A Formally Specified Type System and Operational Semantics for Higher-Order Procedural Variables

Tristan Crolard, Emmanuel Polonowski

▶ **To cite this version:**

Tristan Crolard, Emmanuel Polonowski. A Formally Specified Type System and Operational Semantics for Higher-Order Procedural Variables. 2009. hal-00385416

HAL Id: hal-00385416
https://hal.science/hal-00385416v1

Submitted on 19 May 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Formally Specified Type System and Operational Semantics for Higher-Order Procedural Variables

Tristan Crolard **Emmanuel Polonowski**

May 2009

TR–LACL–2009–3

Laboratoire d'Algorithmique, Complexité et Logique (LACL)
Département d'Informatique
Université Paris 12 – Val de Marne, Faculté des Science et Technologie
61, Avenue du Général de Gaulle, 94010 Créteil cedex, France
Tel.: (33)(1) 45 17 16 47, Fax: (33)(1) 45 17 66 01

Laboratory of Algorithmics, Complexity and Logic (LACL)
University Paris 12 (Paris Est)

Technical Report **TR–LACL–2009–3**

T. Crolard, E. Polonowski.

A Formally Specified Type System and Operational Semantics for Higher-Order Procedural Variables

© T. Crolard, E. Polonowski, May 2009.

A Formally Specified Type System and Operational Semantics for Higher-Order Procedural Variables

T. Crolard and E. Polonowski

May 14, 2009

Abstract

We formally specified the type system and operational semantics of Loop$^\omega$ with Ott and Isabelle/HOL proof assistant. Moreover, both the type system and the semantics of Loop$^\omega$ have been tested using Isabelle/HOL program extraction facility for inductively defined relations. In particular, the program that computes the Ackermann function type checks and behaves as expected. The main difference (apart from the choice of an Ada-like concrete syntax) with Loop$^\omega$ comes from the treatment of parameter passing. Indeed, since Ott does not currently fully support α-conversion, we rephrased the operational semantics with explicit aliasing in order to implement the **out** parameter passing mode.

Introduction

We formally specified the type system and operational semantics of Loop$^\omega$ as described in [CPV09] with Ott [SNO+07] and Isabelle/HOL proof assistant [NPW02]. Moreover, both the type system and the semantics of Loop$^\omega$ have been tested using Isabelle/HOL program extraction facility for inductively defined relations [BN02]. In particular, the program that computes the Ackermann function (reproduced below) type checks and behaves as expected.

The main difference (apart from the choice of an Ada-like concrete syntax) with the description given in [CPV09] comes from the treatment of parameter passing. Indeed, since Ott does not currently fully support α-conversion, we rephrased the operational semantics with explicit aliasing in order to implement the **out** parameter passing mode (instead of a simpler substitution-based semantics as in [CPV09]). On the other hand, the **in** parameter passing mode is implemented exactly as in [CPV09] and relies on Ott generated substitution (see the Isabelle/HOL code given in appendix).

Section 1 contains the description of an Ada-like grammar for Loop$^\omega$. We then present the type system in Section 2 and the structural operational semantic in section 3. Finally, in the appendix we include the Isabelle/HOL theory generated by Ott (all source files are available on request).

Example: the Ackermann function

```
procedure Ack(M : in int; N : in int; R : out int) is
    P : proc(in int, out int) := Incr;
begin
    for I in 1 . . M loop
      declare
        Q : constant proc(in int, out int) := P;
        procedure Aux(S : in int; R : out int) is
          X : int := 0;
        begin
          Q(1, X);
          for J in 1 . . S loop
            Q(X, X);
          end loop;
          R := X;
        end;
      begin
        P := Aux;
      end;
    end loop;
    P(N, R);
end;
```

1 Syntax

index, i, j, l, n			indices
ident, x, y, z, p, f			idents
number, q			
terminals	::=		
	|	$\longrightarrow$	
	|	$\rightarrow$	
	|	$\Rightarrow$	
	|	$\leftarrow$	
	|	$\mapsto$	
	|	$\rightsquigarrow$	
	|	$\vdash$	
	|	$\varnothing$	
	|	$\times$	
	|	$\neq$	
	|	$:=$	
	|	$\langle$	
	|	$\rangle$	
	|	$\sim$	
	|	$\notin$	
	|	$\triangleright$	

mode, m	::=			modes:
	|		S	
	|	in		
	|	out		
	|	in out		

integer, k	::=		
	|	q	
	|	$\{ k_1 + k_2 \}$	
	|	$\{ k_1 - k_2 \}$	
	|	$\{ k_1 \times k_2 \}$	

boolean, b	::=		
	|	true	
	|	false	
	|	$\{ b_1$ and $b_2 \}$	
	|	$\{ b_1$ or $b_2 \}$	
	|	$\{$ not $b \}$	
	|	$\{ k_1 = k_2 \}$	
	|	$k_1 > k_2$	
	|	$k_1 < k_2$	

exp, e	::=			terms:
	|	x		var
	|	v		value
	|	$e_1 + e_2$		addition
	|	$e_1 - e_2$		subtraction
	|	$e_1 \times e_2$		multiplication
	|	$e_1 = e_2$		equality
	|	$e_1 > e_2$		greater
	|	$e_1 < e_2$		less
	|	e_1 and e_2		conjunction
	|	e_1 or e_2		disjunction
	|	not e		negation
	|	$(\ e\ )$	S	parentheses

store, μ	::=	store

	?	S	
	[]		
	$(\,\mu\,,\,x\,\leftarrow\,v\,)$		
	$[\,z_1\,\leftarrow\,v_1\,,\,\ldots\,,\,z_n\,\leftarrow\,v_n\,]$		
trace, *tr*	::=		
	?	S	
	$[\,\langle\,c_1\,,\,\mu_1\,\rangle\,..\,\langle\,c_n\,,\,\mu_n\,\rangle\,]$		
formula	::=		
	$formula_1\quad..\quad formula_n$		
	judgement		
	$x\,=\,x'$		
	$x\,\neq\,x'$		
	$\delta\,=\,\delta'$		
	$\delta\,\neq\,\delta'$		
	$m\,=\,m'$		
	$m\,\neq\,m'$		
	$k\,>\,k'$		
	$k\,\leq\,k'$		
env, Γ	::=		contexts:
	$\{\}$		empty context
	$\{\,x_1\,\delta_1\,,\,\ldots\,,\,x_n\,\delta_n\,\}$		explicit context
	$\Gamma\,,\,x\,\delta$		ident declaration
	Γ	S	parentheses
	$\Gamma\,,\,x_1\,\delta_1\,,\,....\,,\,x_n\,\delta_n$		idents declaration
	$\Gamma\,,\,\delta$		anonymous declaration
cmd, *c*	::=		commands:
	null		null
	$x\,:=\,e$		assignment
	$c_1\,;\,c_2$		sequence
	if e then c_1 ; else c_2 ; end if		conditional
	while e loop c ; end loop		while loop
	$(\,c\,)$	S	
	?	S	
	declare d		declaration
	for x in $e\,..\,e'$ loop c ; end loop	bind x in c	for loop
	$e\,(\,e_1\,,\,..\,,\,e_n\,)$		Procedure call
va, *v*	::=		constants:
	?	S	
	k		integer constant
	b		boolean true
	proc $(\,x_1\,:\,m_1\,\tau_1\,;\,..\,;\,x_n\,:\,m_n\,\tau_n\,)$ is d	bind $x_1..x_n$ in d	
ty, τ	::=		types:
	int		
	bool		
	proc $(\,m_1\,\tau_1\,,\,..\,,\,m_n\,\tau_n\,)$		Procedure
	void		void
δ	::=		
	$:\,m\,\tau$		
	$\hookrightarrow\,\tau$		
dcl, *d*	::=		

	$d\,[\,v\,/\,x\,]$	M	
	begin end		Empty
	begin c ; end		Block
	$x\ :\ \tau\,;\ d$	bind x in d	Uninit. variable
	$x\ :\ \tau\ :=\ e\,;\ d$	bind x in d	Init. variable
	$x\ :$ constant $\tau\ :=\ e\,;\ d$	bind x in d	Constant
	procedure $p\,(\,x_1\ :\ m_1\,\tau_1\,;\,..\,;\,x_n\ :\ m_n\,\tau_n\,)$ is $d_1\,;\ d_2$	bind $x_1..x_n$ in d_1	Proc
	$[\,x_1\ :\ m_1\,\tau_1\ =\ e_1\,,\,..\,,\,x_n\ :\ m_n\,\tau_n\ =\ e_n\,]\,d$	bind $x_1..x_n$ in d	Aliases
	$(\,x\ :\ m\,\tau\ =\ e\,)\,d$	bind x in d	Alias

value ::=

	v	value

eval_exp ::=

	$\mu\,(\,x\,)\ =\ v$	Fetch
	$e\ =_\mu\ v$	Expression evaluation

typing ::=

	$x\ \delta\ \in\ \Gamma$	Lookup
	$\Gamma\ \vdash\ e\ :\ \tau$	Expression typing
	$\delta\ \in\ \Gamma$	LookupD
	$\Gamma\ \vdash\ e\ \sim\ m\,\tau$	Match
	$\Gamma\ \vdash\ (\,e_1\,,\,..\,,\,e_l\,)\ \sim\ (\,m_1\,\tau_1\,,\,..\,,\,m_n\,\tau_n\,)$	MatchList
	$\Gamma\ \vdash\ d$: decl	Declaration typing
	$\Gamma\ \vdash\ c$: comm	Command typing

eval_comm ::=

	$\mu\,\{\,x\ \leftarrow\ v\,\}\ \mapsto\ \mu'$	Store Update
	$\langle c, \mu\rangle \mapsto^k \langle c', \mu'\rangle$	Many Steps
	$\langle c; \mu\rangle \Rightarrow^k tr$	Trace
	$\langle\ c\,;\,\mu\ \rangle\ \rightsquigarrow\ \mu'$	Full evaluation
	$(\,\lvert x_i' : m_i'\tau_i'\rvert\,)\,\#\,(\,\lvert e_j'\rvert\,)\ =\ [\,\lvert x_n : m_n\tau_n = e_n\rvert\,]$	Compatibility
	$\langle c, \mu\rangle \mapsto \langle c', \mu'\rangle$	One step evaluation
	$\langle\ d\,,\,\mu\ \rangle\ \mapsto\ \langle\ d'\,,\,\mu'\ \rangle$	Declaration evaluation

judgement ::=

	eval_exp
	typing
	eval_comm

user_syntax ::=

	index
	ident
	number
	terminals
	mode
	integer
	boolean
	exp
	store
	trace
	formula
	env
	cmd
	va
	ty
	δ
	dcl
	value

