

1 GhostLambda: Lambda-Calculus with Ghost Code

1.1 $g\lambda$ -Calculus Syntax and Semantics

$p ::=$ $\text{var } r_{\text{ref } \tau}^{\mathfrak{B}} = v; \quad p$ t	PROGRAMS <i>reference declaration</i> <i>body</i>	$c ::=$ $0 \mid 1 \mid \dots \mid n$ $\text{true} \mid \text{false}$ $()$ op	CONSTANTS <i>integer</i> <i>boolean</i> <i>unit</i> <i>built-in operators</i>
$t ::=$ c $x_{\tau}^{\mathfrak{B}}$ $\lambda x_{\tau}^{\mathfrak{B}}. t$ $v \ v$ $\text{ghost } t$ $\text{let } x_{\tau}^{\mathfrak{B}} = t \text{ in } t$ $\text{if } v \text{ then } t \text{ else } t$ $\text{rec } f^{\mathfrak{B}} \ x_{\tau}^{\mathfrak{B}} : \tau. t$ $!r_{\text{ref } \tau}^{\mathfrak{B}}$ $r_{\text{ref } \tau}^{\mathfrak{B}} := v$	TERMS <i>constant</i> <i>variable</i> <i>abstraction</i> <i>application</i> <i>ghost term</i> <i>local binding</i> <i>if-branching</i> <i>recursive function</i> <i>reference access</i> <i>reference assignment</i>	$op ::=$ $+ \mid - \mid = \mid \text{not}$	BUILT-IN OPERATORS <i>built-in operators</i>
$v ::=$ c $x_{\tau}^{\mathfrak{B}}$ $\lambda x_{\tau}^{\mathfrak{B}}. t$ $\text{rec } f^{\mathfrak{B}} \ x_{\tau}^{\mathfrak{B}} : \tau. t$	VALUES <i>constant</i> <i>variable</i> <i>abstraction</i> <i>recursive function</i>	$\tau ::=$ $\text{int} \mid \text{bool} \mid \text{unit}$ $\tau^{\mathfrak{B}} \rightarrow \tau$	TYPES <i>built-in types</i> <i>function type</i>
		$\text{ref } \tau$	REFERENCE'S TYPE
		$\mathfrak{B} ::=$ $\perp (*\text{false}*)$ $\top (*\text{true}*)$	GHOST INDICATOR <i>raw code</i> <i>ghost code</i>

Figure 1: *ghost-ml* Syntax

$$\begin{aligned}
\delta(+, n, m) &\triangleq ||n + m|| \text{ (where } n, m \text{ are integers)} \\
\delta(-, n, m) &\triangleq ||n - m|| \text{ (idem)} \\
\delta(\text{not}, b) &\triangleq ||\neg b|| \text{ (where } b \text{ is a boolean)} \\
\delta(=, t, u) &\triangleq ||t =_\tau u|| \text{ (where } t \text{ and } u \text{ are both of the same type } \tau)
\end{aligned}$$

Figure 2: *ghost-ml* Semantics (Delta Rules)

$$\begin{aligned}
op \ v_1 \dots v_k &\xrightarrow{\epsilon}_{|\mu} \delta(c, v_1 \dots v_k)_{|\mu} \text{ if } k = \text{Arity}(op) \text{ and } \delta(c, v) \text{ is defined} \quad (\text{E-OP}) \\
op \ v_1 \dots v_k_{|\mu} &\xrightarrow{\epsilon} \lambda x_\tau^{\mathfrak{B}}. op \ v_1 \dots v_k \ x_\tau^{\mathfrak{B}} \text{ if } 1 \leq k < \text{arity}(op) \quad (\text{E-CLO}) \\
(\lambda x_\tau^{\mathfrak{B}}. t) v_{|\mu} &\xrightarrow{\epsilon} t[x_\tau^{\mathfrak{B}} \leftarrow v]_{|\mu} \quad (\text{E-APPFUN}) \\
(\text{rec } f_{\tau_2^{\mathfrak{B}} \rightarrow \tau_1}^{\mathfrak{B}_1} \ x_{\tau_2^{\mathfrak{B}_2}}^{\mathfrak{B}_2}. t) v_{|\mu} &\xrightarrow{\epsilon} t[x_{\tau_2^{\mathfrak{B}_2}}^{\mathfrak{B}_2} \leftarrow v, f_{\tau_2^{\mathfrak{B}} \rightarrow \tau_1}^1 \leftarrow \text{rec } f_{\tau_2^{\mathfrak{B}} \rightarrow \tau_1}^{\mathfrak{B}_1} \ x_{\tau_2^{\mathfrak{B}_2}}^{\mathfrak{B}_2}. t]_{|\mu} \quad (\text{E-APPREC}) \\
\text{let } x_\tau^{\mathfrak{B}} = v_1 \text{ in } t_2_{|\mu} &\xrightarrow{\epsilon} t_2[x_\tau^{\mathfrak{B}} \leftarrow v_1]_{|\mu} \quad (\text{E-LETV}) \\
\text{if true then } t_1 \text{ else } t_2_{|\mu} &\xrightarrow{\epsilon} t_1_{|\mu} \quad (\text{E-IF-TRUE}) \\
\text{if false then } t_1 \text{ else } t_2_{|\mu} &\xrightarrow{\epsilon} t_2_{|\mu} \quad (\text{E-IF-FALSE}) \\
\text{ghost } t_{|\mu} &\xrightarrow{\epsilon} t_{|\mu} \quad (\text{E-DEGHOST}) \\
!r_{\text{ref } \tau}^{\mathfrak{B}}_{|\mu} &\xrightarrow{\epsilon} \mu(r_{\text{ref } \tau}^{\mathfrak{B}}) \quad (\text{E-DEREF}) \\
r_{\text{ref } \tau}^{\mathfrak{B}} := v_{|\mu} &\xrightarrow{\epsilon} ()_{|\mu[r_{\text{ref } \tau}^{\mathfrak{B}} \leftarrow v]} \quad (\text{E-ASSIGN})
\end{aligned}$$

