. Based on the aforementioned semantics, a proof of this goal gets more complicated, thus to avoid this complication, we have derived (in the Appendix H- Derivations) two rules conforming the definition of while-loop semantics which do not involve exceptions anymore. These two derivations are as follows:

$$\frac{<t, e1> \longrightarrow <t', false>}{<t, \textbf{while}\ e1\ \textbf{do}\ e2> \longrightarrow <t', void>} \qquad (2.4.3.15)$$

$$\frac{\begin{array}{r}<t, e1> \longrightarrow <t'', true>\\ <t'', e2> \longrightarrow <t''', void>\\ <t''', \textbf{while}\ e1\ \textbf{do}\ e2> \longrightarrow <t', void>\end{array}}{<t, \textbf{while}\ e1\ \textbf{do}\ e2> \longrightarrow <t', void>} \qquad (2.4.3.16)$$

We prove this goal (e) by rule induction [5] on the operational semantics of while-loop which is defined above by the two derivation rules (2.4.3.15) and (2.4.3.16). By the strategy of principle of rule induction for while-loop, the goal (e) can be re-formulated as:

$\forall t, t' \in Statew, vw \in Valuew : <t, \textbf{while}\ e1\ \textbf{do}\ e2> \longrightarrow <t', vw>$
$\Rightarrow P(t, t', vw)$ (e')

where

$P(t, t', vw) \Leftrightarrow$
$[\exists s, s' \in State : equals(s, t) \wedge [\![\textbf{while}\ E\ \textbf{do}\ Cseq\ \textbf{end}]\!](em)(s, s')] \wedge$
$[\forall s, s' \in State, dm \in InfoData :$
$equals(s', t') \wedge [\![\textbf{while}\ E\ \textbf{do}\ Cseq\ \textbf{end}]\!](em)(s, s') \wedge dm = infoData(s')$
$\Rightarrow equals(s', t') \wedge equals(dm, vw)]$ (D-p)

where $E, Cseq$ and em are fixed as defined above.

To show goal (e'), based on the principle of rule induction it suffices to show the followings for while-loop for the corresponding derivation rules respectively:

$\forall t, t' \in Statew, vw \in Valuew, e1 \in Expressionw :$
$<t, e1> \longrightarrow <t', false> \Rightarrow P(t, t', vw)$ (e.a)

$\forall t, t', t'', t''' \in Statew, vw \in Valuew, e1, e2 \in Expressionw :$
$<t, e1> \longrightarrow <t'', true> \wedge <t'', e2> \longrightarrow <t''', void> \wedge$
$<t''', \textbf{while}\ e1\ \textbf{do}\ e2> \longrightarrow <t', void> \wedge P(t''', t', void)$
$\Rightarrow P(t, t', vw)$ (e.b)

In the following, we prove these two sub-goals (e.a) and (e.b) in order to prove the goal (e).

Sub-Goal (e.a)

We assume:

$$<t, e1> \longrightarrow <t', false> \qquad (2.4.3.17)$$

We show:
$P(t, t', vw)$
By expanding the definition of $P(t, t', vw)$, we get

$[\exists s, s' \in State : equals(s, t) \wedge [\![\textbf{while}\ E\ \textbf{do}\ Cseq\ \textbf{end}]\!](em)(s, s')]$ (e.a.1)

$[\forall s, s' \in State, dm \in InfoData :$
$equals(s', t') \wedge [\![\textbf{while}\ E\ \textbf{do}\ Cseq\ \textbf{end}]\!](em)(s, s') \wedge dm = infoData(s')$
$\Rightarrow equals(s', t') \wedge equals(dm, vw)]$ (e.a.2)

In the following, we show the sub-goals (e.a.1) and (e.a.2).

Sub-Goal (e.a.1)

We split this goal to show:

$equals(s,t)$ (e.a.1.1)

$[\![\textbf{while } E \textbf{ do } Cseq \textbf{ end}]\!](em)(s,s')$ (e.a.1.2)

We define:

$$s := constructs(t) \tag{2.4.3.18}$$
$$s' := constructs(t') \tag{2.4.3.19}$$
$$inValue(False) := constructs(false) \tag{2.4.3.20}$$

In the following, we prove the sub-goals (e.a.1.1) and (e.a.1.2).

Sub-Goal (e.a.1.1)

We instantiate lemma $(L - cseq5)$ with s as s and t as t to get

$s = construct(t) \Rightarrow equals(s,t)$ (D)

The sub-goal (e.a.1.1) follows from (D) with assumption (2.4.3.18). Hence proved.

Sub-Goal (e.a.1.2)

We instantiate the soundness statement for E with
em as em, $expw$ as $e1$, ew as ew, ew' as ew'', dw as dw, dw' as dw'', tw as tw, tw' as tw'' to get

$wellTyped(em, E) \wedge consistent(em, ew, dw, tw) \wedge$
$<e1, ew'', dw'', tw''> = \text{T}[\![E]\!](em, ew, dw, tw)$
$\Rightarrow$
$wellTyped(e1, ew'', dw'', tw'') \wedge extendsEnv(ew'', e1, ew) \wedge$
$extendsDecl(dw'', e1, dw) \wedge extendsTheory(tw'', e1, tw) \wedge$
$\forall t, t' \in State_w, vw \in Value_w, : <t, e1> \longrightarrow <t', vw>$
$\Rightarrow$
$\exists s, s' \in State_m, vm \in Value : equals(s,t) \wedge [\![E]\!](em)(s, s', vm) \wedge$
$\forall s, s' \in State_m, vm \in Value : equals(s,t) \wedge$
$[\![E]\!](em)(s, s', wm)$
$\Rightarrow equals(s', t') \wedge equals(vm, vw)$ (E)

From (E) and assumptions (2.4.3.9), (2.4.3.2) and (2.4.3.6), it follows that

$\forall t, t' \in State_w, vw \in Value_w, : <t, e1> \longrightarrow <t', vw>$
$\Rightarrow$
$\exists s, s' \in State_m, vm \in Value : equals(s,t) \wedge [\![E]\!](em)(s, s', vm) \wedge$
$\forall s, s' \in State_m, vm \in Value : equals(s,t) \wedge$
$[\![E]\!](em)(s, s', wm)$
$\Rightarrow equals(s', t') \wedge equals(vm, vw)$ (E.1)

We instantiate above formula (E.1) with
t as t, t' as t', vw as $false$ to get

$$
\begin{array}{l}
<t, e1> \longrightarrow <t', false> \\
\Rightarrow \\
\quad \exists s, s' \in State_m, vm \in Value : equals(s,t) \wedge [\![E]\!](em)(s,s',vm) \wedge \\
\quad \forall s, s' \in State_m, vm \in Value : equals(s,t) \wedge \\
\quad [\![E]\!](em)(s,s',wm) \\
\quad\ \Rightarrow equals(s',t') \wedge equals(vm, vw) \qquad\qquad (E.2)
\end{array}
$$

From (E.2) with assumption (2.4.3.17), we get

$$\exists s, s' \in State, vm \in Value : equals(s,t) \wedge [\![E]\!](em)(s,s',vm) \tag{E.3}$$

Taking s as s, s' as s', vm as $inValue(False)$ with (E.3), we know from assumptions (2.4.3.18), (2.4.3.19), (2.4.3.20) and (2.4.3.4) that
there is s, s', $inValue(False)$ and E for which

$$[E](em)(s, s', inValue(False)) \tag{2.4.3.21}$$

We instantiate lemma $(L - c12)$ with
em as em, E as E, $Cseq$ as $Cseq$, s as s and s' as s' to get

$$[\![\mathrm{E}]\!](em)(s,s',inValue(False)) \Rightarrow [\![\textbf{while } E \textbf{ do } Cseq \textbf{ end}]\!](em)(s,s') \tag{E.4}$$

The sub-goal (e.a.1.2) follows from (E.4) with assumption (2.4.3.21).
Consequently, the goal (e.a.1) follows from (e.a.1.1) and (e.a.1.2).

Sub-Goal (e.a.2)

Let s, s', dm, t be arbitrary but fixed.
We assume:

$$equals(s,t) \tag{2.4.3.22}$$

$$[\textbf{while}E\textbf{do}Cseq\textbf{end}](em)(s,s') \tag{2.4.3.23}$$

$$dm = infoData(s') \tag{2.4.3.24}$$

We define:

$$vw := constructs(dm) \tag{2.4.3.25}$$

We split the original goal (e.a.2) and show the following sub-goals:

equals(s', t') (e.a.2.1)

$equals(dm, vw)$ (e.a.2.2)

Now, we prove the following two further sub-goals (e.a.2.1) and (e.a.2.2) in order to prove the goal (e.a.2).

Sub-Goal (e.a.2.1)

We instantiate lemma $(L - cseq5)$ with s as s and t as t to get

$s = construct(t) \Rightarrow equals(s, t)$ (F)

The sub-goal (e.a.2.1) follows from (F) with assumption (2.4.3.22). Hence proved.

Sub-Goal (e.a.2.2)

We instantiate lemma $(L - cseq6)$ with v as vm and v' as dm to get

$vw = construct(dm) \Rightarrow equals(dm, vw)$ (G)

The sub-goal (e.a.2.2) follows from (G) with assumption (2.4.3.25). Hence proved.

Consequently, the goal (e.a.2) follows from (e.a.2.1) and (e.a.2.2). Finally, the goal (e.a) follows from goals (e.a.1) and (e.a.2).

Sub-Goal (e.b)

We assume:

$$<t, e1> \longrightarrow <t'', true> \tag{2.4.3.26}$$

$$<t'', e2> \longrightarrow <t''', void> \tag{2.4.3.27}$$

$$<t''', \textbf{while } e1 \textbf{ do } e2> \longrightarrow <t', void> \tag{2.4.3.28}$$

$$P(t''', t', void) \tag{2.4.3.29}$$

We show:
$P(t, t', vw)$
By expanding the definition of $P(t, t', vw)$, we get

$[\exists s, s' \in State : equals(s, t) \wedge$ $[\![\textbf{while } E \textbf{ do } Cseq \textbf{ end}]\!](em)(s, s')]$ (e.b.1)

$[\forall s, s' \in State, dm \in InfoData :$
$equals(s', t') \wedge$ $[\![\textbf{while } E \textbf{ do } Cseq \textbf{ end}]\!](em)(s, s') \wedge dm = infoData(s')$
$\Rightarrow equals(s', t') \wedge equals(dm, vw)]$ (e.b.2)

We define:

$$s := constructs(t) \tag{2.4.3.30}$$

$$s'' := constructs(t'') \tag{2.4.3.31}$$

$$s''' := constructs(t''') \tag{2.4.3.32}$$

$$inValue(True) := constructs(true) \tag{2.4.3.33}$$

$$inValue(Void) := constructs(void) \tag{2.4.3.34}$$

In the following, we prove the sub-goals (e.b.1) and (e.b.2) in order to prove (e.b).

Sub-Goal (e.b.1)

We show:

$equals(s,t)$ (e.b.1.1)

$[\![\textbf{while } E \textbf{ do } Cseq \textbf{ end}]\!](em)(s,s')$ (e.b.1.2)

In the following, we show the sub-goals (e.b.1.1) and (e.b.1.2) to show the goal (e.b.1).

Sub-Goal (e.b.1.1)

We instantiate lemma $(L - cseq5)$ with s as s and t as t to get

$s = construct(t) \Rightarrow equals(s,t)$ (G)

The sub-goal (e.b.1.1) follows from (G) with assumption (2.4.3.30). Hence proved.

Sub-Goal (e.b.1.2)

We instantiate the soundness statement for E with
em as em, $expw$ as $e1$, ew as ew, ew' as ew'', dw as dw, dw' as dw'', tw as tw, tw' as tw'' to get

$$
\begin{aligned}
&wellTyped(em, E) \wedge consistent(em, ew, dw, tw) \wedge \\
&<e1, ew'', dw'', tw''> = \mathrm{T}[\![E]\!](em, ew, dw, tw) \\
&\Rightarrow \\
&\quad wellTyped(e1, ew'', dw'', tw'') \wedge extendsEnv(ew'', e1, ew) \wedge \\
&\quad extendsDecl(dw'', e1, dw) \wedge extendsTheory(tw'', e1, tw) \wedge \\
&\quad \forall t, t' \in State_w, vw \in Value_w, : <t, e1> \longrightarrow <t', vw> \\
&\quad\ \Rightarrow \\
&\qquad \exists s, s' \in State_m, vm \in Value : equals(s,t) \wedge [\![E]\!](em)(s, s', vm) \wedge \\
&\qquad \forall s, s' \in State_m, vm \in Value : equals(s,t) \wedge \\
&\qquad [\![E]\!](em)(s, s', wm) \\
&\qquad\ \Rightarrow equals(s', t') \wedge equals(vm, vw) \qquad \text{(H)}
\end{aligned}
$$

From (H) and assumptions (2.4.3.9), (2.4.3.2) and (2.4.3.6), it follows that

$$
\begin{aligned}
&\forall t, t' \in State_w, vw \in Value_w, : <t, e1> \longrightarrow <t', vw> \\
&\ \Rightarrow \\
&\quad \exists s, s' \in State_m, vm \in Value : equals(s,t) \wedge [\![E]\!](em)(s, s', vm) \wedge \\
&\quad \forall s, s' \in State_m, vm \in Value : equals(s,t) \wedge \\
&\quad [\![E]\!](em)(s, s', wm) \\
&\quad\ \Rightarrow equals(s', t') \wedge equals(vm, vw) \qquad \text{(H.1)}
\end{aligned}
$$

We instantiate above formula (H.1) with
t as t, t' as t'', vw as $true$ to get

$<t, e1> \longrightarrow <t'', true>$
$\Rightarrow$
$\quad \exists s, s' \in State_m, vm \in Value : equals(s,t) \wedge [\![E]\!](em)(s, s', vm) \wedge$
$\quad \forall s, s' \in State_m, vm \in Value : equals(s,t) \wedge$
$\quad [\![E]\!](em)(s, s', wm)$
$\quad\quad \Rightarrow equals(s', t'') \wedge equals(vm, vw) \qquad$ (H.2)

From (H.2) with assumption (2.4.3.26), we get

$$\exists s, s' \in State, vm \in Value : equals(s,t) \wedge [\![E]\!](em)(s, s', vm) \tag{H.3}$$

Taking s as s, s' as s'', vm as $inValue(True)$ with (H.3), we know from assumptions (2.4.3.30), (2.4.3.31), (2.4.3.33) and (2.4.3.4) that
there is s, s'', $inValue(True)$ and E for which

$$[E](em)(s, s'', inValue(True)) \tag{2.4.3.35}$$

We instantiate the soundness statement for $Cseq$ with
em as em', cw as $e2$, ew as ew'', ew' as ew', dw as dw'', dw' as dw', tw as tw'', tw' as tw' to get

$wellTyped(em', Cseq) \wedge consistent(em', ew'', dw'', tw'') \wedge$
$<e2, ew', dw', tw'> = \mathrm{T}[\![Cseq]\!](em', ew'', dw'', tw'')$
$\Rightarrow$
$\quad wellTyped(e2, ew', dw', tw') \wedge extendsEnv(ew', e2, ew'') \wedge$
$\quad extendsDecl(dw', e2, dw'') \wedge extendsTheory(tw', e2, tw'') \wedge$
$\quad \forall t, t' \in State_w, vw \in Value_w, : <t, e2> \longrightarrow <t', vw>$
$\quad\quad \Rightarrow$
$\quad\quad\quad \exists s, s' \in State_m : equals(s,t) \wedge [\![Cseq]\!](em)(s, s') \wedge$
$\quad\quad\quad \forall s, s' \in State_m, dm \in InfoData : equals(s,t) \wedge$
$\quad\quad\quad [\![Cseq]\!](em)(s, s') \wedge dm = infoData(s')$
$\quad\quad\quad\quad \Rightarrow equals(s', t') \wedge equals(dm, vw) \qquad$ (I)

From (I) and assumptions (2.4.3.9), (2.4.3.2) and (2.4.3.6), it follows that

$\forall t, t' \in State_w, vw \in Value_w, : <t, e2> \longrightarrow <t', vw>$
$\quad \Rightarrow$
$\quad\quad \exists s, s' \in State_m : equals(s,t) \wedge [\![Cseq]\!](em)(s, s') \wedge$
$\quad\quad \forall s, s' \in State_m, dm \in InfoData : equals(s,t) \wedge$
$\quad\quad [\![Cseq]\!](em)(s, s') \wedge dm = infoData(dm)$
$\quad\quad\quad \Rightarrow equals(s', t') \wedge equals(dm, vw) \qquad$ (I.1)

We instantiate above formula (I.1) with
t as t'', t' as t''', vw as $void$ to get

$<t'', e2> \longrightarrow <t''', void>$
$\Rightarrow$
$\quad \exists s, s' \in State_m : equals(s,t) \wedge [\![Cseq]\!](em)(s, s') \wedge$
$\quad \forall s, s' \in State_m, dm \in InfoData : equals(s,t) \wedge$
$\quad [\![Cseq]\!](em)(s, s') \wedge dm = infoData(s')$
$\quad\quad \Rightarrow equals(s', t''') \wedge equals(dm, void) \qquad$ (I.2)

From (I.2) with assumption (2.4.3.27), we get

$\exists s, s' \in State : equals(s,t) \wedge [\![Cseq]\!](em)(s,s')$ (I.3)

Taking s as s", s' as s"' in the above formula, we know from (9.b), (9.c), (1.a') and (3.b) that

Taking s as s'', s' as s''' with (I.3), we know from assumptions (2.4.3.31), (2.4.3.32), (2.4.3.8) and (2.4.3.6) that

there is s'', s''', em' and $Cseq$ for which

$$[Cseq](em')(s'', s''') \tag{2.4.3.36}$$

By expanding assumption (2.4.3.29), we get

$[\exists s, s' \in State : equals(s, t''') \wedge [\![\textbf{while } E \textbf{ do } Cseq \textbf{ end}]\!](em)(s,s')]$ (J.1)

$[\forall s, s' \in State, dm \in InfoData :$
$equals(s',t') \wedge [\![\textbf{while } E \textbf{ do } Cseq \textbf{ end}]\!](em)(s,s') \wedge dm = infoData(s')$
$\Rightarrow equals(s',t') \wedge equals(dm, void)]$ (J.2)

From (J.1), we know there is s, s' for which

$$equals(s, t''') \tag{2.4.3.37}$$

$$[\textbf{while}E\textbf{do}Cseq\textbf{end}](em)(s, s') \tag{2.4.3.38}$$

We instantiate lemma $(L - cseq5)$ with

s as s, t as t''' to get

$s = constructs(t''') \Leftrightarrow equals(s, t''')$ (K)

From (K) and assumption (2.4.3.29), we get

$$s = constructs(t''') \tag{2.4.3.39}$$

From assumptions (2.4.3.29) and (2.4.3.31), we can rewrite (2.4.3.37) and (2.4.3.38) as

$$equals(s''', t''') \tag{2.4.3.40}$$

$$[\textbf{while}E\textbf{do}Cseq\textbf{end}](em)(s''', s') \tag{2.4.3.41}$$

We instantiate lemma $(L - c13)$ with

em as em, em' as em', E as E, $Cseq$ as $Cseq$, s as s, s' as s', s'' as s'', s''' as s''' to get

$[\![E]\!](em)(s, s'', inValue(True)) \wedge em' = Env(em, E) \wedge [\![Cseq]\!](em')(s'', s''')$
$[\![\textbf{while } E \textbf{ do } Cseq \textbf{ end}]\!](em)(s''', s')$
$\Rightarrow [\![\textbf{while } E \textbf{ do } Cseq \textbf{ end}]\!](em)(s, s')$ (L)

The goal (e.b.1.2) follows from (L) with assumptions (2.4.3.35), (2.4.3.8), (2.4.3.36) and (2.4.3.41). Consequently (e.b.1) follows from the proofs of (e.b.1.1) and (e.b.1.2).

Sub-Goal (e.b.2)

Let s, s', dm, t be arbitrary but fixed.
We assume:

$$equals(s, t) \tag{2.4.3.42}$$

$$[\mathbf{while} E \mathbf{do} Cseq \mathbf{end}](em)(s, s') \tag{2.4.3.43}$$

$$dm = infoData(s') \tag{2.4.3.44}$$

We show:

$equals(s', t')$ (e.b.2.1)

$equals(dm, vw)$ (e.b.2.1)

We define:

$$s' := constructs(t') \tag{2.4.3.45}$$

$$vw := constructs(dm) \tag{2.4.3.46}$$

In the following, we prove the sub-goals (e.b.2.1) and (e.b.2.2) in order to show the original goal (e.b.2).

