Mechanized Operational Semantics of WSL

Xingyuan Zhang and Malcolm Munro

University of Durham
South Road, Durhram
DH1 3LE, U.K.

Mark Harman and Lin Hu

Brunel University,
Uxbridge, Middlesex
UB8 3PH, U.K.

Abstract

This paper presents an experiment on computer assisted formal verification of program transformations. The operational semantics of WSL is formalized in the type theoretical proof assistant Coq, which forms the basis, on which the correctness of program transformations can be stated and proved as formulæ in Coq. A group of program transformations frequently used for software maintenance have been proved correct. The existence of a machine checked formal verification increases significantly the confidence in the correctness of program transformations, which is crucial for the reliability of software maintenance systems.

Keywords: Program Transformation, Computer Assisted Formal Reasoning

1 Introduction

The formal semantics of programming languages forms a foundation on which program transformation can be investigated and proved correct rigorously. However, as observed by DeMillo, Lipton and Perlis [1], such proofs typically consist of lengthy and tedious proofs, containing many detailed cases. This makes the proofs difficult to read and check for a human. Often the proofs are less interesting, less appealing and less elegant than the proofs found in mathematics and so there is an additional 'social' discouragement to the wider community to perform the crucial proof-checking role.

It has also been argued [2] that the very idea of program verification is flawed, partly because of the tremendous cost involved in generating and checking proofs.

In this paper we adopt an approach to formal verification of program transformation which addresses both of these two concerns:

Cost

While it is true that verification may be expensive, the cost of verification is more likely to be balanced by a suitable benefit in the case of program transformations than it is with program proving. This is because a program transformation is essentially, a meta-program, which is applied repeatedly to many programs to produce new versions. A bug in a program transformation therefore has potentially far-reaching consequences.

Checkability

It is true that computer science proofs can be long and tedious to read. Proofs of transformations are likely to be more interesting (because of their meta level) than proofs of individual programs to which the transformation may be applied. However, it remains the case that the community has insufficiently many trained individuals with the time, inclination and expertise required to check all the proof that the transformation developer community might be likely to produce.

Fortunately, recent work on proof assistants such as HOL [3, 4], Isabelle [5, 6], PVS [7, 8], LEGO [9, 10], ALF [11, 12] and Coq [13, 14, 15], has opened the possibility for machine-checked proofs, in which the human discharges the proof obligation, but the proof is checked by a machine. This paper shows an example how the mechanical proof assistant Coq is used as a tool to support program transformation proofs.

This paper establishes a foundation for further work on program transformation using WSL under-pinned by correctness proofs expressed in Coq and mechanically checked. Our goal is to establish this foundation by providing a semantic description of WSL and useful lemmas for program transformation. Space only allows us to present the full proofs of two lemmas ('`while` unrolling' and '`do` expansion'). More details and additional lemmas can be found on the website at `http://www.dur.ac.uk/xingyuan.zhang/wslcoq/`.

This paper makes the following contributions:

- An operational semantics of WSL is presented. This can be used to construct proofs about transformations. Because the semantics is defined within the constraints of constructive type theory, it is expressible as a Coq script and can therefore form the basis of automatically checked proofs. The full ASCII Coq script corresponding to the semantics presented here is available on the web.
- The paper illustrates the way in which transformations can be defined and proved correct with respect to the operational semantics.
- A number of lemmas, useful in program transformation, are defined. The website contains proofs of these.

All transformations have been defined as Coq scripts and their proofs have been mechanically checked using the Coq proof assistant.

The rest of this paper is organized as follows: Section 2 presents the syntax of WSL. Section 3 gives the operational semantics of WSL. Section 4 presents the correctness criterion for program transformations in terms of the operational semantics defined in Section 3. A number of useful program transformations from Ward's thesis [16] have been proved correct using such a criterion. Unfortunately, space only allows us to present two of them in full. Section 5 presents these two proofs. The purpose is to illustrate the approach. We do not claim that these proofs are useful, in themselves; they are well-known, after all. Rather, we claim that the approach is useful and believe that this paper serves as an introduction to and overview of the website resources built on top of the operational semantics set out in Section 3. Section 6 concludes. Some auxiliary definitions are given in the Appendix.

In the remainder of the paper, free variables in formulæ are assumed to be universally quantified, for example, $n_1 + n_2 = n_2 + n_1$ is an abbreviation of $\forall n_1, n_2.\ n_1 + n_2 = n_2 + n_1$.

2 Syntax definitions

The abstract syntax of WSL is given in Figure 1, where $\mathcal{C}$ is the type for statements of WSL and $\mathcal{IT}$ is the the type of initial statements. The type of expressions is formalized as an abstract type $\mathcal{E}$. The evaluation of expressions is formalized as the operation $[\![e]\!]_s$, where the expression $[\![e]\!]_s = v$ means that expression e evaluate to value v under program store s and the expression $[\![e]\!]_s = \bot$ means there is no valuation of expression e under program store s.

The representation of multi-way selection statement needs to be explained further. A selection statement:

$$\mathsf{sel}\ e_1 \to c_1\ [\!]\ e_2 \to c_2\ [\!]\ \ldots\ [\!]\ e_n \to c_n\ \mathsf{les}$$

in concrete syntax is represented using functional lists (see Appendix A.3 for the definition of functional list):

$$\mathsf{sel}\ \langle\, (e_1 \diamond e_2 \diamond \cdots \diamond e_n \diamond \ominus_{\overline{\mathsf{false}}}),\ (c_1 \diamond c_2 \diamond \cdots \diamond c_n \diamond \ominus_{\mathsf{abort}})\,\rangle\ \mathsf{les}$$

in abstract syntax, with the f_e in **s_sel** corresponding to $(e_1 \diamond e_2 \diamond \cdots \diamond e_n \diamond \ominus_{\overline{\mathsf{false}}})$, and f_m corresponding to $(c_1 \diamond c_2 \diamond \cdots \diamond c_n \diamond \ominus_{\mathsf{abort}})$. Such a representation is chosen, instead of the more 'natural' representation:

$$\frac{bd : [\![\mathcal{E} \times \mathcal{C}]\!]}{\mathsf{sel}\ bd\ \mathsf{les}\ : \mathcal{C}}\ \textbf{s\_sel\_natural} \qquad (1)$$

to avoid the non-positive occurrence of $\mathcal{C}$ in **s_sel_natural**. For similar reason, an action statement:

$$\mathsf{action}\ i:\ \{\ \ i_1 \equiv\ c_1\ [\!]\ i_2 \equiv c_2\ [\!]\ \ldots\ [\!]\ i_n \equiv\ c_n\ \}$$

in surface syntax is internally represented as:

$$\begin{array}{l}\mathsf{action}\ i:\ \{\\ \quad \langle\, (i_1 \diamond i_2 \diamond \cdots \diamond i_n \diamond \ominus_{\mathsf{za}}),\ (c_1 \diamond c_2 \diamond \cdots \diamond c_n \diamond \ominus_{\mathsf{abort}})\,\rangle\ \}\end{array}$$

in abstract syntax, where za is a reserved identifier which does not equal any identifer in user programs.