2 Type System

Lookup $\boxed{x\ \delta\ \in\ \Gamma}$

$$\frac{}{x\ \delta\ \in\ \Gamma\,,\, x\,\delta}\quad (\mathit{Lookup1})$$

$$\frac{x\ \neq\ x' \qquad x\ \delta\ \in\ \Gamma}{x\ \delta\ \in\ \Gamma\,,\, x'\,\delta'}\quad (\mathit{Lookup2})$$

Expression typing $\boxed{\Gamma\ \vdash\ e\ :\ \tau}$

$$\frac{m\ \neq\ \mathsf{out} \qquad x\ :\ m\,\tau\ \in\ \Gamma}{\Gamma\ \vdash\ x\ :\ \tau}\quad (\mathit{Var})$$

$$\frac{}{\Gamma\ \vdash\ q\ :\ \mathsf{int}}\quad (\mathit{IntCst})$$

$$\frac{}{\Gamma\ \vdash\ \mathsf{true}\ :\ \mathsf{bool}}\quad (\mathit{BoolTrue})$$

$$\frac{}{\Gamma\ \vdash\ \mathsf{false}\ :\ \mathsf{bool}}\quad (\mathit{BoolFalse})$$

$$\frac{\Gamma\ \vdash\ e_1\ :\ \mathsf{int} \qquad \Gamma\ \vdash\ e_2\ :\ \mathsf{int}}{\Gamma\ \vdash\ e_1\ +\ e_2\ :\ \mathsf{int}}\quad (\mathit{Plus})$$

$$\frac{\Gamma\ \vdash\ e_1\ :\ \mathsf{int} \qquad \Gamma\ \vdash\ e_2\ :\ \mathsf{int}}{\Gamma\ \vdash\ e_1\ -\ e_2\ :\ \mathsf{int}}\quad (\mathit{Minus})$$

$$\frac{\Gamma\ \vdash\ e_1\ :\ \mathsf{int} \qquad \Gamma\ \vdash\ e_2\ :\ \mathsf{int}}{\Gamma\ \vdash\ e_1\ \times\ e_2\ :\ \mathsf{int}}\quad (\mathit{Times})$$

$$\frac{\Gamma\ \vdash\ e_1\ :\ \tau \qquad \Gamma\ \vdash\ e_2\ :\ \tau}{\Gamma\ \vdash\ e_1\ =\ e_2\ :\ \mathsf{bool}}\quad (\mathit{Equal})$$

$$\frac{\Gamma\ \vdash\ e_1\ :\ \mathsf{int} \qquad \Gamma\ \vdash\ e_2\ :\ \mathsf{int}}{\Gamma\ \vdash\ e_1 > e_2\ :\ \mathsf{bool}}\quad (\mathit{Greater})$$

$$\frac{\Gamma\ \vdash\ e_1\ :\ \mathsf{int} \qquad \Gamma\ \vdash\ e_2\ :\ \mathsf{int}}{\Gamma\ \vdash\ e_1 < e_2\ :\ \mathsf{bool}}\quad (\mathit{Less})$$

$$\frac{\Gamma\ \vdash\ e_1\ :\ \mathsf{bool} \qquad \Gamma\ \vdash\ e_2\ :\ \mathsf{bool}}{\Gamma\ \vdash\ e_1\ \mathsf{and}\ e_2\ :\ \mathsf{bool}}\quad (\mathit{And})$$

$$\frac{\Gamma\ \vdash\ e_1\ :\ \mathsf{bool} \qquad \Gamma\ \vdash\ e_2\ :\ \mathsf{bool}}{\Gamma\ \vdash\ e_1\ \mathsf{or}\ e_2\ :\ \mathsf{bool}}\quad (\mathit{Or})$$

$$\frac{\Gamma\ \vdash\ e\ :\ \mathsf{bool}}{\Gamma\ \vdash\ \mathsf{not}\ e\ :\ \mathsf{bool}}\quad (\mathit{Not})$$

LookupD $\boxed{\delta\ \in\ \Gamma}$

$$\frac{}{\delta\ \in\ \Gamma\,,\, x\,\delta}\quad (\mathit{LookupD1})$$

$$\frac{\delta\ \neq\ \delta' \qquad \delta\ \in\ \Gamma}{\delta\ \in\ \Gamma\,,\, x\,\delta'}\quad (\mathit{LookupD2})$$

Match $\boxed{\Gamma \vdash e \sim m\,\tau}$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \sim \mathsf{in}\,\tau} \quad (Match1)$$

$$\frac{x : m\,\tau \in \Gamma \qquad m \neq \mathsf{in}}{\Gamma \vdash x \sim \mathsf{out}\,\tau} \quad (Match2)$$

$$\frac{x : \mathsf{in\,out}\,\tau \in \Gamma}{\Gamma \vdash x \sim \mathsf{in\,out}\,\tau} \quad (Match3)$$

MatchList $\boxed{\Gamma \vdash (e_1, .., e_l) \sim (m_1\,\tau_1, .., m_n\,\tau_n)}$

$$\frac{}{\Gamma \vdash ( ) \sim ( )} \quad (MatchList1)$$

$$\frac{\Gamma \vdash e \sim m\,\tau \qquad \Gamma \vdash (e_1, .., e_l) \sim (m_1\,\tau_1, .., m_n\,\tau_n)}{\Gamma \vdash (e, e_1, .., e_l) \sim (m\,\tau, m_1\,\tau_1, .., m_n\,\tau_n)} \quad (MatchList2)$$

Declaration typing $\boxed{\Gamma \vdash d : \mathsf{decl}}$

$$\frac{}{\Gamma \vdash \mathsf{begin\,end} : \mathsf{decl}} \quad (Empty)$$

$$\frac{\Gamma \vdash c : \mathsf{comm}}{\Gamma \vdash \mathsf{begin}\,c\,;\,\mathsf{end} : \mathsf{decl}} \quad (Block)$$

$$\frac{\Gamma, x : \mathsf{in\,out}\,\tau \vdash d : \mathsf{decl}}{\Gamma \vdash x : \tau\,;\,d : \mathsf{decl}} \quad (UninitVar)$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \mathsf{in\,out}\,\tau \vdash d : \mathsf{decl}}{\Gamma \vdash x : \tau := e\,;\,d : \mathsf{decl}} \quad (InitVar)$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \mathsf{in}\,\tau \vdash d : \mathsf{decl}}{\Gamma \vdash x : \mathsf{constant}\,\tau := e\,;\,d : \mathsf{decl}} \quad (Constant)$$

$$\frac{\Gamma, x_1 : m_1\,\tau_1, .., x_n : m_n\,\tau_n \vdash d_1 : \mathsf{decl} \qquad \Gamma, p : \mathsf{in\,proc}\,(m_1\,\tau_1, .., m_n\,\tau_n) \vdash d_2 : \mathsf{decl}}{\Gamma \vdash \mathsf{procedure}\,p\,(x_1 : m_1\,\tau_1\,;\,..\,;\,x_n : m_n\,\tau_n)\,\mathsf{is}\,d_1\,;\,d_2 : \mathsf{decl}} \quad (Proc)$$

Command typing $\boxed{\Gamma \vdash c : \mathsf{comm}}$

$$\frac{}{\Gamma \vdash \mathsf{null} : \mathsf{comm}} \quad (Null)$$

$$\frac{\Gamma \vdash c_1 : \mathsf{comm} \qquad \Gamma \vdash c_2 : \mathsf{comm}}{\Gamma \vdash c_1\,;\,c_2 : \mathsf{comm}} \quad (Seq)$$

$$\frac{m \neq \mathsf{in} \qquad x : m\,\tau \in \Gamma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \mathsf{comm}} \quad (Assign)$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash c_1 : \mathsf{comm} \qquad \Gamma \vdash c_2 : \mathsf{comm}}{\Gamma \vdash \mathsf{if}\,e\,\mathsf{then}\,c_1\,;\,\mathsf{else}\,c_2\,;\,\mathsf{end\,if} : \mathsf{comm}} \quad (IfThenElse)$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash c : \mathsf{comm}}{\Gamma \vdash \mathsf{while}\,e\,\mathsf{loop}\,c\,;\,\mathsf{end\,loop} : \mathsf{comm}} \quad (While)$$

$$\frac{\Gamma \vdash e : \mathsf{int} \qquad \Gamma \vdash e' : \mathsf{int} \qquad \Gamma, x : \mathsf{in\,int} \vdash c : \mathsf{comm}}{\Gamma \vdash \mathsf{for}\,x\,\mathsf{in}\,e\,..\,e'\,\mathsf{loop}\,c\,;\,\mathsf{end\,loop} : \mathsf{comm}} \quad (For)$$

$$\frac{\Gamma \vdash d : \mathsf{decl}}{\Gamma \vdash \mathsf{declare}\ d : \mathsf{comm}} \quad (Decl)$$

$$\frac{\Gamma \vdash e : \mathsf{proc}(m_1\,\tau_1, .., m_n\,\tau_n) \qquad \Gamma \vdash (e_1, .., e_l) \sim (m_1\,\tau_1, .., m_n\,\tau_n)}{\Gamma \vdash e(e_1, .., e_l) : \mathsf{comm}} \quad (ProcCall)$$

3 Structural Operational Semantics

Fetch $\boxed{\mu(x) = v}$

$$\frac{}{(\mu, x \leftarrow v)(x) = v} \quad (Fetch1)$$

$$\frac{x \neq x' \qquad \mu(x) = v}{(\mu, x' \leftarrow v')(x) = v} \quad (Fetch2)$$