Sub-Goal (e.b.2.1)

We instantiate lemma $(L - cseq5)$ with s as s' and t as t' to get

$s' = construct(t') \Rightarrow equals(s', t')$ (M)

The sub-goal (e.b.2.1) follows from (M) with assumption (2.4.3.45).

Sub-Goal (e.b.2.2)

We instantiate lemma $(L - cseq6)$ with v as vw, v' as dm to get

$vw = constructs(dm) \Rightarrow equals(dm, vw)$ (N)

This sub-goal (e.b.2.2) follows from (N) with assumption (2.4.3.46).
Consequently,

- the goal (e.b.2) follows from (e.b.2.1) and (e.b.2.2);
- the goal (e.b) follows from (e.b.1) and (e.b.2);
- the goal (e) follows from (e.a) and (e.b).

Finally, the soundness of the while-loop command follows from the proofs of goals (a), (b), (c), (d) and (e).

2.5 Lemmas

In Appendix E, we discuss the lemmas for the proof of the soundness statements of command sequence, command and expression. Also some auxiliary lemmas are defined. For the complete definition of the lemmas, please see the corresponding sections of the Appendix E. The lemmas say the absence of internal inconsistencies and are essentially about the well-typing, consistency of environments and the extensions of the corresponding intermediate theory and module declarations.

2.6 Definitions

Appendix F includes various definitions required for the proof, e.g. definitions of the translation functions.

2.7 Why3 Semantics

Appendix G defines the corresponding big-step operational semantics of Why3ML as introduced in [1].

2.8 Derivations

In Appendix H, we give the derivation of the rules for the while-loop command. As mentioned earlier that the semantics of a Why3 while-loop is defined by a complex exception handling mechanism. Therefore, the goal here was to introduce two new rules for the while-loop (i.e. (d.a) and (d.b)), which operate directly on the level of while-loop (without expansion). We also showed that these rules follows from the basic rule calculus, i.e. adding these rules does not change the loop semantics.

3 Conclusions and Future Work

In this paper we have sketched the structure and strategy for the soundness statements of the selected constructs of *MiniMaple*, e.g. command sequences, conditional commands, assignment commands and while-loops. The proof was essentially based on structural induction along-with various auxiliary lemmas. However, the proof for the soundness of while-loop required some additional derivations and was proved by rule induction. A proof for some selected cases of expressions is planned as a future goal.

Acknowledgment

The author cordially thanks Wolfgang Schreiner for his valuable and constructive comments and suggestions throughout this work.

4 References

[1] Filliâtre, Jean-Christophe. Why: an Intermediate Language for Program Verification. TYPES Summer School 2007 – http://typessummerschool07.cs.unibo.it/, 2007.

[2] Muhammad Taimoor Khan. Formal Semantics of a Specification Language for *MiniMaple*. DK Technical Report 2012-06, Research Institute for Symbolic Computation, University of Linz, April 2012.

[3] Muhammad Taimoor Khan. Formal Semantics of *MiniMaple*. DK Technical Report 2012-01, Research Institute for Symbolic Computation, University of Linz, January 2012.

[4] Muhammad Taimoor Khan. On the Formal Semantics of *MiniMaple* and its Specification Language. In *Proceedings of Frontiers of Information Technology*, pages 169–174. IEEE Computer Society, 2012.

[5] Winskel, Glynn. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.

Appendices

A Semantic Algebras

A.1 For *MiniMaple*

All the syntactic and semantic domains of *MiniMaple* are included. Here we give the definitions of those domains, which are used.

A.1.1 Truth Values

Domain Boolean = {True, False}

A.1.2 Numeral Values

Domain Nat' = N\{0}, Nat = N, Integer = Z, Float = R

A.1.3 Environment Values

Domains
Environment = Context x Space
Context = Identifier → EnvValue
EnvValue = Value + Type-Tag
Space = $\mathbb{P}$(Variable)
Variable := n, n ∈ N // represents location

A.1.4 State Values

Domains
State = Store x Data
Store = Variable → Value
Data = Flag x Exception x Return
Flag = {execute, exception, return, leave}
Exception = Identifier x ValueU
Return = ValueU

Operations
state : Store x Data → State
state(s,d) = <s,d>

exception : Identifier x ValueU → Exception
exception(i,v) = <i,v>

ide : Exception → Identifier
ide(i,v) → i

valuee : Exception → ValueU
valuee(i,v) → v

data : State → Data
data(s,d) = d

store : State → Store
store(s,d) → s

flag : Data → Flag
flag(f,e,r) = f

exception : Data → Exception
exception(f,e,r) = e

return : Data → Return
return(f,e,r) = r

data : Flag x Exception x Return → Data
data(f,e,r) = <f,e,r>

execute : State → State
execute(s) = LET d = data(s) IN state(store(s), data(execute, exception(d), return(d))

exception : State x String x ValueU → State
exception(s,st,v) = LET d = data(s) IN state(store(s), data(exception, (st,v), return(d))

return : State x ValueU → State
return(s,v) = LET d = data(s) IN state(store(s), data(return, exception(d), v))

executes ⊂ Data
executes(d) ⇔ flag(d) = execute

exceptions ⊂ Data
exceptions(d) ⇔ flag(d) = exception

returns ⊂ Data
returns(d) ⇔ flag(d) = return

A.1.5 Semantic Values

Domain

Value = Procedure + Module + List + Set + Tuple + Boolean + Integer + String + Rational + Float
+ Symbol

A.1.6 Information Values

Domain
InfoData = Value + Data + Void

A.1.7 List Values

Domain List = Value*

A.1.8 Unordered Values

Domain Set = List

A.1.9 Tuple Values

Domain Tuple= List

A.1.10 Procedure Values

Domain Procedure = $\mathbb{P}$(Value* x State x StateU x ValueU)

A.1.11 Lifted Value domain

Domains ValueU = Value + Undefined, Undefined = Unit, StateU = State + Error, Error = Unit

A.2 For Why3

All the syntactic domains of Why3 are included. Here we give the definitions of those semantic domains, which are used. The syntactic domains of Why3 are also suffixed with "w".

A.2.1 Variable Values

Domains
Variable := n, n $\in$ N // represents location

A.2.2 State Values

Domains
Statew = Variable $\rightarrow$ Valuew

A.2.3 Environment Values

Domains
Environmentw // is a mapping from identifiers to type and represents Why3 type environment.

A.2.4 Semantic Values

Domain
Valuew = c + Exceptionw + Functionw + Void

A.2.5 Exception Values

Domain Exceptionw = Identifier x c

A.2.6 Function Values

Domain Functionw = rec f x = c

A.2.7 Constant Values

Domain c = Integerw + Booleanw + Listw + Setw + Tuplew + ...

// this domain hides all other corresponding Why3 domains of values. All suffixed “w” domains
// represent the corresponding built-in domains.

A.2.8 Declaration Values

Domain Declw

A.2.9 Theory Values

Domain Theoryw

A.2.10 Why3 Types

Domain Typew = int + real + tuple + list(Typew) + set(Typew) + . . .
// also includes other built in and extended (abstract) types of Why3

B Auxiliary Functions and Predicates

equals $\subset$ **State** $\times$ **Statew**

This predicate is true, if all the pairs of identifier and value in the former state have a pair of the same identifier and a corresponding value in the latter state.

equals(s, t) $\Leftrightarrow \forall$ i: Identifier: i $\in$ dom(s) $\rightarrow\ \exists$ vm $\in$ Value, vw $\in$ Valuew: (i,vm) $\in$ s $\wedge$ (i,vw) $\in$ t $\wedge$ equals(vm, vw)

equals $\subset$ **Value** $\times$ **Valuew**

This predicate returns true, if the former value is a semantic equivalent to the latter value.

```
equals(vm, vw) ⇔
  cases vm of
   []isInteger(intm) →   cases vw of
       isIntegerw(intw) → valueOf(intm) = valueOf(invw)
       [] _ → false
       end
   []isBoolean(bm) →     cases vw of
       isBooleanw(bw) →  valueOf(bm) = valueOf(bw)
       [] _ → false
       end
   [] ... →  ...
  end
```

equals $\subset$ **InfoData** $\times$ **Valuew**

This predicate returns true, if the corresponding element of the InfoData is semantically equivalent to the given value.

```
equals(d, vw) ⇔
  cases d of
   []isValue(vm) → equals(vm, vw)
   []isData(dm) →   IF exceptions(dm) THEN
       cases vw of
       isExceptionw(ew) →
         equals(getId(dm), getId(ew)) ∧
         equals(getValue(dm), getValue(ew))
       [] _→ false
       end
       ELSE
       ...
       END
   []isVoid(mv) →   cases vw of
       isVoid(wv) → true
       [] _ → false
       end
  end
```

wellTyped $\subset$ **Environment** $\times$ **Syntactic_Domain_of**_*MiniMaple*_

The predicate returns true, if all the identifiers appearing in the given syntactic domain has a corresponding value in the given environment.

```
wellTyped(em, D) ⇔
  cases D of
    inCommand_Sequence(Cseq) →
      cases Cseq of
        isCseqC(C) → wellTyped(em, C)
        isCseqCseq(C;Cseq) → wellTyped(em, C) ∧
            LET em' = Env(e, C) IN wellTyped(em', Cseq)
      end
    inCommand(C) →
      cases C of
        isCCond(if E then Cseq1 else Cseq2) →
          wellTyped(em, E) ∧
          LET em' = Env(em, E) IN
            wellTyped(em', Cseq1)
            ∧ LET em'' = Env(em', Cseq1) IN
            wellTyped(em'', Cseq2)
        [] ... →
      end
    inExpression(E) →
      cases E of
        isEIdentifier(I) → isDefined(I, em)
        isEBoolean(B) → ∀ I : Identifier: I ∈ extractIdentifiers(B)
              → isDefined(I, em)
      end
    inExpression_Sequence(Eseq) → ...
  end
```

wellTyped ⊂ Expressionw × Environmentw × Declw × Theoryw

The predicate returns true, if all the identifiers appearing in the given syntactic Why3 expression has a corresponding value in the given environment or declaration or theory.

wellTyped(cw, ew, dw, tw) ⇔ consistent(ew, dw, tw) ∧ wellFormed(cw) ∧
∀ i: Identifier: i ∈ extractIdentifiers(cw) → isDefined(i, ew, dw, tw)

isDefined ⊂ Identifier × Environmentw × Declw × Theoryw

The predicate returns true only if identifier has a corresponding definition in any of the given Why3 environment, declarations or theory.

isDefined ⊂ Identifier × Environment

The predicate returns true only if identifier has a corresponding value in the given environment.

consistent ⊂ Environment × Environmentw × Declw × Theoryw

This predicate returns true, if the given *MiniMaple* environment is consistent with the definitions as provided in the given Why3 environment, declaration and theory.

consistent ⊂ Environmentw × Declw × Theoryw

This predicate returns true, if the given Why3 environment has the definitions accessible in the given Why3 declaration and theory.

wellFormed ⊂ **Expressionw**
This predicate returns true, if the given Why3 expression is syntactically correct.

infoData : **State** → **InfoData**
The function returns the information data or value extracted from the given command and state. This depends on the syntax of the given command and the control data of the given state.
infoData(s) = inInfoData(data(s)) , if exception(data(s)) is true
inInfoData(Void) , if exception(data(s))
. . .

extractIdentifiers : **Syntactic_Domain_of** *MiniMaple* → **Identifier_Sequence**
The function extracts the identifiers appearing in the given *MiniMaple* syntactic domain.

extractIdentifiers : **Expressionw** → **Identifier_Sequence**
The function extracts the identifiers appearing in the given Why3 expression.

extractDeclarations : **Expressionw** → **Declw**
The function extracts the module declaration sequence appearing in the given Why3 expression.

extractTheoryDeclarations : **Expressionw** → **Theoryw**
The function extracts the theory declarations appearing in the given Why3 expression.

combines : **Declw** × **Declw** → **Declw**
This function combines the given declaration and declaration sequence, it removes the duplicate declarations.

combines : **Theoryw** × **Theoryw** → **Theoryw**
This function combines the given theory and theory declaration sequence, it removes the duplicate theory declarations.

extendsEnv ⊂ **Environmentw** × **Expressionw** × **Environmentw**
This predicate returns true, if the former Why3 environment extends the latter.
extendsEnv(e1, c, e2) ⇔
∀ I: Identifier, v ∈ Value, Iseq ∈ Identifier_Sequence, vseq ∈ Value_Sequence:
[(I,v) ∈ e2 ⇒ (I,v) ∈ e1]
∧ [Iseq = extractIdentifiers(c) ∧ vseq getValues(Iseq,c) ⇒ e1 = e2 U IVSeqtoSet(Iseq, vseq)]

extendsDecl ⊂ **Environmentw** × **Expressionw** × **Environmentw**
This predicate returns true, if the former sequence of Why3 declaration extends the latter.

extendsDecl(d1, c, d2) ⇔
∀ d, dseq ∈ Declw: [d ∈ decltoSet(d2) ⇒ d ∈ decltoSet(e1)]
∧ [dseq = extractDeclarations(c) ⇒ length(d2) + length(dseq) = length(d1) ∧ d1 = combine(d2,dseq)]

extendsTheory ⊂ **Environmentw** × **Environmentw**
This predicate returns true, if the former sequence of Why3 theory extends the latter.
extendsTheory(t1, c, t2) ⇔
∀ t, tseq ∈ Theoryw: [t ∈ theorytoSet(t2) ⇒ t ∈ theorytoSet(t1)]
∧ [tseq = extractTheoryDeclarations(c) ⇒ length(t2) + length(tseq) = length(t1) ∧ t1 = combine(t2, tseq)]

extendsEnv ⊂ **Environmentw** × **Expressionw** × **Environmentw**
This predicate returns true, if the latter Why3 environment extends the former environment with the identifiers appearing in the given Why3 expression.
extendsEnv(e1, c, e2) ⇔ LET iseq = extractIdentifiers(c), vseq = getValues(iseq, c) IN
e1 U IVSeqtoSet(iseq, vseq) = e2

extendsDecl ⊂ **Declw** × **Expressionw** × **Declw**
This predicate returns true, if the latter Why3 declaration extends the former declaration with the declarations appearing in the given Why3 expression.
extendsDecl(d1, c, d2) ⇔ LET dseq = extractDeclarations(c) IN
combine(d1, dseq) = d2

extendsTheory ⊂ **Theoryw** × **Expressionw** × **Theoryw**
This function returns a Why3 theory declaration sequence, which extends the given theory declaration sequence with the theory declarations appearing in the given Why3 expression.
extendsTheory(t1, c, t2) ⇔ LET tseq = extractTheoryDeclarations(c) IN
combine(t1, tseq) = t2

getId : **Exceptionw** → **Identifier**
This function returns the identifier of the given Why3 exception.
getId(ew) = LET ew = (id, val) IN id

getId : **Date** → **Identifier**
This function returns the identifier of the exception in the given Data.
getId(d) = LET id = ide(exception(d)) IN id

getId : **Exceptionw** → **Valuew**
This function returns the value of the given Why3 exception.
getId(ew) = LET ew = (id, val) IN val

getId : **Data** → **Value**
This function returns the value of the exception in the given Data.
getId(d) = LET val = valuee(exception(d)) IN val

ValueOf : **Valuew** → **Valuew**

This function returns the value of the Why3 semantic domain of value.

ValueOf : **Value** $\rightarrow$ **Value**
This function returns the value of the *MiniMaple* semantic domain of value.

$\longrightarrow \subset$ (**Statew** $\times$ **Expressionw**) $\times$ (**Statew** $\times$ **Valuew**)
This predicate holds for the big step semantics of Why3. The $<t, c> \longrightarrow <t', vw>$ is a syntactic sugar for this predicate.

IdSeqtoSet : **Identifier_Sequence** $\rightarrow$ **Set**
This function coverts a given identifier sequence to a set.

Env : **Environment** $\times$ **Syntactic_Domain_of** _*MiniMaple* $\rightarrow$ **Environment**
This function, constructs an extends the given environment for the given syntactic *MiniMaple* domain.

constructs : **Statew** $\rightarrow$ **State**
This function constructs a corresponding *MiniMaple* state for a given Why3 state.

C Soundness Statements

Let's define the soundness statements for the translation of a *MiniMaple* command sequence (Cseq), command (C) and an expression (E) by the corresponding predicates as follows.

C.1 For Command Sequence

Soundness_cseq ⊂ Command_Sequence
Soundness_cseq(Cseq) ⇔
∀ em ∈ Environment, cw ∈ Expressionw, ew, ew' ∈ Environmentw, dw, dw' ∈ Declw, tw, tw' ∈ Theoryw:

wellTyped(em, Cseq) ∧ consistent(em, ew, dw, tw) ∧
<cw, ew', dw', tw'> = T⟦Cseq⟧(em, ew, dw, tw)
⇒ [wellTyped(cw, ew', dw', tw') ∧ extendsEnv(ew', cw, ew) ∧ extendsDecl(dw', cw, dw)
∧ extendsTheory(tw', cw, tw) ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, cw> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq⟧(em)(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq⟧(em)(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]
]

C.2 For Command

Soundness_c ⊂ Command
Soundness_c(C) ⇔
∀ em ∈ Environment, cw ∈ Expressionw, ew, ew' ∈ Environmentw, dw, dw' ∈ Declw, tw, tw' ∈ Theoryw:

wellTyped(em, C) ∧ consistent(em, ew, dw, tw) ∧
<cw, ew', dw', tw'> = T⟦C⟧(em, ew, dw, tw)
⇒ [wellTyped(cw, ew', dw', tw') ∧ extendsEnv(ew', cw, ew) ∧ extendsDecl(dw', cw, dw)
∧ extendsTheory(tw', cw, tw) ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, cw> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦C⟧(em)(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦C⟧(em)(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]

]

C.3 For Expression

Soundness_e ⊂ Expression

Soundness_e(E) ⇔

∀ em ∈ Environment, expw, ∈ Exprw, ew, ew' ∈ Environmentw, dw, dw' ∈ Declw, tw, tw' ∈ Theoryw:

wellTyped(em, E) ∧ consistent(em, ew, dw, tw) ∧
<expw, ew', dw', tw'> = T⟦E⟧(em, ew, dw, tw)
⇒ [wellTyped(expw, ew', dw', tw') ∧ extendsEnv(ew', expw, ew) ∧ extendsDecl(dw', expw, dw)
∧ extendsTheory(tw', expw, tw) ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, expw> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, vw)
]
]
]

C.4 For Identifier

Soundness_e ⊂ Identifier

Soundness_e(I) ⇔

∀ em ∈ Environment, i ∈ Constantw, ew, ew' ∈ Environmentw, dw, dw' ∈ Declw, tw ∈ Theoryw:

wellTyped(em, I) ∧ consistent(em, ew, dw, tw) ∧
<i, ew', dw', tw> = T⟦I⟧(em, ew, dw, tw)
⇒ [wellTyped(i, ew', dw', tw') ∧ extendsEnv(ew', i, ew) ∧ extendsDecl(dw', i, dw) ∧
[∀ t ∈ Statew: <t, i> ⟶ <t, i>
⇒ [∃ v ∈ Variable: ⟦I⟧(em)(v)]
∧
[∀ v ∈ Variable: ⟦I⟧(em)(v) ⇒ equals(v, I)]
]
]

C.5 Goal

We need to prove the following goal:

∀ Cseq ∈ Command, C ∈ Command, E ∈ Expression, I ∈ Identifier:
Soundness_cseq(Cseq) ∧ Soundness_c(C) ∧ Soundness_e(E)

D Proof

In the following we give definition of some constructs of related syntactic domains of *MiniMaple*.

Cseq := C | C;Cseq   // originally was EMPTY | C;Cseq
C := ... | **if** E **then** Cseq **else** Cseq **end if** | **while** E **do** Cseq **end do** | ...
E := ... | E **and** E | E **or** E | E = E | E < E | E <= E | E > E | E >= E | **not** E | ...
Eseq := E | E;Eseq   // originally was EMPTY | E;Eseq
I := is a MiniMaple identifier

We have modified the syntactic domain of command sequence, because no corresponding Why3 semantics is defined for skip command, which is a corresponding translation of an empty command sequence.

Our goal is formulated as follows:

Goal:

$\forall$ Cseq $\in$ Command_Sequence, C $\in$ Command, E $\in$ Expression:
Soundness_cseq(Cseq) $\wedge$ Soundness_c(C) $\wedge$ Soundness_e(E) ------------------- (G)

Proof:

We prove the goal by structural induction on Cseq, C and E whose formal grammar rules are defined. Also the rules for the questioned semantics are defined in Why3 by " $\_ \longrightarrow \_$" notation.