Figure 3: *ghost-ml* Semantics (Head Reduction Rules)

$$\frac{t|_{\mu} \xrightarrow{g\lambda} t'|_{\mu'}}{t|_{\mu} \rightarrow_{g\lambda} t'|_{\mu'}} \quad (\text{E-HEAD})$$

$$\frac{t_1|_{\mu} \rightarrow t'_1|_{\mu'}}{\text{let } x_{\tau}^{\mathfrak{B}} = t_1 \text{ in } t_2|_{\mu} \rightarrow \text{let } x_{\tau}^{\mathfrak{B}} = t'_1 \text{ in } t_2|_{\mu'}} \quad (\text{E-CONTEXT})$$

Figure 4: *ghost-ml* Semantics Context Rules

1.2 Typing Relation

$\frac{\text{Typeof}(c) = \tau}{\vdash_{g\lambda} c : (\tau, \perp, \perp)} \quad (\text{T-CONST})$	$\frac{}{\vdash_{g\lambda} x_{\tau}^{\mathfrak{B}} : (\tau, \mathfrak{B}, \perp)} \quad (\text{T-VAR})$
$\frac{\vdash_{g\lambda} v : (\mathfrak{B}, \tau, \perp)}{\vdash_{g\lambda} \text{var } r_{\text{ref } \tau}^{\mathfrak{B}} = v : (\text{ref } \tau, \mathfrak{B}, \perp)} \quad (\text{T-DECL})$	
$\frac{\vdash_{g\lambda} t_1 : (\tau_1, \mathfrak{B}_1, \Sigma_1) \quad \overline{\mathfrak{B}_1 \wedge \Sigma_1}}{\vdash_{g\lambda} \lambda x_{\tau_2}^{\mathfrak{B}_2}. t_1 : (\tau_2^{\mathfrak{B}_2} \xrightarrow{\Sigma_1} \tau_1, \mathfrak{B}_1, \perp)} \quad (\text{T-ABS})$	
$\frac{\vdash_{g\lambda} t_1 : (\tau_2^{\mathfrak{B}_2} \xrightarrow{\Sigma_0} \tau_1, \mathfrak{B}_1, \Sigma_1) \quad \vdash_{g\lambda} t_2 : (\tau_2, \mathfrak{B}'_2, \Sigma_2) \quad \overline{(\mathfrak{B}_1 \vee (\overline{\mathfrak{B}_2 \wedge \mathfrak{B}'_2})) \wedge (\Sigma_0 \vee \Sigma_1 \vee \Sigma_2)} \quad \overline{\mathfrak{B}'_2 \wedge \Sigma_2} \quad \mathfrak{B}_2 \Rightarrow \mathfrak{B}'_2}{\vdash_{g\lambda} t_1 t_2 : (\tau_1, \mathfrak{B}_1 \vee (\overline{\mathfrak{B}_2 \wedge \mathfrak{B}'_2}), \Sigma_0 \vee \Sigma_1 \vee \Sigma_2)} \quad (\text{T-APP})$	
$\frac{\vdash_{g\lambda} t_1 : (\text{bool}, \mathfrak{B}_1, \Sigma_1) \quad \vdash_{g\lambda} t_2 : (\tau_0, \mathfrak{B}_2, \Sigma_2) \quad \vdash_{g\lambda} t_3 : (\tau_0, \mathfrak{B}_3, \Sigma_3) \quad \overline{(\mathfrak{B}_1 \vee \mathfrak{B}_2 \vee \mathfrak{B}_3) \wedge (\Sigma_1 \vee \Sigma_2 \vee \Sigma_3)}}{\vdash_{g\lambda} \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : (\tau_0, \mathfrak{B}_1 \vee \mathfrak{B}_2 \vee \mathfrak{B}_3, \Sigma_1 \vee \Sigma_2 \vee \Sigma_3)} \quad (\text{T-IF})$	
$\frac{\vdash_{g\lambda} t_1 : (\tau_1, \mathfrak{B}_1, \Sigma_1) \quad \overline{\mathfrak{B}_1 \wedge \Sigma_1}}{\vdash_{g\lambda} \text{rec } f^{\mathfrak{B}_1} x_{\tau_2}^{\mathfrak{B}_2} : \tau_1. t : (\tau_2^{\mathfrak{B}_2} \xrightarrow{\Sigma_1} \tau_1, \mathfrak{B}_1, \perp)} \quad (\text{T-REC})$	
$\frac{\vdash_{g\lambda} t_1 : (\tau_1, \mathfrak{B}_1, \Sigma_1) \quad \vdash_{g\lambda} t_2 : (\tau_2, \mathfrak{B}'_2, \Sigma_2) \quad \overline{\mathfrak{B}'_2 \wedge \Sigma_2} \quad \overline{(\mathfrak{B}_1 \vee (\overline{\mathfrak{B}_2 \wedge \mathfrak{B}'_2})) \wedge (\Sigma_1 \vee \Sigma_2)} \quad \mathfrak{B}_2 \Rightarrow \mathfrak{B}'_2}{\vdash_{g\lambda} \text{let } x_{\tau_2}^{\mathfrak{B}_2} = t_2 \text{ in } t_1 : (\tau_1, \mathfrak{B}_1 \vee (\overline{\mathfrak{B}_2 \wedge \mathfrak{B}'_2}), \Sigma_1 \vee \Sigma_2)} \quad (\text{T-LET})$	
$\frac{\vdash_{g\lambda} t : (\tau, \mathfrak{B}, \perp)}{\vdash_{g\lambda} \text{ghost } t : (\tau, \top, \perp)} \quad (\text{T-GHOST})$	$\frac{}{\vdash_{g\lambda} !r_{\text{ref } \tau}^{\mathfrak{B}} : (\tau, \mathfrak{B}, \perp)} \quad (\text{T-DEREF})$
$\frac{\vdash_{g\lambda} t_2 : (\tau_2, \mathfrak{B}_2, \Sigma_2) \quad \overline{(\mathfrak{B}_1 \vee \mathfrak{B}_2) \wedge \Sigma_2} \quad \mathfrak{B}_2 \Rightarrow \mathfrak{B}_1}{\vdash_{g\lambda} r_{\text{ref } \tau_2}^{\mathfrak{B}_1} := t_2 : (\text{unit}, \mathfrak{B}_1, \overline{\mathfrak{B}_1 \vee \Sigma_2})} \quad (\text{T-ASSIGN})$	