3 Operational semantics

The operational semantics of WSL is given in Figure 2, where the expression:

$$\left| s \xrightarrow[{[cs,dd,acs]}]{c} s' \right|_{er}$$

means the execution of the WSL statement c transforms program store from s to s'. Program store is represented as a finite mapping from variable names to values. The definition of finite mapping is given in Appendix A.4.

For example, the rule **e_asgn** concludes that the execution of $i := e$ transforms s to $s\left[v^{\,i}\right]$, the program store obtained from s by setting the value of i to v. And this transformation is under the premise $[\![e]\!]_s = v$, which stipulates that e evaluates to v under s.

The cs is a 'call stack', the type of which is $[\![\mathcal{ID} \times \mathcal{C}]\!]$. The use of cs is manifested in **e_proc** and **e_pcall**. As indicated by the $(pn, c) \centerdot cs$ in the premise of **e_proc**, when $\mathsf{proc}\ pn\ \equiv\ c$ is executed, a pair (pn, c) is added to head of the call stack cs. The pairs added to cs

$$\frac{}{\mathsf{abort} : \mathcal{C}}\ \textbf{s\_abort} \quad \frac{}{\mathsf{skip} : \mathcal{C}}\ \textbf{s\_skip} \quad \frac{e : \mathcal{E}}{\mathsf{assert}(e) : \mathcal{C}}\ \textbf{s\_asrt} \quad \frac{i : \mathcal{ID} \quad e : \mathcal{E}}{i := e : \mathcal{C}}\ \textbf{s\_asgn}$$

$$\frac{c_1 : \mathcal{C} \quad c_2 : \mathcal{C}}{c_1;\ c_2 : \mathcal{C}}\ \textbf{s\_seq} \quad \frac{e : \mathcal{E} \quad c_1 : \mathcal{C} \quad c_2 : \mathcal{C}}{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi} : \mathcal{C}}\ \textbf{s\_ifs} \quad \frac{its : [\![\mathcal{IT}]\!] \quad c : \mathcal{C}}{\{\langle its \rangle\ c\} : \mathcal{C}}\ \textbf{s\_local}$$

$$\frac{f\_e : \mathcal{FL}(\mathcal{E}) \quad f\_m : \mathcal{FL}(\mathcal{C})}{\mathsf{sel}\ \langle f\_e,\ f\_m \rangle\ \mathsf{les}\ : \mathcal{C}}\ \textbf{s\_sel} \quad \frac{e : \mathcal{E} \quad c : \mathcal{C}}{\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}\ : \mathcal{C}}\ \textbf{s\_while}$$

$$\frac{i : \mathcal{ID} \quad c : \mathcal{C}}{\mathsf{proc}\ i\ \equiv\ c : \mathcal{C}}\ \textbf{s\_proc} \quad \frac{i : \mathcal{ID}}{\mathsf{pcall}\ i\ : \mathcal{C}}\ \textbf{s\_pcall} \quad \frac{c : \mathcal{C}}{\mathsf{do}\ c\ \mathsf{od} : \mathcal{C}}\ \textbf{s\_do}$$

$$\frac{k : \mathcal{N}at}{\mathsf{exit}\ k\ : \mathcal{C}}\ \textbf{s\_exit} \quad \frac{i : \mathcal{ID} \quad f\_i : \mathcal{FL}(\mathcal{ID}) \quad f\_m : \mathcal{FL}(\mathcal{C})}{\mathsf{action}\ i :\ \{\ \langle f\_i,\ f\_m \rangle\} : \mathcal{C}}\ \textbf{s\_action}$$

$$\frac{i : \mathcal{ID}}{\mathsf{acall}\ i\ : \mathcal{C}}\ \textbf{s\_acall} \quad \frac{}{\mathsf{zaction} : \mathcal{C}}\ \textbf{s\_zaction} \quad \frac{i : \mathcal{ID} \quad e : \mathcal{E}}{i \leftarrow e : \mathcal{IT}}\ \textbf{init\_e} \quad \frac{i : \mathcal{ID}}{i \leftarrow \perp : \mathcal{IT}}\ \textbf{init\_n}$$

Figure 1: The syntax of WSL

by **e_proc** are used in **e_pcall**, where the premise $\mathsf{lookup}(pn, cs) = (c, \widetilde{cs})$ stipulates that the body of procedure pn is c and $\mathsf{proc}\ pn\ \equiv\ c$ was started under the call stack $\widetilde{cs}$. The definition of lookup is:

$$\begin{cases} \mathsf{lookup}(pn, (\widetilde{pn}, c) \centerdot cs) \stackrel{\text{def}}{\Longrightarrow} (c, cs) & \text{if } pn = \widetilde{pn} \\ \mathsf{lookup}(pn, (\widetilde{pn}, c) \centerdot cs) \stackrel{\text{def}}{\Longrightarrow} \mathsf{lookup}(pn, cs) & \text{if } pn \neq \widetilde{pn} \\ \mathsf{lookup}(pn, \langle\rangle) \stackrel{\text{def}}{\Longrightarrow} \perp & \end{cases} \tag{2}$$

The dd is called 'do depth', which is a natural number used to record the current nested depth of do statements. The dd is used by the rules **e_do_ag**, **e_do_tm**, **e_do_o**, **e_do_z** and **e_exit**. A common character of **e_do_ag**, **e_do_tm**, **e_do_o** and **e_do_z** is that: as indicated by the $dd+1$ in their premises, dd is incremented by 1 on the entering of do body. In **e_exit**, when an exit n is executed, since the value of dd reflects the number of enclosing do statements, the premise $n \leq dd$ stipulates that the number of enclosing do statements to be escaped (represented by n) should not be larger than the enclosing do statements.