By splitting G, we have following three sub-goals:

Soundness_cseq(Cseq) ----------------- (G1)
Soundness_c(C) ----------------- (G2)
Soundness_e(E) ----------------- (G3)

In the following, we prove the above sub-goals respectively.

D.1 Case G1: Soundness of Command Sequence

Case 1: Cseq := C

From induction assumption, we know that
Soundness_c(C) ----------------- (1)

Also from the definitions of semantics, we know that the semantics of C and Cseq are equivalent, s.t.

⟦Cseq⟧(e)(s,s') ˜ ⟦C⟧(e)(s,s') ------------------ (2)

and also the corresponding translation functions are equal, s.t.

T⟦Cseq⟧(em)(ew, dw, tw) ˜ T⟦C⟧(em)(ew, dw, tw) -------------- (3)
The goal (G1) follows from (1), (2) and (3). Hence proved.

Case 2: Cseq := C;Cseq

Let em, cw, em, ew', dw, dw', tw, tw', be arbitrary but fixed.

We assume:

wellTyped(em,C;Cseq) -------------- (1)
consistent(em, ew, dw, tw) -------------- (2)
(cw, ew', dw', tw') = T⟦C;Cseq⟧(em, ew, dw, tw) -------------- (3)

We show:

wellTyped(cw, ew', dw', tw') -------------- (a)
extendsEnv(ew', cw, ew) -------------- (b)
extendsDecl(dw', cw, dw) -------------- (c)
extendsTheory(tw', cw, tw) -------------- (d)
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, cw> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦C;Cseq⟧(em)(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦C;Cseq⟧(em)(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
] -------------- (e)

Sub-Goal (a)

We instantiate lemma (L-cseq1) with
cseq as C;Cseq, em as em, e as cw, ew as ew, ew' as ew', dw as dw, dw' as dw', tw as tw, tw' as tw'
and get

wellTyped(em, C;Cseq) ∧ (cw, ew', dw', tw') = T⟦C;Cseq⟧(em, ew, dw, tw)
⇒ wellTyped(cw, ew', dw', tw')

This goal follows from assumptions (1) and (3).

Sub-Goal (b)

By definition of translation function (D2) of T⟦C;Cseq⟧, there are e1, e2, ew'', dw'', tw'' for which

(cw, ew', dw', tw') = T⟦C;Cseq⟧(em, ew, dw, tw) ------------- (3)

where
cw = e1;e2 ------------------ (3.a)
(e1, ew'', dw'', tw'') = T⟦C⟧(em, ew, dw, tw) ------------------- (3.b)
em' = Env(em, C) ------------------- (3.b')
(e2, ew', dw', tw') = T⟦Cseq⟧(em', ew'', dw'', tw'') ------------------- (3.c)

and e1;e2 is a syntactic sugar for let _ = e1 in e2

We instantiate lemma (L-cseq3) with
em as em, em' as em', C as C and Cseq as Cseq

from which following holds

wellTyped(em, C) --------------------- (1.a)
em' = Env(em, C) --------------------- (1.b)
wellTyped(em', Cseq) -------------------- (1.c)

We instantiate the soundness statement for C with
em as em, cw as e1, ew as ew, ew' as ew'', dw as dw, dw' as dw'', tw as tw, tw' as tw''

to get

wellTyped(em, C) ∧ consistent(em, ew, dw, tw) ∧
<e1, ew'', dw'', tw''> = T⟦C⟧(em, ew, dw, tw)
⇒ [wellTyped(e1, ew'', dw'', tw'') ∧ extendsEnv(ew'', e1, ew) ∧ extendsDecl(dw'', e1, dw)
∧ extendsTheory(tw'', e1, tw) ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦C⟧(em)(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦C⟧(em)(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]
]

From assumptions (1.a), (2) and (3.b), it follows that

extendsEnv(ew'', e1, ew) ----------------------- (3.d)

We instantiate lemma (L-cseq4) with
em as em, em', C as C, Cseq as Cseq, ew as ew, ew' as ew', e1 as e1, e2 as e2, dw as dw, dw' as dw', tw as tw, tw' as tw', ew'' as ew'', dw'' as dw'', tw'' as tw''

to get

<e1, ew'', dw'', tw''> = T⟦C⟧(em, ew, dw, tw) ∧ em' = Env(em, C) ∧

<e2, ew', dw', tw'> = T⟦Cseq⟧(em', ew'', dw'', tw'') ∧ consistent(em, ew, dw, tw)
⇒ consistent(em', dw'', dw'', tw'')

From assumptions (3.b), (3.b'), (3.c) and (2), it follows that

consistent(em', ew'', dw'', tw'') -------------------------- (3.e)

We instantiate the induction assumption for Cseq with
em as em', cw as e2, ew as ew'', ew' as ew', dw as dw'', dw' as dw', tw as tw'', tw' as tw'

to get

wellTyped(em', Cseq) ∧ consistent(em', ew'', dw'', tw'') ∧
<e2, ew', dw', tw'> = T⟦Cseq⟧(em', ew'', dw'', tw'')
⇒ [wellTyped(e2, ew', dw', tw') ∧ extendsEnv(ew', e2, ew'') ∧ extendsDecl(dw', e2, dw'')
∧ extendsTheory(tw', e2, tw'') ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq⟧(em')(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq⟧(em')(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]
]

From assumptions (1.c), (3.e) and (3.c), it follows that

extendsEnv(ew', e2, ew'') ----------------------- (3.f)

From (3.a), we can re-write the goal (b) as

extendsEnv(ew', e1;e2, ew) -------------------- (b)

We instantiate lemma (L-cseq2) with
em as em, C as C, Cseq as Cseq, ew as ew, ew' as ew', ew'' as ew'', e1 as e1, e2 as e2, dw as dw, dw' as dw', dw'' as dw'', tw as tw, tw' as tw', tw'' as tw''

to get

wellTyped(em, C;Cseq) ∧ <e1;e2, ew', dw', tw'> = T⟦C;Cseq⟧(em, ew, dw, tw)
⇒
[extendsEnv(ew'', e1, ew) ∧ extendsEnv(ew', e2, ew'') ⇒ extendsEnv(ew', e1;e2, ew)] ∧
[extendsDecl(dw'', e1, dw) ∧ extendsDecl(dw', e2, dw'') ⇒ extendsDecl(dw', e1;e2, dw)] ∧

[extendsTheory(tw'', e1, tw) ∧ extendsTheory(tw', e2, tw'') ⇒ extendsTheory(tw', e1;e2, tw)]

The goal (b) follows from assumptions (1), (3), (3.a), (3.d) and (3.f). Hence proved.

Sub-Goal (c)

We instantiate the soundness statement for C with
em as em, cw as e1, ew as ew, ew' as ew'', dw as dw, dw' as dw'', tw as tw, tw' as tw''

to get

wellTyped(em, C) ∧ consistent(em, ew, dw, tw) ∧
<e1, ew'', dw'', tw''> = T⟦C⟧(em, ew, dw, tw)
⇒ [wellTyped(e1, ew'', dw'', tw'') ∧ extendsEnv(ew'', e1, ew) ∧ extendsDecl(dw'', e1, dw)
∧ extendsTheory(tw'', e1, tw) ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦C⟧(em)(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦C⟧(em)(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]
]

From assumptions (1.a), (2) and (3.b), it follows that

extendsDecl(dw'', e1, dw) ----------------------- (3.g)

We instantiate the induction assumption for Cseq with
em as em', cw as e2, ew as ew'', ew' as ew', dw as dw'', dw' as dw', tw as tw'', tw' as tw'

to get

wellTyped(em', Cseq) ∧ consistent(em', ew'', dw'', tw'') ∧
<e2, ew', dw', tw'> = T⟦Cseq⟧(em', ew'', dw'', tw'')
⇒ [wellTyped(e2, ew', dw', tw') ∧ extendsEnv(ew', e2, ew'') ∧ extendsDecl(dw', e2, dw'')
∧ extendsTheory(tw', e2, tw'') ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq⟧(em')(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)

∧ ⟦Cseq⟧(em')(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]
]

From assumptions (1.c), (3.e) and (3.c), it follows that

extendsDecl(dw', e2, dw'') ----------------------- (3.h)

From (3.a), we can re-write the goal (c) as

extendsDecl(dw', e1;e2, dw) -------------------- (b)

We instantiate lemma (L-cseq2) with
em as em, C as C, Cseq as Cseq, ew as ew, ew' as ew', ew'' as ew'', e1 as e1, e2 as e2, dw as dw, dw' as dw', dw'' as dw'', tw as tw, tw' as tw', tw'' as tw''

to get

wellTyped(em, C;Cseq) ∧ <e1;e2, ew', dw', tw'> = T⟦C;Cseq⟧(em, ew, dw, tw)
⇒
[extendsEnv(ew'', e1, ew) ∧ extendsEnv(ew', e2, ew'') ⇒ extendsEnv(ew', e1;e2, ew)] ∧
[extendsDecl(dw'', e1, dw) ∧ extendsDecl(dw', e2, dw'') ⇒ extendsDecl(dw', e1;e2, dw)] ∧
[extendsTheory(tw'', e1, tw) ∧ extendsTheory(tw', e2, tw'') ⇒ extendsTheory(tw', e1;e2, tw)]

The goal (c) follows from assumptions (1), (3), (3.a), (3.g) and (3.h). Hence proved.

Sub-Goal (d)

We instantiate the soundness statement for C with
em as em, cw as e1, ew as ew, ew' as ew'', dw as dw, dw' as dw'', tw as tw, tw' as tw''

to get

wellTyped(em, C) ∧ consistent(em, ew, dw, tw) ∧
<e1, ew'', dw'', tw''> = T⟦C⟧(em, ew, dw, tw)
⇒ [wellTyped(e1, ew'', dw'', tw'') ∧ extendsEnv(ew'', e1, ew) ∧ extendsDecl(dw'', e1, dw)
∧ extendsTheory(tw'', e1, tw) ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦C⟧(em)(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)

∧ ⟦C⟧(em)(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]
]

From assumptions (1.a), (2) and (3.b), it follows that

extendsTheory(tw'', e1, tw) ----------------------- (3.i)

We instantiate the induction assumption for Cseq with
em as em', cw as e2, ew as ew'', ew' as ew', dw as dw'', dw' as dw', tw as tw'', tw' as tw'

to get

wellTyped(em', Cseq) ∧ consistent(em', ew'', dw'', tw'') ∧
<e2, ew', dw', tw'> = T⟦Cseq⟧(em', ew'', dw'', tw'')
⇒ [wellTyped(e2, ew', dw', tw') ∧ extendsEnv(ew', e2, ew'') ∧ extendsDecl(dw', e2, dw'')
∧ extendsTheory(tw', e2, tw'') ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq⟧(em')(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq⟧(em')(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]
]

From assumptions (1.c), (3.e) and (3.c), it follows that

extendsTheory(tw', e2, tw'') ----------------------- (3.j)

From (3.a), we can re-write the goal (d) as

extendsTheory(tw', e1;e2, tw) -------------------- (b)

We instantiate lemma (L-cseq2) with
em as em, C as C, Cseq as Cseq, ew as ew, ew' as ew', ew'' as ew'', e1 as e1, e2 as e2, dw as dw, dw' as dw', dw'' as dw'', tw as tw, tw' as tw', tw'' as tw''

to get

wellTyped(em, C;Cseq) ∧ <e1;e2, ew', dw', tw'> = T⟦C;Cseq⟧(em, ew, dw, tw)
⇒
[extendsEnv(ew'', e1, ew) ∧ extendsEnv(ew', e2, ew'') ⇒ extendsEnv(ew', e1;e2, ew)] ∧

[extendsDecl(dw'', e1, dw) ∧ extendsDecl(dw', e2, dw'') ⇒ extendsDecl(dw', e1;e2, dw)] ∧
[extendsTheory(tw'', e1, tw) ∧ extendsTheory(tw', e2, tw'') ⇒ extendsTheory(tw', e1;e2, tw)]

The goal (c) follows from assumptions (1), (3), (3.a), (3.i) and (3.j). Hence proved.

Sub-Goal (e)

Let t, t', cw, vw be arbitrary but fixed.
We assume:

<t, cw> ⟶ <t', vw> ------------------ (4)

From (3.a), we know

cw = e1;e2 which is a syntactic sugar for let _ = e1 in e2.

From (com-s), we get

<t, let _ = e1 in e2> ⟶ <t', vw> -------------------- (4')

<t, e1> ⟶ <t'', vw'> for some t'', where vw' in not exception ------------------ (5)

<t'', e2> ⟶ <t', vw> for some t'' ------------------ (6)

We show:

∃ s, s' ∈ State: equals(s, t) ∧ ⟦C;Cseq⟧(em)(s, s') ------------------ (e.a)

∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t) ∧ ⟦C;Cseq⟧(em)(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw) ------------------ (e.b)

Sub-Goal (e.a)

We define:

s := constructs(t) ----------------- (4.a)

We show:

equals(s, t) --------------------- (e.a.1)
⟦C;Cseq⟧(em)(s, s')] --------------------- (e.a.2)

Sub-Goal (e.a.1)

We instantiate lemma (L-cseq5) with s as s and t as t
to get

s = construct(t) ⇒ equals(s,t)

From (4.a) and lemma (L-cseq5) we know

equals(s,t)

which is the goal (e.a.1). Hence proved.

Sub-Goal (e.a.2)

We instantiate the soundness statement for C with
em as em, cw as e1, ew as ew, ew' as ew'', dw as dw, dw' as dw'', tw as tw, tw' as tw''

to get

wellTyped(em, C) ∧ consistent(em, ew, dw, tw) ∧
<e1, ew'', dw'', tw''> = T⟦C⟧(em, ew, dw, tw)
⇒ [wellTyped(e1, ew'', dw'', tw'') ∧ extendsEnv(ew'', e1, ew) ∧ extendsDecl(dw'', e1, dw)
∧ extendsTheory(tw'', e1, tw) ∧
[∀ t, t' ∈ Statew, vw' ∈ Valuew: <t, e1> ⟶ <t'', vw'>
⇒ [∃ s, s'' ∈ State: equals(s, t) ∧ ⟦C⟧(em)(s, s'')]
∧
[∀ s, s'' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦C⟧(em)(s, s'') ∧ dm = infoData(s'')
⇒ equals(s'', t'') ∧ equals(dm, vw)
]
]
]

From assumptions (1.a), (2), (3.b) and soundness statement of C, we know

[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦C⟧(em)(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦C⟧(em)(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]

We instantiate the above formula with
t as t and t' as t'', vw as vw' to get

[∀ t, t'' ∈ Statew, vw' ∈ Valuew: <t, e1> ⟶ <t'', vw'>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦C⟧(em)(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦C⟧(em)(s, s') ∧ dm = infoData(s')

⇒ equals(s', t') ∧ equals(dm, vw')
]
]

From assumption (1.d), we know

[∃ s, s' ∈ State: equals(s, t) ∧ ⟦C⟧(em)(s, s')]

By instantiating above with s as s, s' as s'', we know that

there is s, s'' s.t.

⟦C⟧(em)(s,s'') ---------------- (e.a.2.1)

We instantiate the induction assumption for Cseq with
em as em', cw as e2, ew as ew'', ew' as ew', dw as dw'', dw' as dw', tw as tw'', tw' as tw'

to get

wellTyped(em', Cseq) ∧ consistent(em', ew'', dw'', tw'') ∧
<e2, ew', dw', tw'> = T⟦Cseq⟧(em', ew'', dw'', tw'')
⇒ [wellTyped(e2, ew', dw', tw') ∧ extendsEnv(ew', e2, ew'') ∧ extendsDecl(dw', e2, dw'')
∧ extendsTheory(tw', e2, tw'') ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e2> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq⟧(em')(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq⟧(em')(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]
]

From assumptions (1.c), (3.e), (3.c) and induction assumption of Cseq, we know

[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e2> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq⟧(em')(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq⟧(em')(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]

We instantiate the above formula with t as t'', t' as t', vw as vw to get

[∀ t'', t' ∈ Statew, vw ∈ Valuew: <t'', e2> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq⟧(em')(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq⟧(em')(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]

From assumption (6) and above formula we get

[∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq⟧(em')(s, s')]

By instantiating the above formula with s as s'', s' as s', we know that

there is s'', s' s.t.

⟦Cseq⟧(em')(s'',s') ---------------- (e.a.2.2)

From (e.a.2.1) and (e.a.2.2) the definition of semantics of command sequence follows, which is the goal (e.b.2). Hence proved.

From goals (e.b.1) and (e.b.2), the goal (e.b) follows. Hence (e.b) is proved.

Sub-Goal (e.b)

Let s, s', dm be arbitrary but fixed.

We assume:

equals(s,t) --------------------- (7)
⟦C;Cseq⟧(em)(s,s') --------------------- (8)
dm = infoData(s') --------------------- (9)

We define:

s' := constructs(t') ------------------- (9.a)
vw := constructs(dm) ------------------- (9.b)

We show:

equals(s', t') --------------------- (e.b.1)
equals(dm, vw) --------------------- (e.b.2)

Sub-Sub-Goal (e.b.1)

We instantiate lemma (L-cseq5) with
s as s' and t as t'
to get

s' = constructs(t') ⇒ equals(s', t')

From (9.a) and (L-cseq5), we know

equals(s', t')
which is the goal (e.b.1). Hence proved.

Sub-Sub-Goal (e.b.2)
We instantiate lemma (L-cseq6) with
v as vw, v' as dm
to get

vw = constructs(dm) $\Rightarrow$ equals(dm, vw)
From (9.b) and (L-cseq6), we know

equals(dm, vw)

which is the goal (e.b.2). Hence proved.

Consequently, the goal (e.b) follows from (e.b.1) and (e.b.2). Hence (e.b) is proved.

Finally, the goal (e) follows from goals (e.a) and (e.b).

Also the goal (G1) follows from goals (a), (b), (c), (d) and (e).

Hence (G1) proved.

D.2 Case G2: Soundness of Command

We prove it by structural induction on C for some selected cases.

D.2.1 Case 1: C := if E then Cseq1 else Cseq2 end if

The goal (G2) can be re-stated as follows:

∀ em ∈ Environment, e1,e2,e3 ∈ Expressionw, ew, ew' ∈ Environmentw, dw, dw' ∈ Declw, tw, tw' ∈ Theoryw:

wellTyped(em, **if** E **then** Cseq1 **else** Cseq2 **end if**) ∧ consistent(em, ew, dw, tw) ∧
<if e1 then e2 else e3, ew', dw', tw') = T⟦if E **then** Cseq1 **else** Cseq2 **end if**⟧(em, ew, dw, tw)
⇒ [wellTyped(if e1 then e2 else e3, ew', dw', tw') ∧ extendsEnv(ew', if e1 then e2 else e3, ew)
∧ extendsDecl(dw', if e1 then e2 else e3 , dw)
∧ extendsTheory(tw', if e1 then e2 else e3 , tw) ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, if e1 then e2 else e3> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t)
∧ ⟦**if** E **then** Cseq1 **else** Cseq2 **end if**⟧(em)(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦**if** E **then** Cseq1 **else** Cseq2 **end if**⟧(em)(s, s')
∧ dm = infoData(C, s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]
] ------------------- (G21)

Let em, e1,e2,e3, ew, ew', dw, dw', tw, tw', dm and vw be arbitrary but fixed.