Expression evaluation $\boxed{e =_\mu v}$

$$\frac{}{v =_\mu v} \quad (E\_Value)$$

$$\frac{\mu(x) = v}{x =_\mu v} \quad (E\_Ident)$$

$$\frac{e_1 =_\mu k_1 \qquad e_2 =_\mu k_2}{e_1 + e_2 =_\mu \{k_1 + k_2\}} \quad (E\_Plus)$$

$$\frac{e_1 =_\mu k_1 \qquad e_2 =_\mu k_2}{e_1 - e_2 =_\mu \{k_1 - k_2\}} \quad (E\_Minus)$$

$$\frac{e_1 =_\mu k_1 \qquad e_2 =_\mu k_2}{e_1 \times e_2 =_\mu \{k_1 \times k_2\}} \quad (E\_Times)$$

$$\frac{e_1 =_\mu k_1 \qquad e_2 =_\mu k_2}{e_1 > e_2 =_\mu k_1 > k_2} \quad (E\_Greater)$$

$$\frac{e_1 =_\mu k_1 \qquad e_2 =_\mu k_2}{e_1 < e_2 =_\mu k_1 < k_2} \quad (E\_Less)$$

$$\frac{e_1 =_\mu k_1 \qquad e_2 =_\mu k_2}{e_1 = e_2 =_\mu \{k_1 = k_2\}} \quad (E\_Equal)$$

$$\frac{e_1 =_\mu b_1 \qquad e_2 =_\mu b_2}{e_1\ \mathsf{and}\ e_2 =_\mu \{b_1\ \mathsf{and}\ b_2\}} \quad (E\_And)$$

$$\frac{e_1 =_\mu b_1 \qquad e_2 =_\mu b_2}{e_1\ \mathsf{or}\ e_2 =_\mu \{b_1\ \mathsf{or}\ b_2\}} \quad (E\_Or)$$

$$\frac{e =_\mu b}{\mathsf{not}\ e =_\mu \{\mathsf{not}\ b\}} \quad (E\_Not)$$

Store Update $\boxed{\mu\{x \leftarrow v\} \mapsto \mu'}$

$$\frac{}{(\mu, x \leftarrow v)\{x \leftarrow v'\} \mapsto (\mu, x \leftarrow v')} \quad (Update1)$$

$$\frac{x \neq x' \qquad \mu\{x \leftarrow v'\} \mapsto \mu'}{(\mu, x' \leftarrow v)\{x \leftarrow v'\} \mapsto (\mu', x' \leftarrow v)} \quad (Update2)$$

Many Steps $\boxed{\langle c, \mu\rangle \mapsto^{k} \langle c', \mu'\rangle}$

$$\frac{}{\langle c, \mu\rangle \mapsto^{0} \langle c, \mu\rangle} \quad (ManySteps1)$$

$$\frac{}{\langle \mathsf{null}, \mu\rangle \mapsto^{k} \langle \mathsf{null}, \mu\rangle} \quad (ManySteps2)$$

$$\frac{\langle c, \mu\rangle \mapsto \langle c', \mu'\rangle \qquad \langle c', \mu'\rangle \mapsto^{\{\,k-1\,\}} \langle c'', \mu''\rangle}{\langle c, \mu\rangle \mapsto^{k} \langle c'', \mu''\rangle} \quad (ManySteps3)$$

Trace $\boxed{\langle c; \mu\rangle \Rightarrow^{k} tr}$

$$\frac{}{\langle c; \mu\rangle \Rightarrow^{0} [\ ]} \quad (Trace1)$$

$$\frac{}{\langle \mathsf{null}; \mu\rangle \Rightarrow^{k} [\ ]} \quad (Trace2)$$

$$\frac{\langle c, \mu\rangle \mapsto \langle c', \mu'\rangle \qquad \langle c'; \mu'\rangle \Rightarrow^{\{\,k-1\,\}} [\,\langle\, c'_1\,,\, \mu'_1\,\rangle .. \langle\, c'_n\,,\, \mu'_n\,\rangle\,]}{\langle c; \mu\rangle \Rightarrow^{k} [\,\langle\, c'\,,\, \mu'\,\rangle\,\langle\, c'_1\,,\, \mu'_1\,\rangle .. \langle\, c'_n\,,\, \mu'_n\,\rangle\,]} \quad (Trace3)$$

Full evaluation $\boxed{\langle\, c\,;\, \mu\,\rangle \rightsquigarrow \mu'}$

$$\frac{}{\langle\, \mathsf{null}\,;\, \mu\,\rangle \rightsquigarrow \mu} \quad (Eval1)$$

$$\frac{\langle c, \mu\rangle \mapsto \langle c', \mu'\rangle \qquad \langle\, c'\,;\, \mu'\,\rangle \rightsquigarrow \mu''}{\langle\, c\,;\, \mu\,\rangle \rightsquigarrow \mu''} \quad (Eval2)$$

Compatibility $\boxed{(\,|x'_i : m'_i\tau'_i|\,)\,\#\,(\,|e'_j|\,) \;=\; [\,|x_n : m_n\tau_n = e_n|\,]}$

$$\frac{}{(\,)\,\#\,(\,) \;=\; [\,]} \quad (E\_Compat1)$$

$$\frac{(\,|x'_i : m'_i\tau'_i|\,)\,\#\,(\,|e'_j|\,) \;=\; [\,|x_n : m_n\tau_n = e_n|\,]}{(\,x \;:\; m\,\tau\,|x'_i : m'_i\tau'_i|\,)\,\#\,(\,e\,|e'_j|\,) \;=\; [\,x \;:\; m\,\tau \;=\; e\,|x_n : m_n\tau_n = e_n|\,]} \quad (E\_Compat2)$$

One step evaluation $\boxed{\langle c, \mu\rangle \mapsto \langle c', \mu'\rangle}$

$$\frac{}{\langle(\,\mathsf{null}\,;\; c\,), \mu\rangle \mapsto \langle c, \mu\rangle} \quad (E\_Null)$$

$$\frac{\langle c_1, \mu\rangle \mapsto \langle c'_1, \mu'\rangle}{\langle(\,c_1\,;\; c_2\,), \mu\rangle \mapsto \langle(\,c'_1\,;\; c_2\,), \mu'\rangle} \quad (E\_Seq)$$

$$\frac{e =_{\mu} v \qquad \mu\,\{\,x \;\leftarrow\; v\,\} \mapsto \mu'}{\langle(\,x \;:=\; e\,), \mu\rangle \mapsto \langle \mathsf{null}, \mu'\rangle} \quad (E\_Assign)$$

$$\frac{e =_{\mu} \mathsf{true}}{\langle(\,\mathsf{if}\; e\; \mathsf{then}\; c_1\,;\; \mathsf{else}\; c_2\,;\; \mathsf{end\ if}\,), \mu\rangle \mapsto \langle c_1, \mu\rangle} \quad (E\_IfThenElse1)$$

$$\frac{e =_{\mu} \mathsf{false}}{\langle(\,\mathsf{if}\; e\; \mathsf{then}\; c_1\,;\; \mathsf{else}\; c_2\,;\; \mathsf{end\ if}\,), \mu\rangle \mapsto \langle c_2, \mu\rangle} \quad (E\_IfThenElse2)$$

$$\frac{e =_{\mu} \mathsf{false}}{\langle(\,\mathsf{while}\; e\; \mathsf{loop}\; c\,;\; \mathsf{end\ loop}\,), \mu\rangle \mapsto \langle \mathsf{null}, \mu\rangle} \quad (E\_While1)$$

$$\frac{e =_\mu \mathsf{true}}{\langle(\,\mathsf{while}\ e\ \mathsf{loop}\ c\,;\ \mathsf{end\ loop}\,),\mu\rangle \mapsto \langle(\,c\,;\ \mathsf{while}\ e\ \mathsf{loop}\ c\,;\ \mathsf{end\ loop}\,),\mu\rangle} \quad (E\_While2)$$

$$\frac{}{\langle \mathsf{declare\ begin\ end}, \mu\rangle \mapsto \langle \mathsf{null}, \mu\rangle} \quad (E\_Decl1)$$

$$\frac{\langle\ d\ ,\ \mu\ \rangle \mapsto \langle\ d'\ ,\ \mu'\ \rangle}{\langle \mathsf{declare}\ d, \mu\rangle \mapsto \langle \mathsf{declare}\ d', \mu'\rangle} \quad (E\_Decl2)$$

$$\frac{e =_\mu k \qquad e' =_\mu k' \qquad k > k'}{\langle(\,\mathsf{for}\ x\ \mathsf{in}\ e\,..\,e'\ \mathsf{loop}\ c\,;\ \mathsf{end\ loop}\,),\mu\rangle \mapsto \langle \mathsf{null}, \mu\rangle} \quad (E\_For1)$$

$$\frac{e =_\mu k \qquad e' =_\mu k' \qquad k \leq k'}{\langle(\,\mathsf{for}\ x\ \mathsf{in}\ e\,..\,e'\ \mathsf{loop}\ c\,;\ \mathsf{end\ loop}\,),\mu\rangle \mapsto \langle(\,\mathsf{declare}\ x\ :\ \mathsf{constant\ int}\ :=\ k\,;\ \mathsf{begin}\ c\,;\ \mathsf{end}\,;\ \mathsf{for}\ x\ \mathsf{in}\ \{\ k\ +\ 1\ \}\,..\,k'\ \mathsf{loop}\ c\,;\ \mathsf{end\ loop}\,),\mu\rangle} \quad (E\_For2)$$

$$\frac{e =_\mu \mathsf{proc}\,(\,|x_i' : m_i'\tau_i'|\,)\ \mathsf{is}\ d \qquad (\,|x_i' : m_i'\tau_i'|\,)\,\#\,(\,|e_j'|\,)\ =\ [\,|x_n : m_n\tau_n = e_n|\,]}{\langle e\,(\,|e_j'|\,),\mu\rangle \mapsto \langle \mathsf{declare}\,[\,|x_n : m_n\tau_n = e_n|\,]\ d, \mu\rangle} \quad (E\_ProcCall)$$