The acs is an 'action stack' used for the execution of action and acall statements in the rules **e_action** and **e_acall**. In rule **e_action**, when an action statement action $L :\ \{\ \langle f\_i,\ f\_m \rangle\}$ is executed, the body of the action statement (f_i, f_m) is pushed on the action stack by the operation $(f\_i, f\_m) \centerdot acs$ in the premise. The action stack is represented by a list of type $[\![\mathcal{FL}(\mathcal{ID}) \times \mathcal{FL}(\mathcal{C})]\!]$, with the body of the most recently activated action statement at the head. In **e_acall**, when a acall statement acall L is executed, the action branch L being called is assumed to be in the most recently activated action statement and this assumption manifests in the $(f\_i, f\_m) \centerdot acs$ in the conclusion of **e_acall**. The premise $f\_i(i) = L$ in **e_acall** indicates that the i-th branch in f_i is labelled with L, the branch statement corresponding to i is $f\_m(i)$, which is executed in the second premise of **e_acall**.

The er is an 'execution result' used to accommodate the 'jumping out' effect of the exit and zaction statements. The type of er is $\mathcal{ER}$ which is defined as:

$$\frac{}{\mathcal{ER} : \mathsf{Set}}\ \textbf{er\_formation} \quad \frac{}{\mathsf{nm} : \mathcal{ER}}\ \textbf{nm\_intro}$$
$$\frac{n : \mathcal{N}at}{\mathsf{et}(n) : \mathcal{ER}}\ \textbf{et\_intro} \quad \frac{}{\mathsf{ez} : \mathcal{ER}}\ \textbf{ez\_intro} \tag{3}$$

The $\mathsf{et}(\ldots)$ is generated by the rule **e_exit**, where the execution of exit n generates $\mathsf{et}(n - dd)$, where the $n - dd$ indicates the level of enclosing do statement up to which the exit n statement intends to jump. The ez is generated by **e_zaction** which is for the execution of zaction, the intention of which is to jump out of the immediate enclosing action statement. Statements with no 'jumping out' effect generate nm, for example, the execution of $i := e$ generates nm.

Since the value of er indicates the terminating reason of execution, it may affect how the ensuing statements is going to be executed. For example, in **e_seq_c**, if the execution of c_1 terminates with nm, the following statement c_2 is executed. On the other hand, if c_1 is terminated by some value other than nm, as indicated by the premise $er \neq \mathsf{nm}$ in **e_seq_o**, the following c_2 will not be executed. The value of er is also checked extensively on the termination of loop bodies in both while and do statements.

The value of dd is reset to 0 when proc, pcall, action and acall are executed, this is to prevent the execution of these statements from being escaped by exit statement. After resetting dd to 0, if an exit is encountered during execution which intends to exit outside the enclosing proc, pcall, action or acall, the premise $n \leq dd$ in **e_exit** will prevent the exit statement from doing so.

The execution of $\{\langle its \rangle\ c\}$ is formalized by the rule **e_local**. The execution starts with the execution of the initialization statements its, which is formalized by the premise $\mathrm{init\_s}(s, its) = s'$, where the operation $\mathrm{init\_s}(s, its)$ computes the program store after the execution of its:

$$\begin{cases} \mathrm{init\_s}(s, i \leftarrow e \centerdot its) \stackrel{\mathsf{def}}{\Longrightarrow} \\ \qquad \mathrm{init\_s}(s, its) \succcurlyeq \left(\lambda\, \widetilde{s}\,.\ \left(\llbracket e \rrbracket_s \succcurlyeq \lambda\, v\,.\, \widetilde{s}\,[v^{\,i}]\right)\right) \\ \mathrm{init\_s}(s, i \leftarrow \bot \centerdot its) \stackrel{\mathsf{def}}{\Longrightarrow} \mathrm{init\_s}(s, its) \\ \qquad \mathrm{init\_s}(s, \langle\rangle) \stackrel{\mathsf{def}}{\Longrightarrow} s \end{cases} \tag{4}$$

where the definition of $\succcurlyeq$ is in (51). The semantics formalized by init_s is described as the following: the initialization statement $i \leftarrow e$ declares the local variable i and initializes it to the value of e. The initialization statement $i \leftarrow \bot$ declares the local variable i without initialization, if i has a value before the execution of $\{\langle its \rangle\ c\}$, that value is inherited by i; otherwise, the value of i is $\bot$.

The execution of its is followed by the execution of c, which is formalized by the premise $\left| s' \xrightarrow[{[cs,dd,acs]}]{c} s'' \right|_{er}$. And finally, all local variables are restored to their values before the execution of $\{\langle its \rangle\ c\}$, which is formalized by the $(s \blacktriangleleft \mathrm{vsi}(its) \blacktriangleright s'')$ in the conclusion, where $\mathrm{vsi}(its)$ is the set of local variables declared by its, which is defined as:

$$\begin{cases} \mathrm{vsi}(i \leftarrow e \centerdot its) \stackrel{\mathsf{def}}{\Longrightarrow} \{i\} \cup \mathrm{vsi}(its) \\ \mathrm{vsi}(i \leftarrow \bot \centerdot its) \stackrel{\mathsf{def}}{\Longrightarrow} \{i\} \cup \mathrm{vsi}(its) \\ \qquad \mathrm{vsi}(\langle\rangle) \stackrel{\mathsf{def}}{\Longrightarrow} \emptyset \end{cases} \tag{5}$$

The definition of $(\bullet \blacktriangleleft \bullet \blacktriangleright \bullet)$ is given in (54).

The execution of multi-way selection statement sel $\langle f\_e,\ f\_m \rangle$ les is by selecting a branch the guard expression of which evaluates to true, this is manifested by the premise $\llbracket f\_e(i) \rrbracket_s = \mathsf{true}$ in **e_sel**, which means the guard expression of the i-th branch evaluates to true, the execution of the branch statement is represented by the premise $\left| s \xrightarrow[{[cs,dd,acs]}]{f\_m(i)} s' \right|_{er}$.

The execution of action L : $\{\ \langle f\_i,\ f\_m \rangle\}$ starts with the branch labelled with L, this is manifests by the premise $f\_i(i) = L$ in **e_action**, which means the i-th branch is labelled with L, the execution of the branch statement is presented by the premise $\left| s \xrightarrow[{[cs,0,(f\_i,f\_m)\centerdot acs]}]{f\_m(i)} s' \right|_{er}$.

4 Program Transformations

Program transformations can be expressed as a function from program terms to program terms. Let $f : \mathcal{C} \rightarrow \mathcal{C}$ be such a program transformation, $c : \mathcal{C}$ a program, the transformation of c can be expressed as $f(c)$. A program transformation is correct if it preserves semantics while changing the syntax.