We assume:

wellTyped(em,**if** E **then** Cseq1 **else** Cseq2 **end if**) -------------- (1)
consistent(em, ew, dw, tw) -------------- (2)
<if e1 then e2 else e3, ew', dw', tw'> = T⟦**if** E **then** Cseq1 **else** Cseq2 **end if**⟧(em, ew, dw, tw) -------------- (3)

By expanding the definition of (3), we know
<e1, ew''', dw''', tw'''> = T⟦E⟧(em, ew, dw, tw) ----------------- (3.a)
em' = Env(em, E) ---------------- (3.a')
<e2, ew'', dw'', tw''> = T⟦Cseq1⟧(em', ew''', dw''', tw''') ---------------- (3.b)
<e3, ew', dw', tw'> = T⟦Cseq2⟧(em', ew'', dw'', tw'') ---------------- (3.c)

We show:

wellTyped(if e1 then e2 else e3, ew', dw', tw')   -------------- (a)
extendsEnv(ew', if e1 then e2 else e3, ew)   -------------- (b)
extendsDecl(dw', if e1 then e2 else e3 , dw)   -------------- (c)
extendsTheory(tw', if e1 then e2 else e3 , tw)   -------------- (d)
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, if e1 then e2 else e3> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦**if** E **then** Cseq1 **else** Cseq2 **end if**⟧(em)(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦**if** E **then** Cseq1 **else** Cseq2 **end if**⟧(em)(s, s')
∧ dm = infoData(**if** E **then** Cseq1 **else** Cseq2 **end if**, s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]   -------------- (e)

Sub-Goal (a)

We instantiate lemma (L-c1) with
c as if E then Cseq1 else Cseq2 end, em as em, e as if e1 then e2 else e3, ew as ew, ew' as ew', dw as dw, dw' as dw', tw as tw, tw' as tw'
and get

wellTyped(em, if E then Cseq1 else Cseq2 end)
∧ (if e1 then e2 else e3, ew', dw', tw') = T⟦if E then Cseq1 else Cseq2 end⟧(em, ew, dw, tw)
⇒ wellTyped(if e1 then e2 else e3, ew', dw', tw')

From assumptions (1), (3) and (L-c1), we know

wellTyped(if e1 then e2 else e3, ew', dw', tw')

which is the goal (a). Hence (a) proved.

Sub-Goal (b)

We instantiate lemma (L-c3) with
em as em, em' as em', E as E, Cseq1 as Cseq1, Cseq2 as Cseq2
to get

wellTyped(em, if E then Cseq1 else Cseq2 end) ⇒
wellTyped(em, E) ∧ em' = Env(em, E) ∧ wellTyped(em', Cseq1) ∧ wellTyped(em', Cseq2)

From (1) and (L-c3), we know

wellTyped(em, E)   --------------------- (1.a)
em' = Env(em, E)   --------------------- (1.a')
wellTyped(em', Cseq1)   -------------------- (1.b)
wellTyped(em', Cseq2)   -------------------- (1.c)

We instantiate lemma (L-c4) with
em as em, em’ as em’, E as E, Cseq1 as Cseq1, Cseq2 as Cseq2, ew as ew, ew’ ew’, ew” as ew”, ew”’ as ew”’, dw as dw, dw’ as dw’, dw” as dw”, dw”’ as dw”’, tw as tw, tw’ as tw’, tw” as tw”, tw”’ as tw”’
to get

<e1, ew”’, dw”’, tw”’> = T⟦E⟧(em, ew, dw, tw) ∧ em’ = Env(em, E) ∧
<e2, ew”, dw”, tw”> = T⟦Cseq1⟧(em’, ew”’, dw”’, tw”’) ∧
<e3, ew’, dw’, tw’> = T⟦Cseq2⟧(em’, ew”, dw”, tw”) ∧ consistent(em, ew, dw, tw)
⇒ consistent(em’, ew”’, dw”’, tw”’) ∧ consistent(em’, ew”, dw”, tw”)

From assumptions (3.a), (3.a’), (3.b), (3.c), (2) and (L-c4), we know

consistent(em’, ew”’, dw”’, tw”’) ---------------------- (2.a)
consistent(em’, ew”, dw”, tw”) ---------------------- (2.b)

We instantiate soundness statement of E with
em as em, expw as e1, ew as ew, ew’ as ew”’, dw as dw, dw’ as dw”’, tw as tw, tw’ as tw”’
and get

wellTyped(em, E) ∧ consistent(em, ew, dw, tw) ∧
<e1, ew”’, dw”’, tw”’> = T⟦E⟧(em, ew, dw, tw)
⇒ [wellTyped(e1, ew”’, dw”’, tw”’) ∧ extendsEnv(ew”’, e1, ew) ∧ extendsDecl(dw”’, e1, dw)
∧ extendsTheory(tw”’, e1, tw) ∧
[∀ t, t’ ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t’, vw>
⇒ [∃ s, s’ ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s’, vm)]
∧
[∀ s, s’ ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s’, vm)
⇒ equals(s’, t’) ∧ equals(vm, vw)
]
]
]

From assumptions (1.a), (2), (3.a) and the soundness statement of E, we know

extendsEnv(ew”’, e1, ew) ---------------------------- (b.1)

We instantiate the soundness statement of Cseq for Cseq1 with
em as em’ cw as e2, ew as ew”’, ew’ as ew”, dw as dw”’, dw’ as dw”, tw as tw”’, tw’ as tw”

to get

wellTyped(em’, Cseq1) ∧ consistent(em’, ew”’, dw”’, tw”’) ∧
<e2, ew”, dw”, tw”> = T⟦Cseq1⟧(em’, ew”’, dw”’, tw”’)

⇒ [wellTyped(e2, ew'', dw'', tw'') ∧ extendsEnv(ew'', e2, ew''') ∧ extendsDecl(dw'', e2, dw''')
∧ extendsTheory(tw'', e2, tw''') ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t', e2> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq1⟧(em')(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq1⟧(em')(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]
]

From assumptions (1.b), (2.a), (3.b) and soundness statement of Cseq, we know

extendsEnv(ew'', e2, ew''') -------------------------- (b.2)

We instantiate the soundness statement of Cseq for Cseq2 with
em as em' cw as e3, ew as ew'', ew' as ew', dw as dw'', dw' as dw', tw as tw'', tw' as tw'

to get

wellTyped(em', Cseq2) ∧ consistent(em', ew'', dw'', tw'') ∧
<e3, ew', dw', tw'> = T⟦Cseq2⟧(em', ew'', dw'', tw'')
⇒ [wellTyped(e3, ew', dw', tw') ∧ extendsEnv(ew', e3, ew'') ∧ extendsDecl(dw', e3, dw'')
∧ extendsTheory(tw', e3, tw'') ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t', e3> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq2⟧(em')(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq2⟧(em')(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]
]

From assumptions (1.c), (2.b), (3.c) and soundness statement of Cseq, we know

extendsEnv(ew', e3, ew'') ------------------------- (b.3)

We instantiate lemma (L-c2) with
em as em, E as E, Cseq1 as Cseq1, Cseq2 as Cseq2, e1 as e1, e2 as e2, e3 as e3, ew as ew, ew', ew'', ew'', ew''' as ew''', dw as dw, dw' as dw', dw'' as dw'', dw''' as dw''', tw as tw, tw' as tw', tw'' as tw'', tw''' as tw'''
to get

wellTyped(em, if E then Cseq1 else Cseq2 end) ∧
<if e1 then e2 else e3, ew', dw', tw'> = T⟦if E then Cseq1 else Cseq2 end⟧(em, ew, dw, tw) ∧
<e1, ew''', dw''', tw'''> = T⟦E⟧(em, ew, dw, tw) ∧
em' = Env(em, E) ∧
<e2, ew'', dw'', tw''> = T⟦Cseq1⟧(em', ew''', dw''', tw''') ∧
<e3, ew', dw', tw'> = T⟦Cseq2⟧(em', ew'', dw'', tw'') ∧
⇒
[extendsEnv(ew''', e1, ew) ∧ extendsEnv(ew'', e2, ew''') ∧ extendsEnv(ew', e3, ew'')
⇒ extendsEnv(ew', if e1 then e2 else e3, ew)] ∧
[extendsDecl(dw''', e1, dw) ∧ extendsDecl(dw'', e2, dw''') ∧ extendsDecl(dw', e3, dw'')
⇒ extendsDecl(dw', if e1 then e2 else e3, dw)] ∧
[extendsTheory(tw''', e1, tw) ∧ extendsTheory(tw'', e2, tw''') ∧ extendsTheory(tw', e3, tw'')
⇒ extendsTheory(tw', if e1 then e2 else e3, tw)]

From assumptions (1), (3), (3.a), (3.a'), (3.b), (3.c), (b.1), (b.2), (b.3) and lemma (L-c2), we know

extendsEnv(ew', if e1 then e2 else e3, ew)

which is the goal. Hence (b) proved.

Sub-Goal (c)

We instantiate soundness statement of E with
em as em, expw as e1, ew as ew, ew' as ew''', dw as dw, dw' as dw''', tw as tw, tw' as tw'''
and get

wellTyped(em, E) ∧ consistent(em, ew, dw, tw) ∧
<e1, ew''', dw''', tw'''> = T⟦E⟧(em, ew, dw, tw)
⇒ [wellTyped(e1, ew''', dw''', tw''') ∧ extendsEnv(ew''', e1, ew) ∧ extendsDecl(dw''', e1, dw)
∧ extendsTheory(tw''', e1, tw) ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, vw)
]
]
]

From assumptions (1.a), (2), (3.a) and the soundness statement of E, we know extendsDecl(dw''', e1, dw) --------------------------- (b.4)

We instantiate the soundness statement of Cseq for Cseq1 with
em as em' cw as e2, ew as ew''', ew' as ew'', dw as dw''', dw' as dw'', tw as tw''', tw' as tw''

to get

wellTyped(em', Cseq1) ∧ consistent(em', ew''', dw''', tw''') ∧
<e2, ew'', dw'', tw''> = T⟦Cseq1⟧(em', ew''', dw''', tw''')
⇒ [wellTyped(e2, ew'', dw'', tw'') ∧ extendsEnv(ew'', e2, ew''') ∧ extendsDecl(dw'', e2, dw''')
∧ extendsTheory(tw'', e2, tw''') ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t', e2> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq1⟧(em')(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq1⟧(em')(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]
]

From assumptions (1.b), (2.a), (3.b) and soundness statement of Cseq, we know

extendsDecl(dw'', e2, dw''') -------------------------- (b.5)

We instantiate the soundness statement of Cseq for Cseq2 with
em as em' cw as e3, ew as ew'', ew' as ew', dw as dw'', dw' as dw', tw as tw'', tw' as tw'

to get

wellTyped(em', Cseq2) ∧ consistent(em', ew'', dw'', tw'') ∧
<e3, ew', dw', tw'> = T⟦Cseq2⟧(em', ew'', dw'', tw'')
⇒ [wellTyped(e3, ew', dw', tw') ∧ extendsEnv(ew', e3, ew'') ∧ extendsDecl(dw', e3, dw'')
∧ extendsTheory(tw', e3, tw'') ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t', e3> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq2⟧(em')(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq2⟧(em')(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]
]

From assumptions (1.c), (2.b), (3.c) and soundness statement of Cseq, we know extendsDecl(dw', e3, dw'') -------------------------- (b.6)

We instantiate lemma (L-c2) with
em as em, E as E, Cseq1 as Cseq1, Cseq2 as Cseq2, e1 as e1, e2 as e2, e3 as e3, ew as ew, ew', ew'', ew'', ew''' as ew''', dw as dw, dw' as dw', dw'' as dw'', dw''' as dw''', tw as tw, tw' as tw', tw'' as tw'', tw''' as tw'''
to get

wellTyped(em, if E then Cseq1 else Cseq2 end) ∧
<if e1 then e2 else e3, ew', dw', tw'> = T⟦if E then Cseq1 else Cseq2 end⟧(em, ew, dw, tw) ∧
<e1, ew''', dw''', tw'''> = T⟦E⟧(em, ew, dw, tw) ∧
em' = Env(em, E) ∧
<e2, ew'', dw'', tw''> = T⟦Cseq1⟧(em', ew''', dw''', tw''') ∧
<e3, ew', dw', tw'> = T⟦Cseq2⟧(em', ew'', dw'', tw'') ∧
⇒
[extendsEnv(ew''', e1, ew) ∧ extendsEnv(ew'', e2, ew''') ∧ extendsEnv(ew', e3, ew'')
⇒ extendsEnv(ew', if e1 then e2 else e3, ew)] ∧
[extendsDecl(dw''', e1, dw) ∧ extendsDecl(dw'', e2, dw''') ∧ extendsDecl(dw', e3, dw'')
⇒ extendsDecl(dw', if e1 then e2 else e3, dw)] ∧
[extendsTheory(tw''', e1, tw) ∧ extendsTheory(tw'', e2, tw''') ∧ extendsTheory(tw', e3, tw'')
⇒ extendsTheory(tw', if e1 then e2 else e3, tw)]

From assumptions (1), (3), (3.a), (3.a'), (3.b), (3.c), (b.4), (b.5), (b.6) and lemma (L-c2), we know

extendsDecl(dw', if e1 then e2 else e3, dw)

which is the goal. Hence (c) proved.

Sub-Goal (d)

We instantiate soundness statement of E with
em as em, expw as e1, ew as ew, ew' as ew''', dw as dw, dw' as dw''', tw as tw, tw' as tw'''
and get

wellTyped(em, E) ∧ consistent(em, ew, dw, tw) ∧
<e1, ew''', dw''', tw'''> = T⟦E⟧(em, ew, dw, tw)
⇒ [wellTyped(e1, ew''', dw''', tw''') ∧ extendsEnv(ew''', e1, ew) ∧ extendsDecl(dw''', e1, dw)
∧ extendsTheory(tw''', e1, tw) ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)

⇒ equals(s', t') ∧ equals(vm, vw)
]
]
]

From assumptions (1.a), (2), (3.a) and the soundness statement of E, we know

extendsTheory(tw''', e1, tw) ---------------------------- (b.7)

We instantiate the soundness statement of Cseq for Cseq1 with
em as em' cw as e2, ew as ew''', ew' as ew'', dw as dw''', dw' as dw'', tw as tw''', tw' as tw''

to get

wellTyped(em', Cseq1) ∧ consistent(em', ew''', dw''', tw''') ∧
<e2, ew'', dw'', tw''> = T⟦Cseq1⟧(em', ew''', dw''', tw''')
⇒ [wellTyped(e2, ew'', dw'', tw'') ∧ extendsEnv(ew'', e2, ew''') ∧ extendsDecl(dw'', e2, dw''')
∧ extendsTheory(tw'', e2, tw''') ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t', e2> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq1⟧(em')(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq1⟧(em')(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]
]

From assumptions (1.b), (2.a), (3.b) and soundness statement of Cseq, we know

extendsTheory(tw'', e2, tw''') --------------------------- (b.8)

We instantiate the soundness statement of Cseq for Cseq2 with
em as em' cw as e3, ew as ew'', ew' as ew', dw as dw'', dw' as dw', tw as tw'', tw' as tw'

to get

wellTyped(em', Cseq2) ∧ consistent(em', ew'', dw'', tw'') ∧
<e3, ew', dw', tw'> = T⟦Cseq2⟧(em', ew'', dw'', tw'')
⇒ [wellTyped(e3, ew', dw', tw') ∧ extendsEnv(ew', e3, ew'') ∧ extendsDecl(dw', e3, dw'')
∧ extendsTheory(tw', e3, tw'') ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t', e3> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq2⟧(em')(s, s')]
∧

[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq2⟧(em')(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]
]

From assumptions (1.c), (2.b), (3.c) and soundness statement of Cseq, we know

extendsTheory(tw', e3, tw'') -------------------------- (b.9)

We instantiate lemma (L-c2) with
em as em, E as E, Cseq1 as Cseq1, Cseq2 as Cseq2, e1 as e1, e2 as e2, e3 as e3, ew as ew, ew', ew'', ew'', ew''' as ew''', dw as dw, dw' as dw', dw'' as dw'', dw''' as dw''', tw as tw, tw' as tw', tw'' as tw'', tw''' as tw'''
to get

wellTyped(em, if E then Cseq1 else Cseq2 end) ∧
<if e1 then e2 else e3, ew', dw', tw'> = T⟦if E then Cseq1 else Cseq2 end⟧(em, ew, dw, tw) ∧
<e1, ew''', dw''', tw'''> = T⟦E⟧(em, ew, dw, tw) ∧
em' = Env(em, E) ∧
<e2, ew'', dw'', tw''> = T⟦Cseq1⟧(em', ew''', dw''', tw''') ∧
<e3, ew', dw', tw'> = T⟦Cseq2⟧(em', ew'', dw'', tw'') ∧
⇒
[extendsEnv(ew''', e1, ew) ∧ extendsEnv(ew'', e2, ew''') ∧ extendsEnv(ew', e3, ew'')
⇒ extendsEnv(ew', if e1 then e2 else e3, ew)] ∧
[extendsDecl(dw''', e1, dw) ∧ extendsDecl(dw'', e2, dw''') ∧ extendsDecl(dw', e3, dw'')
⇒ extendsDecl(dw', if e1 then e2 else e3, dw)] ∧
[extendsTheory(tw''', e1, tw) ∧ extendsTheory(tw'', e2, tw''') ∧ extendsTheory(tw', e3, tw'')
⇒ extendsTheory(tw', if e1 then e2 else e3, tw)]

From assumptions (1), (3), (3.a), (3.a'), (3.b), (3.c), (b.7), (b.8), (b.9) and lemma (L-c2), we know

extendsTheory(tw', if e1 then e2 else e3, tw)

which is the goal. Hence (d) proved.

Sub-Goal (e)

Let t, t', cw, vw be arbitrary but fixed.
We assume:

<t, cw> ⟶ <t', vw> ------------------ (4)

From (3), we know

<t, if e1 then e2 else e3> ⟶ <t', vw> -------------------- (4')

From rule (cond-t), we assume

<t, e1> ⟶ <t'', true> for some t'' ------------------ (5)
<t'', e2> ⟶ <t', vw> for some t'' ------------------ (6)

From rule (cond-f), we assume

<t, e1> ⟶ <t'', false> for some t'' ------------------ (7)
<t'', e3> ⟶ <t', vw> for some t'' ------------------ (8)

We define:

s := constructs(t) ----------------- (4.a)
s'' := constructs(t'') ----------------- (4.b)
s' := constructs(t') ----------------- (4.c)

We show:

∃ s, s' ∈ State: equals(s, t) ∧ ⟦if E then Cseq1 else Cseq2 end⟧(em)(s, s')
------------------ (e.a)

∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t) ∧ ⟦if E then Cseq1 else Cseq2⟧(em)(s, s')
∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw) ------------------ (e.b)

Sub-Goal (e.a)

We show:

equals(s, t) --------------------- (e.a.1)
⟦if E then Cseq1 else Cseq2 end⟧(em)(s, s')] --------------------- (e.a.2)

Sub-Goal (e.a.1)

We instantiate lemma (L-cseq5) with
s as s, t as t
to get

s = constructs(t) ⇒ equals(s,t)

From assumption (4.a) and (L-cseq5), we know

equals(s,t)

which is the goal (e.a.1). Hence (e.a.1) is proved.

Sub-Goal (e.a.2)

We instantiate soundness statement of E with
em as em, expw as e1, ew as ew, ew' as ew''', dw as dw, dw' as dw''', tw as tw, tw' as tw'''
and get

wellTyped(em, E) ∧ consistent(em, ew, dw, tw) ∧
<e1, ew''', dw''', tw'''> = T⟦E⟧(em, ew, dw, tw)
⇒ [wellTyped(e1, ew''', dw''', tw''') ∧ extendsEnv(ew''', e1, ew) ∧ extendsDecl(dw''', e1, dw)
∧ extendsTheory(tw''', e1, tw) ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, vw)
]
]
]

From assumptions (1.a), (2), (3.a) and the soundness statement of E, we know

[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, vw)
]
] ----------------------------- (T)

We have two cases here for (T)

Case 1: When vw = True

We instantiate (T) with
t as t, t' as t''', vw as true to get

<t, e1> ⟶ <t'', true>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, vw)
]

From assumption (5), we know

∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)

By instantiating above formula with s as s, s' as s'', vm as inValue(True), we know

there is s, s'', inValue(True)

⟦E⟧(em)(s,s'',inValue(True)) --------------------- (e.a.2.1)

We instantiate the soundness statement of Cseq for Cseq1 with
em as em', cw as e2, ew as ew''', ew' as ew'', dw as dw''', dw' as dw'', tw as tw''', tw' as tw''

to get

wellTyped(em', Cseq1) ∧ consistent(em', ew''', dw''', tw''') ∧
<e2, ew'', dw'', tw''> = T⟦Cseq1⟧(em', ew''', dw''', tw''')
⇒ [wellTyped(e2, ew'', dw'', tw'') ∧ extendsEnv(ew'', e2, ew''') ∧ extendsDecl(dw'', e2, dw''')
∧ extendsTheory(tw'', e2, tw''') ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e2> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq1⟧(em')(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq1⟧(em')(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]
]

From assumptions (1.c), (3.e), (3.c) and soundness statement of Cseq, we know

[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e2> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq1⟧(em')(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq1⟧(em')(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]

We instantiate the above formula with t as t'', t' as t', vw as vw to get

[∀ t'', t' ∈ Statew, vw ∈ Valuew: <t'', e2> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq1⟧(em')(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq1⟧(em')(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]

From assumption (6) and above formula we get

[∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq1⟧(em')(s, s')]

By instantiating the above formula with s as s'', s' as s', we know that

there is s'', s' s.t.