Figure 5: *ghost*-ml Typing Rules with Effects

Invariant of Typing Relation

The following lemma states that in well typed terms the ghost code does not write in any not ghost global references.

Lemma 1.1 [INVARIANT OF TYPING RELATION].

If $\vdash_{g\lambda} t : (\tau, \mathfrak{B}, \Sigma)$ is a well typed term, then for any sub-term t_1 of t , such that $\vdash_{g\lambda} t_1 : (\tau_1, \mathfrak{B}_1, \Sigma_1)$, the condition $\overline{\mathfrak{B}_1 \wedge \Sigma_1}$ holds.

Proof. By induction on typing derivations and case analysis (the interesting cases are (T-APP) and (T-LET) where the resulting invariant condition does not cover the case when ghost code is used as argument by a non-ghost function, so that argument invariant condition must be written in typing rules explicitly). \square

Ghost-Code Propagation

$\mathbf{t_1 \ t_2 : \vdash_{g\lambda} t_1 : (\tau_2^{\mathfrak{B}_2} \xrightarrow{\Sigma_0} \tau_1, \mathfrak{B}_1, \Sigma_1) \quad \vdash_{g\lambda} t_2 : (\tau_2, \mathfrak{B}'_2, \Sigma_2) \quad \mathfrak{B}_2 \Rightarrow \mathfrak{B}'_2}$				
\mathfrak{B}_2	\mathfrak{B}_1	\mathfrak{B}'_2	result	
\perp	\perp	\perp	\perp	raw code application
\top	\perp	\top	\perp	ghost-code passing inside normal code
\top	\top	\top	\top	ghost code application
\perp	\top	\perp	\top	raw code passing inside ghost code
\perp	\top	\top	\top	formal parameter contamination
\perp	\perp	\top	\top	function parameter and body contamination
\top	\perp	\perp	—	impossible
\top	\top	\perp	—	impossible

1.3 Ghost Code Erasure

Definition 1.1 (Type-Erasure). We define type-erasure function (parametrized by $\{\perp, \text{top}\}$) by induction on the structure of types :

$$\begin{aligned}\mathcal{E}_{\top}(\tau) &= \text{unit} \\ \mathcal{E}_{\perp}(\tau_2^{\mathfrak{B}_2} \rightarrow \tau_1) &= \mathcal{E}_{\mathfrak{B}_2}(\tau_2) \rightarrow \mathcal{E}_{\perp}(\tau_1). \\ \mathcal{E}_{\perp}(\tau_1) &= \tau_1 \quad \text{otherwise.}\end{aligned}$$

Also $\mathcal{E}_{\mathfrak{B}}(\text{ref } \tau) = \text{ref } \mathcal{E}_{\mathfrak{B}}(\tau)$.