Declaration evaluation $\boxed{\langle\ d\ ,\ \mu\ \rangle \mapsto \langle\ d'\ ,\ \mu'\ \rangle}$

$$\frac{}{\langle\ \mathsf{begin\ null}\,;\ \mathsf{end}\ ,\ \mu\ \rangle \mapsto \langle\ \mathsf{begin\ end}\ ,\ \mu\ \rangle} \quad (E\_Block1)$$

$$\frac{\langle c, \mu\rangle \mapsto \langle c', \mu'\rangle}{\langle\ \mathsf{begin}\ c\,;\ \mathsf{end}\ ,\ \mu\ \rangle \mapsto \langle\ \mathsf{begin}\ c'\,;\ \mathsf{end}\ ,\ \mu'\ \rangle} \quad (E\_Block2)$$

$$\frac{}{\langle\ x\ :\ \tau\ :=\ e\,;\ \mathsf{begin\ end}\ ,\ \mu\ \rangle \mapsto \langle\ \mathsf{begin\ end}\ ,\ \mu\ \rangle} \quad (E\_InitVar1)$$

$$\frac{e =_\mu v \qquad \langle\ d\ ,\ (\,\mu\ ,\ x\ \leftarrow\ v\,)\ \rangle \mapsto \langle\ d'\ ,\ (\,\mu'\ ,\ x\ \leftarrow\ v'\,)\ \rangle}{\langle\ x\ :\ \tau\ :=\ e\,;\ d\ ,\ \mu\ \rangle \mapsto \langle\ x\ :\ \tau\ :=\ v'\,;\ d'\ ,\ \mu'\ \rangle} \quad (E\_InitVar2)$$

$$\frac{}{\langle\ x\ :\ \mathsf{constant}\,\tau\ :=\ e\,;\ \mathsf{begin\ end}\ ,\ \mu\ \rangle \mapsto \langle\ \mathsf{begin\ end}\ ,\ \mu\ \rangle} \quad (E\_Const1)$$

$$\frac{e =_\mu v \qquad \langle\ d\,[\,v\,/\,x\,]\ ,\ \mu\ \rangle \mapsto \langle\ d'\ ,\ \mu'\ \rangle}{\langle\ x\ :\ \mathsf{constant}\,\tau\ :=\ e\,;\ d\ ,\ \mu\ \rangle \mapsto \langle\ x\ :\ \mathsf{constant}\,\tau\ :=\ v\,;\ d'\ ,\ \mu'\ \rangle} \quad (E\_Const2)$$

$$\frac{}{\langle\ \mathsf{procedure}\ p\,(\,|x_n : m_n\tau_n|\,)\ \mathsf{is}\ d_1\,;\ d\ ,\ \mu\ \rangle \mapsto \langle\ d\,[\,\mathsf{proc}\,(\,|x_n : m_n\tau_n|\,)\ \mathsf{is}\ d_1\,/\,p\,]\ ,\ \mu\ \rangle} \quad (E\_Proc)$$

$$\frac{}{\langle\,(\ x\ :\ m\,\tau\ =\ e\ )\,\mathsf{begin\ end}\ ,\ \mu\ \rangle \mapsto \langle\ \mathsf{begin\ end}\ ,\ \mu\ \rangle} \quad (E\_Alias1)$$

$$\frac{e =_\mu v \qquad \langle\ d\,[\,v\,/\,x\,]\ ,\ \mu\ \rangle \mapsto \langle\ d'\ ,\ \mu'\ \rangle}{\langle\,(\ x\ :\ \mathsf{in}\,\tau\ =\ e\ )\,d\ ,\ \mu\ \rangle \mapsto \langle\ d'\ ,\ \mu'\ \rangle} \quad (E\_Alias2)$$

$$\frac{m\ \neq\ \mathsf{in} \qquad \mu\,(\,y\,)\ =\ v \qquad \langle\ d\ ,\ (\,\mu\ ,\ x\ \leftarrow\ v\,)\ \rangle \mapsto \langle\ d'\ ,\ (\,\mu'\ ,\ x\ \leftarrow\ v'\,)\ \rangle \qquad \mu'\,\{\,y\ \leftarrow\ v'\,\} \mapsto \mu''}{\langle\,(\ x\ :\ m\,\tau\ =\ y\ )\,d\ ,\ \mu\ \rangle \mapsto \langle\,(\ x\ :\ m\,\tau\ =\ y\ )\,d'\ ,\ \mu''\ \rangle} \quad (E\_Alias3)$$

$$\frac{}{\langle\ [\ ]\,d\ ,\ \mu\ \rangle \mapsto \langle\ d\ ,\ \mu\ \rangle} \quad (E\_Aliases1)$$

$$\frac{}{\langle\ [\,|x_n : m_n\tau_n = e_n|\,]\,\mathsf{begin\ end}\ ,\ \mu\ \rangle \mapsto \langle\ \mathsf{begin\ end}\ ,\ \mu\ \rangle} \quad (E\_Aliases2)$$

$$\frac{\langle\,(\ x\ :\ m\,\tau\ =\ e\ )\,[\,|x_n : m_n\tau_n = e_n|\,]\,d\ ,\ \mu\ \rangle \mapsto \langle\ d'\ ,\ \mu'\ \rangle}{\langle\ [\,x\ :\ m\,\tau\ =\ e\ ,\ |x_n : m_n\tau_n = e_n|\,]\,d\ ,\ \mu\ \rangle \mapsto \langle\ d'\ ,\ \mu'\ \rangle} \quad (E\_Aliases3)$$