⟦Cseq1⟧(em')(s'',s') ---------------- (e.a.2.2)

From (e.a.2.1), (e.a.2.2) and the definition of semantics of conditional command (when E evaluates to True) follows, which proves Case 1 of the goal (e.a.2).

Case 2: When vw = False

We instantiate above (T) with
t as t, t' as t''', vw as false to get

<t, e1> ⟶ <t'', false>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, vw)
]

From assumption (7), we know

∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)

By instantiating above formula with s as s, s' as s'', vm as inValue(False), we know

there is s, s'', inValue(False)

⟦E⟧(em)(s,s'',inValue(False)) --------------------- (e.a.2.3)

We instantiate the soundness statement of Cseq for Cseq2 with
em as em', cw as e3, ew as ew'', ew' as ew', dw as dw'', dw' as dw', tw as tw'', tw' as tw''

to get

wellTyped(em', Cseq2) ∧ consistent(em', ew'', dw'', tw'') ∧
<e3, ew', dw', tw'> = T⟦Cseq2⟧(em', ew'', dw'', tw'')
⇒ [wellTyped(e3, ew', dw', tw') ∧ extendsEnv(ew', e3, ew'') ∧ extendsDecl(dw', e3, dw'')
∧ extendsTheory(tw', e3, tw'') ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e3> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq2⟧(em')(s, s')]

```
        ∧
        [ ∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
            ∧ ⟦Cseq2⟧(em')(s, s') ∧ dm = infoData(s')
          ⇒ equals(s', t') ∧ equals(dm, vw)
        ]
    ]
  ]
```

From assumptions (1.c), (3.e), (3.c) and soundness statement of Cseq, we know

```
[ ∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e3> ⟶ <t', vw>
  ⇒ [ ∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq2⟧(em')(s, s') ]
     ∧
     [ ∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
         ∧ ⟦Cseq2⟧(em')(s, s') ∧ dm = infoData(s')
       ⇒ equals(s', t') ∧ equals(dm, vw)
     ]
]
```

We instantiate the above formula with t as t'', t' as t', vw as vw to get

```
[ ∀ t'', t' ∈ Statew, vw ∈ Valuew: <t'', e3> ⟶ <t', vw>
  ⇒ [ ∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq2⟧(em')(s, s') ]
     ∧
     [ ∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
         ∧ ⟦Cseq1⟧(em')(s, s') ∧ dm = infoData(s')
       ⇒ equals(s', t') ∧ equals(dm, vw)
     ]
]
```

From assumption (8) and above formula we get

[∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq2⟧(em')(s, s')]

By instantiating the above formula with s as s'', s' as s', we know that

there is s'', s' s.t.

⟦Cseq2⟧(em')(s'',s') ---------------- (e.a.2.4)

From (e.a.2.3), (e.a.2.4) and the definition of semantics of conditional command (when E evaluates to False) follows, which proves Case 2 of the goal (e.a.2).

The full definition of (e.a) follows from (e.a.2.1), (e.a.2.2), (e.a.2.3), (e.a.2.4) and (3.a'). Hence (e.a) is proved.

Sub-Goal (e.b)

Let s, s', dm, t be arbitrary but fixed.

We assume:

equals(s,t) --------------------- (7)
⟦if E then Cseq1 else Cseq2 end⟧(em)(s,s') --------------------- (8)
dm = infoData(s') --------------------- (9)

We define:

s' := constructs(t') ------------------- (9.a)
vw := constructs(dm) ------------------- (9.b)

We show:

equals(s', t') --------------------- (e.b.1)
equals(dm, vw) --------------------- (e.b.2)

Sub-Sub-Goal (e.b.1)

We instantiate lemma (L-cseq5) with
s as s' and t as t'
to get

s' = constructs(t') ⇒ equals(s', t')

From (9.a) and (L-cseq5), we know

equals(s', t')
which is the goal (e.b.1). Hence proved.

Sub-Sub-Goal (e.b.2)

We instantiate lemma (L-cseq6) with
v as vw, v' as dm
to get

vw = constructs(dm) ⇒ equals(dm, vw)
From (9.b) and (L-cseq6), we know

equals(dm, vw)

which is the goal (e.b.2). Hence proved.

Consequently, the goal (e.b) follows from (e.b.1) and (e.b.2). Hence (e.b) is proved.

Finally, the goal (e) follows from goals (e.a) and (e.b).

Also the goal (G21) follows from goals (a), (b), (c), (d) and (e).

Hence (G21) proved.

D.2.2 Case 2: C := I, Iseq := E, Eseq

Based on the available semantics definition of corresponding Why3 constructs (Iseq), we limit the proof here as explain next; we have many sub-cases depending on the grammar of Iseq and Eseq; however, we prove the usual case (when Iseq and Eseq are EMPTY) and the rests are left as an exercise.

As the behavior respectively translation of an assignment command is depends on whether it has occurred in the “global” and “local” context. We consider only the “local” context, when the variables are already declared.

Also, we assume the case, when an expression E evaluates to some value other than a module or a procedure because of the missing semantics definition of corresponding Why3 constructs.

The goal (G2) can be re-stated as follows:

∀ em ∈ Environment, x, e ∈ Expressionw, ew, ew' ∈ Environmentw, dw, dw' ∈ Declw, tw, tw' ∈ Theoryw:

wellTyped(em, I := E) ∧ consistent(em, ew, dw, tw) ∧
<x:=e, ew', dw', tw') = T⟦I := E⟧(em, ew, dw, tw)
⇒ [wellTyped(x:=e, ew', dw', tw') ∧ extendsEnv(ew', x:=e, ew)
∧ extendsDecl(dw', x:=e, dw)
∧ extendsTheory(tw', x:=e , tw) ∧
[∀ t, t' ∈ Statew, void ∈ Valuew: <t, x:=e> ⟶ <t', void>
⇒ [∃ s, s' ∈ State: equals(s, t)
∧ ⟦I := E⟧(em)(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦I := E⟧(em)(s, s')
∧ dm = infoData(I:=E, s')
⇒ equals(s', t') ∧ equals(dm, void)
]
]
] ------------------- (G22)

Let em, x, e, ew, ew', dw, dw', tw, tw', dm be arbitrary but fixed.

We assume:

wellTyped(em,I:=E) -------------- (1)
consistent(em, ew, dw, tw) -------------- (2)
<x:=e, ew', dw', tw'> = T⟦I:=E⟧(em, ew, dw, tw) -------------- (3)

By expanding the definition of (3), we know
<e, ew'', dw'', tw''> = T⟦E⟧(em, ew, dw, tw) ----------------- (3.a)
em' = Env(em, E) ---------------- (3.a')
<x, ew', dw', tw'> = T⟦I⟧(em', ew'', dw'', tw'') ---------------- (3.b)

We show:

wellTyped(x:=e, ew', dw', tw') -------------- (a)
extendsEnv(ew', x:=e, ew) -------------- (b)
extendsDecl(dw', x:=e, dw) -------------- (c)
extendsTheory(tw', x:=e , tw) -------------- (d)
[∀ t, t' ∈ Statew, void ∈ Valuew: <t, x:=e> ⟶ <t', void>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦I:=E⟧(em)(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦I:=E⟧(em)(s, s')
∧ dm = infoData(I:=E, s')
⇒ equals(s', t') ∧ equals(dm, void)
]
] -------------- (e)

Sub-Goal (a)

We instantiate lemma (L-c1) with
c as I:=E, em as em, e as x:=e, ew as ew, ew' as ew', dw as dw, dw' as dw', tw as tw, tw' as tw'
and get

wellTyped(em, I:=E)
∧ (x:=e, ew', dw', tw') = T⟦I:=E⟧(em, ew, dw, tw)
⇒ wellTyped(x:=e, ew', dw', tw')

From assumptions (1), (3) and (L-c1), we know

wellTyped(x:=e, ew', dw', tw')

which is the goal (a). Hence (a) proved.

Sub-Goal (b)

We instantiate lemma (L-c5) with
em as em, em' as em', I as I, E as E
to get

wellTyped(em, I:=E) ⇒
wellTyped(em, E) ∧ em' = Env(em, E) ∧ wellTyped(em', I)

From (1) and (L-c5), we know

wellTyped(em, E) --------------------- (1.a)
em' = Env(em, E) --------------------- (1.a')
wellTyped(em', I) -------------------- (1.b)

We instantiate lemma (L-c6) with

em as em, em’ as em’, I as I, E as E, x as x, e as e, ew as ew, ew’ ew’, ew” as ew”, dw as dw, dw’ as dw’, dw” as dw”, tw as tw, tw’ as tw’, tw” as tw”
to get

<e, ew”, dw”, tw”> = T⟦E⟧(em, ew, dw, tw) ∧ em’ = Env(em, E) ∧
<x, ew’, dw’, tw’> = T⟦I⟧(em’, ew”, dw”, tw”) ∧ consistent(em, ew, dw, tw)
⇒ consistent(em’, ew”, dw”, tw”)

From assumptions (3.a), (3.a’), (3.b), (2) and (L-c6), we know

consistent(em’, ew”, dw”, tw”) ---------------------- (2.a)

We instantiate soundness statement of E with
em as em, expw as e, ew as ew, ew’ as ew”, dw as dw, dw’ as dw”, tw as tw, tw’ as tw”
and get

wellTyped(em, E) ∧ consistent(em, ew, dw, tw) ∧
<e, ew”, dw”, tw”> = T⟦E⟧(em, ew, dw, tw)
⇒ [wellTyped(e, ew”, dw”, tw”) ∧ extendsEnv(ew”, e, ew) ∧ extendsDecl(dw”, e, dw)
∧ extendsTheory(tw”, e, tw) ∧
[∀ t, t’ ∈ Statew, vw ∈ Valuew: <t, e> ⟶ <t’, vw>
⇒ [∃ s, s’ ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s’, vm)]
∧
[∀ s, s’ ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s’, vm)
⇒ equals(s’, t’) ∧ equals(vm, vw)
]
]
]

From assumptions (1.a), (2), (3.a) and the soundness statement of E, we know

extendsEnv(ew”, e, ew) ---------------------------- (b.1)

We instantiate soundness statement of E for identifier expression with
em as em’, expw as x, ew as ew”, ew’ as ew’, dw as dw”, dw’ as dw’, tw as tw”, tw’ as tw’
and get

wellTyped(em’, I) ∧ consistent(em’, ew”, dw”, tw”) ∧
<x, ew’, dw’, tw’> = T⟦I⟧(em’, ew”, dw”, tw”)
⇒ [wellTyped(x, ew’, dw’, tw’) ∧ extendsEnv(ew’, x, ew”) ∧ extendsDecl(dw’, x, dw”)
∧ extendsTheory(tw’, x, tw”) ∧
[∀ t, t’ ∈ Statew, vw ∈ Valuew: <t, x> ⟶ <t’, vw>
⇒ [∃ s, s’ ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦I⟧(em)(s, s’, vm)]
∧

[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦I⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, vw)
]
]
]

From assumptions (1.b), (2.a), (3.b) and soundness statement of E, we know

extendsEnv(ew', x, ew'') --------------------------- (b.2)

We instantiate lemma (L-c7) with
em as em, em' as em', I as I, E as E, x as x, e as e, ew as ew, ew' as ew', ew'' as ew'', dw as dw, dw' as dw', dw'' as dw'', tw as tw, tw' as tw', tw'' as tw''
to get

wellTyped(em, E) ∧
<x:=e, ew', dw', tw'> = T⟦I:=E⟧(em, ew, dw, tw) ∧
<e, ew'', dw'', tw''> = T⟦E⟧(em, ew, dw, tw) ∧
em' = Env(em, E) ∧
<x, ew', dw', tw'> = T⟦I⟧(em', ew'', dw'', tw'') ∧
⇒
[extendsEnv(ew'', e, ew) ∧ extendsEnv(ew', x, ew'') ⇒ extendsEnv(ew', x:=e, ew)] ∧
[extendsDecl(dw'', e, dw) ∧ extendsDecl(dw', x, dw'') ⇒ extendsDecl(dw', x:=e, dw)] ∧
[extendsTheory(tw'', e, tw) ∧ extendsTheory(tw', x, tw'') ⇒ extendsTheory(tw', x:=e, tw)]

From assumptions (1), (3), (3.a), (3.a'), (3.b), (b.1), (b.2) and lemma (L-c7), we know

extendsEnv(ew', x:=e, ew)

which is the goal. Hence (b) proved.

Sub-Goal (c)

We instantiate soundness statement of E with
em as em, expw as e, ew as ew, ew' as ew'', dw as dw, dw' as dw'', tw as tw, tw' as tw''
and get

wellTyped(em, E) ∧ consistent(em, ew, dw, tw) ∧
<e, ew'', dw'', tw''> = T⟦E⟧(em, ew, dw, tw)
⇒ [wellTyped(e, ew'', dw'', tw'') ∧ extendsEnv(ew'', e, ew) ∧ extendsDecl(dw'', e, dw)
∧ extendsTheory(tw'', e, tw) ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)]
∧

[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, vw)
]
]
]

From assumptions (1.a), (2), (3.a) and the soundness statement of E, we know

extendsDecl(dw'', e, dw) ---------------------------- (b.3)

We instantiate soundness statement of E for identifier expression with
em as em', expw as x, ew as ew'', ew' as ew', dw as dw'', dw' as dw', tw as tw'', tw' as tw'
and get

wellTyped(em', I) ∧ consistent(em', ew'', dw'', tw'') ∧
<x, ew', dw', tw'> = T⟦I⟧(em', ew'', dw'', tw'')
⇒ [wellTyped(x, ew', dw', tw') ∧ extendsEnv(ew', x, ew'') ∧ extendsDecl(dw', x, dw'')
∧ extendsTheory(tw', x, tw'') ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, x> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦I⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦I⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, vw)
]
]
]

From assumptions (1.b), (2.a), (3.b) and soundness statement of E, we know

extendsDecl(dw', x, dw'') --------------------------- (b.4)

We instantiate lemma (L-c7) with
em as em, em' as em', I as I, E as E, x as x, e as e, ew as ew, ew' as ew', ew'' as ew'', dw as dw, dw' as dw', dw'' as dw'', tw as tw, tw' as tw', tw'' as tw''
to get

wellTyped(em, E) ∧
<x:=e, ew', dw', tw'> = T⟦I:=E⟧(em, ew, dw, tw) ∧
<e, ew'', dw'', tw''> = T⟦E⟧(em, ew, dw, tw) ∧
em' = Env(em, E) ∧
<x, ew', dw', tw'> = T⟦I⟧(em', ew'', dw'', tw'') ∧
⇒
[extendsEnv(ew'', x, ew) ∧ extendsEnv(ew', e, ew'') ⇒ extendsEnv(ew', x:=e, ew)] ∧
[extendsDecl(dw'', x, dw) ∧ extendsDecl(dw', e, dw'') ⇒ extendsDecl(dw', x:=e, dw)] ∧

[extendsTheory(tw'', x, tw) ∧ extendsTheory(tw', e, tw'') ⇒ extendsTheory(tw', x:=e, tw)]

From assumptions (1), (3), (3.a), (3.a'), (3.b), (b.3), (b.4) and lemma (L-c7), we know

extendsDecl(dw', x:=e, dw)

which is the goal. Hence (c) proved.

Sub-Goal (d)

We instantiate soundness statement of E with
em as em, expw as e, ew as ew, ew' as ew'', dw as dw, dw' as dw'', tw as tw, tw' as tw''
and get

wellTyped(em, E) ∧ consistent(em, ew, dw, tw) ∧
<e, ew'', dw'', tw''> = T⟦E⟧(em, ew, dw, tw)
⇒ [wellTyped(e, ew'', dw'', tw'') ∧ extendsEnv(ew'', e, ew) ∧ extendsDecl(dw'', e, dw)
∧ extendsTheory(tw'', e, tw) ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, vw)
]
]
]

From assumptions (1.a), (2), (3.a) and the soundness statement of E, we know

extendsTheory(tw'', e, tw) ---------------------------- (b.5)

We instantiate soundness statement of E for identifier expression with
em as em', expw as x, ew as ew'', ew' as ew', dw as dw'', dw' as dw', tw as tw'', tw' as tw'
and get

wellTyped(em', I) ∧ consistent(em', ew'', dw'', tw'') ∧
<x, ew', dw', tw'> = T⟦I⟧(em', ew'', dw'', tw'')
⇒ [wellTyped(x, ew', dw', tw') ∧ extendsEnv(ew', x, ew'') ∧ extendsDecl(dw', x, dw'')
∧ extendsTheory(tw', x, tw'') ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, x> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦I⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦I⟧(em)(s, s', vm)

⇒ equals(s', t') ∧ equals(vm, vw)
]
]
]

From assumptions (1.b), (2.a), (3.b) and soundness statement of E, we know

extendsTheory(tw', x, tw'') --------------------------- (b.6)

We instantiate lemma (L-c7) with
em as em, em' as em', I as I, E as E, x as x, e as e, ew as ew, ew' as ew', ew'' as ew'', dw as dw, dw' as dw', dw'' as dw'', tw as tw, tw' as tw', tw'' as tw''
to get

wellTyped(em, E) ∧
<x:=e, ew', dw', tw'> = T⟦I:=E⟧(em, ew, dw, tw) ∧
<e, ew'', dw'', tw''> = T⟦E⟧(em, ew, dw, tw) ∧
em' = Env(em, E) ∧
<x, ew', dw', tw'> = T⟦I⟧(em', ew'', dw'', tw'') ∧
⇒
[extendsEnv(ew'', x, ew) ∧ extendsEnv(ew', e, ew'') ⇒ extendsEnv(ew', x:=e, ew)] ∧
[extendsDecl(dw'', x, dw) ∧ extendsDecl(dw', e, dw'') ⇒ extendsDecl(dw', x:=e, dw)] ∧
[extendsTheory(tw'', x, tw) ∧ extendsTheory(tw', e, tw'') ⇒ extendsTheory(tw', x:=e, tw)]

From assumptions (1), (3), (3.a), (3.a'), (3.b), (b.5), (b.6) and lemma (L-c7), we know

extendsTheory(tw', x:=e, tw)

which is the goal. Hence (d) proved.

Sub-Goal (e)

Let t, t', cw, vw be arbitrary but fixed.
We assume:

<t, x:=e> ⟶ <t', void> ------------------ (4)

From (3), we know

<t, e> ⟶ <t'', vw> -------------------- (5)

From (4) and definition of corresponding Why3 semantics, we know

t' = t''+[x|->vw] ------------------- (6)

We define:

s := constructs(t) ----------------- (4.a)
s'' := constructs(t'') ----------------- (4.b)
s' := constructs(t') ----------------- (4.c)

We show:

∃ s, s' ∈ State: equals(s, t) ∧ ⟦I:=E⟧(em)(s, s') ------------------ (e.a)

∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t) ∧ ⟦I:=E⟧(em)(s, s')
∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, void) ------------------ (e.b)

Sub-Goal (e.a)

We show:

equals(s, t) --------------------- (e.a.1)
⟦I:=E⟧(em)(s, s')] --------------------- (e.a.2)

Sub-Goal (e.a.1)

We instantiate lemma (L-cseq5) with
s as s, t as t
to get

s = constructs(t) ⇒ equals(s,t)

From assumption (4.a) and (L-cseq5), we know

equals(s,t)

which is the goal (e.a.1). Hence (e.a.1) is proved.

Sub-Goal (e.a.2)

We instantiate soundness statement of E with
em as em, expw as e, ew as ew, ew' as ew'', dw as dw, dw' as dw'', tw as tw, tw' as tw''
and get

wellTyped(em, E) ∧ consistent(em, ew, dw, tw) ∧
<e, ew'', dw'', tw''> = T⟦E⟧(em, ew, dw, tw)
⇒ [wellTyped(e, ew'', dw'', tw'') ∧ extendsEnv(ew'', e, ew) ∧ extendsDecl(dw'', e, dw)
∧ extendsTheory(tw'', e, tw) ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, vw)

]
]
]

From assumptions (1.a), (2), (3.a) and the soundness statement of E, we know

[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, vw)
]
]

We instantiate the above formula with
t as t, t' as t'', vw as vw
to get

<t, e> ⟶ <t'', vw>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, vw)
]

From assumption (5) and above formula, we know

∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)

By instantiating above formula with s as s, s' as s'', vm as vm, we know

there is s, s'', vm

⟦E⟧(em)(s,s'',vm) --------------------- (e.a.2.1)

We define:

vw = constructs(vm) --------------------- (e.a.2.2)

We instantiate lemma (L-c8) with
x as x, e as e, s' as s', s'' as s'', t' as t', t'' as t'', vw as vw, vm as vm
to get

s' = constructs(t') ∧ s'' = constructs(t'') ∧ t' = t''+[x|->vw] ∧ vm = constructs(vw)
⇒ s' = update(x, vm, s'')

From (4.b), (4.c), (6), (e.a.2.2) and (L-c8), we know s' = update(x, vm, s'') ---------------- (e.a.2.3)

The definition of semantics of an assignment command (when Iseq and Eseq are EMPTY) follows from (e.a.2.1) and (e.a.2.3).

