Weakest Precondition for General Recursive Programs Formalized in Coq *

Xingyuan Zhang[1], Malcolm Munro[1], Mark Harman[2], and Lin Hu[2]

[1] Department of Computer Science, University of Durham, Science Laboratories, South Road, Durhram, DH1 3LE, U.K.
{Xingyuan.Zhang, Malcolm.Munro}@durham.ac.uk
[2] Department of Information Systems and Computing, Brunel University, Uxbridge, Middlesex, UB8 3PH, U.K.
{Mark.Harman, Lin.Hu}@brunel.ac.uk

Abstract. This paper describes a formalization of the weakest precondition, wp, for general recursive programs using the type-theoretical proof assistant Coq. The formalization is a deep embedding using the computational power intrinsic to type theory. Since Coq accepts only structural recursive functions, the computational embedding of general recursive programs is non-trivial. To justify the embedding, an operational semantics is defined and the equivalence between wp and the operational semantics is proved. Three major healthiness conditions, namely: Strictness, Monotonicity and Conjunctivity are proved as well.
Keywords: Weakest Precondition, Operational Semantics, Formal Verification, Coq

1 Introduction

The weakest precondition, wp, proposed by E. W. Dijkstra [5] proved to be useful in various areas of software development and has been investigated extensively [1, 13, 12]. There have been a number of attempts to support wp and refinement calculus with computer assisted reasoning systems such as HOL [16, 19, 9], Isabelle [14, 17, 18], Ergo [3], PVS [8] and Alf [10]. Unfortunately, among these works, only Laibinis and Wright [9] deals with general recursion.

In this paper, an embedding of wp is given for general recursive programs using the intentional type theory supported by Coq [2]. Since the computational mechanism peculiar to type theory is used, we name such a style of embedding 'computational embedding'. The importance of computational embedding is that it can be seen as a bridge between deep and shallow embedding. Since wp is defined as a function from statement terms to predicate transformers, before the definition of wp in wp (c) is expanded, by accessing the syntax structure of c, we enjoy the benefit of deep embedding, so that meta-level operations such as program transformations (usually expressed as functions on program terms)

* The work in this paper is sponsored by the EPSRC project GUSTT

can be verified. On the other hand, after expanding wp, $\mathsf{wp}\,(c)$ becomes the semantics of c. With the computational mechanism taking care of the expanding process, we can focus on the semantics without worrying about syntax details, a benefit traditionally enjoyed by shallow embedding. Therefore, computational embedding enables us to switch between deep and shallow embedding with ease.

The language investigated in this paper is general recursive in the sense that it contains a statement which takes the form of a parameter-less recursive procedure : $\mathsf{proc}\ p \equiv c$, the execution of which starts with the execution of the procedure body c. When a recursive call $\mathsf{pcall}\ p$ is encountered during the execution, the $\mathsf{pcall}\ p$ is replaced by the procedure body c and the execution continues from the beginning of c.

A naïve definition of $\mathsf{wp}\,(\mathsf{proc}\ p \equiv c)$ could be:

$$\mathsf{wp}\,(\mathsf{proc}\ p \equiv c) \overset{\mathsf{def}}{\Longrightarrow} \bigvee_{n<\omega} \mathsf{wp}\left(\boxed{\mathsf{proc}\ p \equiv c}^{\,n}\right) \tag{1}$$

where the expansion operation $\boxed{\mathsf{proc}\ p \equiv c}^{\,n}$ is defined as:

$$\begin{cases} \boxed{\mathsf{proc}\ p \equiv c}^{\,0} \overset{\mathsf{def}}{\Longrightarrow} \mathsf{assert}(\overline{\mathsf{false}}) \\ \boxed{\mathsf{proc}\ p \equiv c}^{\,n+1} \overset{\mathsf{def}}{\Longrightarrow} c\left\{p/\boxed{\mathsf{proc}\ p \equiv c}^{\,n}\right\} \end{cases} \tag{2}$$

where in each round of expansion, p in c is substituted by $\boxed{\mathsf{proc}\ p \equiv c}^{\,n}$.

The problem with this definition is that: in (1), the recursive call of wp:

$$\mathsf{wp}\left(\boxed{\mathsf{proc}\ p \equiv c}^{\,n}\right)$$

is not structural recursive, since the argument $\boxed{\mathsf{proc}\ p \equiv c}^{\,n}$ is structurally larger than the argument $\mathsf{proc}\ p \equiv c$ on the left. Therefore, (1) is rejected by the function definition mechanism of Coq, which only accepts structural recursive functions.

This paper proposes a formulation of wp which solves this problem. To justify such a formulation, an operational semantics is defined and the equivalence of wp and the operational semantics is proved. Three major healthiness conditions are proved for wp as well, namely: Strictness, Monotonicity and Conjunctivity. For brevity, in this paper, only the equivalence proof is discussed in full detail.

The rest of the paper is arranged as follows: Section 2 defines the notions of predicate, predicate transformer, transformer of predicate transformers together with various partial orders and monotonicity predicates. Section 3 gives the definition wp. Section 4 presents the operational semantics. Section 5 relates operational semantics to wp. Section 6 relates wp to operational semantics. Section 7 concludes the whole paper. Additional technical detail can be found in Appendices. The technical development of this paper has been fully formalized and checked by Coq. The Coq scripts are available from

`http : //www.dur.ac.uk/xingyuan.zhang/tphol`

Conventions. Because type theory [4, 15, 11] was first proposed to formalize constructive mathematics, we are able to present the work in standard mathematical notation. For brevity, we use the name of a variable to suggest its type. For example, $s, s', \overline{s}, \ldots$ in this paper always represent program stores.

Type theory has a notion of computation, which is used as a definition mechanism, where the equation $a \stackrel{\mathsf{def}}{\Longrightarrow} b$ represents a 'computational rule' used to expand the definition of a to b.

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

For $A : \mathtt{Set}$, the exceptional set $\mathcal{M}(A)$ is defined as:

$$\begin{array}{c} \dfrac{A : \mathtt{Set}}{\mathcal{M}(A) : \mathtt{Set}}\ \mathcal{M}\textbf{\_formation} \\ \dfrac{A : \mathtt{Set}}{\bot : \mathcal{M}(A)}\ \textbf{bottom\_value} \quad \dfrac{A : \mathtt{Set} \quad a : A}{\mathsf{unit}\,(a) : \mathcal{M}(A)}\ \textbf{normal\_value} \end{array} \tag{3}$$

Intuitively, $\mathcal{M}(A)$ represents the type obtained from A by adding a special element $\bot$ to represent undefined value. A normal element a of the original type A is represented as $\mathsf{unit}\,(a)$ in exceptional set $\mathcal{M}(A)$. However, the unit is usually omitted, unless there is possibility of confusion.