Definition 1.2 (Term-Erasure). Let \mathfrak{t} be a term such that $\vdash_{g\lambda} t : (\tau, \mathfrak{B}, \Sigma)$ holds. We define term-erasure function $\mathcal{E}_{\mathfrak{B}}(t)$ by induction on the structure of \mathfrak{t} :

$$\begin{aligned}\mathcal{E}_{\top}(t_1) &= () \quad \text{where } \vdash_{g\lambda} t_1 : (\tau_1, \top, \perp). \\ \mathcal{E}_{\perp}(c) &= c \\ \mathcal{E}_{\perp}(x_{\tau}^{\mathfrak{B}}) &= x_{\mathcal{E}_{\perp}(\tau)} \\ \mathcal{E}_{\perp}(\lambda x_{\tau_2^{\mathfrak{B}_2}}.t) &= \lambda x_{\mathcal{E}_{\mathfrak{B}_2}(\tau_2)}. \mathcal{E}_{\perp}(t_1) \quad \text{where } \vdash_{g\lambda} t_1 : (\tau_1, \perp, \Sigma). \\ \mathcal{E}_{\perp}(t_1 \ t_2) &= \mathcal{E}_{\perp}(t_1) \ \mathcal{E}_{\mathfrak{B}_2}(t_2) \\ \text{where } \vdash_{g\lambda} t_1 : (\tau_2^{\mathfrak{B}_2} \xrightarrow{\Sigma_0} \tau_1, \perp, \Sigma_1) \text{ and } \vdash_{g\lambda} t_2 : (\tau_2, \mathfrak{B}'_2, \Sigma_2). \\ \mathcal{E}_{\perp}(\text{let } x_{\tau_2^{\mathfrak{B}_2}}^{\mathfrak{B}_2} = t_2 \text{ in } t_1) &= \text{let } x_{\mathcal{E}_{\mathfrak{B}_2}(\tau_2)} = \mathcal{E}_{\mathfrak{B}_2}(t_2) \text{ in } \mathcal{E}_{\perp}(t_1) \\ \mathcal{E}_{\perp}(\text{rec } f^{\perp} x_{\tau_2^{\mathfrak{B}_2}}^{\mathfrak{B}_2} : \tau_1. t_1) &= \text{rec } f \ x_{\mathcal{E}_{\mathfrak{B}_2}(\tau_2)} : \mathcal{E}_{\perp}(\tau_1). \mathcal{E}_{\perp}(t_1) \\ \mathcal{E}_{\perp}(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) &= \text{if } \mathcal{E}_{\perp}(t_1) \text{ then } \mathcal{E}_{\perp}(t_2) \text{ else } \mathcal{E}_{\perp}(t_3) \\ \mathcal{E}_{\perp}(!r_{\text{ref } \tau}^{\perp}) &= !r_{\text{ref } \mathcal{E}_{\perp}(\tau)} \\ \mathcal{E}_{\perp}(r_{\text{ref } \tau}^{\perp} := t_2) &= r_{\text{ref } \mathcal{E}_{\perp}(\tau)} := \mathcal{E}_{\perp}(t_2)\end{aligned}$$

Definition 1.3 (Global-Variable-Erasure).

$$\begin{aligned}\mathcal{E}_{\top}(\text{var } r_{\text{ref } \tau}^{\top} = v) &= \emptyset \\ \mathcal{E}_{\perp}(\text{var } r_{\text{ref } \tau}^{\perp} = v) &= \text{var } r_{\text{ref } \mathcal{E}_{\perp}(\tau)} : \mathcal{E}_{\perp}(v)\end{aligned}$$

Definition 1.4 (Memory-Erasure).

$$\mathcal{E}_{\perp}(\mu) = \left\{ \left(r_{\text{ref } \mathcal{E}_{\perp}(\tau)}, \mathcal{E}_{\perp}(\mu(r_{\text{ref } \tau}^{\perp})) \right) \right\}$$

Figure 6: *ghost-ml* Ghost-Code Erasure

1.4 Properties of ghost erasure

Ghost code is a part of program specification. Suppose we have a source program p that we want to specify using ghost code inside.

First of all, if our specification predicts statically some dynamic behaviour that p does not even have, it is simply unsound.

$$\begin{array}{ccc}
 t|_{\mu} : (_, \perp, _) & \xrightarrow[g\lambda]{\epsilon} & t'|_{\mu'} : (_, \perp, _) \\
 \mathcal{E}_{\perp}(t|_{\mu}) \Downarrow & & \Downarrow \mathcal{E}_{\perp}(t'|_{\mu'}) \\
 p|_{\mu_0} & \cdots \cdots \cdots / \cdots \cdots \cdots & q|_{\mu_1}
 \end{array}$$

Figure 7: The ghost-ML "non ghost" head reduction from a ghostML term t (program p 's specification) to t' , to t' does not correspond to any of reductions of p .