Hence (e.a) is proved.

Sub-Goal (e.b)

Let s, s', dm, t be arbitrary but fixed.

We assume:

equals(s,t) --------------------- (7)
⟦I:=E⟧(em)(s,s') --------------------- (8)
dm = infoData(s') --------------------- (9)

We define:

s' := constructs(t') ------------------- (9.a)
vw := constructs(dm) ------------------- (9.b)

We show:

equals(s', t') --------------------- (e.b.1)
equals(dm, void) --------------------- (e.b.2)

Sub-Sub-Goal (e.b.1)

We instantiate lemma (L-cseq5) with
s as s' and t as t'
to get

s' = constructs(t') ⇒ equals(s', t')

From (9.a) and (L-cseq5), we know

equals(s', t')
which is the goal (e.b.1). Hence proved.

Sub-Sub-Goal (e.b.2)

We instantiate lemma (L-cseq6) with
v as void, v' as dm
to get

void = constructs(dm) ⇒ equals(dm, void)
From (9.b) and (L-cseq6), we know

equals(dm, void)

which is the goal (e.b.2). Hence proved.

Consequently, the goal (e.b) follows from (e.b.1) and (e.b.2). Hence (e.b) is proved.

Finally, the goal (e) follows from goals (e.a) and (e.b).

Also the goal (G22) follows from goals (a), (b), (c), (d) and (e).

Hence (G22) proved.

D.2.3 Case 3: C := while E do Cseq end

The goal (G2) can be re-stated as follows:

∀ em ∈ Environment, e1, e2 ∈ Expressionw, ew, ew' ∈ Environmentw, dw, dw' ∈ Declw, tw, tw' ∈ Theoryw:

wellTyped(em, **while** E **do** Cseq **end**) ∧ consistent(em, ew, dw, tw) ∧
<while e1 do e2, ew', dw', tw'> = T⟦**while** E **do** Cseq **end**⟧(em, ew, dw, tw)
⇒ [wellTyped(while e1 do e2, ew', dw', tw')
∧ extendsEnv(ew', while e1 do e2, ew)
∧ extendsDecl(dw', while e1 do e2, dw)
∧ extendsTheory(tw', while e1 do e2, tw) ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, while e1 do e2> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t)
∧ ⟦**while** E **do** Cseq **end**⟧(em)(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦**while** E **do** Cseq **end**⟧(em)(s, s')
∧ dm = infoData(**while** E **do** Cseq **end**, s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]
] ------------------- (G23)

Let em, e1,e2, ew, ew', dw, dw', tw, tw', dm and vw be arbitrary but fixed.

We assume:

wellTyped(em,**while** E **do** Cseq **end**) -------------- (1)
consistent(em, ew, dw, tw) -------------- (2)
<while e1 do e2, ew', dw', tw'> = T⟦**while** E **do** Cseq **end**⟧(em, ew, dw, tw) -------------- (3)

By expanding the definition of (3), we know
<e1, ew'', dw'', tw''> = T⟦E⟧(em, ew, dw, tw) ----------------- (3.a)
em' = Env(em, E) ---------------- (3.a')
<e2, ew', dw', tw'> = T⟦Cseq⟧(em', ew'', dw'', tw'') ---------------- (3.b)

We show:

wellTyped(while e1 do e2, ew', dw', tw') -------------- (a)
extendsEnv(ew', while e1 do e2, ew) -------------- (b)
extendsDecl(dw', while e1 do e2, dw) -------------- (c)
extendsTheory(tw', while e1 do e2, tw) -------------- (d)
∀ t, t' ∈ Statew, vw ∈ Valuew: <t, while e1 do e2> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t)
∧ ⟦**while** E **do** Cseq **end**⟧(em)(s, s')]

∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦**while** E **do** Cseq **end**⟧(em)(s, s')
∧ dm = infoData(**while** E **do** Cseq **end**, s')
⇒ equals(s', t') ∧ equals(dm, vw)
] -------------- (e)

Sub-Goal (a)

We instantiate lemma (L-c1) with
c as while E do Cseq end, em as em, e as while e1 do e2, ew as ew, ew' as ew', dw as dw, dw' as dw', tw as tw, tw' as tw'
and get

wellTyped(em, while E do Cseq end)
∧ (while e1 do e2, ew', dw', tw') = T⟦while E do Cseq end⟧(em, ew, dw, tw)
⇒ wellTyped(while e1 do e2, ew', dw', tw')

From assumptions (1), (3) and (L-c1), we know

wellTyped(while e1 do e2, ew', dw', tw')

which is the goal (a). Hence (a) proved.

Sub-Goal (b)

We instantiate lemma (L-c9) with
em as em, em' as em', E as E, Cseq as Cseq
to get

wellTyped(em, while E do Cseq end) ⇒
wellTyped(em, E) ∧ em' = Env(em, E) ∧ wellTyped(em', Cseq)

From (1) and (L-c9), we know

wellTyped(em, E) --------------------- (1.a)
em' = Env(em, E) --------------------- (1.a')
wellTyped(em', Cseq) -------------------- (1.b)

We instantiate lemma (L-c10) with
em as em, em' as em', E as E, Cseq as Cseq, ew as ew, ew' ew', ew'' as ew'', dw as dw, dw' as dw', dw'' as dw'', tw as tw, tw' as tw', tw'' as tw''
to get

(e1, ew'', dw'', tw'') = T⟦E⟧(em, ew, dw, tw) ∧ em' = Env(em, E) ∧
(e2, ew', dw', tw') = T⟦Cseq⟧(em', ew'', dw'', tw'') ∧ consistent(em, ew, dw, tw)
⇒ consistent(em', dw'', dw'', tw'')

From assumptions (3.a), (3.a'), (3.b), (2) and (L-c10), we know consistent(em’, ew’’, dw’’, tw’’)   --------------------- (2.a)

We instantiate soundness statement of E with
em as em, expw as e1, ew as ew, ew’ as ew’’, dw as dw, dw’ as dw’’, tw as tw, tw’ as tw’’
and get

wellTyped(em, E) ∧ consistent(em, ew, dw, tw) ∧
<e1, ew’’, dw’’, tw’’> = T⟦E⟧(em, ew, dw, tw)
⇒ [wellTyped(e1, ew’’, dw’’, tw’’) ∧ extendsEnv(ew’’, e1, ew) ∧ extendsDecl(dw’’, e1, dw)
∧ extendsTheory(tw’’, e1, tw) ∧
[∀ t, t’ ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t’, vw>
⇒ [∃ s, s’ ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s’, vm)]
∧
[∀ s, s’ ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s’, vm)
⇒ equals(s’, t’) ∧ equals(vm, vw)
]
]
]

From assumptions (1.a), (2), (3.a) and the soundness statement of E, we know

extendsEnv(ew’’, e1, ew)   --------------------------- (b.1)

We instantiate the soundness statement of Cseq with
em as em’ cw as e2, ew as ew’’, ew’ as ew’, dw as dw’’, dw’ as dw’, tw as tw’’, tw’ as tw’

to get

wellTyped(em’, Cseq) ∧ consistent(em’, ew’’, dw’’, tw’’) ∧
<e2, ew’, dw’, tw’> = T⟦Cseq⟧(em’, ew’’, dw’’, tw’’)
⇒ [wellTyped(e2, ew’, dw’, tw’) ∧ extendsEnv(ew’, e2, ew’’) ∧ extendsDecl(dw’, e2, dw’’)
∧ extendsTheory(tw’, e2, tw’’) ∧
[∀ t, t’ ∈ Statew, vw ∈ Valuew: <t’, e2> ⟶ <t’, vw>
⇒ [∃ s, s’ ∈ State: equals(s, t) ∧ ⟦Cseq⟧(em’)(s, s’)]
∧
[∀ s, s’ ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq⟧(em’)(s, s’) ∧ dm = infoData(s’)
⇒ equals(s’, t’) ∧ equals(dm, vw)
]
]
]

From assumptions (1.b), (2.a), (3.b) and soundness statement of Cseq, we know extendsEnv(ew’, e2, ew’’) -------------------------- (b.2)

We instantiate lemma (L-c11) with
em as em, E as E, Cseq as Cseq, e1 as e1, e2 as e2, ew as ew, ew’, ew’’, ew’’, dw as dw, dw’ as dw’, dw’’ as dw’’, tw as tw, tw’ as tw’, tw’’ as tw’’
to get

wellTyped(em, while E do Cseq end) ∧
<while e1 do e2, ew’, dw’, tw’> = T⟦while E do Cseq end⟧(em, ew, dw, tw) ∧
<e1, ew’’, dw’’, tw’’> = T⟦E⟧(em, ew, dw, tw) ∧
em’ = Env(em, E) ∧
<e2, ew’, dw’, tw’> = T⟦Cseq⟧(em’, ew’’, dw’’, tw’’)
⇒
[extendsEnv(ew’’, e1, ew) ∧ extendsEnv(ew’, e2, ew’’)
⇒ extendsEnv(ew’, while e1 do e2, ew)] ∧
[extendsDecl(dw’’, e1, dw) ∧ extendsDecl(dw’, e2, dw’’)
⇒ extendsDecl(dw’, while e1 do e2, dw)] ∧
[extendsTheory(tw’’, e1, tw) ∧ extendsTheory(tw’, e2, tw’’)
⇒ extendsTheory(tw’, while e1 do e2, tw)]

From assumptions (1), (3), (3.a), (3.a’), (3.b), (b.1), (b.2) and lemma (L-c11), we know

extendsEnv(ew’, while e1 do e2, ew)

which is the goal. Hence (b) proved.

Sub-Goal (c)

We instantiate soundness statement of E with
em as em, expw as e1, ew as ew, ew’ as ew’’, dw as dw, dw’ as dw’’, tw as tw, tw’ as tw’’
and get

wellTyped(em, E) ∧ consistent(em, ew, dw, tw) ∧
<e1, ew’’, dw’’, tw’’> = T⟦E⟧(em, ew, dw, tw)
⇒ [wellTyped(e1, ew’’, dw’’, tw’’) ∧ extendsEnv(ew’’, e1, ew) ∧ extendsDecl(dw’’, e1, dw)
∧ extendsTheory(tw’’, e1, tw) ∧
[∀ t, t’ ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t’, vw>
⇒ [∃ s, s’ ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s’, vm)]
∧
[∀ s, s’ ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s’, vm)
⇒ equals(s’, t’) ∧ equals(vm, vw)
]
]
]

From assumptions (1.a), (2), (3.a) and the soundness statement of E, we know extendsDecl(dw'', e1, dw) ---------------------------- (b.3)

We instantiate the soundness statement of Cseq with
em as em' cw as e2, ew as ew'', ew' as ew', dw as dw'', dw' as dw', tw as tw'', tw' as tw'

to get

wellTyped(em', Cseq) ∧ consistent(em', ew'', dw'', tw'') ∧
<e2, ew', dw', tw'> = T⟦Cseq⟧(em', ew'', dw'', tw'')
⇒ [wellTyped(e2, ew', dw', tw') ∧ extendsEnv(ew', e2, ew'') ∧ extendsDecl(dw', e2, dw'')
∧ extendsTheory(tw', e2, tw'') ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t', e2> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State: equals(s, t) ∧ ⟦Cseq⟧(em')(s, s')]
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq⟧(em')(s, s') ∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)
]
]
]

From assumptions (1.b), (2.a), (3.b) and soundness statement of Cseq, we know

extendsDecl(dw', e2, dw'') --------------------------- (b.4)

We instantiate lemma (L-c11) with
em as em, E as E, Cseq as Cseq, e1 as e1, e2 as e2, ew as ew, ew', ew'', ew'', dw as dw, dw' as dw', dw'' as dw'', tw as tw, tw' as tw', tw'' as tw''
to get

wellTyped(em, while E do Cseq end) ∧
<while e1 do e2, ew', dw', tw'> = T⟦while E do Cseq end⟧(em, ew, dw, tw) ∧
<e1, ew'', dw'', tw''> = T⟦E⟧(em, ew, dw, tw) ∧
em' = Env(em, E) ∧
<e2, ew', dw', tw'> = T⟦Cseq⟧(em', ew'', dw'', tw'')
⇒
[extendsEnv(ew'', e1, ew) ∧ extendsEnv(ew', e2, ew'')
⇒ extendsEnv(ew', while e1 do e2, ew)] ∧
[extendsDecl(dw'', e1, dw) ∧ extendsDecl(dw', e2, dw'')
⇒ extendsDecl(dw', while e1 do e2, dw)] ∧
[extendsTheory(tw'', e1, tw) ∧ extendsTheory(tw', e2, tw'')
⇒ extendsTheory(tw', while e1 do e2, tw)]

From assumptions (1), (3), (3.a), (3.a'), (3.b), (b.3), (b.4) and lemma (L-c11), we know extendsDecl(dw’, while e1 do e2, dw)

which is the goal. Hence (c) proved.

Sub-Goal (d)

We instantiate soundness statement of E with
em as em, expw as e1, ew as ew, ew’ as ew’’, dw as dw, dw’ as dw’’, tw as tw, tw’ as tw’’
and get

wellTyped(em, E) ∧ consistent(em, ew, dw, tw) ∧
<e1, ew’’, dw’’, tw’’> = T⟦E⟧(em, ew, dw, tw)
⇒ [wellTyped(e1, ew’’, dw’’, tw’’) ∧ extendsEnv(ew’’, e1, ew) ∧ extendsDecl(dw’’, e1, dw)
∧ extendsTheory(tw’’, e1, tw) ∧
[∀ t, t’ ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t’, vw>
⇒ [∃ s, s’ ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s’, vm)]
∧
[∀ s, s’ ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s’, vm)
⇒ equals(s’, t’) ∧ equals(vm, vw)
]
]
]

From assumptions (1.a), (2), (3.a) and the soundness statement of E, we know

extendsTheory(tw’’, e1, tw) ---------------------------- (b.5)

We instantiate the soundness statement of Cseq with
em as em’ cw as e2, ew as ew’’, ew’ as ew’, dw as dw’’, dw’ as dw’, tw as tw’’, tw’ as tw’

to get

wellTyped(em’, Cseq) ∧ consistent(em’, ew’’, dw’’, tw’’) ∧
<e2, ew’, dw’, tw’> = T⟦Cseq⟧(em’, ew’’, dw’’, tw’’)
⇒ [wellTyped(e2, ew’, dw’, tw’) ∧ extendsEnv(ew’, e2, ew’’) ∧ extendsDecl(dw’, e2, dw’’)
∧ extendsTheory(tw’, e2, tw’’) ∧
[∀ t, t’ ∈ Statew, vw ∈ Valuew: <t’, e2> ⟶ <t’, vw>
⇒ [∃ s, s’ ∈ State: equals(s, t) ∧ ⟦Cseq⟧(em’)(s, s’)]
∧
[∀ s, s’ ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq⟧(em’)(s, s’) ∧ dm = infoData(s’)
⇒ equals(s’, t’) ∧ equals(dm, vw)
]
]
]

From assumptions (1.b), (2.a), (3.b) and soundness statement of Cseq, we know

extendsTheory(tw', e2, tw'') -------------------------- (b.6)

We instantiate lemma (L-c11) with
em as em, E as E, Cseq as Cseq, e1 as e1, e2 as e2, ew as ew, ew', ew'', ew'', dw as dw, dw' as dw', dw'' as dw'', tw as tw, tw' as tw', tw'' as tw''
to get

wellTyped(em, while E do Cseq end) ∧
<while e1 do e2, ew', dw', tw'> = T⟦while E do Cseq end⟧(em, ew, dw, tw) ∧
<e1, ew'', dw'', tw''> = T⟦E⟧(em, ew, dw, tw) ∧
em' = Env(em, E) ∧
<e2, ew', dw', tw'> = T⟦Cseq⟧(em', ew'', dw'', tw'')
⇒
[extendsEnv(ew'', e1, ew) ∧ extendsEnv(ew', e2, ew'')
⇒ extendsEnv(ew', while e1 do e2, ew)] ∧
[extendsDecl(dw'', e1, dw) ∧ extendsDecl(dw', e2, dw'')
⇒ extendsDecl(dw', while e1 do e2, dw)] ∧
[extendsTheory(tw'', e1, tw) ∧ extendsTheory(tw', e2, tw'')
⇒ extendsTheory(tw', while e1 do e2, tw)]

From assumptions (1), (3), (3.a), (3.a'), (3.b), (b.5), (b.6) and lemma (L-c11), we know

extendsTheory(tw', while e1 do e2, tw)

which is the goal. Hence (d) proved.

Sub-Goal (e)

Let t, t', cw, vw be arbitrary but fixed s.t.
We assume:

<t, while e1 do e2> ⟶ <t', vw> ------------------ (4)

We show:

[∃ s, s' ∈ State: equals(s, t) ∧ ⟦while E do Cseq end⟧(em)(s, s')] ------------------ (e.a)

[∀ s, s' ∈ State, dm ∈ InfoData: equals(s, t) ∧ ⟦while E do Cseq end⟧(em)(s, s')
∧ dm = infoData(s')
⇒ equals(s', t') ∧ equals(dm, vw)] ------------------ (e.b)

The semantics of the classical Why3 while-loop is defined by a complex exception-handling mechanism. Based on the aforementioned semantics, a proof of this goal gets more complicated, thus to avoid this complication, we have derived (in the Appendix - Derivations) two rules conforming the definition of while-loop semantics which do not involve exceptions anymore. These two derivations are as follows:

<t, e1> ⟶ <t', false>
-- ------------------- (d.a)
<t, while e1 do e2> ⟶ <t', void>

<t, e1> ⟶ <t'', true>
<t'', e2> ⟶ <t''', void>
<t''', while e1 do e2> ⟶ <t', void>
-- ------------------- (d.b)
<t, while e1 do e2> ⟶ <t', void>

We prove this goal (e) by rule induction on the operational semantics of while-loop which is defined above by the two derivation rules (d.a) and (d.b). By the strategy of principle of rule induction for while-loop, the goal (e) can be re-formulated as:

∀ t, t' ∈ Statew, vw ∈ Valuew: <t, while e1 do e2> ⟶ <t', vw> ⇒ ℙ(t,t',vw) -----(G-e)

where
ℙ(t, t', vw) :⇔
[∃ s, s' ∈ State: equals(s,t) ∧ ⟦while E do Cseq⟧(em)(s,s')]
∧ [∀ s, s' ∈ State, dm ∈ InfoData:
equals(s',t') ∧ ⟦while E do Cseq⟧(em)(s,s') ∧ dm=infoData(s')
⇒ equals(s',t') ∧ equals(dm, vw)]

where E, Cseq and em are fixed as defined above ------------------------- (D-p)

To show (G-e), based on the principle of rule induction it suffices to show the followings for while-loop for the corresponding derivation rules respectively:

∀ t, t' ∈ Statew, vw ∈ Valuew, e1 ∈ Expressionw:
<t, e1> ⟶ <t', false> ⇒ ℙ(t,t',vw) ------------ (G-e.1)

∀ t, t',t'',t''' ∈ Statew, vw ∈ Valuew, e1, e2 ∈ Expressionw:
<t, e1> ⟶ <t'', true> ∧ <t'', e2> ⟶ <t''', void>
∧ <t''', while e1 do e2> ⟶ <t', void> ∧ ℙ(t''',t',void) ⇒ ℙ(t,t',vw) ------------ (G-e.2)

Goal (G-e.1):

We assume:

<t, e1> ⟶ <t', false> ------------------- (5)

We show:

$\mathbb{P}$(t,t',vw)

By expanding the definition of $\mathbb{P}$(t,t',vw), we get

[∃ s, s' ∈ State: equals(s,t) ∧ ⟦while E do Cseq⟧(em)(s,s')] ----------- (G-e.1.a)
∧ [∀ s, s' ∈ State, dm ∈ InfoData: equals(s',t') ∧ ⟦while E do Cseq⟧(em)(s,s') ∧ dm=infoData(s')
⇒ equals(s',t') ∧ equals(dm, vw)] ----------- (G-e.1.b)

Sub-Goal (G-e.1.a)

We show:

equals(s, t) --------------------- (G-e.1.a.1)
⟦while E do Cseq end⟧(em)(s, s')] --------------------- (G-e.1.a.2)

We define:

s := constructs(t) ----------------- (5.a)
s' := constructs(t') ----------------- (5.b)
inValue(False) := constructs(false) --------------------- (5.c)

Sub-Goal (G-e.1.a.1)

We instantiate lemma (L-cseq5) with
s as s, t as t
to get

s = constructs(t) ⇒ equals(s,t)

From assumption (5.a) and (L-cseq5), we know

equals(s,t)

which is the goal (G-e.1.a.1). Hence (G-e.1.a.1) is proved.