2 Predicates and predicate transformers

We take the view that the predicates transformed by wp are predicates on program stores. Therefore, the type of predicates $\mathcal{PD}$ is defined as:

$$\mathcal{PD} \stackrel{\mathsf{def}}{\Longrightarrow} \mathcal{PS} \rightarrow \mathtt{Prop} \tag{4}$$

where $\mathtt{Prop}$ is the type of propositions, $\mathcal{PS}$ is the type of program stores. For simplicity, this paper is abstract about program stores and denotable values. However, there is a comprehensive treatment of program stores, denotable values and expression evaluation in the Coq scripts.

The type of predicate transformers $\mathcal{PT}$ is defined as:

$$\mathcal{PT} \stackrel{\mathsf{def}}{\Longrightarrow} \mathcal{PD} \rightarrow \mathcal{PD} \tag{5}$$

The type of transformers of predicate transformers $\mathcal{PTT}$ is defined as:

$$\mathcal{PTT} \stackrel{\mathsf{def}}{\Longrightarrow} \mathcal{PT} \rightarrow \mathcal{PT} \tag{6}$$

The partial order $\preccurlyeq_P$ between predicates is defined as:

$$P_1 \preccurlyeq_P P_2 \stackrel{\mathsf{def}}{\Longrightarrow} \forall\, s\,.\, P_1\,(s) \Rightarrow P_2\,(s) \tag{7}$$

The partial order $\preccurlyeq_{pt}$ between predicate transformers is defined as:

$$pt_1 \preccurlyeq_{pt} pt_2 \stackrel{\mathsf{def}}{\Longrightarrow} \forall\, P\,.\, pt_1\,(P) \preccurlyeq_P pt_2\,(P) \tag{8}$$

The monotonicity predicate on $\mathcal{PT}$ is defined as:

$$\text{mono}(pt) \overset{\mathsf{def}}{\Longrightarrow} \forall P_1, P_2 .\, P_1 \preccurlyeq_P P_2 \Rightarrow pt\,(P_1) \preccurlyeq_P pt\,(P_2) \tag{9}$$

The partial order on $\preccurlyeq_\sigma$ between environments is defined as:

$$\sigma_1 \preccurlyeq_\sigma \sigma_2 \overset{\mathsf{def}}{\Longrightarrow} \forall\, p\,.\, \sigma_1\,(p) \preccurlyeq_{pt} \sigma_2\,(p) \tag{10}$$

The partial order $\preccurlyeq_{ptt}$ between transformers of predicate transformers is defined as:

$$ptt_1 \preccurlyeq_{ptt} ptt_2 \overset{\mathsf{def}}{\Longrightarrow} \forall\, pt\,.\, \text{mono}(pt) \Rightarrow ptt_1\,(pt) \preccurlyeq_{pt} ptt_2\,(pt) \tag{11}$$

The corresponding derived equivalence relations are defined as:

$$\begin{cases} P_1 \approx_P P_2 \overset{\mathsf{def}}{\Longrightarrow} P_1 \preccurlyeq_P P_2 \wedge P_2 \preccurlyeq_P P_1 \\ pt_1 \approx_{\mathsf{pt}} pt_2 \overset{\mathsf{def}}{\Longrightarrow} pt_1 \preccurlyeq_{pt} pt_2 \wedge pt_2 \preccurlyeq_{pt} pt_1 \\ ptt_1 \approx_{\mathsf{ptt}} ptt_2 \overset{\mathsf{def}}{\Longrightarrow} ptt_1 \preccurlyeq_{ptt} ptt_2 \wedge ptt_2 \preccurlyeq_{ptt} ptt_1 \end{cases} \tag{12}$$

The monotonicity predicate on environments is defined as:

$$\text{mono}(\sigma) \overset{\mathsf{def}}{\Longrightarrow} \forall\, p\,.\, \text{mono}(\sigma\,(p)) \tag{13}$$

3 Formalization of wp

In order to overcome the problem mentioned in the Introduction, a notion of environment is introduced, which is represented by the type Σ:

$$\Sigma \overset{\mathsf{def}}{\Longrightarrow} \mathcal{ID} \rightarrow \mathcal{PT} \tag{14}$$

which is a function from identifiers to predicate transformers. In this paper, procedure names are represented as identifiers.

The empty environment $\varepsilon : \Sigma$, which maps all procedure names to the false predicate transformer $\lambda\, P .\, \mathsf{F}$, is defined as:

$$\varepsilon \overset{\mathsf{def}}{\Longrightarrow} \lambda\, p .\, \lambda\, P .\, \mathsf{F} \tag{15}$$

where p is procedure name, P is predicate and F is the false predicate, which is defined as:

$$\mathsf{F} \overset{\mathsf{def}}{\Longrightarrow} \lambda\, s .\, \mathsf{False} \tag{16}$$

where s is program store, False is the false proposition in Coq. Therefore, F does not hold on any program store.

The operation $\sigma\,[p \mapsto pt]$ is defined, to add the mapping $p \mapsto pt$ to environment σ:

$$\begin{cases} \sigma\,[p \mapsto pt]\,(\overline{p}) \overset{\mathsf{def}}{\Longrightarrow} pt & \text{if } \overline{p} = p \\ \sigma\,[p \mapsto pt]\,(\overline{p}) \overset{\mathsf{def}}{\Longrightarrow} \sigma\,(\overline{p}) & \text{if } \overline{p} \neq p \end{cases} \tag{17}$$

$$\frac{}{\mathcal{C} : \mathtt{Set}}\ \mathbf{\mathcal{C}\_fmt}$$

$$\frac{i : \mathcal{ID} \quad e : \mathcal{E}}{i := e : \mathcal{C}}\ \mathbf{s\_asgn} \qquad \frac{e : \mathcal{E}}{\mathsf{assert}(e) : \mathcal{C}}\ \mathbf{s\_assert} \qquad \frac{c_1 : \mathcal{C} \quad c_2 : \mathcal{C}}{c_1;\ c_2 : \mathcal{C}}\ \mathbf{s\_seq}$$

$$\frac{e : \mathcal{E} \quad c_1 : \mathcal{C} \quad c_2 : \mathcal{C}}{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : \mathcal{C}}\ \mathbf{s\_ifs} \qquad \frac{p : \mathcal{ID} \quad c : \mathcal{C}}{\mathsf{proc}\ p\ \equiv\ c : \mathcal{C}}\ \mathbf{s\_proc} \qquad \frac{p : \mathcal{ID}}{\mathsf{pcall}\ p : \mathcal{C}}\ \mathbf{s\_pcall}$$