A Generated Isabelle/HOL theory

```
(* generated by Ott 0.10.15 from: _source-1-s.ott _source-1.ott _source-2-s.ott _source-2.ott
_source-3-s.ott _source-3.ott _source-4.ott source.ott *)
theory source imports Main Multiset begin

(** syntax *)
types index = "nat"
types ident = "string"
types number = "int"
types integer = "int"
datatype
mode =
   M_In
 | M_Out
 | M_InOut

types boolean = "bool"
datatype
ty =
   T_Int
 | T_Bool
 | T_Proc "(mode*ty) list"
 | T_Void

datatype
dcl =
   D_Empty
 | D_Block "cmd"
 | D_UninitVar "ident" "ty" "dcl"
 | D_InitVar "ident" "ty" "exp" "dcl"
 | D_Constant "ident" "ty" "exp" "dcl"
 | D_Proc "ident" "(ident*mode*ty) list" "dcl" "dcl"
 | D_Aliases "(ident*mode*ty*exp) list" "dcl"
 | D_Alias "ident" "mode" "ty" "exp" "dcl"
and va =
   V_Int "integer"
 | V_Bool "boolean"
 | V_Proc "(ident*mode*ty) list" "dcl"
and cmd =
   C_Null
 | C_Assign "ident" "exp"
 | C_Seq "cmd" "cmd"
 | C_IfThenElse "exp" "cmd" "cmd"
 | C_While "exp" "cmd"
 | C_Decl "dcl"
 | C_For "ident" "exp" "exp" "cmd"
 | C_ProcCall "exp" "exp list"
and exp =
   E_Var "ident"
 | E_Value "va"
 | E_Plus "exp" "exp"
 | E_Minus "exp" "exp"
 | E_Times "exp" "exp"
 | E_Equal "exp" "exp"
 | E_Greater "exp" "exp"
 | E_Less "exp" "exp"
 | E_And "exp" "exp"
 | E_Or "exp" "exp"
 | E_Not "exp"

datatype
df =
   VarDecl "mode" "ty"
 | ReturnType "ty"

types store = "(ident*va) list"
```

```
types env = "(ident*df) list"
types trace = "(cmd*store) list"


(** library functions *)
lemma [mono]:"
         (!! x. f x --> g x) ==> list_all (%b. b) (map f foo_list)-->
                     list_all (%b. b) (map g foo_list) "
   apply(induct_tac foo_list, auto) done

lemma [mono]: "split f p = f (fst p) (snd p)" by (simp add: split_def)

(** subrules *)
consts
is_value_of_exp :: "exp => bool"
primrec
"is_value_of_exp (E_Var x) = (False)"
"is_value_of_exp (E_Value v) = ((True))"
"is_value_of_exp (E_Plus e1 e2) = (False)"
"is_value_of_exp (E_Minus e1 e2) = (False)"
"is_value_of_exp (E_Times e1 e2) = (False)"
"is_value_of_exp (E_Equal e1 e2) = (False)"
"is_value_of_exp (E_Greater e1 e2) = (False)"
"is_value_of_exp (E_Less e1 e2) = (False)"
"is_value_of_exp (E_And e1 e2) = (False)"
"is_value_of_exp (E_Or e1 e2) = (False)"
"is_value_of_exp (E_Not e) = (False)"


(** substitutions *)
consts
subst_ty_exp :: "exp => ident => (ty*exp) => (ty*exp)"
subst_mode_ty_exp :: "exp => ident => mode*(ty*exp) => mode*(ty*exp)"
subst_ident_mode_ty_exp :: "exp => ident => ident*(mode*ty*exp) => ident*(mode*ty*exp)"
subst_ident_mode_ty_exp_list :: "exp => ident => (ident*mode*ty*exp) list =>
(ident*mode*ty*exp) list"
subst_dcl :: "exp => ident => dcl => dcl"
subst_va :: "exp => ident => va => va"
subst_exp_list :: "exp => ident => exp list => exp list"
subst_cmd :: "exp => ident => cmd => cmd"
subst_exp :: "exp => ident => exp => exp"
primrec
"subst_ty_exp e_5 x_5 (ty1,exp1) = (ty1 , subst_exp e_5 x_5 exp1)"
"subst_mode_ty_exp e_5 x_5 (mode1,ty_exp1) = (mode1 , subst_ty_exp e_5 x_5 ty_exp1)"
"subst_ident_mode_ty_exp e_5 x_5 (ident1,mode_ty_exp1) = (ident1 , subst_mode_ty_exp e_5 x_5
mode_ty_exp1)"
"subst_ident_mode_ty_exp_list e_5 x_5 Nil = (Nil)"
"subst_ident_mode_ty_exp_list e_5 x_5 (ident_mode_ty_exp_0#ident_mode_ty_exp_list_0) =
((subst_ident_mode_ty_exp e_5 x_5 ident_mode_ty_exp_0) # (subst_ident_mode_ty_exp_list e_5 x_5
ident_mode_ty_exp_list_0))"
"subst_dcl e_5 x_5 D_Empty = (D_Empty )"
"subst_dcl e_5 x_5 (D_Block c) = (D_Block (subst_cmd e_5 x_5 c))"
"subst_dcl e_5 x_5 (D_UninitVar x T d) = (D_UninitVar x T (if x_5 mem [x] then d else
(subst_dcl e_5 x_5 d)))"
"subst_dcl e_5 x_5 (D_InitVar x T e d) = (D_InitVar x T (subst_exp e_5 x_5 e) (if x_5 mem [x]
then d else (subst_dcl e_5 x_5 d)))"
"subst_dcl e_5 x_5 (D_Constant x T e d) = (D_Constant x T (subst_exp e_5 x_5 e) (if x_5 mem [x]
then d else (subst_dcl e_5 x_5 d)))"
"subst_dcl e_5 x_5 (D_Proc p (x_m_T_list) d1 d2) = (D_Proc p x_m_T_list (if x_5 mem (List.map
(%((x_0::ident),(m_0::mode),(T_0::ty)).x_0) x_m_T_list) then d1 else (subst_dcl e_5 x_5 d1))
(subst_dcl e_5 x_5 d2))"
"subst_dcl e_5 x_5 (D_Aliases (x_m_T_e_list) d) = (D_Aliases (subst_ident_mode_ty_exp_list e_5
x_5 x_m_T_e_list) (if x_5 mem (List.map (%((x_0::ident),(m_0::mode),(T_0::ty),(e_0::exp)).x_0)
x_m_T_e_list) then d else (subst_dcl e_5 x_5 d)))"
"subst_dcl e_5 x_5 (D_Alias x m T e d) = (D_Alias x m T (subst_exp e_5 x_5 e) (if x_5 mem [x]
then d else (subst_dcl e_5 x_5 d)))"
"subst_va e5 x_5 (V_Int k) = (V_Int k)"
```

```
"subst_va e5 x_5 (V_Bool b) = (V_Bool b)"
"subst_va e5 x_5 (V_Proc (x_m_T_list) d) = (V_Proc x_m_T_list (if x_5 mem (List.map
(%((x_0::ident),(m_0::mode),(T_0::ty)).x_0) x_m_T_list) then d else (subst_dcl e5 x_5 d)))"
"subst_exp_list e_5 x5 Nil = (Nil)"
"subst_exp_list e_5 x5 (exp_0#exp_list_0) = ((subst_exp e_5 x5 exp_0) # (subst_exp_list e_5 x5
exp_list_0))"
"subst_cmd e_5 x5 C_Null = (C_Null )"
"subst_cmd e_5 x5 (C_Assign x e) = (C_Assign x (subst_exp e_5 x5 e))"
"subst_cmd e_5 x5 (C_Seq c1 c2) = (C_Seq (subst_cmd e_5 x5 c1) (subst_cmd e_5 x5 c2))"
"subst_cmd e_5 x5 (C_IfThenElse e c1 c2) = (C_IfThenElse (subst_exp e_5 x5 e) (subst_cmd e_5 x5
c1) (subst_cmd e_5 x5 c2))"
"subst_cmd e_5 x5 (C_While e c) = (C_While (subst_exp e_5 x5 e) (subst_cmd e_5 x5 c))"
"subst_cmd e_5 x5 (C_Decl d) = (C_Decl (subst_dcl e_5 x5 d))"
"subst_cmd e_5 x5 (C_For x e e' c) = (C_For x (subst_exp e_5 x5 e) (subst_exp e_5 x5 e') (if x5
mem [x] then c else (subst_cmd e_5 x5 c)))"
"subst_cmd e_5 x5 (C_ProcCall e (e_list)) = (C_ProcCall (subst_exp e_5 x5 e) (subst_exp_list
e_5 x5 e_list))"
"subst_exp e_5 x5 (E_Var x) = ((if x=x5 then e_5 else (E_Var x)))"
"subst_exp e_5 x5 (E_Value v) = (E_Value (subst_va e_5 x5 v))"
"subst_exp e_5 x5 (E_Plus e1 e2) = (E_Plus (subst_exp e_5 x5 e1) (subst_exp e_5 x5 e2))"
"subst_exp e_5 x5 (E_Minus e1 e2) = (E_Minus (subst_exp e_5 x5 e1) (subst_exp e_5 x5 e2))"
"subst_exp e_5 x5 (E_Times e1 e2) = (E_Times (subst_exp e_5 x5 e1) (subst_exp e_5 x5 e2))"
"subst_exp e_5 x5 (E_Equal e1 e2) = (E_Equal (subst_exp e_5 x5 e1) (subst_exp e_5 x5 e2))"
"subst_exp e_5 x5 (E_Greater e1 e2) = (E_Greater (subst_exp e_5 x5 e1) (subst_exp e_5 x5 e2))"
"subst_exp e_5 x5 (E_Less e1 e2) = (E_Less (subst_exp e_5 x5 e1) (subst_exp e_5 x5 e2))"
"subst_exp e_5 x5 (E_And e1 e2) = (E_And (subst_exp e_5 x5 e1) (subst_exp e_5 x5 e2))"
"subst_exp e_5 x5 (E_Or e1 e2) = (E_Or (subst_exp e_5 x5 e1) (subst_exp e_5 x5 e2))"
"subst_exp e_5 x5 (E_Not e) = (E_Not (subst_exp e_5 x5 e))"


(** definitions *)
(*defns eval_exp *)
inductive_set Fetch :: "(store*ident*va) set"
 and ExpEval :: "(exp*store*va) set"
where
(* defn Fetch *)

Fetch1I: " (  (( x , v )# mu )  , x , v ) : Fetch"

| Fetch2I: "[| x  ~=  x'  ;
 ( mu , x , v ) : Fetch|] ==>
 (  (( x' , v' )# mu )  , x , v ) : Fetch"

| (* defn ExpEval *)

E_ValueI: " ( (E_Value v) , mu , v ) : ExpEval"

| E_IdentI: "[| ( mu , x , v ) : Fetch|] ==>
 ( (E_Var x) , mu , v ) : ExpEval"

| E_PlusI: "[| ( e1 , mu , (V_Int k1) ) : ExpEval ;
 ( e2 , mu , (V_Int k2) ) : ExpEval|] ==>
 ( (E_Plus e1 e2) , mu , (V_Int  ( k1  +  k2 ) ) ) : ExpEval"

| E_MinusI: "[| ( e1 , mu , (V_Int k1) ) : ExpEval ;
 ( e2 , mu , (V_Int k2) ) : ExpEval|] ==>
 ( (E_Minus e1 e2) , mu , (V_Int  ( k1  -  k2 ) ) ) : ExpEval"

| E_TimesI: "[| ( e1 , mu , (V_Int k1) ) : ExpEval ;
 ( e2 , mu , (V_Int k2) ) : ExpEval|] ==>
 ( (E_Times e1 e2) , mu , (V_Int  ( k1  *  k2 ) ) ) : ExpEval"

| E_GreaterI: "[| ( e1 , mu , (V_Int k1) ) : ExpEval ;