The assertion $c_1 \cong c_2$ means c_1 and c_2 are semantically equal, which is formally defined as:

$$c_1 \cong c_2 \stackrel{\mathsf{def}}{\Longrightarrow} \left| s \xrightarrow[{[cs,dd,acs]}]{c_1} s' \right|_{er} \Leftrightarrow \left| s \xrightarrow[{[cs,dd,acs]}]{c_2} s' \right|_{er} \tag{6}$$

With $\cong$, the correctness of a program transformation f can be expressed as $f(c) \cong c$. The proof of $c_1 \cong c_2$ usually consists of two lemmas, the one for

$$\left| s \xrightarrow[{[cs,dd,acs]}]{c_1} s' \right|_{er} \Rightarrow \left| s \xrightarrow[{[cs,dd,acs]}]{c_2} s' \right|_{er}$$

and the one for

$$\left| s \xrightarrow[{[cs,dd,acs]}]{c_2} s' \right|_{er} \Rightarrow \left| s \xrightarrow[{[cs,dd,acs]}]{c_1} s' \right|_{er}$$

so do the correctness proofs of program transformations.

A number of program transformations from Ward's thesis [16] have been expressed and proved correct in such a setting. These transformations, although most of them are quite simple, have been proved useful in reverse engineering by the practise of MA [17, 18].

For example, the program transformation which swaps the two branches of if e then c_1 else c_2 fi can be defined as:

$$\begin{cases} \mathsf{swpt}(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}) \\ \qquad \stackrel{\mathsf{def}}{\Longrightarrow} \mathsf{if}\ \neg e\ \mathsf{then}\ c_2\ \mathsf{else}\ c_1\ \mathsf{fi} \\ \mathsf{swpt}(\_) \stackrel{\mathsf{def}}{\Longrightarrow} \_ \end{cases} \tag{7}$$

where the equation $\mathsf{swpt}(\_) \stackrel{\mathsf{def}}{\Longrightarrow} \_$ means swpt makes no change for all other cases. The correctness of swpt can be expressed as $\mathsf{swpt}(c) \cong c$ and proved easily.

The transformation astif used to simplify the chaining of assert and if statements is defined as:

$$\begin{cases} \mathsf{astif}(\mathsf{assert}(e);\ \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}) \\ \qquad \stackrel{\mathsf{def}}{\Longrightarrow} \mathsf{assert}(e);\ c_1 \\ \mathsf{astif}(\_) \stackrel{\mathsf{def}}{\Longrightarrow} \_ \end{cases} \tag{8}$$

$$\frac{}{\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{skip}} s \right|_{\mathsf{nm}}}\ \textbf{e\_skip} \qquad \frac{[\![e]\!]_s = \mathsf{true}}{\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{assert}(e)} s \right|_{\mathsf{nm}}}\ \textbf{e\_asrt} \qquad \frac{[\![e]\!]_s = v}{\left| s \xrightarrow[{[cs,dd,acs]}]{i:=e} s\left[v\ {}^{i}\right] \right|_{\mathsf{nm}}}\ \textbf{e\_asgn}$$

$$\frac{[\![e]\!]_s = \mathsf{true} \qquad \left| s \xrightarrow[{[cs,dd,acs]}]{c_1} s' \right|_{er}}{\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}} s' \right|_{er}}\ \textbf{e\_ifs\_t} \qquad \frac{[\![e]\!]_s = \mathsf{false} \qquad \left| s \xrightarrow[{[cs,dd,acs]}]{c_2} s' \right|_{er}}{\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}} s' \right|_{er}}\ \textbf{e\_ifs\_f}$$

$$\frac{\left| s \xrightarrow[{[cs,dd,acs]}]{c_1} s' \right|_{\mathsf{nm}} \qquad \left| s' \xrightarrow[{[cs,dd,acs]}]{c_2} s'' \right|_{er}}{\left| s \xrightarrow[{[cs,dd,acs]}]{c_1;\ c_2} s'' \right|_{er}}\ \textbf{e\_seq\_c} \qquad \frac{\left| s \xrightarrow[{[cs,dd,acs]}]{c_1} s' \right|_{er} \qquad er \neq \mathsf{nm}}{\left| s \xrightarrow[{[cs,dd,acs]}]{c_1;\ c_2} s' \right|_{er}}\ \textbf{e\_seq\_o}$$

$$\frac{[\![e]\!]_s = \mathsf{true} \qquad \left| s \xrightarrow[{[cs,dd,acs]}]{c} s' \right|_{\mathsf{nm}} \qquad \left| s' \xrightarrow[{[cs,dd,acs]}]{\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}} s'' \right|_{er}}{\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}} s'' \right|_{er}}\ \textbf{e\_while\_t}$$

$$\frac{[\![e]\!]_s = \mathsf{true} \qquad \left| s \xrightarrow[{[cs,dd,acs]}]{c} s' \right|_{er} \qquad er \neq \mathsf{nm}}{\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}} s' \right|_{er}}\ \textbf{e\_while\_o} \qquad \frac{[\![e]\!]_s = \mathsf{false}}{\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}} s \right|_{\mathsf{nm}}}\ \textbf{e\_while\_f}$$

$$\frac{\mathrm{init\_s}(s, its) = s' \qquad \left| s' \xrightarrow[{[cs,dd,acs]}]{c} s'' \right|_{er}}{\left| s \xrightarrow[{[cs,dd,acs]}]{\{\langle its\rangle\ c\}} (s \blacktriangleleft \mathrm{vsi}(its) \blacktriangleright s'') \right|_{er}}\ \textbf{e\_local} \qquad \frac{[\![f\_e(i)]\!]_s = \mathsf{true} \qquad \left| s \xrightarrow[{[cs,dd,acs]}]{f\_m(i)} s' \right|_{er}}{\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{sel}\ \langle f\_e,\ f\_m\rangle\ \mathsf{les}} s' \right|_{er}}\ \textbf{e\_sel}$$

$$\frac{\left| s \xrightarrow[{[(pn,c)\bullet cs,0,acs]}]{c} s' \right|_{er}}{\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{proc}\ pn\ \equiv\ c} s' \right|_{er}}\ \textbf{e\_proc} \qquad \frac{\mathsf{lookup}(pn, cs) = (c, \widetilde{cs}) \qquad \left| s \xrightarrow[{[\widetilde{cs},0,acs]}]{c} s' \right|_{er}}{\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{pcall}\ pn} s' \right|_{er}}\ \textbf{e\_pcall}$$