Sub-Goal (G-e.1.a.2)

We instantiate soundness statement of E with
em as em, expw as e1, ew as ew, ew' as ew'', dw as dw, dw' as dw'', tw as tw, tw' as tw''
and get

wellTyped(em, E) ∧ consistent(em, ew, dw, tw) ∧
<e1, ew'', dw'', tw''> = T⟦E⟧(em, ew, dw, tw)
⇒ [wellTyped(e1, ew'', dw'', tw'') ∧ extendsEnv(ew'', e1, ew) ∧ extendsDecl(dw'', e1, dw)
∧ extendsTheory(tw'', e1, tw) ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t', vw>

⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, vw)
]
]
]

From assumptions (1.a), (2), (3.a) and the soundness statement of E, we know

[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, vw)
]
]

We instantiate above formula with
t as t, t' as t', vw as false to get

<t, e1> ⟶ <t', false>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, vw)
]

From assumption (5) and above formula,

∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)

Taking s as s, s' as s', vm as inValue(False) with above formula, we know from (5.a), (5.b), (5.c) and (3.a) that

there is s, s', inValue(False) and E for which

⟦E⟧(em)(s,s',inValue(False)) --------------------- (G-e.1.a.2.1)

We instantiate lemma (L-c12) with
em as em, E as E, Cseq as Cseq, s as s and s' as s' to get

⟦E⟧(em)(s,s',inValue(False)) ⇒ ⟦while E do Cseq end⟧(em)(s,s')

The goal (G-e.1.a.2) follows from assumption (G-e.1.a.2.1) and lemma (L-c12).

Consequently, the goal (G-e.1.a) follows from (G-e.1.a.1) and (G-e.1.a.2). Hence (G-e.1.a) is proved.

Sub-Goal (G-e.1.b)

Let s, s', dm, t be arbitrary but fixed.

We assume:

equals(s,t) --------------------- (6)
⟦while E do Cseq end⟧(em)(s,s') --------------------- (7)
dm = infoData(s') --------------------- (8)

We show:

equals(s', t') --------------------- (G-e.1.b.1)
equals(dm, vw) --------------------- (G-e.1.b.2)

vw := constructs(dm) ------------------- (7.a)

Sub-Goal (G-e.1.b.1)

We instantiate lemma (L-cseq5) with
s as s' and t as t'
to get

s' = constructs(t') ⇒ equals(s', t')

From (5.b) and (L-cseq5), we know

equals(s', t') which is the goal (G-e.1.b.1). Hence proved.

Sub-Goal (G-e.1.b.2)

We instantiate lemma (L-cseq6) with
v as vw, v' as dm
to get

vw = constructs(dm) ⇒ equals(dm, vw)
From (7.a) and (L-cseq6), we know

equals(dm, vw)

which is the goal (G-e.1.b.2). Hence proved.

Consequently, the goal (G-e.1.b) follows from (G-e.1.b.1) and (G-e.1.b.2). Hence (G-e.1.b) is proved.

Finally, the goal (G-e.1) follows from goals (G-e.1.a) and (G-e.1.b).

Goal (G-e.2):

We assume:

<t, e1> ⟶ <t'', true> ------------------ (8)
<t'', e2> ⟶ <t''', void> ------------------ (9)
<t''', while e1 do e2> ⟶ <t', void> ------------------ (10)
ℙ(t''',t',void) ------------------ (11)

We show:

ℙ(t,t',vw)

By expanding the definition of ℙ(t,t',vw), we get

[∃ s, s' ∈ State: equals(s,t) ∧ ⟦while E do Cseq⟧(em)(s,s')] ----------- (G-e.2.a)
∧ [∀ s, s' ∈ State, dm ∈ InfoData: equals(s',t') ∧ ⟦while E do Cseq⟧(em)(s,s') ∧ dm=infoData(s')
⇒ equals(s',t') ∧ equals(dm, vw)] ----------- (G-e.2.b)

We define:

s := constructs(t) ----------------- (9.a)
s'' := constructs(t'') ----------------- (9.b)
s''' := constructs(t''') ----------------- (9.c)
inValue(True) := constructs(true) --------------------- (9.d)
inValue(Void) := constructs(void) --------------------- (9.e)

Sub-Goal (G-e.2.a)

We show:

equals(s, t) --------------------- (G-e.2.a.1)
⟦while E do Cseq end⟧(em)(s, s')] --------------------- (G-e.2.a.2)

Sub-Goal (G-e.2.a.1)

We instantiate lemma (L-cseq5) with
s as s, t as t
to get

s = constructs(t) ⇒ equals(s,t)

From assumption (9.a) and (L-cseq5), we know

equals(s,t)

which is the goal (G-e.2.a.1). Hence (G-e.2.a.1) is proved.
Sub-Goal (G-e.2.a.2)

We instantiate soundness statement of E with
em as em, expw as e1, ew as ew, ew' as ew'', dw as dw, dw' as dw'', tw as tw, tw' as tw''

and get

wellTyped(em, E) ∧ consistent(em, ew, dw, tw) ∧
<e1, ew'', dw'', tw''> = T⟦E⟧(em, ew, dw, tw)
⇒ [wellTyped(e1, ew'', dw'', tw'') ∧ extendsEnv(ew'', e1, ew) ∧ extendsDecl(dw'', e1, dw)
∧ extendsTheory(tw'', e1, tw) ∧
[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, vw)
]
]
]

From assumptions (1.a), (2), (3.a) and the soundness statement of E, we know

[∀ t, t' ∈ Statew, vw ∈ Valuew: <t, e1> ⟶ <t', vw>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, vw)
]
]

We instantiate above formula with
t as t, t' as t'', vw as true to get

<t, e1> ⟶ <t'', true>
⇒ [∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)]
∧
[∀ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)
⇒ equals(s', t') ∧ equals(vm, true)
]

From assumption (8), we know

∃ s, s' ∈ State, vm ∈ Value: equals(s, t) ∧ ⟦E⟧(em)(s, s', vm)

Taking s as s, s' as s'', vm as inValue(True) with above formula, we know from (9.a), (9.c), (9.d) and (3.a) that

there is s, s'', inValue(True) and E, for which

⟦E⟧(em)(s,s'',inValue(True)) -------------------- (G-e.2.a.2.1)

We instantiate the soundness statement of Cseq with em as em’, cw as e2, ew as ew’’, ew’ as ew’, dw as dw’’, dw’ as dw’, tw as tw’’, tw’ as tw’

to get

wellTyped(em’, Cseq) ∧ consistent(em’, ew’’, dw’’, tw’’) ∧
<e2, ew’, dw’, tw’> = T⟦Cseq⟧(em’, ew’’, dw’’, tw’’)
⇒ [wellTyped(e2, ew’, dw’, tw’) ∧ extendsEnv(ew’, e2, ew’’) ∧ extendsDecl(dw’, e2, dw’’)
∧ extendsTheory(tw’, e2, tw’’) ∧
[∀ t, t’ ∈ Statew, vw ∈ Valuew: <t, e2> ⟶ <t’, vw>
⇒ [∃ s, s’ ∈ State: equals(s, t) ∧ ⟦Cseq⟧(em’)(s, s’)]
∧
[∀ s, s’ ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq⟧(em’)(s, s’) ∧ dm = infoData(s’)
⇒ equals(s’, t’) ∧ equals(dm, vw)
]
]
]

From assumptions (1.b), (2a), (3.b) and soundness statement of Cseq, we know

[∀ t, t’ ∈ Statew, vw ∈ Valuew: <t, e2> ⟶ <t’, vw>
⇒ [∃ s, s’ ∈ State: equals(s, t) ∧ ⟦Cseq⟧(em’)(s, s’)]
∧
[∀ s, s’ ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq⟧(em’)(s, s’) ∧ dm = infoData(s’)
⇒ equals(s’, t’) ∧ equals(dm, vw)
]
]

We instantiate the above formula with t as t’’, t’ as t’’’, vw as void to get

[∀ t’’, t’’’ ∈ Statew, void ∈ Valuew: <t’’, e2> ⟶ <t’’’, void>
⇒ [∃ s, s’ ∈ State: equals(s, t) ∧ ⟦Cseq⟧(em’)(s, s’)]
∧
[∀ s, s’ ∈ State, dm ∈ InfoData: equals(s, t)
∧ ⟦Cseq⟧(em’)(s, s’) ∧ dm = infoData(s’)
⇒ equals(s’, t’) ∧ equals(dm, vw)
]
]

From assumption (9) and above formula we get

[∃ s, s’ ∈ State: equals(s, t) ∧ ⟦Cseq⟧(em’)(s, s’)]

Taking s as s’’, s’ as s’’’ in the above formula, we know from (9.b), (9.c), (1.a’) and (3.b) that there is s'', s''', em' and Cseq s.t.

⟦Cseq⟧(em')(s'',s''') ---------------- (G-e.2.a.2.2)

By expanding (11), we get

[∃ s, s' ∈ State: equals(s,t''') ∧ ⟦while E do Cseq⟧(em)(s,s')] ----------- (12)
∧
[∀ s, s' ∈ State, dm ∈ InfoData: equals(s',t') ∧ ⟦while E do Cseq⟧(em)(s,s')
∧ dm=infoData(s') ⇒ equals(s',t') ∧ equals(dm, void)] ----------- (13)

From (12), we know there is s, s'

equals(s,t''') -------------------------- (12.a)
⟦while E do Cseq⟧(em)(s,s') --------------------- (12.b)

We instantiate lemma (L-cseq5) with
s as s, t as t'''
to get

s = constructs(t''') <⇒ equals(s,t''')

From (12.a) and lemma (L-cseq5), we get

s = constructs(t''') --------------------- (12.c)

From (12.c) and (9.b), we can rewrite (12.a) and (12.b) as

equals(s''',t''') -------------------------- (12.a')
⟦while E do Cseq⟧(em)(s''',s') --------------------- (12.b')

We instantiate lemma (L-c13) with
em as em, em' as em', E as E, Cseq as Cseq, s as s, s' as s', s'' as s'', s''' as s''' to get

⟦E⟧(em)(s,s'',inValue(True)) ∧ em' = Env(em, E) ∧ ⟦Cseq⟧(em')(s'',s''')
∧ ⟦while E do Cseq end⟧(em)(s''',s')
⇒ ⟦while E do Cseq end⟧(em)(s,s')

The goal (G-e.2.a.2) follows from assumptions (G-e.2.a.2.1), (1.a'), (G-e.2.a.2.2), (12.b') and lemma (L-c13).
Consequently (G-e.2.a) follows from the proofs of (G-e.2.a.1) and (G-e.2.a.2).

Sub-Goal (G-e.2.b)

Let s, s', dm, t be arbitrary but fixed.

We assume:

equals(s,t) --------------------- (13)
⟦while E do Cseq end⟧(em)(s,s') --------------------- (14)
dm = infoData(s') --------------------- (15)

We show:

equals(s', t') --------------------- (G-e.2.b.1)
equals(dm, vw) --------------------- (G-e.2.b.2)

We define:

s' := constructs(t') ------------------- (14.a)
vw := constructs(dm) ------------------- (14.b)

Sub-Goal (G-e.2.b.1)

We instantiate lemma (L-cseq5) with
s as s' and t as t'
to get

s' = constructs(t') ⇔ equals(s', t')

From (14.a) and (L-cseq5), we know

equals(s', t') which is the goal (G-e.2.b.1). Hence proved.

Sub-Goal (G-e.2.b.2)

We instantiate lemma (L-cseq6) with
v as vw, v' as dm
to get

vw = constructs(dm) ⇒ equals(dm, vw)
From (12.b) and (L-cseq6), we know

equals(dm, vw)

which is the goal (G-e.2.b.2). Hence proved.

Consequently, the goal (G-e.2.b) follows from (G-e.2.b.1) and (G-e.2.b.2).
Finally, the goal (e) follows from the proofs of goals (G-e.a) and (G-e.b).

Also the goal (G23) follows from the proofs of goals (a), (b), (c), (d) and (e).
Hence (G23) proved.

E Lemmas

E.1 For Command_Sequence

:

Lemma cseq1:

$\forall$ cseq $\in$ Command_Sequence, em $\in$ Environment, e $\in$ Expressionw, ew, ew' $\in$ Environmentw, dw, dw' $\in$ Declw, tw, tw' $\in$ Theoryw:

wellTyped(em, cseq) $\wedge$ (e, ew', dw', tw') = T$[\![$cseq$]\!]$(em, ew, dw, tw)
$\Rightarrow$ wellTyped(e, ew', dw', tw') ------------------------- (L-cseq1)

Lemma cseq2:

$\forall$ em $\in$ Environment, C $\in$ Command, Cseq $\in$ Command_Sequence,
ew, ew', ew'' $\in$ Environmentw, e1, e2 $\in$ Expressionw, dw, dw', dw'' $\in$ Declw, tw, tw', tw'' $\in$ Theoryw:
wellTyped(em, C;Cseq) $\wedge$ (e1;e2, ew', dw', tw') = T$[\![$C;Cseq$]\!]$(em, ew, dw, tw)
$\Rightarrow$
[extendsEnv(ew'', e1, ew) $\wedge$ extendsEnv(ew', e2, ew'') $\Rightarrow$ extendsEnv(ew', e1;e2, ew)] $\wedge$
[extendsDecl(dw'', e1, dw) $\wedge$ extendsDecl(dw', e2, dw'') $\Rightarrow$ extendsDecl(dw', e1;e2, dw)] $\wedge$
[extendsTheory(tw'', e1, tw) $\wedge$ extendsTheory(tw', e2, tw'') $\Rightarrow$ extendsTheory(tw', e1;e2, tw)]
------------------------- (L-cseq2)

Lemma cseq3:

$\forall$ em, em' $\in$ Environment, C $\in$ Command, Cseq $\in$ Command_Sequence:
wellTyped(em, C;Cseq) $\Rightarrow$ wellTyped(em, C) $\wedge$ em' = Env(em, C) $\wedge$ wellTyped(em', Cseq)
------------------------- (L-cseq3)

Lemma cseq4:

$\forall$ em, em' $\in$ Environment, C $\in$ Command, Cseq $\in$ Command_Sequence,
ew, ew', ew'' $\in$ Environmentw, e1, e2 $\in$ Expressionw, dw, dw', dw'' $\in$ Declw, tw, tw', tw'' $\in$ Theoryw:
(e1, ew'', dw'', tw'') = T$[\![$C$]\!]$(em, ew, dw, tw) $\wedge$ em' = Env(em, C) $\wedge$
(e2, ew', dw', tw') = T$[\![$Cseq$]\!]$(em', ew'', dw'', tw'') $\wedge$ consistent(em, ew, dw, tw)
$\Rightarrow$ consistent(em', dw'', dw'', tw'') ------------------------- (L-cseq4)

Lemma cseq5:

$\forall$ s $\in$ State, t $\in$ Statew: s = constructs(t) $<\Rightarrow$ equals(s,t) ------------------------- (L-cseq5)

Lemma cseq6:

∀ v ∈ Valuew, v' ∈ InfoData: v' = constructs(v) <⇒ equals(v',v) -----------------------(L-cseq6)

E.2 For Command

:

Lemma c1:

∀ c ∈ Command, em ∈ Environment, e ∈ Expressionw, ew, ew' ∈ Environmentw, dw, dw' ∈ Declw, tw, tw' ∈ Theoryw:

wellTyped(em, c) ∧ (e, ew', dw', tw') = T⟦c⟧(em, ew, dw, tw)
⇒ wellTyped(e, ew', dw', tw') ------------------------- (L-c1)

Lemma c2:

∀ em ∈ Environment, C ∈ Command, Cseq ∈ Command_Sequence,
ew, ew', ew'', ew''' ∈ Environmentw, e1, e2, e3 ∈ Expressionw, dw, dw', dw'', dw''' ∈ Declw, tw, tw', tw'', tw''' ∈ Theoryw:

wellTyped(em, if E then Cseq1 else Cseq2 end) ∧
<if e1 then e2 else e3, ew', dw', tw'> = T⟦if E then Cseq1 else Cseq2 end⟧(em, ew, dw, tw) ∧
<e1, ew''', dw''', tw'''> = T⟦E⟧(em, ew, dw, tw) ∧
em' = Env(em, E) ∧
<e2, ew'', dw'', tw''> = T⟦Cseq1⟧(em', ew''', dw''', tw''') ∧
<e3, ew', dw', tw'> = T⟦Cseq2⟧(em', ew'', dw'', tw'') ∧
⇒
[extendsEnv(ew''', e1, ew) ∧ extendsEnv(ew'', e2, ew''') ∧ extendsEnv(ew', e3, ew'')
⇒ extendsEnv(ew', if e1 then e2 else e3, ew)] ∧
[extendsDecl(dw''', e1, dw) ∧ extendsDecl(dw'', e2, dw''') ∧ extendsDecl(dw', e3, dw'')
⇒ extendsDecl(dw', if e1 then e2 else e3, dw)] ∧
[extendsTheory(tw''', e1, tw) ∧ extendsTheory(tw'', e2, tw''') ∧ extendsTheory(tw', e3, tw'')
⇒ extendsTheory(tw', if e1 then e2 else e3, tw)]
------------------------ (L-c2)

Lemma c3:

∀ em, em' ∈ Environment, E ∈ Expression Cseq1, Cseq2 ∈ Command_Sequence:
wellTyped(em, if E then Cseq1 else Cseq2 end) ⇒ wellTyped(em, E) ∧ em' = Env(em, E) ∧ wellTyped(em', Cseq1) ∧ wellTyped(em, Cseq2)
------------------------ (L-c3)

Lemma c4:

ment c∀ em, em' ∈ Environment, E ∈ Expression, Cseq1, Cseq2 ∈ Command_Sequence,

ew, ew', ew'', ew''' ∈ Environmentw, e1, e2, e3 ∈ Expressionw, dw, dw', dw'', dw''' ∈ Declw, tw, tw', tw'', tw''' ∈ Theoryw:

(e1, ew'', dw'', tw'') = T⟦E⟧(em, ew, dw, tw) ∧ em' = Env(em, E) ∧

(e2, ew''', dw''', tw''') = T⟦Cseq1⟧(em', ew'', dw'', tw'') ∧

(e3, ew', dw', tw') = T⟦Cseq2⟧(em, ew''', dw''', tw''') ∧ consistent(em, ew, dw, tw)

⇒ consistent(em', dw'', dw'', tw'') ∧ consistent(em, ew''', dw''', tw''')

------------------------- (L-c4)

Lemma c5:

∀ em, em' ∈ Environment, I ∈ Identifier, E ∈ Expression:
wellTyped(em, I:=E) ⇒ wellTyped(em, E) ∧ em' = Env(em, E) ∧ wellTyped(em', I)
------------------------- (L-c5)

Lemma c6:

∀ em, em' ∈ Environment, I ∈ Identifer, E ∈ Expression, ew, ew', ew'' ∈ Environmentw, x, e ∈ Expressionw, dw, dw', dw'' ∈ Declw, tw, tw', tw'' ∈ Theoryw:
(e, ew'', dw'', tw'') = T⟦E⟧(em, ew, dw, tw) ∧ em' = Env(em, E) ∧
(x, ew', dw', tw') = T⟦I⟧(em', ew'', dw'', tw'') ∧ consistent(em, ew, dw, tw)
⇒ consistent(em', dw'', dw'', tw'')
------------------------- (L-c6)

Lemma c7:

∀ em, em' ∈ Environment, I ∈ Identifier, E ∈ Expression, ew, ew', ew'' ∈ Environmentw, x, e ∈ Expressionw, dw, dw', dw'' ∈ Declw, tw, tw', tw'' ∈ Theoryw:

wellTyped(em, I:=E) ∧
<x:=e, ew', dw', tw'> = T⟦I:=E⟧(em, ew, dw, tw) ∧
<e, ew'', dw'', tw''> = T⟦E⟧(em, ew, dw, tw) ∧
em' = Env(em, E) ∧
<x, ew', dw', tw'> = T⟦I⟧(em', ew'', dw'', tw'')
⇒
[extendsEnv(ew'', e, ew) ∧ extendsEnv(ew', x, ew'') ⇒ extendsEnv(ew', x:=e, ew)] ∧
[extendsDecl(dw'', e, dw) ∧ extendsDecl(dw', x, dw'') ⇒ extendsDecl(dw', x:=e, dw)] ∧
[extendsTheory(tw'', e, tw) ∧ extendsTheory(tw', x, tw'') ⇒ extendsTheory(tw', x:=e, tw)]
------------------------- (L-c7)