Fig. 1. The Definition of $\mathcal{C}$

$$\left\{\begin{array}{lr}
\mathsf{wpc}\,(\sigma,\ i := e)\ \stackrel{\mathsf{def}}{\Longrightarrow}\ \lambda\, P, s.\, \exists\, v\,.\, [\![e]\!]_s = v\ \wedge\ P\left(s\left[v^{\,i}\right]\right) & \text{(18a)} \\
\mathsf{wpc}\,(\sigma,\ \mathsf{assert}(e))\ \stackrel{\mathsf{def}}{\Longrightarrow}\ \lambda\, P, s.\, [\![e]\!]_s = \mathsf{true}\ \wedge\ P\,(s) & \text{(18b)} \\
\mathsf{wpc}\,(\sigma,\ c_1;\ c_2)\ \stackrel{\mathsf{def}}{\Longrightarrow}\ \lambda\, P, s.\, \mathsf{wpc}\,(\sigma,\ c_1)\,(\mathsf{wpc}\,(\sigma,\ c_2)\,(P))\,(s) & \text{(18c)} \\
\mathsf{wpc}\,(\sigma,\ \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ \stackrel{\mathsf{def}}{\Longrightarrow}\ \lambda\, P, s\,.([\![e]\!]_s = \mathsf{true}\ \wedge\ \mathsf{wpc}\,(\sigma,\ c_1)\,(P)\,(s)) \vee & \\
\qquad\qquad\qquad\qquad\qquad\qquad ([\![e]\!]_s = \mathsf{false}\ \wedge\ \mathsf{wpc}\,(\sigma,\ c_2)\,(P)\,(s)) & \text{(18d)} \\
\mathsf{wpc}\,(\sigma,\ \mathsf{proc}\ p\ \equiv\ c)\ \stackrel{\mathsf{def}}{\Longrightarrow}\ \lambda\, P, s.\, \exists\, n\,.\, \boxed{\lambda\, \overline{pt}.\, \mathsf{wpc}\left(\sigma\left[p \mapsto \overline{pt}\right],\ c\right)}^{\,n}\,(P)\,(s) & \text{(18e)} \\
\mathsf{wpc}\,(\sigma,\ \mathsf{pcall}\ p)\ \stackrel{\mathsf{def}}{\Longrightarrow}\ \lambda\, P, s.\, \sigma\,(p)\,(P)\,(s) & \text{(18f)}
\end{array}\right.$$

Fig. 2. The Definition of wpc

Instead of defining wp directly, an operation $\mathsf{wpc}\,(\sigma,\ c)$ is defined to compute a predicate transformer for command c under the environment σ. By using the environment σ, $\mathsf{wpc}\,(\sigma,\ c)$ can be defined using structural recursion. With wpc, the normal wp is now defined as: $\mathsf{wp}\,(c)\ \stackrel{\mathsf{def}}{\Longrightarrow}\ \mathsf{wpc}\,(\varepsilon,\ c)$.

The syntax of the programming language formalized in this paper is given in Figure 1 as the inductive type $\mathcal{C}$. 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 $[\![e]\!]_s = \perp$ means there is no valuation of expression e under program store s. The definition of $\mathsf{wpc}\,(\sigma,\ c)$ is given in Figure 2, where each command corresponds to an equation. The right-hand-side of each equation is a lambda abstraction $\lambda\, P, s.\,(\ldots)$, where P is the predicate required to hold after execution of the corresponding command, and s is the program store before execution. The operation $s\left[v^{\,i}\right]$ is the program store obtained from program store s by setting the value of variable i to v.

Equations (18a) – (18d) are quite standard. Only (18e) and (18f) need more explanation. Equation (18e) is for $\mathsf{wpc}\,(\sigma,\ \mathsf{proc}\ p\ \equiv\ c)$, the originally problematic case. The recursive call on the right-hand-side is $\mathsf{wpc}\left(\sigma\left[p \mapsto \overline{pt}\right],\ c\right)$, which is structural recursive with respect to the second argument. The key idea is that: a mapping $p \mapsto \overline{pt}$ is added to the environment σ, which maps p to the formal parameter $\overline{pt}$. By abstracting on $\overline{pt}$, a 'transformer of predicate transformers':

$$\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\sigma\left[p \mapsto \overline{pt}\right],\ c\right) : \mathcal{PT} \ \rightarrow\ \mathcal{PT}$$

is obtained. Notice that the term $\boxed{\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\sigma\left[p \mapsto \overline{pt}\right],\ c\right)}^{\,n}$ in (18e) is no longer the expansion operation defined in (2), but a folding operation defined as:

$$\begin{cases} \boxed{ptt}^{\,0} \stackrel{\mathsf{def}}{\Longrightarrow} \lambda\,P\,.\,\mathsf{F} \\ \boxed{ptt}^{\,n+1} \stackrel{\mathsf{def}}{\Longrightarrow} ptt\left(\boxed{ptt}^{\,n}\right) \end{cases} \tag{19}$$

In place of $\bigvee_{n<\omega} \mathsf{wp}\left(\boxed{\mathsf{proc}\ p\ \equiv\ c}^{\,n}\right)$, we write

$$\exists\, n\,.\boxed{\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\sigma\left[p \mapsto \overline{pt}\right],\ c\right)}^{\,n}(P)\,(s)$$

which is semantically identical, but expressible in Coq.

The equation (18f) defines wpc for $\mathsf{pcall}\ p$, which is the predicate transformer assigned to p by σ.

As a sanity checking, three healthiness conditions, namely: Strictness, Monotonicity and Conjunctivity, are proved. For brevity, only the one, which is used in this paper, is listed here:

Lemma 1 (Generalized Monotonicity Lemma).

$$\begin{aligned} &\forall c. \\ &\quad (\sigma_1 \preccurlyeq_\sigma \sigma_2 \Rightarrow \mathrm{mono}(\sigma_1) \Rightarrow \mathrm{mono}(\sigma_2) \Rightarrow \\ &\qquad\qquad \mathsf{wpc}\,(\sigma_1,\ c) \preccurlyeq_{pt} \mathsf{wpc}\,(\sigma_2,\ c)) \wedge && (20a) \\ &\quad (\mathrm{mono}(\sigma) \Rightarrow \mathrm{mono}(\mathsf{wpc}\,(\sigma,\ c))) && (20b) \end{aligned}$$

The proof for other healthiness conditions can be found in the Coq scripts.

4 The operational semantics

To strengthen the justification of wp, an operational semantics is defined and a formal relationship between the operational semantics and wp is established. The operational semantics is given in Figure 3 as an inductively defined relation $s \xrightarrow[cs]{c} s'$, which means that: the execution of command c transforms program store from s to s'. The cs is the 'call stack' under which c is executed. The type of call stack $\mathcal{CS}$ is defined as:

$$\frac{}{\mathcal{CS} : \mathtt{Set}}\ \mathbf{\mathcal{CS}\_form} \qquad \frac{}{\vartheta : \mathcal{CS}}\ \mathbf{nil\_cs} \qquad \frac{p : \mathcal{ID} \quad c : \mathcal{C} \quad cs, \overline{cs} : \mathcal{CS}}{cs[p \rightsquigarrow (c, \overline{cs})] : \mathcal{CS}}\ \mathbf{cons\_cs} \tag{21}$$

where ϑ is the empty call stack, and $cs[p \rightsquigarrow (c, \overline{cs})]$ is the call stack obtained from cs by pushing $(p, c, \overline{cs})$.

In the rule **e_proc**, when a recursive procedure $\mathsf{proc}\ p \equiv c$ is executed, the operation $cs[p \rightsquigarrow (c, cs)]$ pushes procedure body c together with the call stack cs, under which the c is going to be executed onto the call stack, and then the procedure body c is executed.