 ( e2 , mu , (V_Int k2) ) : ExpEval|] ==>
 ( (E_Greater e1 e2) , mu , (V_Bool  ( k1  >  k2 ) ) ) : ExpEval"

| E_LessI: "[| ( e1 , mu , (V_Int k1) ) : ExpEval ;
```

```
 ( e2 , mu , (V_Int k2) ) : ExpEval|] ==>
 ( (E_Less e1 e2) , mu , (V_Bool  ( k1  <  k2 ) ) ) : ExpEval"

| E_EqualI: "[| ( e1 , mu , (V_Int k1) ) : ExpEval ;
 ( e2 , mu , (V_Int k2) ) : ExpEval|] ==>
 ( (E_Equal e1 e2) , mu , (V_Bool  ( k1  =  k2 ) ) ) : ExpEval"

| E_AndI: "[| ( e1 , mu , (V_Bool b1) ) : ExpEval ;
 ( e2 , mu , (V_Bool b2) ) : ExpEval|] ==>
 ( (E_And e1 e2) , mu , (V_Bool  ( b1  \<and>  b2 ) ) ) : ExpEval"

| E_OrI: "[| ( e1 , mu , (V_Bool b1) ) : ExpEval ;
 ( e2 , mu , (V_Bool b2) ) : ExpEval|] ==>
 ( (E_Or e1 e2) , mu , (V_Bool  ( b1  \<or>  b2 ) ) ) : ExpEval"

| E_NotI: "[| ( e , mu , (V_Bool b) ) : ExpEval|] ==>
 ( (E_Not e) , mu , (V_Bool  (\<not>  b ) ) ) : ExpEval"

(*defns typing *)
inductive_set Lookup :: "(ident*df*env) set"
 and ExpTyping :: "(env*exp*ty) set"
 and LookupD :: "(df*env) set"
 and Match :: "(env*exp*mode*ty) set"
 and MatchList :: "(env*exp list*(mode*ty) list) set"
 and DeclTyping :: "(env*dcl) set"
 and CommTyping :: "(env*cmd) set"
where
(* defn Lookup *)

Lookup1I: " ( x , df ,  (( x , df ) #  G )  ) : Lookup"

| Lookup2I: "[| x  ~=  x'  ;
 ( x , df , G ) : Lookup|] ==>
 ( x , df ,  (( x' , df' ) #  G )  ) : Lookup"

| (* defn ExpTyping *)

VarI: "[| m  ~=  M_Out  ;
 ( x , (VarDecl m T) , G ) : Lookup|] ==>
 ( G , (E_Var x) , T ) : ExpTyping"

| IntCstI: " ( G , (E_Value (V_Int  q )) , T_Int ) : ExpTyping"

| BoolTrueI: " ( G , (E_Value (V_Bool  true )) , T_Bool ) : ExpTyping"

| BoolFalseI: " ( G , (E_Value (V_Bool  false )) , T_Bool ) : ExpTyping"

| PlusI: "[| ( G , e1 , T_Int ) : ExpTyping ;
 ( G , e2 , T_Int ) : ExpTyping|] ==>
 ( G , (E_Plus e1 e2) , T_Int ) : ExpTyping"

| MinusI: "[| ( G , e1 , T_Int ) : ExpTyping ;
 ( G , e2 , T_Int ) : ExpTyping|] ==>
 ( G , (E_Minus e1 e2) , T_Int ) : ExpTyping"

| TimesI: "[| ( G , e1 , T_Int ) : ExpTyping ;
 ( G , e2 , T_Int ) : ExpTyping|] ==>
 ( G , (E_Times e1 e2) , T_Int ) : ExpTyping"

| EqualI: "[| ( G , e1 , T ) : ExpTyping ;
 ( G , e2 , T ) : ExpTyping|] ==>
 ( G , (E_Equal e1 e2) , T_Bool ) : ExpTyping"

| GreaterI: "[| ( G , e1 , T_Int ) : ExpTyping ;
 ( G , e2 , T_Int ) : ExpTyping|] ==>
 ( G , (E_Greater e1 e2) , T_Bool ) : ExpTyping"
```

```
| LessI: "[| ( G , e1 , T_Int ) : ExpTyping ;
 ( G , e2 , T_Int ) : ExpTyping|] ==>
 ( G , (E_Less e1 e2) , T_Bool ) : ExpTyping"

| AndI: "[| ( G , e1 , T_Bool ) : ExpTyping ;
 ( G , e2 , T_Bool ) : ExpTyping|] ==>
 ( G , (E_And e1 e2) , T_Bool ) : ExpTyping"

| OrI: "[| ( G , e1 , T_Bool ) : ExpTyping ;
 ( G , e2 , T_Bool ) : ExpTyping|] ==>
 ( G , (E_Or e1 e2) , T_Bool ) : ExpTyping"

| NotI: "[| ( G , e , T_Bool ) : ExpTyping|] ==>
 ( G , (E_Not e) , T_Bool ) : ExpTyping"

| (* defn LookupD *)

LookupD1I: " ( df ,  (( x , df ) #  G )  ) : LookupD"

| LookupD2I: "[| df  ~=  df'  ;
 ( df , G ) : LookupD|] ==>
 ( df ,  (( x , df' ) #  G )  ) : LookupD"

| (* defn Match *)

Match1I: "[| ( G , e , T ) : ExpTyping|] ==>
 ( G , e , M_In , T ) : Match"

| Match2I: "[| ( x , (VarDecl m T) , G ) : Lookup ;
 m  ~=  M_In |] ==>
 ( G , (E_Var x) , M_Out , T ) : Match"

| Match3I: "[| ( x , (VarDecl M_InOut T) , G ) : Lookup|] ==>
 ( G , (E_Var x) , M_InOut , T ) : Match"

| (* defn MatchList *)

MatchList1I: " ( G , [] , [] ) : MatchList"

| MatchList2I: "[| ( G , e , m , T ) : Match ;
 ( G , (e_list) , (m_T_list) ) : MatchList|] ==>
 ( G , ((e) # e_list) , ((m,T) # m_T_list) ) : MatchList"

| (* defn DeclTyping *)

EmptyI: " ( G , D_Empty ) : DeclTyping"

| BlockI: "[| ( G , c ) : CommTyping|] ==>
 ( G , (D_Block c) ) : DeclTyping"

| UninitVarI: "[| (  (( x , (VarDecl M_InOut T) ) #  G )  , d ) : DeclTyping|] ==>
 ( G , (D_UninitVar x T d) ) : DeclTyping"

| InitVarI: "[| ( G , e , T ) : ExpTyping ;
 (  (( x , (VarDecl M_InOut T) ) #  G )  , d ) : DeclTyping|] ==>
 ( G , (D_InitVar x T e d) ) : DeclTyping"

| ConstantI: "[| ( G , e , T ) : ExpTyping ;
 (  (( x , (VarDecl M_In T) ) #  G )  , d ) : DeclTyping|] ==>
 ( G , (D_Constant x T e d) ) : DeclTyping"

| ProcI: "[| (  (   (List.rev  ((List.map (%((x_0::ident),(m_0::mode),(T_0::ty)).(x_0,(VarDecl
m_0 T_0))) x_m_T_list))  @ G)   )  , d1 ) : DeclTyping ;
 (  (( p , (VarDecl M_In (T_Proc ((List.map (%((x_0::ident),(m_0::mode),(T_0::ty)).(m_0,T_0))
x_m_T_list)))) ) #  G )  , d2 ) : DeclTyping|] ==>
 ( G , (D_Proc p (x_m_T_list) d1 d2) ) : DeclTyping"
```

```
| (* defn CommTyping *)

NullI: " ( G , C_Null ) : CommTyping"

| SeqI: "[| ( G , c1 ) : CommTyping ;
 ( G , c2 ) : CommTyping|] ==>
 ( G , (C_Seq c1 c2) ) : CommTyping"

| AssignI: "[| m  ~=  M_In  ;
 ( x , (VarDecl m T) , G ) : Lookup ;
 ( G , e , T ) : ExpTyping|] ==>
 ( G , (C_Assign x e) ) : CommTyping"

| IfThenElseI: "[| ( G , e , T_Bool ) : ExpTyping ;
 ( G , c1 ) : CommTyping ;
 ( G , c2 ) : CommTyping|] ==>
 ( G , (C_IfThenElse e c1 c2) ) : CommTyping"

| WhileI: "[| ( G , e , T_Bool ) : ExpTyping ;
 ( G , c ) : CommTyping|] ==>
 ( G , (C_While e c) ) : CommTyping"

| ForI: "[| ( G , e , T_Int ) : ExpTyping ;
 ( G , e' , T_Int ) : ExpTyping ;
 (  (( x , (VarDecl M_In T_Int) ) #  G )  , c ) : CommTyping|] ==>
 ( G , (C_For x e e' c) ) : CommTyping"

| DeclI: "[| ( G , d ) : DeclTyping|] ==>
 ( G , (C_Decl d) ) : CommTyping"

| ProcCallI: "[| ( G , e , (T_Proc (m_T_list)) ) : ExpTyping ;
 ( G , (e_list) , (m_T_list) ) : MatchList|] ==>
 ( G , (C_ProcCall e (e_list)) ) : CommTyping"

(*defns eval_comm *)
inductive_set StoreUpdate :: "(store*ident*va*store) set"
 and ManySteps :: "(cmd*store*integer*cmd*store) set"
 and Trace :: "(cmd*store*integer*trace) set"
 and FullEvaluation :: "(cmd*store*store) set"
 and Compat :: "((ident*mode*ty) list*exp list*(ident*mode*ty*exp) list) set"
 and OneStep :: "(cmd*store*cmd*store) set"
 and DeclEval :: "(dcl*store*dcl*store) set"
where
(* defn StoreUpdate *)

Update1I: " (  (( x , v )# mu )  , x , v' ,  (( x , v' )# mu )  ) : StoreUpdate"

| Update2I: "[| x  ~=  x'  ;
 ( mu , x , v' , mu' ) : StoreUpdate|] ==>
 (  (( x' , v )# mu )  , x , v' ,  (( x' , v )# mu' )  ) : StoreUpdate"

| (* defn ManySteps *)

ManySteps1I: " ( c , mu ,  0  , c , mu ) : ManySteps"

| ManySteps2I: " ( C_Null , mu , k , C_Null , mu ) : ManySteps"

| ManySteps3I: "[| ( c , mu , c' , mu' ) : OneStep ;
 ( c' , mu' ,  ( k  -  1  )  , c'' , mu'' ) : ManySteps|] ==>
 ( c , mu , k , c'' , mu'' ) : ManySteps"

| (* defn Trace *)

Trace1I: " ( c , mu ,  0  ,  []  ) : Trace"

| Trace2I: " ( C_Null , mu , k ,  []  ) : Trace"
```

```
| Trace3I: "[| ( c , mu , c' , mu' ) : OneStep ;
 ( c' , mu' ,  ( k  -   1  )  ,  (c'_mu'_list)  ) : Trace|] ==>
 ( c , mu , k ,  ((c',mu') # c'_mu'_list)  ) : Trace"

| (* defn FullEvaluation *)

Eval1I: " ( C_Null , mu , mu ) : FullEvaluation"

| Eval2I: "[| ( c , mu , c' , mu' ) : OneStep ;
 ( c' , mu' , mu'' ) : FullEvaluation|] ==>
 ( c , mu , mu'' ) : FullEvaluation"

| (* defn Compat *)

E_Compat1I: " ( [] , [] , [] ) : Compat"

| E_Compat2I: "[| ( (x'_m'_T'_list) , (e'_list) , (x_m_T_e_list) ) : Compat|] ==>
 ( ((x,m,T) # x'_m'_T'_list) , ((e) # e'_list) , ((x,m,T,e) # x_m_T_e_list) ) : Compat"

| (* defn OneStep *)

E_NullI: " (  (C_Seq C_Null c)  , mu , c , mu ) : OneStep"

| E_SeqI: "[| ( c1 , mu , c1' , mu' ) : OneStep|] ==>
 (  (C_Seq c1 c2)  , mu ,  (C_Seq c1' c2)  , mu' ) : OneStep"

| E_AssignI: "[| ( e , mu , v ) : ExpEval ;
 ( mu , x , v , mu' ) : StoreUpdate|] ==>