$$\frac{\left| s \xrightarrow[{[cs,dd+1,acs]}]{c} s' \right|_{\mathsf{nm}} \qquad \left| s' \xrightarrow[{[cs,dd,acs]}]{\mathsf{do}\ c\ \mathsf{od}} s'' \right|_{er}}{\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{do}\ c\ \mathsf{od}} s'' \right|_{er}}\ \textbf{e\_do\_ag} \qquad \frac{\left| s \xrightarrow[{[cs,dd+1,acs]}]{c} s' \right|_{\mathsf{et}(dd)}}{\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{do}\ c\ \mathsf{od}} s' \right|_{\mathsf{nm}}}\ \textbf{e\_do\_tm}$$

$$\frac{\left| s \xrightarrow[{[cs,dd+1,acs]}]{c} s' \right|_{\mathsf{et}(n)} \qquad n < dd}{\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{do}\ c\ \mathsf{od}} s' \right|_{\mathsf{et}(n)}}\ \textbf{e\_do\_o} \qquad \frac{\left| s \xrightarrow[{[cs,dd+1,acs]}]{c} s' \right|_{\mathsf{ez}}}{\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{do}\ m\ \mathsf{od}} s' \right|_{\mathsf{ez}}}\ \textbf{e\_do\_z}$$

$$\frac{0 < n \quad \le dd}{\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{exit}\ n} s \right|_{\mathsf{et}(dd-n)}}\ \textbf{e\_exit} \qquad \frac{f\_i(i) = L \qquad \left| s \xrightarrow[{[cs,0,(f\_i,f\_m)\bullet acs]}]{f\_m(i)} s' \right|_{er}}{\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{action}\ L\text{: }\{\ \langle f\_i,\ f\_m\rangle\}} s' \right|_{\mathsf{nm}}}\ \textbf{e\_action}$$

$$\frac{f\_i(i) = L \qquad \left| s \xrightarrow[{[cs,0,(f\_i,f\_m)\bullet acs]}]{f\_m(i)} s' \right|_{er}}{\left| s \xrightarrow[{[cs,dd,(f\_i,f\_m)\bullet acs]}]{\mathsf{acall}\ L} s' \right|_{er}}\ \textbf{e\_acall} \qquad \frac{}{\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{zaction}} s \right|_{\mathsf{ez}}}\ \textbf{e\_zaction}$$

Figure 2: Operational semantics of WSL

The transformation dve used to simplify a local statement is defined as:

$$\begin{cases} \mathsf{dve}(\{\langle its\rangle\ c_1;\ c_2\}) \\ \qquad \stackrel{\mathsf{def}}{\Longrightarrow}\ \{\langle its\rangle\ c_1\} \quad \text{if } \mathsf{vasgn}(c_2) \subseteq \mathrm{vsi}(its) \\ \mathsf{dve}(\_)\ \stackrel{\mathsf{def}}{\Longrightarrow}\ \_ \end{cases} \tag{9}$$

where $\mathsf{vasgn}(c)$ is an operation used to compute the set of variables assigned to by the program c. For brevity, the definition of $\mathsf{vasgn}(c)$ is omitted.

The transformation fec used to absorb c to the back of $\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}$ is defined as:

$$\begin{cases} \mathsf{fec}(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi};\ c) \\ \qquad \stackrel{\mathsf{def}}{\Longrightarrow}\ \mathsf{if}\ e\ \mathsf{then}\ c_1;\ c\ \mathsf{else}\ c_2;\ c\ \mathsf{fi} \\ \mathsf{fec}(\_)\ \stackrel{\mathsf{def}}{\Longrightarrow}\ \_ \end{cases} \tag{10}$$

The transformation $\mathsf{wunroll}$, which unrolls a while statement, is defined as:

$$\begin{cases} \mathsf{wunroll}(\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}) \\ \quad \stackrel{\mathsf{def}}{\Longrightarrow}\ \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi};\ \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od} \\ \mathsf{wunroll}(\_)\ \stackrel{\mathsf{def}}{\Longrightarrow}\ \_ \end{cases} \tag{11}$$

The transformation dexp which expand the do statement is defined as:

$$\begin{cases} \mathsf{dexp}(\mathsf{do}\ c\ \mathsf{od})\ \stackrel{\mathsf{def}}{\Longrightarrow}\ \mathsf{do}\ c;\ c\ \mathsf{od} \\ \mathsf{dexp}(\_)\ \stackrel{\mathsf{def}}{\Longrightarrow}\ \_ \end{cases} \tag{12}$$

5 Example proofs

5.1 The correctness proof of wunroll

The correctness proof of $\mathsf{wunroll}$ consists of two lemmas. The first one is:

Lemma 5.1

$$\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}} s' \right|_{er} \Rightarrow \tag{13a}$$

$$\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi};\ \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}} s' \right|_{er} \tag{13b}$$

Proof: An inversion on (13a) reveals three possible cases for the execution of $\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}$:

1. When the execution is constructed by **e_while_t**, we have:

$$[\![e]\!]_s = \mathsf{true} \tag{14a}$$

$$\left| s \xrightarrow[{[cs,dd,acs]}]{c} s_m \right|_{\mathsf{nm}} \tag{14b}$$

$$\left| s_m \xrightarrow[{[cs,dd,acs]}]{\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}} s' \right|_{er} \tag{14c}$$

By applying **e_ifs_t** to (14a) and (14b), it can be deduced that:

$$\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi}} s_m \right|_{\mathsf{nm}} \tag{15}$$

By applying **e_seq_c** to (15) and (14c), the goal (13b) can be proved.

2. When the execution is constructed by **e_while_o**, we have:

$$[\![e]\!]_s = \mathsf{true} \tag{16a}$$

$$\left| s \xrightarrow[{[cs,dd,acs]}]{c} s' \right|_{er} \tag{16b}$$

$$er \neq \mathsf{nm} \tag{16c}$$

By applying **e_ifs_t** to (16a) and (16b), it can be deduced that:

$$\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi}} s' \right|_{er} \tag{17}$$

By applying **e_seq_o** to (17) and (16c), the goal (13b) can be deduced.

3. When the execution is constructed by **e_while_f**, we have:

$$[\![e]\!]_s = \mathsf{false} \tag{18a}$$

$$s' = s \tag{18b}$$

$$er = \mathsf{nm} \tag{18c}$$

By applying **e_ifs_f** to (18a), we have:

$$\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi}} s \right|_{\mathsf{nm}} \tag{19}$$

By applying **e_while_f** to (18a), we have:

$$\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}} s \right|_{\mathsf{nm}} \tag{20}$$

By applying **e_seq_c** to (19) and (20), it can be proved that:

$$\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi};\ \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}} s \right|_{\mathsf{nm}}$$

Combining this with (18b) and (18c), the goal (13b) can be proved.