On the other hand, if some dynamic behaviour of p escapes from the specification, then such a specification does not reflect properly it's meaning, so it is useless for establishing the correctness of p :

$$\begin{array}{ccc}
 t|_{\mu} & \xrightarrow{g\lambda} & t'|_{\mu'} \\
 \mathcal{E}_{\perp}(t|_{\mu}) \Downarrow & & \searrow \mathcal{E}_{\perp}(t'|_{\mu'}) \\
 p|_{\mu_0} & \xrightarrow{\lambda} & p'|_{\mu'_0} \neq -
 \end{array}$$

Figure 8: None of ghost-ML term's t (p 's specification) reductions can reflect the reduction step from source program p to p' .

In this section we check the absence of those two pathological situations, using a technique called *bi-simulation*, which consists of the following two theorems:

Theorem 1.5 [FORWARD SIMULATION]. *If t is a closed ghost-ML term, such that $\vdash_{g\lambda} t : (\tau, \perp, \Sigma)$ holds and $t|_v \rightarrow_{g\lambda}^* \mu|_{\mu'}$ for some value v , then $\mathcal{E}_{\perp}(t)|_{\mathcal{E}_{\perp}(\mu)} \rightarrow_{\lambda}^* \mathcal{E}_{\perp}(v)|_{\mathcal{E}_{\perp}(\mu')}$.*

Theorem 1.6 [BACKWARD SIMULATION]. *If t is a closed ghost-ML term, such that $\vdash_{g\lambda} t : (\tau, \perp, \Sigma)$ holds and $\mathcal{E}_{\perp}(t)|_{\mathcal{E}_{\perp}(\mu)} = t_0|_{\mu_0} \rightarrow_{\lambda} t_1|_{\mu_1}$ for some ML value t_1 and*

store μ_1 , then there exist a ghost-ML term t' and a store μ' such that $t_1 = \mathcal{E}_\perp(t')$, $\mu_1 = \mathcal{E}_\perp(\mu')$ and $t|_\mu \rightarrow_{g\lambda}^* t'|_{\mu'}$.

We begin by stating auxiliary lemmas which will help us deal with proving these theorems.

1.4.1 Forward simulation

Lemma 1.2 [ONE STEP FORWARD SIMULATION]. If t is a closed ghost-ML term, such that $\vdash_{g\lambda} t : (\tau, \perp, \Sigma)$ holds and $t|_\mu \rightarrow_{g\lambda} t'|_{\mu'}$, then $\mathcal{E}_\perp(t)_{\mathcal{E}_\perp(\mu)} \rightarrow^{0|1} \mathcal{E}_\perp(t')_{\mathcal{E}_\perp(\mu')}$.

Proof. By induction on the evaluation relation $t|_\mu \rightarrow_{g\lambda} t'|_{\mu'}$.

(α)

(β) Otherwise, t is of the form $\text{let } x_{\tau_2}^{\mathfrak{B}_2} = t_2 \text{ in } t_1$ and the only case to consider is:

$$\frac{t_2|_\mu \rightarrow t'_2|_{\mu'}}{\text{let } x_{\tau_2}^{\mathfrak{B}_2} = t_2 \text{ in } t_1|_\mu \rightarrow \text{let } x_{\tau_2}^{\mathfrak{B}_2} = t'_2 \text{ in } t_1|_{\mu'}} \quad (\text{E-CONTEXT})$$

with the typing

$$\frac{\vdash_{g\lambda} t_1 : (\tau_1, \mathfrak{B}_1, \Sigma_1) \quad \vdash_{g\lambda} t_2 : (\tau_2, \mathfrak{B}'_2, \Sigma_2) \quad \mathfrak{B}_2 \Rightarrow \mathfrak{B}'_2}{\vdash_{g\lambda} \text{let } x_{\tau_2}^{\mathfrak{B}_2} = t_2 \text{ in } t_1 : (\tau_1, \mathfrak{B}_1 \vee (\overline{\mathfrak{B}_2} \wedge \mathfrak{B}'_2), \Sigma_1 \vee \Sigma_2)} \quad (\text{T-LET})$$

By hypothesis, t itself is not ghost, so looking at the typing of let , we can deduce that either $\mathfrak{B}_2 = \mathfrak{B}'_2 = \top$ or $\mathfrak{B}_2 = \mathfrak{B}'_2 = \perp$.

(β_1) If term t_2 is ghost, then

$$\mathcal{E}_\perp(\text{let } x_{\tau_2}^\top = t_2 \text{ in } t_1) = \text{let } x_{\text{unit}} = () \text{ in } \mathcal{E}_\perp(t_1) = \mathcal{E}_\perp(\text{let } x_{\tau_2}^\top = t'_2 \text{ in } t_1)$$

That is, $\mathcal{E}_\perp(t)|_{\mathcal{E}_\perp(\mu)} \rightarrow^0 \mathcal{E}_\perp(t')|_{\mathcal{E}_\perp(\mu')}$.