 (  (C_Assign x e)  , mu , C_Null , mu' ) : OneStep"

| E_IfThenElse1I: "[| ( e , mu , (V_Bool  true ) ) : ExpEval|] ==>
 (  (C_IfThenElse e c1 c2)  , mu , c1 , mu ) : OneStep"

| E_IfThenElse2I: "[| ( e , mu , (V_Bool  false ) ) : ExpEval|] ==>
 (  (C_IfThenElse e c1 c2)  , mu , c2 , mu ) : OneStep"

| E_While1I: "[| ( e , mu , (V_Bool  false ) ) : ExpEval|] ==>
 (  (C_While e c)  , mu , C_Null , mu ) : OneStep"

| E_While2I: "[| ( e , mu , (V_Bool  true ) ) : ExpEval|] ==>
 (  (C_While e c)  , mu ,  (C_Seq c (C_While e c))  , mu ) : OneStep"

| E_Decl1I: " ( (C_Decl D_Empty) , mu , C_Null , mu ) : OneStep"

| E_Decl2I: "[| ( d , mu , d' , mu' ) : DeclEval|] ==>
 ( (C_Decl d) , mu , (C_Decl d') , mu' ) : OneStep"

| E_For1I: "[| ( e , mu , (V_Int k) ) : ExpEval ;
 ( e' , mu , (V_Int k') ) : ExpEval ;
 ( k  >  k' ) |] ==>
 (  (C_For x e e' c)  , mu , C_Null , mu ) : OneStep"

| E_For2I: "[| ( e , mu , (V_Int k) ) : ExpEval ;
 ( e' , mu , (V_Int k') ) : ExpEval ;
 ( k  <=  k' ) |] ==>
 (  (C_For x e e' c)  , mu ,  (C_Seq (C_Decl (D_Constant x T_Int (E_Value (V_Int k)) (D_Block
c))) (C_For x (E_Value (V_Int  ( k  +   1  ) )) (E_Value (V_Int k')) c))  , mu ) : OneStep"

| E_ProcCallI: "[| ( e , mu , (V_Proc (x'_m'_T'_list) d) ) : ExpEval ;
 ( (x'_m'_T'_list) , (e'_list) , (x_m_T_e_list) ) : Compat|] ==>
 ( (C_ProcCall e (e'_list)) , mu , (C_Decl (D_Aliases (x_m_T_e_list) d)) , mu ) : OneStep"

| (* defn DeclEval *)

E_Block1I: " ( (D_Block C_Null) , mu , D_Empty , mu ) : DeclEval"

| E_Block2I: "[| ( c , mu , c' , mu' ) : OneStep|] ==>
```

```
 ( (D_Block c) , mu , (D_Block c') , mu' ) : DeclEval"

| E_InitVar1I: " ( (D_InitVar x T e D_Empty) , mu , D_Empty , mu ) : DeclEval"

| E_InitVar2I: "[| ( e , mu , v ) : ExpEval ;
 ( d ,  (( x , v )# mu )  , d' ,  (( x , v' )# mu' )  ) : DeclEval|] ==>
 ( (D_InitVar x T e d) , mu , (D_InitVar x T (E_Value v') d') , mu' ) : DeclEval"

| E_Const1I: " ( (D_Constant x T e D_Empty) , mu , D_Empty , mu ) : DeclEval"

| E_Const2I: "[| ( e , mu , v ) : ExpEval ;
 (  (subst_dcl (E_Value  v )  x   d )  , mu , d' , mu' ) : DeclEval|] ==>
 ( (D_Constant x T e d) , mu , (D_Constant x T (E_Value v) d') , mu' ) : DeclEval"

| E_ProcI: " ( (D_Proc p (x_m_T_list) d1 d) , mu ,  (subst_dcl (E_Value  (V_Proc (x_m_T_list)
d1) )  p   d )  , mu ) : DeclEval"

| E_Alias1I: " ( (D_Alias x m T e D_Empty) , mu , D_Empty , mu ) : DeclEval"

| E_Alias2I: "[| ( e , mu , v ) : ExpEval ;
 (  (subst_dcl (E_Value  v )  x   d )  , mu , d' , mu' ) : DeclEval|] ==>
 ( (D_Alias x M_In T e d) , mu , d' , mu' ) : DeclEval"

| E_Alias3I: "[| m  ~=  M_In  ;
 ( mu , y , v ) : Fetch ;
 ( d ,  (( x , v )# mu )  , d' ,  (( x , v' )# mu' )  ) : DeclEval ;
 ( mu' , y , v' , mu'' ) : StoreUpdate|] ==>
 ( (D_Alias x m T (E_Var y) d) , mu , (D_Alias x m T (E_Var y) d') , mu'' ) : DeclEval"

| E_Aliases1I: " ( (D_Aliases [] d) , mu , d , mu ) : DeclEval"

| E_Aliases2I: " ( (D_Aliases (x_m_T_e_list) D_Empty) , mu , D_Empty , mu ) : DeclEval"

| E_Aliases3I: "[| ( (D_Alias x m T e (D_Aliases (x_m_T_e_list) d)) , mu , d' , mu' ) :
DeclEval|] ==>
 ( (D_Aliases ((x,m,T,e) # x_m_T_e_list) d) , mu , d' , mu' ) : DeclEval"


code_module Evaluation
contains

test1 = " (  ( (E_Plus (E_Value (V_Int  2 )) (E_Value (V_Int  3 ))) )  ,  Nil  ,  _  ) :
ExpEval"
test2 = " (  ( (E_Plus (E_Var ''X'') (E_Value (V_Int  3 ))) )  ,  ([(''X'',(V_Int  5 ))])  ,  _
) : ExpEval"

ML {* DSeq.hd Evaluation.test1 *}


code_module Typing (* file "Typing.sml" *)
contains

test1 = " (  Nil  , (E_Var ''X'') , T_Int ) : ExpTyping"

test2 = " (  (''X'',(VarDecl  M_In  T_Int)) # [(''Y'',(VarDecl  M_In  T_Int))])  , (E_Equal  (
(E_Plus (E_Var ''X'') (E_Value (V_Int  1 ))) )  (E_Var ''Y'')) , T_Bool ) : ExpTyping"

ML {* Typing.test1 *}
ML {* Typing.test2 *}


code_module Evaluation
contains

test1 = " (  ( (E_Plus (E_Value (V_Int  2 )) (E_Value (V_Int  3 ))) )  ,  Nil  ,  _  ) :
ExpEval"
```

```
test2 = " (  (('’X’’,(V_Int  2 )) # [(’’Y’’,(V_Int  3 ))])  , ’’X’’ , (V_Int  3 ) ,  _  ) :
StoreUpdate"

test3 = " (  (C_Assign ’’X’’ (E_Plus (E_Var ’’X’’) (E_Value (V_Int  1 ))))  ,  ([(’’X’’,(V_Int
2 ))])  ,  1  ,  _  ,  _  ) : ManySteps"

test4 = " (  (C_Seq (C_Assign ’’X’’ (E_Plus (E_Var ’’X’’) (E_Var ’’Y’’))) (C_Assign ’’Y’’
(E_Plus (E_Var ’’X’’) (E_Var ’’Y’’))))  ,  ((’’X’’,(V_Int  42 )) # [(’’Y’’,(V_Int  12 ))])  ,
_  ) : FullEvaluation"

test5 = " (  (C_IfThenElse (E_Var ’’B’’) (C_Assign ’’X’’ (E_Value (V_Int  1 ))) (C_Assign ’’Y’’
(E_Value (V_Int  1 ))))  ,  ((’’B’’,(V_Bool  true )) # (’’X’’,(V_Int  0 )) # [(’’Y’’,(V_Int  0
))])  ,  _  ) : FullEvaluation"

ML {* DSeq.hd Evaluation.test1 *}
ML {* DSeq.hd Evaluation.test2 *}
ML {* DSeq.hd Evaluation.test3 *}
ML {* DSeq.hd Evaluation.test4 *}
ML {* DSeq.hd Evaluation.test5 *}


code_module Typing (* file "Extraction.sml" *)
contains

test1 = " (  ([(’’X’’,(VarDecl M_InOut T_Int))])  , (C_Assign ’’X’’ (E_Plus (E_Var ’’X’’)
(E_Value (V_Int  1 )))) ) : CommTyping"

test2 = " (  (('’X’’,(VarDecl M_InOut T_Int)) # (’’Y’’,(VarDecl M_In T_Bool)) #
[(’’B’’,(VarDecl M_In T_Bool))])  , (C_IfThenElse (E_Var ’’B’’) (C_Assign ’’X’’ (E_Value (V_Int
1 ))) (C_Assign ’’Y’’ (E_Value (V_Int  1 )))) ) : CommTyping"

ML {* Typing.test1 *}
ML {* Typing.test2 *}


code_module Evaluation
contains

test1 = " ( (C_Decl (D_Constant ’’B’’ T_Bool (E_Value (V_Bool  false )) (D_Block (C_IfThenElse
(E_Var ’’B’’) (C_Assign ’’X’’ (E_Value (V_Int  1 ))) (C_Assign ’’Y’’ (E_Value (V_Int  1 )))))))
,  (('’X’’,(V_Int  0 )) # [(’’Y’’,(V_Int  0 ))])  ,  _  ) : FullEvaluation"

test2 = " ( (C_For ’’I’’ (E_Value (V_Int  1 )) (E_Var ’’X’’) (C_Assign ’’Y’’ (E_Plus (E_Var
’’Y’’) (E_Var ’’X’’)))) ,  (('’X’’,(V_Int  5 )) # [(’’Y’’,(V_Int  0 ))])  ,  _  ) :
FullEvaluation"

ML {* DSeq.hd Evaluation.test1 *}
ML {* DSeq.hd Evaluation.test2 *}


code_module Typing (* file "Extraction.sml" *)
contains

test1 = " (  ([(’’X’’,(VarDecl M_InOut T_Int))])  , (C_Assign ’’X’’ (E_Plus (E_Var ’’X’’)
(E_Value (V_Int  1 )))) ) : CommTyping"

test2 = " (  ([(’’X’’,(VarDecl M_InOut T_Int))])  , (C_Decl (D_InitVar ’’X’’ T_Int (E_Value
(V_Int  42 )) (D_Block (C_Seq (C_Assign ’’X’’ (E_Plus (E_Var ’’Y’’) (E_Value (V_Int  1 ))))
(C_Seq (C_Assign ’’X’’ (E_Plus (E_Var ’’X’’) (E_Value (V_Int  1 )))) (C_Assign ’’Y’’ (E_Minus
(E_Var ’’Y’’) (E_Value (V_Int  1 ))))))))) ) : CommTyping"

test3 = " (  ([(’’X’’,(VarDecl M_InOut T_Int))])  , (C_Decl (D_InitVar ’’Y’’ T_Bool (E_Value
(V_Bool  false )) (D_Block (C_For ’’I’’ (E_Value (V_Int  1 )) (E_Var ’’X’’) (C_Assign ’’X’’
(E_Plus (E_Var ’’Y’’) (E_Value (V_Int  1 ))))))) ) : CommTyping"

ML {* Typing.test1 *}
ML {* Typing.test2 *}
```

```
ML {* Typing.test3 *}


code_module Typing
contains

test1 = " (  ([(''R'',(VarDecl M_Out T_Bool))])  , (C_Decl (D_InitVar ''Y'' T_Int (E_Value
(V_Int  42 )) (D_Proc ''P'' (('' I'',M_InOut,T_Int) # [(''B'',M_Out,T_Bool)]) (D_Block (C_Assign
''B''  ( (E_Equal (E_Var ''I'') (E_Value (V_Int  1 ))) ) )) (D_Block (C_ProcCall (E_Var ''P'')
(((E_Var ''Y'')) # [((E_Var ''R''))])))))) ) : CommTyping"

test2 = " (  ([(''R'',(VarDecl M_Out T_Int))])  , (C_Decl (D_Proc ''Incr'' (('' N'',M_In,T_Int)
# [(''R'',M_Out,T_Int)]) (D_Block (C_Assign ''R'' (E_Plus (E_Var ''N'') (E_Value (V_Int  1
))))) (D_Proc ''Ack'' (('' M'',M_In,T_Int) # (''N'',M_In,T_Int) # [(''R'',M_Out,T_Int)])
(D_InitVar ''P'' (T_Proc ((M_In,T_Int) # [(M_Out,T_Int)])) (E_Var ''Incr'') (D_Block (C_Seq
(C_For ''I'' (E_Value (V_Int  1 )) (E_Var ''M'') (C_Decl (D_Proc ''Aux'' (('' S'',M_In,T_Int) #