$\square$

The second one is:

Lemma 5.2

$$\left| s \xrightarrow[{[cs,dd,acs]}]{\text{if } e \text{ then } c \text{ else skip fi; while } e \text{ do } c \text{ od}} s' \right|_{er} \tag{21a}$$

$$\Rightarrow \left| s \xrightarrow[{[cs,dd,acs]}]{\text{while } e \text{ do } c \text{ od}} s' \right|_{er} \tag{21b}$$

Proof: By inversion on (21a), we get two cases:

1. When (21a) is constructed by **e_seq_c**, we have:

$$\left| s \xrightarrow[{[cs,dd,acs]}]{\text{if } e \text{ then } c \text{ else skip fi}} s_m \right|_{\mathsf{nm}} \tag{22a}$$

$$\left| s_m \xrightarrow[{[cs,dd,acs]}]{\text{while } e \text{ do } c \text{ od}} s' \right|_{er} \tag{22b}$$

An inversion on (22a) gives rise to two sub-cases:

(a) When (22a) is constructed by **e_ifs_t**, we have:

$$[\![e]\!]_s = \mathsf{true} \tag{23a}$$

$$\left| s \xrightarrow[{[cs,dd,acs]}]{c} s_m \right|_{\mathsf{nm}} \tag{23b}$$

By applying **e_while_t** to (23a), (23b) and (22b), the goal (21b) is proved.

(b) When (22a) is constructed by **e_ifs_f**, we have:

$$[\![e]\!]_s = \mathsf{false} \tag{24a}$$

$$\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{skip}} s_m \right|_{\mathsf{nm}} \tag{24b}$$

From (24b), it can be derived that:

$$s_m = s \tag{25}$$

Rewriting (22b) using (25), we have:

$$\left| s \xrightarrow[{[cs,dd,acs]}]{\text{while } e \text{ do } c \text{ od}} s' \right|_{er} \tag{26}$$

Which is exactly the goal (21b).

2. When (21a) is constructed by **e_seq_o**, we have:

$$\left| s \xrightarrow[{[cs,dd,acs]}]{\text{if } e \text{ then } c \text{ else skip fi}} s' \right|_{er} \tag{27a}$$

$$er \neq \mathsf{nm} \tag{27b}$$

An inversion on (27a) gives rise to two sub-cases:

(a) When (27a) is constructed by **e_ifs_t**, we have:

$$[\![e]\!]_s = \mathsf{true} \tag{28a}$$

$$\left| s \xrightarrow[{[cs,dd,acs]}]{c} s' \right|_{er} \tag{28b}$$

By applying **e_while_o** to (28a), (28b) and (27b), the goal (21b) is proved.

(b) When (27a) is constructed by **e_ifs_f**, we have:

$$[\![e]\!]_s = \mathsf{false} \tag{29a}$$

$$\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{skip}} s_m \right|_{er} \tag{29b}$$

This case can be refuted, because the execution of skip can never generate an er which is not nm, as specified by (27b).

$\square$

By combining Lemma 5.1 and Lemma 5.2, it can be proved easily that:

Lemma 5.3 (Correctness of wunroll)

$$\mathsf{wunroll}(c) \cong c$$

Proof: After expanding the definition of wunroll, it becomes:

$$\text{if } e \text{ then } c \text{ else skip fi; while } e \text{ do } c \text{ od} \cong \text{while } e \text{ do } c \text{ od}$$

the proof of which naturally splits into Lemma 5.1 and Lemma 5.2. $\square$

5.2 The correctness proof of dexp

The correctness proof of dexp consists of two lemmas. The first one is:

Lemma 5.4

$$\left| s \xrightarrow[{[cs,dd,acs]}]{\text{do } c \text{ od}} s' \right|_{er} \Rightarrow \tag{30a}$$

$$\left| s \xrightarrow[{[cs,dd,acs]}]{\text{do } c;\ c \text{ od}} s' \right|_{er} \tag{30b}$$

Proof: The proof is by induction on the structure of (30a), the non-trivial cases are:

1. When (30a) is constructed by **e_do_ag**, we have:

$$\left| s \xrightarrow[{[cs,dd+1,acs]}]{c} s_m \right|_{\mathsf{nm}} \tag{31a}$$

$$\left| s_m \xrightarrow[{[cs,dd,acs]}]{\text{do } c \text{ od}} s' \right|_{er} \tag{31b}$$

An inversion on (31b) reveals the following sub-cases:

(a) When (31b) is constructed by **e_do_ag**, we have:

$$\left| s_m \xrightarrow[{[cs,dd+1,acs]}]{c} \widetilde{s_m} \right|_{\mathsf{nm}} \tag{32a}$$

$$\left| \widetilde{s_m} \xrightarrow[{[cs,dd,acs]}]{\mathsf{do}\ c\ \mathsf{od}} s' \right|_{er} \tag{32b}$$

By applying **e_seq_c** to (31a) and (32a), we have:

$$\left| s \xrightarrow[{[cs,dd+1,acs]}]{c;\ c} \widetilde{s_m} \right|_{\mathsf{nm}} \tag{33}$$

By induction hypothesis on (32b), we have:

$$\left| \widetilde{s_m} \xrightarrow[{[cs,dd,acs]}]{\mathsf{do}\ c;\ c\ \mathsf{od}} s' \right|_{er} \tag{34}$$

By applying **e_do_ag** to (33) and (34), the goal (30b) is proved.

(b) When (31b) is constructed by **e_do_tm**, we have:

$$\left| s_m \xrightarrow[{[cs,dd+1,acs]}]{c} s' \right|_{\mathsf{et}(dd)} \tag{35a}$$

$$er = \mathsf{nm} \tag{35b}$$

By applying **e_seq_c** to (31a) and (35a), we have:

$$\left| s \xrightarrow[{[cs,dd+1,acs]}]{c;\ c} s' \right|_{\mathsf{et}(dd)} \tag{36}$$

By applying **e_do_tm** to this, we have:

$$\left| s \xrightarrow[{[cs,dd+1,acs]}]{\mathsf{do}\ c;\ c\ \mathsf{od}} s' \right|_{\mathsf{nm}}$$

Combing this with (35b), the goal (30b) is proved.

(c) When (31b) is constructed by **e_do_o** or **e_do_o**, we have:

$$\left| s_m \xrightarrow[{[cs,dd+1,acs]}]{c} s' \right|_{\mathsf{et}(n)|\mathsf{ez}} \tag{37a}$$

$$er = \mathsf{et}(n)|\mathsf{ez} \tag{37b}$$

The proof can be carried out similarly as Case 1b.