Lemma c8:

∀ x ∈ Identifier, s', s'' ∈ State, t', t'' ∈ Statew, vw ∈ Valuew, vm ∈ Value:

s' = constructs(t') ∧ s'' = constructs(t'') ∧ t' = t''+[x|->vw] ∧ vm = constructs(vw)
⇒ s' = update(x, vm, s'') ------------------------- (L-c8)

Lemma c9:

∀ em, em' ∈ Environment, E ∈ Expression Cseq ∈ Command_Sequence:
wellTyped(em, while E do Cseq end)
⇒ wellTyped(em, E) ∧ em' = Env(em, E) ∧ wellTyped(em', Cseq) ------------------------- (L-c9)

Lemma c10:

∀ em, em' ∈ Environment, E ∈ Expression, Cseq ∈ Command_Sequence, ew, ew', ew'' ∈ Environmentw, e1, e2 ∈ Expressionw, dw, dw', dw'' ∈ Declw, tw, tw', tw'' ∈ Theoryw:
(e1, ew'', dw'', tw'') = T⟦E⟧(em, ew, dw, tw) ∧ em' = Env(em, E) ∧
(e2, ew''', dw''', tw''') = T⟦Cseq⟧(em', ew'', dw'', tw'') ∧ consistent(em, ew, dw, tw)
⇒ consistent(em', dw'', dw'', tw'') ------------------------- (L-c10)

Lemma c11:

∀ em ∈ Environment, E ∈ Expression, Cseq ∈ Command_Sequence, ew, ew', ew'' ∈ Environmentw, e1, e2 ∈ Expressionw, dw, dw', dw'' ∈ Declw, tw, tw', tw'' ∈ Theoryw:

wellTyped(em, while E do Cseq end) ∧
<while e1 do e2, ew', dw', tw'> = T⟦while E do Cseq end⟧(em, ew, dw, tw) ∧
<e1, ew'', dw'', tw''> = T⟦E⟧(em, ew, dw, tw) ∧
em' = Env(em, E) ∧
<e2, ew', dw', tw'> = T⟦Cseq⟧(em', ew'', dw'', tw'')
⇒
[extendsEnv(ew'', e1, ew) ∧ extendsEnv(ew', e2, ew'')
⇒ extendsEnv(ew', while e1 do e2, ew)] ∧
[extendsDecl(dw'', e1, dw) ∧ extendsDecl(dw', e2, dw'')
⇒ extendsDecl(dw', while e1 do e2, dw)] ∧
[extendsTheory(tw'', e1, tw) ∧ extendsTheory(tw', e2, tw'')
⇒ extendsTheory(tw', while e1 do e2, tw)]
------------------------- (L-c11)

Lemma c12:

∀ em ∈ Environment, E ∈ Expression, Cseq ∈ Command_Sequence, s,s' ∈ State:

⟦E⟧(em)(s,s',inValue(False)) ⇒ ⟦while E do Cseq end⟧(em)(s,s') ------------------------- (L-c12)

Lemma c13:

∀ em, em' ∈ Environment, E ∈ Expression, Cseq ∈ Command_Sequence, s,s',s'',s''' ∈ State:

⟦E⟧(em)(s,s'',inValue(True)) ∧ em' = Env(em, E) ∧ ⟦Cseq⟧(em')(s'',s''')
∧ ⟦while E do Cseq end⟧(em)(s''',s')
⇒ ⟦while E do Cseq end⟧(em)(s,s') ------------------------- (L-c13)

E.3 For Expression

:

Lemma e1:

$\forall$ E $\in$ Expression, em $\in$ Environment, e $\in$ Expressionw, ew, ew' $\in$ Environmentw, dw, dw' $\in$ Declw, tw, tw' $\in$ Theoryw:

wellTyped(em, E) $\wedge$ (e, ew', dw', tw') = T⟦E⟧(em, ew, dw, tw)
$\Rightarrow$ wellTyped(e, ew', dw', tw') -------------------------- (L-e1)

E.4 Auxiliary Lemmas

:

Lemma a1:

Suppose,
there exists a derivation of
<t''', try loop if e1 then e2 else raise Exit with Exit _ → void end> ⟶ <t', void> ---------- (a)

then there exists a derivation of
<t''', loop if e1 then e2 else raise Exit> ⟶ <t', Exit c> ---------------- (G)

Given (a), we can derive (G) only by one rule (try-1). ------------------ (L-a1)

Proof:

As we have three rules that can be applied to (a), so we prove by case analysis on these rules.

Case 1: rule (try-1)

From rule (try-1), we know that

(a) holds only if derivations of

<t''', loop if e1 then e2 else raise Exit> ⟶ <t', Exit c>
<t', void> ⟶ <t', void>

holds. Thus (G) can directly be obtained as above.

Case 2: rule (try-2)

It can also not be used to derive (G). We prove here by induction on number of iterations.
Suppose n ∈ N is the number of loop iteration:
We start for 0 iteration, when **n = 0**

By the application of rule (try-2), we know that

(a) holds only if derivation of

<t''', loop if e1 then e2 else raise Exit> ⟶ <t', void>

holds, which is not (G).

Now suppose, for iteration **n = n-1**$^{\mathbf{th}}$, by the application of rule (try-2), we know that

(a) holds only if derivation of

<t''', loop if e1 then e2 else raise Exit> ⟶ <t_n-1, void> for some t_n-1

holds, which is again not the same as (G).

Now assume the rule application above for n = n-1, we prove it does not hold for **n = n**. Now at nth iteration, by the application of rule (try-2), we know that

(a) holds only if derivation of

<t''', loop if e1 then e2 else raise Exit> ⟶ <t_n, void> for some t_n

holds, which is different than (G).

As we saw by induction above that (G) cannot be derived by rule (try-2). Hence rule (try-2) is also not applicable.

Case 3: rule (try-3)

(G) can clearly not be derived by rule (try-3) as this rule has conclusion, whose derivation has the consequence with non-exception value, i.e. <t', E' c>, while our assumption has non-exception value, i.e. <t', void>.

Hence, we have proved that the only possible derivation of (G) from (1) is by rule (try-1).

F Definitions

Definition 1:

(cw, ew’, dw’, tw’) = T⟦cseq⟧(em, ew, dw, tw) ------------------- (D1)

where
ew’ = extends(ew, cw)
dw’ = extends(dw, cw)
tw’ = extends(tw, cw)

Definition 2:

(e1;e2, ew’, dw’, tw’) = T⟦c;cseq⟧(em, ew, dw, tw) ------------------- (D2)

where

(e1, ew’’, dw’’, tw’’) = T⟦c⟧(em, ew, dw, tw)
em’ = Env(em, C)
(e2, ew’, dw’, tw’) = T⟦cseq⟧(em’, ew’’, dw’’, tw’’)

and

ew’’ = extends(ew, e1)
ew’ = extends(ew’’, e2)
dw’’ = extends(dw, e1)
dw’ = extends(dw’’, e2)
tw’’ = extends(tw, e1)
tw’ = extends(tw’’, e2)

and e1;e2 is a syntactic sugar for let _ = e1 in e2

Definition 3:

<t, e1> ⟶ <t’’, vw’>, vw’ is not exception <t’’, e2> ⟶ <t’, vw>

<t, e1;e2> ⟶ <t’, vw>
------------------- (D3)

where e1;e2 is a syntactic sugar for let _ = e1 in e2

Definition 4:

(if e1 then e2 else e3, ew’, dw’, tw’) = T⟦**if** E **then** Cseq1 **else** Cseq2 **end**⟧(em, ew, dw, tw) ------------------- (D4)

where
(e1, ew’’’, dw’’’, tw’’’) = T⟦E⟧(em, ew, dw, tw)
(e2, ew’’, dw’’, tw’’) = T⟦Cseq1⟧(em, ew’’’, dw’’’, tw’’’)
(e3, ew’, dw’, tw’) = T⟦Cseq2⟧(em, ew’’, dw’’, tw’’)

and
ew''' = extends(ew, e1)
ew'' = extends(ew''', e2)
ew' = extends(ew'', e3)
dw''' = extends(dw, e1)
dw'' = extends(dw''', e2)
dw' = extends(dw'', e3)
tw''' = extends(tw, e1)
tw'' = extends(tw''', e2)
tw' = extends(tw'', e3)

Definition 5:

```
⟦if E then Cseq Elif end if⟧(e)(s,s') ⇔
∃ v ∈ ValueU, s'' ∈ StateU: ⟦E⟧(e)(s,s'',v) AND
cases v of
  isUndefined() → s' = inError()
  [] isValue(v1)
    → cases s'' of
      isError() → s' = inError()
      [] isState(p)
        →cases v1 of
          isBoolean(v2) → IF v2 THEN ⟦Cseq⟧(e)(p,s')
            ELSE ∃ v' ∈ Tr, p' ∈ StateU: ⟦Elif⟧(e)(s,p',v') AND
            cases p' of
              isError() → s' = inError()
              [] isState(p'')→ IF v'=inTr(True) THEN
                    s' = inStateU(p'')
                    ELSE s' = s
                    END
            END
            END
        END
    END
END                   ------------------------ (D5)
```

Definition 6:

```
<t, e1> ⟶ <t'', true>        <t'', e2> ⟶ <t', vw>
--------------------------------------------------------------------------------
      <t, if e1 then e2 else e3> ⟶ <t', vw>
              ------------------- (D6)
```

Definition 7:

```
<t, e1> ⟶ <t'', false>        <t'', e3> ⟶ <t', vw>
--------------------------------------------------------------------------------
      <t, if e1 then e2 else e3> ⟶ <t', vw>
              ------------------- (D7)
```

Definition 8:
// while loop iterator ...
iterate ⊂ Nat x StateU* x StateU* x Environment x StateValueRelation x StateRelation
iterate(i, t, u, e, E, C) ⇔
cases t(i) of
isError() → false
[] isState(m) → executes(data(m)) AND ∃ v ∈ ValueU, s' ∈ StateU : E(e)(m,s',v) AND
cases s' of
isError() → u(i+1)=inError() AND t(i+1)=u(i+1)
[] isState(p) →
cases v of
isUndefined() → u(i+1)=inError() AND t(i+1)=u(i+1)
[] isValue(v') → cases v' of
isBoolean(b) → b AND LET e'=Env(e,E) IN
C(e')(p,u(i+1)) AND t(i+1)=u(i+1)
[] ... → u(i+1)=inError() AND t(i+1)=u(i+1)
END //cases-v'
END //cases-v
END //cases-s'
END //cases-t(i)

Definition 9:

⟦**while** E **do** Cseq **end do** ⟧(e)(s,s') ⇔
∃ k ∈ Nat, t, u ∈ StateU*:
t(0)=inStataU(s) AND u(0)=inStateU(s) AND
(∀ i ∈ Nat_k: iterate(i, t, u, e, ⟦E⟧, ⟦Cseq⟧)) AND
((u(k)=inError() AND s'=u(k)) OR (returns(data(inState(u(k)))) AND s'=t(k)) OR
(∃ v ∈ ValueU: ⟦E⟧(e)(inState(t(k)), u(k), v)
AND v <> inValue(inBoolean(True)) AND
IF v = inValue(inBoolean(False)) THEN
s'=u(k)
ELSE s' = inError() END //if-v
)
)

G Why3 Semantics

<t, e1> ⟶ <t'', vw'>, vw' is not exception <t'', e2> ⟶ <t', vw>
--- ------------------- (com-s)
<t, e1;e2> ⟶ <t', vw>

----------------------------------- ------------------- (const)

```
  <t, c> ⟶ <t, c>


<t, e1> ⟶ <t'', true>        <t'', e2> ⟶ <t', vw>
------------------------------------------------------------------------------    ----------------
--- (cond-t)
      <t, if e1 then e2 else e3> ⟶ <t', vw>



<t, e1> ⟶ <t'', false>        <t'', e3> ⟶ <t', vw>
------------------------------------------------------------------------------    ----------------
--- (cond-f)
      <t, if e1 then e2 else e3> ⟶ <t', vw>


<t, e> ⟶ <t'', void>        <t'', loop e> ⟶ <t', vw>
------------------------------------------------------------------------------    ----------------
--- (loop-n)
      <t, loop e> ⟶ <t', vw>

<t, e> ⟶ <t', E c>
------------------------------------    ------------------- (loop-e)
  <t, loop e> ⟶ <t', E c>

<t, e> ⟶ <t', c>
------------------------------------    ------------------- (raise)
  <t, raise (E e)> ⟶ <t', E c>

<t, e1> ⟶ <t'', E c>        <t'', e2[x<--c]> ⟶ <t', vw>
------------------------------------------------------------------------------    ----------------
--- (try-1)
      <t, try e1 with E x → e2 end> ⟶ <t', vw>

      <t, e1> ⟶ <t', vw> vw is not exc.
------------------------------------------------------------------------------    ----------------
--- (try-2)
      <t, try e1 with E x → e2 end> ⟶ <t', vw>

      <t, e1> ⟶ <t', E' c>  E' <> E
------------------------------------------------------------------------------    ----------------
--- (try-3)
      <t, try e1 with E x → e2 end> ⟶ <t', E' c>
```

H Derivations

From the semantics of Why3, we know that the while-loop "while e1 do e2" is a syntactic sugar, which is semantically equivalent to as follows

while e1 do e2 ˜ try
loop if e1 then e2 else raise Exit
with Exit _ → void end ------------- (4'')
and

raise Exit ˜ raise Exit _ ------------------------ (4''')

Now we introduce two new rules for while-loop (d.a) and (d.b), which operates directly on the level of while-loop (without expansion). In the following, we show that these rules follows from the basic rule calculus, i.e. adding these rules does not change the semantics.

Derivation 1:

-- //applying (const)
<t', _> ⟶ <t', c> , where c=_
-- //applying (raise)
<t, e1> ⟶ <t', false> <t', raise Exit> ⟶ <t', Exit c>
--- //applying (cond-f)
<t, if e1 then e2 else raise Exit> ⟶ <t', Exit c>
--- //applying (loop-e)
<t, loop if e1 then e2 else raise Exit> ⟶ <t', Exit c>

-- //applying (const)
<t', void> ⟶ <t', void>
-- //rewriting
<t', void[_ ← c]> ⟶ <t', void>
-- //applying (try-1)
<t, try loop if e1 then e2 else raise Exit with Exit _ → void end> ⟶ <t', void>

The above derivation is only possible if following holds:

<t, e1> ⟶ <t', false>

<t, try loop if e1 then e2 else raise Exit with Exit _ → void end> ⟶ <t', void>

-------------------- (d1)
From (4'') and (d1), we get

<t, e1> ⟶ <t', false>

-- -------------------- (d.a)
<t, while e1 do e2> ⟶ <t’, void>

Derivation 2:

<t, e1> ⟶ <t”, true> <t”, e2> ⟶ <t”’, void>
-- //applying (cond-t)
<t, if e1 then e2 else raise Exit> ⟶ <t”’, void>

<t”’, loop if e1 then e2 else raise Exit _> ⟶ <t’, Exit c>
--- //applying (loop-n)
<t, loop if e1 then e2 else raise Exit> ⟶ <t’, Exit c>

-- //applying (const)
<t’, void> ⟶ <t’, void>
-- //rewriting
<t’, void[_ ← c]> ⟶ <t’, void>
-- //applying (try-1)
<t, try loop if e1 then e2 else raise Exit with Exit _ → void end> ⟶ <t’, void>

The above derivation is only possible when following holds:

<t, e1> ⟶ <t”, true> <t”, e2> ⟶ <t”’, void>
<t”’, loop if e1 then e2 else raise Exit> ⟶ <t’, Exit c>

<t, try loop if e1 then e2 else raise Exit with Exit _ → void end> ⟶ <t’, void>

-------------------- (d2)

Based on derivation (d2), we need the following derivation to conform to the rule based definition of while-loop semantics:

<t, e1> ⟶ <t”, true> <t”, e2> ⟶ <t”’, void>
<t”’, try loop if e1 then e2 else raise Exit with Exit _ → void end> ⟶ <t’, void>

<t, try loop if e1 then e2 else raise Exit with Exit _ → void end> ⟶ <t’, void>

-------------------- (d3)

From (4”), (d3) can be rewritten as follows:

<t, e1> ⟶ <t”, true> <t”, e2> ⟶ <t”’, void> <t”’, while e1 do e2> ⟶ <t’, void>

<t, while e1 do e2> ⟶ <t’, void>

-------------------- (d.b)

In order to get rule (d3) from (d2), we need to show that if there exists a derivation of

<t’’’, try loop if e1 then e2 else raise Exit with Exit _ → void end> ⟶ <t’, void> -------------- (p.1)

then there also exists a corresponding derivation of

<t’’’, loop if e1 then e2 else raise Exit> ⟶ <t’, Exit c> ---------------- (p.2)

Because, we want to write (d2) instead of (d3) because (d3) respectively (d.b) is a direct definition of while-loop operational semantics.

Proof:

The goal (p.2) follows from (p.1) and lemma (L-a1). Based on (d2) and derivation of (p.2), we get (d3). Hence (d2) can be derived from (d3), where (d3) can be rewritten to (d.b).

Technical Reports of the Doctoral Program

"Computational Mathematics"

2014

2014-01 E. Pilgerstorfer, B. Jüttler: *Bounding the Influence of Domain Parameterization and Knot Spacing on Numerical Stability in Isogeometric Analysis* February 2014. Eds.: B. Jüttler, P. Paule

2014-02 T. Takacs, B. Jüttler, O. Scherzer: *Derivatives of Isogeometric Functions on Rational Patches* February 2014. Eds.: B. Jüttler, P. Paule

2014-03 M.T. Khan: *On the Soundness of the Translation of MiniMaple to Why3ML* February 2014. Eds.: W. Schreiner, F. Winkler

2013

2013-01 U. Langer, M. Wolfmayr: *Multiharmonic Finite Element Analysis of a Time-Periodic Parabolic Optimal Control Problem* January 2013. Eds.: W. Zulehner, R. Ramlau

2013-02 M.T. Khan: *Translation of* MiniMaple *to Why3ML* February 2013. Eds.: W. Schreiner, F. Winkler

2013-03 J. Kraus, M. Wolfmayr: *On the robustness and optimality of algebraic multilevel methods for reaction-diffusion type problems* March 2013. Eds.: U. Langer, V. Pillwein

2013-04 H. Rahkooy, Z. Zafeirakopoulos: *On Computing Elimination Ideals Using Resultants with Applications to Gröbner Bases* May 2013. Eds.: B. Buchberger, M. Kauers

2013-05 G. Grasegger: *A procedure for solving autonomous AODEs* June 2013. Eds.: F. Winkler, M. Kauers

2013-06 M.T. Khan *On the Formal Verification of Maple Programs* June 2013. Eds.: W. Schreiner, F. Winkler

2013-07 P. Gangl, U. Langer: *Topology Optimization of Electric Machines based on Topological Sensitivity Analysis* August 2013. Eds.: R. Ramlau, V. Pillwein

2013-08 D. Gerth, R. Ramlau: *A stochastic convergence analysis for Tikhonov regularization with sparsity constraints* October 2013. Eds.: U. Langer, W. Zulehner

2013-09 W. Krendl, V. Simoncini, W. Zulehner: *Efficient preconditioning for an optimal control problem with the time-periodic Stokes equations* November 2013. Eds.: U. Langer, V. Pillwein

The complete list since 2009 can be found at
`https://www.dk-compmath.jku.at/publications/`

Doctoral Program
"Computational Mathematics"

Director:

Prof. Dr. Peter Paule
Research Institute for Symbolic Computation

Deputy Director:

Prof. Dr. Bert Jüttler
Institute of Applied Geometry

Address:

Johannes Kepler University Linz
Doctoral Program "Computational Mathematics"
Altenbergerstr. 69
A-4040 Linz
Austria
Tel.: ++43 732-2468-6840

E-Mail:

office@dk-compmath.jku.at

Homepage:

http://www.dk-compmath.jku.at

Submissions to the DK-Report Series are sent to two members of the Editorial Board who communicate their decision to the Managing Editor.