In the rule **e_pcall**, when a recursive call $\mathsf{pcall}\ p$ is executed, the operation $\mathsf{lookup}(cs, p)$ is used to look up the procedure body being called and the call stack under which it is going to be executed. Suppose $(c, \overline{cs})$ is found, then c is executed under the call stack $\overline{cs}$.

The definition of lookup is:

$$\begin{cases} \mathsf{lookup}(cs[\overline{p} \rightsquigarrow (c, \overline{cs})], p) \stackrel{\mathsf{def}}{\Longrightarrow} (c, \overline{cs}) & \text{if } p = \overline{p} \\ \mathsf{lookup}(cs[\overline{p} \rightsquigarrow (c, \overline{cs})], p) \stackrel{\mathsf{def}}{\Longrightarrow} \mathsf{lookup}(cs, p) & \text{if } p \neq \overline{p} \\ \mathsf{lookup}(\vartheta, p) \stackrel{\mathsf{def}}{\Longrightarrow} \bot & \end{cases} \tag{22}$$

$$\frac{[\![e]\!]_s = \mathsf{true}}{s \xrightarrow[cs]{\mathsf{assert}(e)} s}\ \textbf{e\_asrt} \qquad \frac{[\![e]\!]_s = v}{s \xrightarrow[cs]{i:=e} s\left[v\ ^i\right]}\ \textbf{e\_asgn}$$

$$\frac{[\![e]\!]_s = \mathsf{true} \quad s \xrightarrow[cs]{c_1} s'}{s \xrightarrow[cs]{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2} s'}\ \textbf{e\_ifs\_t} \qquad \frac{[\![e]\!]_s = \mathsf{false} \quad s \xrightarrow[cs]{c_2} s'}{s \xrightarrow[cs]{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2} s'}\ \textbf{e\_ifs\_f}$$

$$\frac{s \xrightarrow[cs]{c_1} s' \quad s' \xrightarrow[cs]{c_2} s''}{s \xrightarrow[cs]{c_1;\ c_2} s''}\ \textbf{e\_seq} \qquad \frac{s \xrightarrow[cs[p \rightsquigarrow (c, cs)]]{c} s'}{s \xrightarrow[cs]{\mathsf{proc}\ p\ \equiv\ c} s'}\ \textbf{e\_proc}$$

$$\frac{\mathsf{lookup}(cs, p) = (c, \overline{cs}) \quad s \xrightarrow[\overline{cs}[p \rightsquigarrow (c, \overline{cs})]]{c} s'}{s \xrightarrow[cs]{\mathsf{pcall}\ p} s'}\ \textbf{e\_pcall}$$

Fig. 3. Definition of the operational semantics

5 Relating operational semantics to wp

The operational semantics can be related to wp by the following lemma:

Lemma 2 (operational semantics to wp).

$$s \xrightarrow[\vartheta]{c} s' \Rightarrow P(s') \Rightarrow \mathsf{wp}(c)(P)(s)$$

which says: if the execution of c under the empty call stack yields program store s', then for any predicate P, if P holds on s', then the predicate $\mathsf{wp}(c)(P)$ (the predicate P transformed by $\mathsf{wp}(c)$) holds on the initial program store s.

Instead of proving Lemma 2 directly, the following generalized lemma is proved first, and Lemma 2 is treated as a corollary of Lemma 3.

Lemma 3 ('operational semantics to wp' generalized).

$$s \xrightarrow[cs]{c} s' \Rightarrow \tag{23a}$$
$$\mathsf{can}(cs) \Rightarrow \tag{23b}$$
$$\exists\, n\, .$$
$$(\forall\, P\, .\, P\,(s') \Rightarrow \mathsf{wpc}\,(\{\!\!|cs|\!\!\}^n,\ c)\,(P)\,(s)) \tag{23c}$$

where the predicate can is used to constrain the form of cs, so that it can be guaranteed that the execution $s \xrightarrow[cs]{c} s'$ is a sub-execution of some top level execution $\overline{s} \xrightarrow[\vartheta]{\overline{c}} \overline{s'}$. Therefore, can is formalized as the following inductively defined predicate:

$$\frac{}{\mathsf{can}(\vartheta)}\ \textbf{can\_nil} \qquad \frac{\mathsf{can}(cs)}{\mathsf{can}(cs[p \rightsquigarrow (c, cs)])}\ \textbf{can\_cons} \tag{24}$$

The operation $\{\!\!|cs|\!\!\}^n$ is used to transform a call stack to environment. It is defined as:

$$\begin{cases} \{\!\!|\vartheta|\!\!\}^n \stackrel{\mathsf{def}}{\Longrightarrow} \varepsilon \\ \{\!\!|cs[p \rightsquigarrow (c, \overline{cs})]|\!\!\}^n \stackrel{\mathsf{def}}{\Longrightarrow} \\ \qquad \{\!\!|cs|\!\!\}^n \left[p \mapsto \boxed{\lambda\, \overline{pt}.\, \mathsf{wpc}\left(\{\!\!|\overline{cs}|\!\!\}^n \left[p \mapsto \overline{pt}\right],\ c\right)}^{\,n}\,\right] \end{cases} \tag{25}$$

The idea behind can and $\{\!\!|cs|\!\!\}^n$ is explained in detail in Section 5.1. The proof of Lemma 3 is given in Appendix A. The preliminary lemmas used in the proof are given in Appendix C.1.

Since $\mathsf{can}(\vartheta)$ is trivially true, by instantiating cs to ϑ, Lemma 2 follows directly from Lemma 3.

5.1 Informal explanation of Lemma 3

In (18e), the n is determined by the number of recursive calls to p during the execution of c. n can be any natural number larger than this number. By expanding wpc in $\mathsf{wpc}\,(\sigma,\ \mathsf{proc}\ p\ \equiv\ c)\,(P)\,(s)$, we have:

$$\exists\ n\, .\, \boxed{\lambda\, \overline{pt}.\, \mathsf{wpc}\left(\sigma\left[p \mapsto \overline{pt}\right],\ c\right)}^{\,n}\,(P)\,(s)$$

If $n = \overline{n} + 1$, then by expanding the definition of $\boxed{\cdots}^{\,\overline{n}+1}$, we have:

$$\mathsf{wpc}\left(\sigma\left[p \mapsto \boxed{\lambda\, \overline{pt}.\, \mathsf{wpc}\left(\sigma\left[p \mapsto \overline{pt}\right],\ c\right)}^{\,\overline{n}}\,\right],\ c\right)(P)\,(s) \tag{26}$$

The natural number $\overline{n}$ is the number of recursive calls of p during the execution of c starting from program store s.