(β_2) Otherwise, the binding of $x_{\tau_2}^{\mathfrak{B}_2}$ is non-ghost and we can apply the induction hypothesis on $t_2|_\mu \rightarrow_{g\lambda} t'_2|_{\mu'}$:

$$\mathcal{E}_\perp(t_2)|_{\mathcal{E}_\perp(\mu)} \rightarrow^{0|1} \mathcal{E}_\perp(t'_2)|_{\mathcal{E}_\perp(\mu')}$$

and conclude in both cases $(\mathcal{E}_\perp(t_2)|_{\mathcal{E}_\perp(\mu)} = \mathcal{E}_\perp(t'_2)|_{\mathcal{E}_\perp(\mu')})$ or $\mathcal{E}_\perp(t_2)|_{\mathcal{E}_\perp(\mu)} \rightarrow \mathcal{E}_\perp(t'_2)|_{\mathcal{E}_\perp(\mu')}$ with application of E-CONTEXT rule in ML:

$$\frac{\mathcal{E}_\perp(t_2)|_{\mathcal{E}_\perp(\mu)} \rightarrow^{0|1} \mathcal{E}_\perp(t'_2)|_{\mathcal{E}_\perp(\mu')}}{\mathcal{E}_\perp(t)|_{\mathcal{E}_\perp(\mu)} \rightarrow^{0|1} \mathcal{E}_\perp(t')|_{\mathcal{E}_\perp(\mu')}} \quad (\text{E-CONTEXT})$$

□

Theorem 1.7 [FORWARD SIMULATION]. *If t is a closed ghost-ML term, such that $\vdash_{g\lambda} t : (\tau, \perp, \Sigma)$ holds and $t|_{\mu} \rightarrow_{g\lambda}^* v|_{\mu'}$ for some value v , then $\mathcal{E}_{\perp}(t)|_{\mathcal{E}_{\perp}(\mu)} \rightarrow_{\lambda}^* \mathcal{E}_{\perp}(v)|_{\mathcal{E}_{\perp}(\mu')}$.*

Proof. By induction on the length of the evaluation of $t|_{\mu} \rightarrow_{g\lambda}^* v|_{\mu'}$.

(α) if $t|_{\mu} \xrightarrow{\epsilon}_{g\lambda} v|_{\mu'}$ then by *one-step* forward simulation lemma,

$$\mathcal{E}_{\perp}(t)|_{\mathcal{E}_{\perp}(\mu)} \rightarrow^{0|1} \mathcal{E}_{\perp}(v)|_{\mathcal{E}_{\perp}(\mu')}.$$

(β) Assume now that $t|_{\mu} \rightarrow_{g\lambda} t''|_{\mu''}$ and $t''|_{\mu''} \rightarrow^n v|_{\mu'}$ for some arbitrary $n \in \mathbb{N}$. By the progress of Ghost-ML typing, $\vdash_{g\lambda} t'' : (\tau, \perp, \Sigma'')$ (for some Σ with $\Sigma \Rightarrow \Sigma''$ and some store μ''). By induction hypothesis on t'' , and by *one step* forward lemma applied to t , we have that

$$\mathcal{E}_{\perp}(t)|_{\mathcal{E}_{\perp}(\mu)} \rightarrow^{0|1} \mathcal{E}_{\perp}(t'')|_{\mathcal{E}_{\perp}(\mu'')} \rightarrow^n \mathcal{E}_{\perp}(v)|_{\mathcal{E}_{\perp}(\mu')}.$$

That is, for any $n \in \mathbb{N}$, $\mathcal{E}_{\perp}(t)|_{\mathcal{E}_{\perp}(\mu)} \rightarrow_{g\lambda}^* \mathcal{E}_{\perp}(v)|_{\mathcal{E}_{\perp}(\mu')}$.

□

1.4.2 Evaluation Preservation

Lemma 1.3 [SUBSTITUTION UNDER ERASURE].

If $\vdash_{g\lambda} t_1 : (\tau_1, \perp, \Sigma_1)$ and $\vdash_{g\lambda} v_2 : (\tau_2, \mathfrak{B}_2, \Sigma_2)$ hold, then $\mathcal{E}_{\perp}(t_1[x_{\tau_2}^{\mathfrak{B}_2} \leftarrow v_2]) = \mathcal{E}_{\perp}(t_1)[x_{\mathcal{E}_{\perp}(\tau_2)} \leftarrow \mathcal{E}_{\perp}(v_2)]$

Proof. By induction on the structure of t_1 .

□

Lemma 1.4 [ONE-STEP EVALUATION UNDER ERASURE]. *For any closed $g\lambda$ -term t such that $\vdash_{g\lambda} t : (\tau, \perp, \Sigma_1)$ holds, if $t|_{\mu} \rightarrow_{g\lambda} t'|_{\mu'}$ for some term t' , then either $\mathcal{E}_{\perp}(t) \rightarrow_{\lambda} \mathcal{E}_{\perp}(t')$ or $\mathcal{E}_{\perp}(t) = \mathcal{E}_{\perp}(t')$.*

Proof. By induction on the evaluation relation of $t \rightarrow_{g\lambda} t'$.