2. When (30a) is constructed by **e_do_tm**, we have:

$$\left| s_m \xrightarrow[{[cs,dd+1,acs]}]{c} s' \right|_{\mathsf{et}(dd)} \tag{38a}$$

$$er = \mathsf{nm} \tag{38b}$$

By applying **e_seq_o** to (38a), we have:

$$\left| s \xrightarrow[{[cs,dd+1,acs]}]{c;\ c} s' \right|_{\mathsf{et}(dd)} \tag{39}$$

By applying **e_do_tm** to this, we have:

$$\left| s \xrightarrow[{[cs,dd+1,acs]}]{\mathsf{do}\ c;\ c\ \mathsf{od}} s' \right|_{\mathsf{nm}}$$

Combing this with (38b), the goal (30b) is proved.

3. When (30a) is constructed by **e_do_o** or **e_do_o**, we have:

$$\left| s_m \xrightarrow[{[cs,dd+1,acs]}]{c} s' \right|_{\mathsf{et}(n)|\mathsf{ez}} \tag{40a}$$

$$er = \mathsf{et}(n)|\mathsf{ez} \tag{40b}$$

The proof can be carried out similarly as Case 2.

□

The second one is:

Lemma 5.5

$$\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{do}\ c;\ c\ \mathsf{od}} s' \right|_{er} \tag{41a}$$

$$\Rightarrow \left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{do}\ c\ \mathsf{od}} s' \right|_{er} \tag{41b}$$

Proof: The proof is by induction on the structure of (41a). There are four non-trivial cases:

1. When the execution is constructed by **e_do_ag**, we have:

$$\left| s \xrightarrow[{[cs,dd+1,acs]}]{c;\ c} s_1 \right|_{\mathsf{nm}} \tag{42a}$$

$$\left| s_1 \xrightarrow[{[cs,dd,acs]}]{\mathsf{do}\ c;\ c\ \mathsf{od}} s' \right|_{er} \tag{42b}$$

From (42a), it can be deduced that:

$$\left| s \xrightarrow[{[cs,dd+1,acs]}]{c} s_2 \right|_{\mathsf{nm}} \tag{43a}$$

$$\left| s_2 \xrightarrow[{[cs,dd+1,acs]}]{c} s_1 \right|_{\mathsf{nm}} \tag{43b}$$

From the induction hypothesis of (42b), it can be deduced that:

$$\left| s_1 \xrightarrow[{[cs,dd,acs]}]{\mathsf{do}\ c\ \mathsf{od}} s' \right|_{er} \tag{44}$$

By applying **e_do_ag** to (43b) and (44), it can be deduced that:

$$\left| s_2 \xrightarrow[{[cs,dd,acs]}]{\mathsf{do}\ c\ \mathsf{od}} s' \right|_{er} \tag{45}$$

By applying **e_do_ag** to (43a) and (45), the goal (41b) can be proved.

2. When the execution is constructed by **e_do_tm**, we have:

$$\left| s \xrightarrow[{[cs,dd+1,acs]}]{c;\ c} s' \right|_{\mathsf{et}(dd)} \tag{46a}$$

$$er = \mathsf{nm} \tag{46b}$$

An analysis of (46a) gives rise to two cases:

(a) When

$$\left| s \xrightarrow[{[cs,dd+1,acs]}]{c} s_1 \right|_{\mathsf{nm}} \tag{47a}$$

$$\left| s_1 \xrightarrow[{[cs,dd+1,acs]}]{c} s' \right|_{\mathsf{et}(dd)} \tag{47b}$$

by applying **e_do_tm** to (47b), it can be deduced that:

$$\left| s_1 \xrightarrow[{[cs,dd,acs]}]{\mathsf{do}\ c\ \mathsf{od}} s' \right|_{\mathsf{nm}} \tag{48}$$

By applying **e_do_ag** to (47a) and (48), it can be deduced that $\left| s \xrightarrow[{[cs,dd,acs]}]{\mathsf{do}\ c\ \mathsf{od}} s' \right|_{\mathsf{nm}}$. Combing this with (46b), the goal can be proved.

(b) When

$$\left| s \xrightarrow[{[cs,dd+1,acs]}]{c} s' \right|_{\mathsf{et}(dd)} \tag{49}$$

By applying **e_do_tm**, the goal (41b) can be proved.

3. When the execution is constructed by **e_do_o**, the proof is similar to Case 2.

4. When the execution is constructed by **e_do_z**, the proof is similar to Case 2.

□

By combing Lemma 5.4 and Lemma 5.5, the correctness of dexp can be proved:

Lemma 5.6 (Correctness of dexp) $\mathsf{dexp}(c) \cong c$

6 Conclusion

This paper presents a mechanized operational semantics of WSL and shows how it can be used to verify the correctness of program transformations. It constitutes a first step toward a framework in which program transformations can be developed rigorously so that the correctness of them is guaranteed by automatic proof checking.

7 References

[1] R. A. De Millo, R. J. Lipton, and A. J. Perlis, "Social processes and proofs of theorems and programs," *Communications of the ACM*, vol. 22, pp. 271–280, May 1979. An earlier version appeared in *ACM Symposium on Principles of Programming Languages (POPL) , Los Angeles, California* , 1977 pp. 206–214.

[2] J. H. Fetzer, "Program verification: the very idea," *Communications of the ACM*, vol. 31, pp. 1048–1063, Sept. 1988.

[3] M. Gordon, "Introduction to the HOL system," in *Proceedigns of the International Workshop on the HOL Theorem Proving System and its Applications* (M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, eds.), (Los Alamitos, CA, USA), pp. 2–3, IEEE Computer Society Press, Aug. 1992.

[4] M. Gordon and T. F. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[5] L. C. Paulson, "Isabelle: A generic theorem prover," *Lecture Notes in Computer Science*, vol. 828, pp. xvii + 321, 1994.

[6] L. C. Paulson and T. Nipkow, "Isabelle tutorial and user's manual," Tech. Rep. TR-189, Computer Laboratory, University of Cambridge, Jan. 1990.

[7] N. Shankar, S. Owre, and J. M. Rushby, "A tutorial on specification and verification using PVS," preliminary draft, Csl report, SRI International, Menlo Park, CA, USA, Feb. 1993.

[8] S. Owre, N. Shankar, and J. M. Rushby, *User Guide for the PVS Specification and Verification System.* Computer Science Laboratory, SRI International, 1993.