An analysis of the operational semantics in Figure 3 may reveal that: if the execution $s \xrightarrow[cs]{c} s'$ is a sub-execution of some 'top level execution' $\overline{s} \xrightarrow[\vartheta]{\overline{c}} \overline{s'}$, then cs must be of the form:

$$\vartheta[p_0 \rightsquigarrow (c_0, cs_0)][p_1 \rightsquigarrow (c_1, cs_1)] \ldots [p_m \rightsquigarrow (c_m, cs_m)] \tag{27}$$

where the $[p_i \rightsquigarrow (c_i, cs_i)]$ $(i \in \{0, \ldots, m\})$ are pushed onto cs through the execution of $\mathsf{proc}\ p_i \equiv c_i$ $(i \in \{0, \ldots, m\})$. During the execution of c, there must be a number of recursive calls on each of the procedures p_i $(i \in \{0, \ldots, m\})$. Let these numbers be $n_{p_0}, n_{p_1}, \ldots, n_{p_m}$.

In the design of $\{|cs|\}^n$, inspired by the analysis in (26), we first intended to transform cs into:

$$\begin{aligned} &\varepsilon\,[p_0 \mapsto \mathsf{fd}(\{|cs_0|\}, n_{p_0}, p_0, c_0)]\,[p_1 \mapsto \mathsf{fd}(\{|cs_1|\}, n_{p_1}, p_1, c_1)] \ldots \\ &\qquad [p_m \mapsto \mathsf{fd}(\{|cs_m|\}, n_{p_m}, p_m, c_m)] \end{aligned} \tag{28}$$

where $\mathsf{fd}(\sigma, n, p, c)$ is the abbreviation defined as:

$$\mathsf{fd}(\sigma, n, p, c) \stackrel{\mathsf{def}}{\Longrightarrow} \boxed{\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\sigma\left[p \mapsto \overline{pt}\right],\ c\right)}^{\,n} \tag{29}$$

However, it is quite inconvenient to find the values for all the natural numbers $n_{p_0}, n_{p_1}, \ldots, n_{p_m}$. Fortunately, since $\mathsf{fd}(\sigma, n, p, c)$ is monotonous with respect to n, it is sufficient to deal only with the upper bound of them. It is usually easier to deal with one natural number than a group of natural numbers. Therefore, the cs is finally transformed into:

$$\begin{aligned} &\varepsilon\,[p_0 \mapsto \mathsf{fd}(\{|cs_0|\}^{n_{max}}, n_{max}, p_0, c_0)]\,[p_1 \mapsto \mathsf{fd}(\{|cs_1|\}^{n_{max}}, n_{max}, p_1, c_1)] \ldots \\ &\qquad [p_m \mapsto \mathsf{fd}(\{|cs_m|\}^{n_{max}}, n_{max}, p_m, c_m)] \end{aligned} \tag{30}$$

where n_{max} is a natural number larger than any n_{p_i} $(i \in \{0, \ldots, m\})$. The n in Lemma 3 is actually the n_{max} in (30). The equation (30) also explains the definition of $\{|cs|\}^n$ in (25).

6 Relating wp to operational semantics

wp can be related to the operational semantics by the following lemma:

Lemma 4 (wp to operational semantics).

$$\mathsf{wp}\,(c)\,(P)\,(s) \Rightarrow \exists\, s'\,.\left(s \xrightarrow[\vartheta]{c} s' \wedge P\,(s')\right)$$

which says: for any predicate P, if the transformation of any predicate P by $\mathsf{wp}\,(c)$ holds on s, then the execution of c under the empty call stack terminates and transforms program store from s to s', and P holds on s'.

Instead of proving Lemma 4 directly, the following generalized lemma is proved first and Lemma 4 follows as a corollary of Lemma 5.

Lemma 5 ('wp to operational semantics' generalized).

$$\begin{aligned}
&\mathsf{wpc}\,(\mathsf{envof}(ecs),\ c)\,(P)\,(s) \Rightarrow && \text{(31a)}\\
&\mathsf{ecan}(ecs) \Rightarrow && \text{(31b)}\\
&\exists\, s' \,.\\
&\quad (s \xrightarrow[\mathsf{csof}(ecs)]{c} s' \wedge && \text{(31c)}\\
&\quad P\,(s')) && \text{(31d)}
\end{aligned}$$

where, ecs is an 'extended call stack', the type of which – $\mathcal{ECS}$ is defined as:

$$\frac{}{\mathcal{ECS} : \mathsf{Type}}\ \textbf{ECS\_form} \qquad \frac{}{\theta : \mathcal{ECS}}\ \textbf{nil\_ecs}$$
$$\frac{p : \mathcal{ID} \quad c : \mathcal{C} \quad \overline{ecs} : \mathcal{ECS} \quad pt : \mathcal{PT} \quad ecs : \mathcal{ECS}}{ecs[p \hookrightarrow (c, \overline{ecs}, pt)] : \mathcal{ECS}}\ \textbf{cons\_ecs} \tag{32}$$

where θ is the 'empty extended call stack' and $ecs[p \hookrightarrow (c, \overline{ecs}, pt)]$ is the extended call stack obtained by adding $(p, c, \overline{ecs}, pt)$ to the head of ecs. It is obvious from the definition of $\mathcal{ECS}$ that an extended call stack is a combination of call stack and environment, with each procedure name p being mapped to a triple $(c, \overline{ecs}, pt)$. Therefore, by forgetting pt in the triple, a normal call stack is obtained. This is implemented by the operation $\mathsf{csof}(ecs)$:

$$\begin{cases} \mathsf{csof}(\theta) \overset{\mathsf{def}}{\Longrightarrow} \vartheta \\ \mathsf{csof}(ecs[p \hookrightarrow (c, \overline{ecs}, pt)]) \overset{\mathsf{def}}{\Longrightarrow} \mathsf{csof}(ecs)[p \rightsquigarrow (c, \mathsf{csof}(\overline{ecs}))] \end{cases} \tag{33}$$

By forgetting $c, \overline{ecs}$ in the triple, a normal environment is obtained. This is implemented by the operation $\mathsf{envof}(ecs)$:

$$\begin{cases} \mathsf{envof}(\theta) \overset{\mathsf{def}}{\Longrightarrow} \varepsilon \\ \mathsf{envof}(ecs[p \hookrightarrow (c, \overline{ecs}, pt)]) \overset{\mathsf{def}}{\Longrightarrow} \mathsf{envof}(ecs)\,[p \mapsto pt] \end{cases} \tag{34}$$

The operation $\mathsf{lookup\_ecs}(p, ecs)$ is defined to lookup (in ecs) the triple $(c, \overline{ecs}, pt)$ mapped to p:

$$\begin{cases} \mathsf{lookup\_ecs}(p, ecs[\overline{p} \hookrightarrow (c, \overline{ecs}, pt)]) \overset{\mathsf{def}}{\Longrightarrow} (\overline{p}, c, \overline{ecs}, pt) & \text{if } p = \overline{p} \\ \mathsf{lookup\_ecs}(p, ecs[\overline{p} \hookrightarrow (c, \overline{ecs}, pt)]) \overset{\mathsf{def}}{\Longrightarrow} \mathsf{lookup\_ecs}(p, ecs) & \text{if } p \neq \overline{p} \\ \mathsf{lookup\_ecs}(p, \theta) \overset{\mathsf{def}}{\Longrightarrow} \bot \end{cases} \tag{35}$$

The notation $\perp$ is overloaded here, it is an undefined value in exceptional type, instead of exceptional set.