[(''R'',M_Out,T_Int)]) (D_InitVar ''X'' T_Int (E_Value (V_Int  0 )) (D_Block (C_Seq (C_ProcCall
(E_Var ''P'') (((E_Value (V_Int  1 ))) # [((E_Var ''X''))])) (C_Seq (C_For ''J'' (E_Value
(V_Int  1 )) (E_Var ''S'') (C_ProcCall (E_Var ''P'') (((E_Var ''X'')) # [((E_Var ''X'')
)])))) (C_Assign ''R'' (E_Var ''X'')))))) (D_Block (C_Assign ''P'' (E_Var ''Aux'')))))) (C_ProcCall
(E_Var ''P'') (((E_Var ''N'')) # [((E_Var ''R''))])))))) (D_Block (C_ProcCall (E_Var ''Ack'')
(((E_Value (V_Int  2 ))) # ((E_Value (V_Int  2 ))) # [((E_Var ''R''))]))))))) ) : CommTyping"

test24 = " (  ([(''R'',(VarDecl M_Out T_Int))])  , (C_Decl (D_Proc ''Comp''
([(''P1'',M_In,(T_Proc ((M_In,T_Int) # [(M_Out,T_Int)])))] @ [(''P2'',M_In,(T_Proc
((M_In,T_Int) # [(M_Out,T_Int)])))] @ [(''P3'',M_Out,(T_Proc ((M_In,T_Int) #
[(M_Out,T_Int)])))]) (D_Proc ''P'' (('' N'',M_In,T_Int) # [(''R'',M_Out,T_Int)]) (D_InitVar
''X'' T_Int (E_Value (V_Int  0 )) (D_Block (C_Seq (C_ProcCall (E_Var ''P1'') (((E_Var ''N'')) #
[((E_Var ''X''))])) (C_ProcCall (E_Var ''P2'') (((E_Var ''X'')) # [((E_Var ''R''))])))))
(D_Block (C_Assign ''P3'' (E_Var ''P'')))) (D_Proc ''Incr'' (('' N'',M_In,T_Int) #
[(''R'',M_Out,T_Int)]) (D_Block (C_Assign ''R'' (E_Plus (E_Var ''N'') (E_Value (V_Int  1 )))))
(D_Proc ''IncrN'' (('' M'',M_In,T_Int) # (''N'',M_In,T_Int) # [(''R'',M_Out,T_Int)]) (D_InitVar
''P'' (T_Proc ((M_In,T_Int) # [(M_Out,T_Int)])) (E_Var ''Incr'') (D_Block (C_Seq (C_For ''I''
(E_Value (V_Int  1 )) (E_Var ''N'') (C_ProcCall (E_Var ''Comp'') (((E_Var ''P'')) # ((E_Var
''P'')) # [((E_Var ''P''))])))) (C_ProcCall (E_Var ''P'') (((E_Var ''M'')) # [((E_Var
''R''))])))))) (D_Block (C_ProcCall (E_Var ''IncrN'') (((E_Value (V_Int  3 ))) # ((E_Value
(V_Int  3 ))) # [((E_Var ''R''))])))))))) ) : CommTyping"

ML {* Typing.test1 *}
ML {* Typing.test2 *}
ML {* Typing.test24 *}


code_module Evaluation
contains

test1 = " ( (C_Decl (D_InitVar ''Y'' T_Int (E_Value (V_Int  42 )) (D_Proc ''P''
(('' I'',M_InOut,T_Int) # [(''B'',M_Out,T_Bool)]) (D_Block (C_Assign ''B''  ( (E_Equal (E_Var
''I'') (E_Value (V_Int  1 ))) ) )) (D_Block (C_ProcCall (E_Var ''P'') (((E_Var ''Y'')) #
[((E_Var ''R''))])))))) ,  ([(''R'',(V_Bool  false ))])  ,  1  ,  _  ,  _  ) : ManySteps"

test2 = " ( (C_Decl (D_InitVar ''Y'' T_Int (E_Value (V_Int  42 )) (D_Proc ''P''
(('' I'',M_InOut,T_Int) # [(''B'',M_Out,T_Bool)]) (D_Block (C_Assign ''B''  ( (E_Equal (E_Var
''I'') (E_Value (V_Int  1 ))) ) )) (D_Block (C_ProcCall (E_Var ''P'') (((E_Var ''Y'')) #
[((E_Var ''R''))])))))) ,  ([(''R'',(V_Bool  false ))])  ,  2  ,  _  ,  _  ) : ManySteps"

test3 = " ( (C_Decl (D_InitVar ''Y'' T_Int (E_Value (V_Int  42 )) (D_Proc ''P''
(('' I'',M_In,T_Int) # [(''B'',M_Out,T_Bool)]) (D_Block (C_Assign ''B''  ( (E_Equal (E_Var
''I'') (E_Value (V_Int  1 ))) ) )) (D_Block (C_ProcCall (E_Var ''P'') (((E_Var ''Y'')) #
[((E_Var ''R''))])))))) ,  ([(''R'',(V_Bool  false ))])  ,  3  ,  _  ,  _  ) : ManySteps"

test10 = " ( (C_Decl (D_InitVar ''Y'' T_Int (E_Value (V_Int  42 )) (D_Proc ''P''
(('' I'',M_In,T_Int) # [(''B'',M_Out,T_Bool)]) (D_Block (C_Assign ''B''  ( (E_Equal (E_Var
''I'') (E_Value (V_Int  1 ))) ) )) (D_Block (C_ProcCall (E_Var ''P'') (((E_Var ''Y'')) #
[((E_Var ''R''))])))))) ,  ([(''R'',(V_Bool  true ))])  ,  _  ) : FullEvaluation"

test20 = " ( (C_Decl (D_InitVar ''Y'' T_Int (E_Value (V_Int  42 )) (D_Proc ''P''
```

```
((''I'',M_In,T_Int) # [(''B'',M_Out,T_Bool)]) (D_Block (C_Assign ''B''  ( (E_Equal (E_Var
''I'') (E_Value (V_Int  1 ))) ) )) (D_Block (C_ProcCall (E_Var ''P'') (((E_Var ''Y'')) #
[((E_Var ''R''))])))))) ,  ([(''R'',(V_Bool  false ))])  ,  20  ,  _  ) : Trace"

test24 = " ( (C_Decl (D_Proc ''Incr'' ((''N'',M_In,T_Int) # [(''R'',M_Out,T_Int)]) (D_Block
(C_Assign ''R'' (E_Plus (E_Var ''N'') (E_Value (V_Int  1 ))))) (D_Proc ''Plus''
((''M'',M_In,T_Int) # (''N'',M_In,T_Int) # [(''R'',M_Out,T_Int)]) (D_InitVar ''X'' T_Int (E_Var
''M'') (D_Block (C_Seq (C_For ''I'' (E_Value (V_Int  1 )) (E_Var ''N'') (C_ProcCall (E_Var
''Incr'') (((E_Var ''X'')) # [((E_Var ''X''))])))) (C_Assign ''R'' (E_Var ''X'')))))) (D_Block
(C_ProcCall (E_Var ''Plus'') (((E_Value (V_Int  3 ))) # ((E_Value (V_Int  5 ))) # [((E_Var
''R''))])))))) ,  ([(''R'',(V_Int  0 ))])  ,  _  ) : FullEvaluation"

test25 = " ( (C_Decl (D_Proc ''Comp'' ([(''P1'',M_In,(T_Proc ((M_In,T_Int) #
[(M_Out,T_Int)])))] @ [(''P2'',M_In,(T_Proc ((M_In,T_Int) # [(M_Out,T_Int)])))] @
[(''P3'',M_Out,(T_Proc ((M_In,T_Int) # [(M_Out,T_Int)])))]) (D_Proc ''P'' ((''N'',M_In,T_Int) #
[(''R'',M_Out,T_Int)]) (D_InitVar ''X'' T_Int (E_Value (V_Int  0 )) (D_Block (C_Seq (C_ProcCall
(E_Var ''P1'') (((E_Var ''N'')) # [((E_Var ''X''))])) (C_ProcCall (E_Var ''P2'') (((E_Var
''X'')) # [((E_Var ''R''))])))))) (D_Block (C_Assign ''P3'' (E_Var ''P'')))) (D_Proc ''Incr''
((''N'',M_In,T_Int) # [(''R'',M_Out,T_Int)]) (D_Block (C_Assign ''R'' (E_Plus (E_Var ''N'')
(E_Value (V_Int  1 ))))) (D_Proc ''IncrN'' ((''M'',M_In,T_Int) # (''N'',M_In,T_Int) #
[(''R'',M_Out,T_Int)]) (D_InitVar ''P'' (T_Proc ((M_In,T_Int) # [(M_Out,T_Int)])) (E_Var
''Incr'') (D_Block (C_Seq (C_For ''I'' (E_Value (V_Int  1 )) (E_Var ''N'') (C_ProcCall (E_Var
''Comp'') (((E_Var ''P'')) # ((E_Var ''P'')) # [((E_Var ''P''))])))) (C_ProcCall (E_Var ''P'')
(((E_Var ''M'')) # [((E_Var ''R''))])))))) (D_Block (C_ProcCall (E_Var ''IncrN'') (((E_Value
(V_Int  3 ))) # ((E_Value (V_Int  3 ))) # [((E_Var ''R''))])))))))) ,  ([(''R'',(V_Int  0 ))])
,  _  ) : FullEvaluation"

test30 = " ( (C_Decl (D_Proc ''Incr'' ((''N'',M_In,T_Int) # [(''R'',M_Out,T_Int)]) (D_Block
(C_Assign ''R'' (E_Plus (E_Var ''N'') (E_Value (V_Int  1 ))))) (D_Proc ''Ack''
((''M'',M_In,T_Int) # (''N'',M_In,T_Int) # [(''R'',M_Out,T_Int)]) (D_InitVar ''P'' (T_Proc
((M_In,T_Int) # [(M_Out,T_Int)])) (E_Var ''Incr'') (D_Block (C_Seq (C_For ''I'' (E_Value (V_Int
1 )) (E_Var ''M'') (C_Decl (D_Constant ''Q'' (T_Proc ((M_In,T_Int) # [(M_Out,T_Int)])) (E_Var
''P'') (D_Proc ''Aux'' ((''S'',M_In,T_Int) # [(''R'',M_Out,T_Int)]) (D_InitVar ''X'' T_Int
(E_Value (V_Int  0 )) (D_Block (C_Seq (C_ProcCall (E_Var ''Q'') (((E_Value (V_Int  1 ))) #
[((E_Var ''X''))])) (C_Seq (C_For ''J'' (E_Value (V_Int  1 )) (E_Var ''S'') (C_ProcCall (E_Var
''Q'') (((E_Var ''X'')) # [((E_Var ''X''))]))) (C_Assign ''R'' (E_Var ''X'')))))) (D_Block
(C_Assign ''P'' (E_Var ''Aux''))))))) (C_ProcCall (E_Var ''P'') (((E_Var ''N'')) # [((E_Var
''R''))])))))) (D_Block (C_ProcCall (E_Var ''Ack'') (((E_Value (V_Int  3 ))) # ((E_Value (V_Int
2 ))) # [((E_Var ''R''))])))))) ,  ([(''R'',(V_Int  0 ))])  ,  _  ) : FullEvaluation"

ML {* print_depth 1000 *}
ML {* DSeq.hd Evaluation.test24 *}
ML {* DSeq.hd Evaluation.test25 *}
ML {* DSeq.hd Evaluation.test30 *}
ML {* val trace = DSeq.hd Evaluation.test24 *}
ML {* List.nth (trace, 0) *}
ML {* List.nth (trace, 1) *}
ML {* List.nth (trace, 2) *}



end
```

References

[BN02] S. Berghofer and T. Nipkow. Executing higher order logic. In *In Proc. TYPES Working Group Annual Meeting 2000, LNCS*, pages 24–40. Springer-Verlag, 2002.

[CPV09] T. Crolard, E. Polonowski, and P. Valarcher. Extending the loop language with higher-order procedural variables. *Special issue of ACM TOCL on Implicit Computational Complexity. To appear.*, 10(4):1–36, 2009.

[NPW02] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[SNO+07] P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: effective tool support for the working semanticist. *SIGPLAN Not.*, 42(9):1–12, 2007.