Case E-APPABS: $t = (\lambda x_{\tau_2}^{\mathfrak{B}_2}. t_1) v_1$ with $(\lambda x_{\tau_2}^{\mathfrak{B}_2}. t_1) v_1 \xrightarrow{\epsilon}_{g\lambda} t_1[x_{\tau_2}^{\mathfrak{B}_2} \leftarrow v_1]$
 $\vdash_{g\lambda} (\lambda x_{\tau_2}^{\mathfrak{B}_2}. t_1) v_1 : (\tau_1, \mathfrak{B}_1), \quad \vdash_{g\lambda} v_1 : (\tau_2, \mathfrak{B}'_2),$
 $\mathfrak{B}_1 = \perp, \quad \models \mathfrak{B}_2 \Leftrightarrow \mathfrak{B}'_2$

$$\begin{aligned} & \mathcal{E}_{\perp}[(\lambda x_{\tau_2}^{\mathfrak{B}_2}. t_1) v_1] \\ &= \lambda x_{\mathcal{E}_{\perp}(\tau_2)}. \mathcal{E}_{\perp}(t_1) \mathcal{E}_{\perp}(v_1) \quad (\text{as } \mathfrak{B}_1 = \perp) \\ & \xrightarrow{\epsilon}_{\lambda} \mathcal{E}_{\perp}(t_1)[x_{\mathcal{E}_{\perp}(\tau_2)} \leftarrow \mathcal{E}_{\perp}(v_1)] \quad (\text{head red.}) \\ &= \mathcal{E}_{\perp}(t_1[x_{\tau_2}^{\mathfrak{B}_2} \leftarrow v_1]) \quad (\text{by Substitution under erasure lemma}) \end{aligned}$$

Case E-DEGHOST:

Trivially verified, as for any instance of $\vdash_{g\lambda} (\text{ghost } t_1) : (\tau_1, \mathfrak{B}_1)$, $\mathfrak{B}_1 = \top$.

Case E-APPLEFT: $t = t_1 t_2$, $t' = t'_1 t_2$, with $t_1 \rightarrow_{g\lambda} t'_1$
 $\vdash_{g\lambda} t_1 : (\tau_2^{\mathfrak{B}_2} \rightarrow \tau_1, \mathfrak{B}_1)$, $\vdash_{g\lambda} t_2 : (\tau_2, \mathfrak{B}'_2)$,
 $\mathfrak{B}_1 = \perp$, $\models \mathfrak{B}_2 \Leftrightarrow \mathfrak{B}'_2$

As $\mathfrak{B}_1 = \perp$, we can apply induction hypothesis on t_1 which gives $\mathcal{E}_\perp(t_1) \rightarrow_\lambda \mathcal{E}_\perp(t'_1)$. Then, applying E-APPRIGHT rule, we obtain:

$$\mathcal{E}_\perp(t) = \mathcal{E}_\perp(t_1)\mathcal{E}_\perp(t_2) \rightarrow_\lambda \mathcal{E}_\perp(t'_1)\mathcal{E}_\perp(t_2) = \mathcal{E}_\perp(t').$$

Case E-APPRIGHT: $t = v_1 t_2$, $t' = v_1 t'_2$, with $t_2 \rightarrow_{g\lambda} t'_2$
 $\vdash_{g\lambda} v_1 : (\tau_2^{\mathfrak{B}_2} \rightarrow \tau_1, \mathfrak{B}_1)$, $\vdash_{g\lambda} t_2 : (\tau_2, \mathfrak{B}'_2)$,
 $\mathfrak{B}_1 = \perp$, $\models \mathfrak{B}_2 \Leftrightarrow \mathfrak{B}'_2$

If $\mathfrak{B}_2 = \mathfrak{B}'_2 = \top$, then

$$\mathcal{E}_\perp(t) = \mathcal{E}_\perp(v_1)\mathcal{E}_\top(t_2) = (\mathcal{E}_\perp(v_1))() = \mathcal{E}_\perp(v_1)\mathcal{E}_\top(t'_2) = \mathcal{E}_\perp(t').$$

Otherwise, $\mathfrak{B}_2 = \mathfrak{B}'_2 = \perp$. By induction hypothesis, $\mathcal{E}_\perp(t_2) \rightarrow_\lambda \mathcal{E}_\perp(t'_2)$. Then, applying E-APPRIGHT rule, we obtain:

$$\mathcal{E}_\perp(t) = \mathcal{E}_\perp(v_1)\mathcal{E}_\perp(t_2) \rightarrow_\lambda \mathcal{E}_\perp(v_1)\mathcal{E}_\perp(t'_2) = \mathcal{E}_\perp(t').$$

□

Now we can prove the main theorem.

Theorem 1.8 [VALUE PRESERVATION UNDER ERASURE]. For any closed $g\lambda$ -term t such that $\vdash_{g\lambda} t : (\tau, \perp)$ holds, if $t \rightarrow_{g\lambda}^* v$ for some value v , then $\mathcal{E}(t) \rightarrow_\lambda^* \mathcal{E}(v)$.

Proof. By induction on the length of the evaluation of $t \rightarrow_{g\lambda}^* v$.

We already have proved the base case : indeed, if $t \rightarrow_{g\lambda} v$ then by the one-step evaluation lemma, $\mathcal{E}_\perp(t) \rightarrow_\lambda^{0|1} \mathcal{E}_\perp(v)$.