The predicate $\mathsf{ecan}(ecs)$ is defined to constrain the form of ecs, so that in each triple $(c, \overline{ecs}, pt)$, pt can be related to the $(c, \overline{ecs})$ in the following sense:

$$\mathsf{ecan}(ecs) \stackrel{\mathsf{def}}{\Longrightarrow} \mathsf{lookup\_ecs}(p, ecs) = (c, \overline{ecs}, pt) \Rightarrow \\ pt\,(P)\,(s) \Rightarrow \exists\, s' \,.\, \left(s \xrightarrow[\mathsf{csof}(\overline{ecs})]{\mathsf{proc}\ p\ \equiv\ c} s' \wedge P\,(s') \right) \quad (36)$$

Since $\mathsf{lookup\_ecs}(p, \theta) = \perp$, it clear that $\mathsf{ecan}(\theta)$ holds. Therefore, by instantiating ecs to θ, Lemma 4 can be proved as a corollary of Lemma 5.

The proof of Lemma 5 is given in Appendix B. The preliminary lemmas used in the proof are given in Appendix C.2.

7 Conclusion

We have given a computational embedding of wp in Coq. The definition is verified by relating it to an operational semantics. Since such a style of embedding has the benefits of both deep and shallow embedding, it can be used to verify both program transformations and concrete programs.

Laibinis and Wright [9] treats general recursion in HOL. But that is a shallow embedding and there is no relation between wp and operational semantics.

There have been some efforts to verification imperative programs using type theory. Fillitre [6] implemented an extension of Coq to generate proof obligations from annotated imperative programs. The proof of these proof obligations in Coq will guarantee the correctness of the annotated imperative programs. Since it uses shallow embedding, meta programming (such as program transformation) can not be verified in Fillitre's setting.

Kleymann [7] derived Hoare logic directly from operational semantics. Since Kleymann's treatment is a deep embedding, program transformations can be verified. However, because the operational semantics is formalized as an inductive relation (rather than using computation), verifying concrete programs in Kleymann's setting is not very convenient. This paper can be seen as an effort to overcome this problem through a computational treatment of wp. In our setting, computation mechanism can be used to simplify proof obligations when verifying concrete programs. Hoare triple can be defined as:

$$\{P_1\}\ c\ \{P_2\} \stackrel{\mathsf{def}}{\Longrightarrow} P_1\,(s) \Rightarrow \mathsf{wp}\,(c)\,(P_2)\,(s) \quad (37)$$

From this, Hoare logic rules for structure statements can be derived. For example, the rule for $\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$ is:

Lemma 6 (The proof rule for $\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$).

$$\{\lambda\, s.\, ([\![e]\!]_s = \mathsf{true} \wedge P_1\,(s))\}\ c_1\ \{P_2\} \Rightarrow \\ \{\lambda\, s.\, ([\![e]\!]_s = \mathsf{false} \wedge P_1\,(s))\}\ c_2\ \{P_2\} \Rightarrow \\ \{P_1\}\ \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \{P_2\}$$

Hoare logic rules can be used to propagate proof obligations from parent statements to its sub-statements. When the propagation process reaches atomic statements (such as $i := e$), by expanding the definition of wp, the proof obligation can be simplified by the computation mechanism. Since users have access to the definition of wp all the time, they can choose whatever convenient for their purpose, either Hoare logic rules (such as Lemma 6) or the direct definition of wp.

We have gone as far as the verification of insertion sorting program. Admittedly, verification of concrete program in our setting is slightly complex than in Fillitre's. However, the ability to verify both program transformations and concrete programs makes our approach unique. The treatment of program verification in our setting will be detailed in a separate paper.

Acknowledgement. I must first thank Prof. Zhaohui Luo for a lot of technical directions into type theory. Dr. James McKinna provided many good comments on an earlier version of this paper. The anonymous referees are thanked for the helpful suggestions of improvement.

A Proof of Lemma 3

The proof is by induction on the structure of the $s \xrightarrow[cs]{c} s'$ in (23a). Some preliminary lemmas used in the proof are listed in Appendix C.1. There is one case corresponding to each execution rule. For brevity, only two of the more interesting cases are discussed here:

1. When the execution is constructed using **e_proc**, we have $c = (\mathsf{proc}\ p \ \equiv\ \overline{c})$ and $s \xrightarrow[cs[p \rightsquigarrow (\overline{c}, cs)]]{\overline{c}} s'$. From the induction hypothesis for this execution and (23b), it can be derived that:

$$\exists\, \overline{n}\,.\,\forall\, P\,.\, P\,(s') \Rightarrow \mathsf{wpc}\left(\{\!|cs[p \rightsquigarrow (\overline{c}, cs)]|\!\}^{\overline{n}},\ \overline{c}\right)(P)\,(s) \tag{38}$$

   By assigning $\overline{n}$ to n and expanding the definition of wpc, the goal (23c) becomes:

$$\forall P\,.\, P\,(s') \Rightarrow \exists\, n\,.\, \boxed{\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\{\!|cs|\!\}^{\overline{n}}\left[p \mapsto \overline{pt}\right],\ \overline{c}\right)}^{\,n}\,(P)\,(s) \tag{39}$$

   By assigning $\overline{n}+1$ to n and expanding the definition of $\boxed{\cdots}^{\,\overline{n}+1}$, it becomes:

$$\forall\, P\,.\, P\,(s') \Rightarrow \mathsf{wpc}\left(\{\!|cs|\!\}^{\overline{n}}\left[p \mapsto \boxed{\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\{\!|cs|\!\}^{\overline{n}}\left[p \mapsto \overline{pt}\right],\ \overline{c}\right)}^{\,\overline{n}}\right],\ \overline{c}\right)(P)\,(s) \tag{40}$$

   which is exactly (38) with the definition of $\{\!|\cdots|\!\}^{\overline{n}}$ expanded.
2. When the execution is constructed using **e_proc**, we have $c = (\mathsf{pcall}\ p)$ and

$$\mathsf{lookup}(cs, p) = (\overline{c}, \overline{cs}) \tag{41a}$$

$$s \xrightarrow[\overline{cs}[p \rightsquigarrow (\overline{c}, \overline{cs})]]{\overline{c}} s' \tag{41b}$$

By applying Lemma 7 to (23b) and (41a), it can be deduced that $\mathsf{can}(\overline{cs})$, from which, $\mathsf{can}(\overline{cs}[p \rightsquigarrow (\overline{c}, \overline{cs})])$ can be deduced. By applying induction hypothesis for (41b) to this, it can be deduced that:

$$\exists\, \overline{n} \,.\, \forall\, P \,.\, P\,(s') \Rightarrow \mathsf{wpc}\left(\{\!|\overline{cs}|\!\}^{\overline{n}}\left[p \mapsto \boxed{\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\{\!|\overline{cs}|\!\}^{\overline{n}}\left[p \mapsto \overline{pt}\right],\ \overline{c}\right)}^{\overline{n}}\right],\ \overline{c}\right)(P)\,(s) \tag{42}$$