[9] Z. Luo and R. Pollack, "LEGO proof development system: User's manual," Tech. Rep. ECS-LFCS-92-211, LFCS, Computer Science Dept., University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, May 1992. Updated version.

[10] Z. Luo, *Computation and Reasoning: A Type Theory for Computer Science.* No. 11 in International Series of Monographs on Computer Science, Oxford University Press, 1994.

[11] L. Magnusson, *The Implementation of ALF—a Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution.* Phd thesis, Dept. of Computing Science, Chalmers Univ. of Technology and Univ. of Göteborg, 1994.

[12] B. Nordström, K. Peterson, and J. M. Smith, *Programming in Martin-Lof's Type Theory*, vol. 7 of *International Series of Monographs on Computer Science.* New York, NY: Oxford University Press, 1990.

[13] G. Huet, G. Kahn, and C. Paulin-Mohring, "The Coq proof assistant, A tutorial, version 5.10," Technical Report RT-0178, INRIA (Institut National de Recherche en Informatique et en Automatique), 1995.

[14] E. Gimenez, "A tutorial on recursive types in Coq," Technical Report RT-0221, INRIA (Institut National de Recherche en Informatique et en Automatique), 1998.

[15] C. Cornes, J. Courant, J.-C. Filliatre, G. Huet, P. Manoury, C. Munoz, C. Murthy, Christine, A. Saibi, and B. Werner, "The Coq proof assistant, reference manual, version 7.0," Technical Report RT-0177, INRIA (Institut National de Recherche en Informatique et en Automatique), 2001.

[16] M. Ward, *Proving Program Refinements And Transformations.* D.Phil Thesis., University of Oxford, 1989.

[17] M. Ward, F. W. Calliss, and M. Munro, "The maintainer's assistant," in *Proceedings of the International Conference on Software Maintenance 1989*, p. 307, IEEE Computer Society Press, Los Alamitos, California, USA, 1989.

[18] M. Ward, "Assembler to c migration using the fermat transformation system," in *IEEE International Conference on Software Maintenance (ICSM'99)*, (Oxford, UK), IEEE Computer Society Press, Los Alamitos, California, USA, Aug. 1999.

A Auxiliary definitions

A number of auxiliary types and functions used in this paper are defined in this Appendix.

A.1 List

For any data type $A : \mathtt{Set}$, $[\![A]\!]$ is the type of lists consisting of elements from A. In $[\![A]\!]$, the empty list is written as $\langle\rangle$, $a \centerdot l$ is the list obtained from l by putting a to the head of it.

A.2 Exceptional set

For $A : \mathtt{Set}$, the exceptional set $\mathcal{M}(A)$ is the type obtained from A by adding a special element $\perp$ to represent undefined value. The formal definition is:

$$\begin{array}{c} \dfrac{A : \mathsf{Set}}{\perp : \mathcal{M}(A)} \ \textbf{bottom\_value} \\ \dfrac{A : \mathsf{Set} \quad a : A}{\mathsf{return}(a) : \mathcal{M}(A)} \ \textbf{normal\_value} \end{array} \tag{50}$$

A normal element a of the original type A is represented as $\mathsf{return}(a)$ in the exceptional set $\mathcal{M}(A)$. However, the return is usually omitted in the sequel, unless there is possibility of confusion.

For $ev : \mathcal{M}(A)$ and $f : A \to \mathcal{M}(B)$, the operator $ev \succcurlyeq f$ is defined as:

$$\begin{cases} \mathsf{return}(a) \succcurlyeq f \stackrel{\mathsf{def}}{\Longrightarrow} f(a) \\ \perp \succcurlyeq f \stackrel{\mathsf{def}}{\Longrightarrow} \perp \end{cases} \tag{51}$$

A.3 Functional list

Functional list is another way to representation lists. The type $\mathcal{FL}(A)$ is defined as a function from $\mathcal{N}at$ to $\mathcal{M}(A)$: $\mathcal{FL}(A) \stackrel{\mathsf{def}}{\Longrightarrow} \mathcal{N}at \to A$, where $\mathcal{N}at$ is the type of natural numbers in the underlying type theory. The intuition of this definition is that: $fl(n) = a$ means the element a is at the n-th position of the functional list fl. To represent empty list, a default value $a' : A$ is needed, so that the empty list can be defined as: $\ominus_{a'} \stackrel{\mathsf{def}}{\Longrightarrow} \lambda n.\, a'$. An function $\diamond$ is defined to add an element to the head of a functional list:

$$\begin{cases} (a \diamond fl)(0) \stackrel{\mathsf{def}}{\Longrightarrow} a \\ (a \diamond fl)(n) \stackrel{\mathsf{def}}{\Longrightarrow} fl(n-1) \quad \text{if } n > 0 \end{cases} \tag{52}$$

A.4 Finite mapping

Given any two type A and B, $A \rightharpoonup B$ is the type of finite mappings from A to B, which is defined as: $A \rightharpoonup B \stackrel{\mathsf{def}}{\Longrightarrow} A \to \mathcal{M}(B)$. The empty mapping, which does not map anything, is written as ϵ, and is defined as: $\epsilon \stackrel{\mathsf{def}}{\Longrightarrow} \lambda a : A. \perp$. Intuitively, for finite mapping $s : A \rightharpoonup B$, $a : A$, $b : B$, the assertion $s(a) = b$ means: the mapping s maps a to b.

The operation $s\,[b\,^a]$ is used to add a new map 'a to b' to s, overriding the original maps for a in s, it is formally defined as:

$$\begin{cases} s\,[b\,^a]\,(\widetilde{a}) \stackrel{\mathsf{def}}{\Longrightarrow} b & \text{if } \widetilde{a} = a \\ s\,[b\,^a]\,(\widetilde{a}) \stackrel{\mathsf{def}}{\Longrightarrow} s(\widetilde{a}) & \text{otherwise} \end{cases} \tag{53}$$

The operation $(s_1 \blacktriangleleft vs \blacktriangleright s_2)$ is defined to shadow the maps in s_1 over the maps in s_2, the shadowing is restricted to the maps for those in vs. It is formally defined as:

$$\begin{cases} (s_1 \blacktriangleleft vs \blacktriangleright s_2)(a) \stackrel{\mathsf{def}}{\Longrightarrow} s_1(a) & \text{if } a \in vs \\ (s_1 \blacktriangleleft vs \blacktriangleright s_2)(a) \stackrel{\mathsf{def}}{\Longrightarrow} s_2(a) & \text{if } a \notin vs \end{cases} \tag{54}$$