By assigning $\overline{n}+1$ to n, the goal (23c) is specialized to:

$$\overline{P}\,(s') \Rightarrow \{\!|cs|\!\}^{\overline{n}+1}\,(p)\,\left(\overline{P}\right)(s) \tag{43}$$

By applying Lemma 8 to (41a) and expanding the definition of $\boxed{\cdots}^{\overline{n}+1}$, it can be deduced that:

$$\{\!|cs|\!\}^{\overline{n}+1}\,(p) = \mathsf{wpc}\left(\{\!|\overline{cs}|\!\}^{\overline{n}+1}\left[p \mapsto \boxed{\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\{\!|\overline{cs}|\!\}^{\overline{n}+1}\left[p \mapsto \overline{pt}\right],\ \overline{c}\right)}^{\overline{n}}\right],\ \overline{c}\right) \tag{44}$$

After rewritten using (44), the goal (43) becomes:

$$\overline{P}\,(s') \Rightarrow \mathsf{wpc}\left(\{\!|\overline{cs}|\!\}^{\overline{n}+1}\left[p \mapsto \boxed{\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\{\!|\overline{cs}|\!\}^{\overline{n}+1}\left[p \mapsto \overline{pt}\right],\ \overline{c}\right)}^{\overline{n}}\right],\ \overline{c}\right)\left(\overline{P}\right)(s) \tag{45}$$

By applying (42) to the $\overline{P}\,(s')$ in (45), we have:

$$\mathsf{wpc}\left(\{\!|\overline{cs}|\!\}^{\overline{n}}\left[p \mapsto \boxed{\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\{\!|\overline{cs}|\!\}^{\overline{n}}\left[p \mapsto \overline{pt}\right],\ \overline{c}\right)}^{\overline{n}}\right],\ \overline{c}\right)\left(\overline{P}\right)(s) \tag{46}$$

By applying Lemma 9 to the fact that $\overline{n} \le \overline{n}+1$ and (23b), it can be deduced that $\{\!|\overline{cs}|\!\}^{\overline{n}} \preccurlyeq_{\sigma} \{\!|\overline{cs}|\!\}^{\overline{n}+1}$. By applying Lemma 10 to this, it can be deduced that:

$$\boxed{\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\{\!|\overline{cs}|\!\}^{\overline{n}}\left[p \mapsto \overline{pt}\right],\ \overline{c}\right)}^{\overline{n}} \preccurlyeq_{pt} \boxed{\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\{\!|\overline{cs}|\!\}^{\overline{n}+1}\left[p \mapsto \overline{pt}\right],\ \overline{c}\right)}^{\overline{n}} \tag{47}$$

By combining $\{\!|\overline{cs}|\!\}^{\overline{n}} \preccurlyeq_{\sigma} \{\!|\overline{cs}|\!\}^{\overline{n}+1}$ and (47), it can be deduced that:

$$\begin{aligned}&\{\!|\overline{cs}|\!\}^{\overline{n}}\left[p \mapsto \boxed{\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\{\!|\overline{cs}|\!\}^{\overline{n}}\left[p \mapsto \overline{pt}\right],\ \overline{c}\right)}^{\overline{n}}\right] \preccurlyeq_{\sigma}\\ &\qquad \{\!|\overline{cs}|\!\}^{\overline{n}+1}\left[p \mapsto \boxed{\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\{\!|\overline{cs}|\!\}^{\overline{n}+1}\left[p \mapsto \overline{pt}\right],\ \overline{c}\right)}^{\overline{n}}\right]\end{aligned} \tag{48}$$

By applying (20a) to this, it can be deduced that:

$$\begin{aligned}&\mathsf{wpc}\left(\{\!|\overline{cs}|\!\}^{\overline{n}}\left[p \mapsto \boxed{\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\{\!|\overline{cs}|\!\}^{\overline{n}}\left[p \mapsto \overline{pt}\right],\ \overline{c}\right)}^{\overline{n}}\right],\ \overline{c}\right) \preccurlyeq_{pt}\\ &\qquad \mathsf{wpc}\left(\{\!|\overline{cs}|\!\}^{\overline{n}+1}\left[p \mapsto \boxed{\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\{\!|\overline{cs}|\!\}^{\overline{n}+1}\left[p \mapsto \overline{pt}\right],\ \overline{c}\right)}^{\overline{n}}\right],\ \overline{c}\right)\end{aligned} \tag{49}$$

From this and (46), the goal (45) can be proved.

B Proof of Lemma 5

The proof is by induction on the structure of c. Some preliminary lemmas used in the proof are listed in Appendix C.2. There is one case for each type of command. For brevity, only two of the more interesting cases are discussed here:

1. When $c = (\mathsf{proc}\ p\ \equiv\ \overline{c})$, after expanding the definition of wpc, the premise (31a) becomes:

$$\exists\, \overline{n} \,.\, \forall\, P \,.\, \boxed{\lambda\, \overline{pt}.\, \mathsf{wpc}\left(\mathsf{envof}(ecs)\left[p \mapsto \overline{pt}\right],\ \overline{c}\right)}^{\overline{n}} (P)\,(s) \tag{50}$$

A nested induction on $\overline{n}$ is used to prove the goal, which gives rise to two cases:

(a) When $\overline{n} = 0$, this case can be refuted. Since

$$\boxed{\lambda\, \overline{pt}.\, \mathsf{wpc}\left(\mathsf{envof}(ecs)\left[p \mapsto \overline{pt}\right],\ \overline{c}\right)}^{\overline{n}}$$

reduces to $\lambda\, P \,.\, \mathsf{F}$, it can not hold on P and s. And this is in contradiction with (50).

(b) When $\overline{n} = \underline{n}{+}1$, after expanding the definition of $\boxed{\cdots}^{\underline{n}+1}$, (50) becomes:

$$\mathsf{wpc}\left(\mathsf{envof}(ecs)\left[p \mapsto \boxed{\lambda\, \overline{pt}.\, \mathsf{wpc}\left(\mathsf{envof}(ecs)\left[p \mapsto \overline{pt}\right],\ c\right)}^{\underline{n}}\right],\ \overline{c}\right)(P)\,(s) \tag{51}$$

which is exactly

$$\mathsf{wpc}\left(\mathsf{envof}(ecs[p \hookrightarrow (\overline{c}, ecs, \boxed{\lambda\, \overline{pt}.\, \mathsf{wpc}\left(\mathsf{envof}(ecs)\left[p \mapsto \overline{pt}\right],\ \overline{c}\right)}^{\underline{n}})]),\ \overline{c}\right)(P)\,(s) \tag{52}$$

with the definition of $\mathsf{envof}(\cdots)$ expanded. From the nested induction hypothesis for $\underline{n}$, it can be proved that:

$$\mathsf{ecan}(ecs[p \hookrightarrow (\overline{c}, ecs, \boxed{\lambda\, \overline{pt}.\, \mathsf{wpc}\left(\mathsf{envof}(ecs)\left[p \mapsto \overline{pt}\right],\ \overline{c}\right)}^{\underline{n}})])$$

By applying the main induction hypothesis to this and (52), after expanding the definition of csof, it can be deduced that:

$$\exists\, \underline{s} \,.\, s \xrightarrow[\mathsf{csof}(ecs)[p \rightsquigarrow (\overline{c}, \mathsf{csof}(ecs))]]{\overline{c}} \underline{s}\ \ \wedge P\,(\underline{s}) \tag{53}$$

By assigning $\underline{s}$ to s', the goal (31d) can be proved directly from the $P\,(\underline{s})$ in (53). Also the goal (31c) can be proved by applying **e_proc** to the $s \xrightarrow[\mathsf{csof}(ecs)[p \rightsquigarrow (\overline{c}, \mathsf{csof}(ecs))]]{\overline{c}} \underline{s}$ in (53).

2. When $c = (\mathsf{pcall}\ p)$, after expanding the definition of wpc, the premise (31a) becomes $\mathsf{envof}(ecs)\,(p)\,(P)\,(s)$. By applying Lemma 11 to this, we have:

$$\exists\, \overline{c}, \overline{ecs} \,.\, \mathsf{lookup\_ecs}(p, ecs) = (\overline{c}, \overline{ecs}, \mathsf{envof}(ecs)\,(p)) \tag{54}$$

After expanding the definition of ecan in (31b), it can be applied to (54) and the $\mathsf{envof}(ecs)\,(p)\,(P)\,(s)$ at the beginning to yield:

$$\exists\, \underline{s}\,.\; s \xrightarrow[\mathsf{csof}(ecs)]{\mathsf{proc}\ p\ \equiv\ \overline{c}} \underline{s} \;\;\wedge\;\; P\,(\underline{s}) \tag{55}$$

By assigning $\underline{s}$ to s', the goal (31d) can be proved directly from the $P\,(\underline{s})$ in (55). By inversion on the $s \xrightarrow[\mathsf{csof}(ecs)]{\mathsf{proc}\ p\ \equiv\ \overline{c}} \underline{s}$ in (55), we have: $s \xrightarrow[\mathsf{csof}(ecs)[p\rightsquigarrow(\overline{c},\mathsf{csof}(ecs))]]{\overline{c}} \underline{s}$. By applying Lemma 12 to (54), we have: $\mathsf{lookup}(p, \mathsf{csof}(ecs)) = (\overline{c}, \mathsf{csof}(\overline{ecs}))$. Therefore, the goal (31c) can be proved by applying **e_pcall** to these two results.

C Preliminary lemmas

C.1 Lemmas used in the proof of Lemma 3

Lemma 7. $\mathsf{can}(cs) \Rightarrow \mathsf{lookup}(p, cs) = (\overline{c}, \overline{cs}) \Rightarrow \mathsf{can}(\overline{cs})$

Lemma 8. $\mathsf{lookup}(p, cs) = (\overline{c}, \overline{cs}) \Rightarrow \{\!|cs|\!\}^n\,(p) = \boxed{\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\{\!|\overline{cs}|\!\}^n\left[p \mapsto \overline{pt}\right],\ \overline{c}\right)}^{\,n}$

Lemma 9. $n_1 \leq n_2 \Rightarrow \mathsf{can}(cs) \Rightarrow \{\!|cs|\!\}^{n_1} \preccurlyeq_\sigma \{\!|cs|\!\}^{n_2}$

Lemma 10.

$$\sigma_1 \preccurlyeq_\sigma \sigma_2 \Rightarrow \mathrm{mono}(\sigma_1) \Rightarrow \mathrm{mono}(\sigma_2) \Rightarrow$$
$$\boxed{\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\sigma_1\left[p \mapsto \overline{pt}\right],\ c\right)}^{\,n} \preccurlyeq_{pt} \boxed{\lambda\,\overline{pt}.\,\mathsf{wpc}\left(\sigma_2\left[p \mapsto \overline{pt}\right],\ c\right)}^{\,n}$$

C.2 Lemmas used in the proof of Lemma 5

Lemma 11.

$$\mathsf{envof}(ecs)\,(p)\,(P)\,(s) \Rightarrow \exists\,\overline{c}, \overline{ecs}\,.$$
$$\mathsf{lookup\_ecs}(p, ecs) = (\overline{c}, \overline{ecs}, \mathsf{envof}(ecs)\,(p))$$

Lemma 12. $\mathsf{lookup\_ecs}(p, ecs) = (\overline{c}, \overline{ecs}, \overline{pt}) \Rightarrow \mathsf{lookup}(p, \mathsf{csof}(ecs)) = (\overline{c}, \mathsf{csof}(\overline{ecs}))$

References

1. R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, August 1988.

2. B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Mu noz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997.
3. D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. Refinement in Ergo. Technical report 94-44, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane 4072. Australia, November 1994.
4. T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:96–120, 1988.
5. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
6. J.-C. Filliâtre. Proof of Imperative Programs in Type Theory. In *International Workshop, TYPES '98, Kloster Irsee, Germany*, volume 1657 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1998.
7. T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. Ph.D. thesis, University of Edinburgh, 1998.
8. J. Knappmann. *A PVS based tool for developing programs in the refinement calculus*. Marster's Thesis, Christian-Albrechts-University, 1996.
9. L. Laibinis and J. von Wright. Functional procedures in higher-order logic. Technical Report TUCS-TR-252, Turku Centre for Computer Science, Finland, March 15, 1999.
10. L. Lindqvist. *A formalization of Dijkstra's predicate transformer* wp *in Martin-Lof type theory*. Master's Thesis, Linkopin University, Sweden, 1997.
11. Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Number 11 in International Series of Monographs on Computer Science. Oxford University Press, 1994.
12. C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.
13. J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.
14. T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 180–192. Springer-Verlag LNCS 1180, 1996.
15. B. Nordström, K. Peterson, and J. M. Smith. *Programming in Martin-Lof's Type Theory*, volume 7 of *International Series of Monographs on Computer Science*. Oxford University Press, New York, NY, 1990.
16. R. J. R. Back and J. von Wright. Refinement concepts formalized in higher-order logic. Reports on Computer Science & Mathematics Series A—85, Institutionen för Informationsbehandling & Mathematiska Institutet, Åbo Akademi, Lemminkäinengatan 14, SF-20520 Turku, Finland, September 1989.
17. M. Staples. *A Mechanised Theory of Refinement*. Ph.D. Dissertation, Computer Laboratory, University of Cambridge, 1998.
18. M. Staples. Program transformations and refinements in HOL. In Y. Bertot G. Dowek, C. Paulin, editor, *TPHOLs: The 12th International Conference on Theorem Proving in Higher-Order Logics. LNCS, Springer-Verlag.*, 1999.
19. J. von Wright and K. Sere. Program transformations and refinements in HOL. In Myla Archer, Jennifer J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *Proceedigns of the International Workshop on the HOL Theorem Proving System and its Applications*, pages 231–241, Los Alamitos, CA, USA, August 1992. IEEE Computer Society Press.