Now, assume that $t \rightarrow_{g\lambda}^1 t' \rightarrow_{g\lambda}^n v$ for some arbitrary $n \in \mathbb{N}$. By the progress of typing, $\vdash_{g\lambda} t' : (\tau, \perp)$, so we can apply induction hypothesis on t' which gives $\mathcal{E}(t') \rightarrow_\lambda^* \mathcal{E}(v)$. By the one-step evaluation lemma again, we have $\mathcal{E}_\perp(t) \rightarrow_\lambda^{0|1} \mathcal{E}_\perp(t')$. That is, $\mathcal{E}_\perp(t) \rightarrow_\lambda^* \mathcal{E}_\perp(v)$. □

1.4.3 Typing Erasure

Lemma 1.5 [TYPING RELATION UNDER ERASURE].

If $\vdash_{g\lambda} t : (\tau, \perp)$ then $\vdash_{\lambda} \mathcal{E}_{\perp}(t) : \mathcal{E}_{\perp}(\tau)$.

Proof. By induction on a derivation of the statement $\vdash_{g\lambda} \mathcal{E}_{\perp}(t) : \mathcal{E}_{\perp}(\tau)$. For a given derivation, we proceed by case analysis on the final typing rule used in the proof.

Case T-UNIT: $\vdash_{g\lambda} () : (\text{unit}, \perp)$

Immediately by definition of \mathcal{E}_{\perp} .

Case T-VAR: $\vdash_{g\lambda} x_{\tau}^{\perp} : (\tau, \perp)$

$\mathcal{E}_{\perp}(x_{\tau}^{\perp}) = x_{\mathcal{E}_{\perp}(\tau)}$ gives immediately $\vdash_{\lambda} x_{\mathcal{E}_{\perp}(\tau)} : \mathcal{E}(\tau)$.

Case T-ABS: $\vdash_{g\lambda} \lambda x_{\tau_2}^{\mathfrak{B}_2}. t_1 : (\tau_2^2 \rightarrow \tau_1, \perp)$ with $\vdash_{g\lambda} t_1 : (\tau_1, \perp)$

By induction hypothesis $\vdash_{\lambda} \mathcal{E}_{\perp}(t_1) : \mathcal{E}_{\perp}(\tau_1)$. There are two cases to consider, depending on whether the parameter of the abstraction is ghost or not. If $\mathfrak{B}_2 = \top$ then $\mathcal{E}_{\perp}(\lambda x_{\tau_2}^{\top}. t_1) = \lambda x_{\text{unit}}. \mathcal{E}(t_1)$ and therefore

$$\frac{\vdash_{\lambda} \mathcal{E}_{\perp}(t_1) : \mathcal{E}_{\perp}(\tau_1)}{\vdash_{\lambda} \lambda x_{\text{unit}}. \mathcal{E}_{\perp}(t_1) : \text{unit} \rightarrow \mathcal{E}_{\perp}(\tau_1)} \quad (\text{T-ABS})$$

Otherwise $\mathfrak{B}_2 = \perp$ and again by the rule T-ABS we obtain :

$$\frac{\vdash_{\lambda} \mathcal{E}_{\perp}(t_1) : \mathcal{E}_{\perp}(\tau_1)}{\vdash_{\lambda} \lambda x_{\mathcal{E}_{\perp}(\tau_2)}. \mathcal{E}_{\perp}(t_1) : \mathcal{E}_{\perp}(\tau_2) \rightarrow \mathcal{E}_{\perp}(\tau_1)} \quad (\text{T-ABS})$$

Case T-APP: $\vdash_{g\lambda} t_1 t_2 : (\tau_1, \perp)$ with sub-derivations:

$$\vdash_{g\lambda} t_1 : (\tau_2^{\mathfrak{B}_2} \rightarrow \tau_1, \mathfrak{B}_1)$$

$$\vdash_{g\lambda} t_2 : (\tau_2, \mathfrak{B}'_2),$$

As $\vdash_{g\lambda} t_1 t_2 : (\tau_1, \perp)$, the inversion lemma gives as By inversion that $\models \mathfrak{B}_2 \Leftrightarrow \mathfrak{B}'_2$. That is, we have two cases to consider.

If $\mathfrak{B}_2 = \mathfrak{B}'_2 = \perp$ then by induction hypotheses

$\vdash_{\lambda} \mathcal{E}_{\perp}(t_1) : \mathcal{E}_{\perp}(\tau_2) \rightarrow \mathcal{E}_{\perp}(\tau_1)$ and $\vdash_{\lambda} \mathcal{E}_{\perp}(t_2) : \mathcal{E}_{\perp}(\tau_2)$. By T-APP rule,

$$\frac{\vdash_{\lambda} \mathcal{E}_{\perp}(t_1) : \mathcal{E}_{\perp}(\tau_2) \rightarrow \mathcal{E}_{\perp}(\tau_1) \quad \vdash_{\lambda} \mathcal{E}_{\perp}(t_2) : \mathcal{E}_{\perp}(\tau_2)}{\vdash_{\lambda} \mathcal{E}_{\perp}(t_1 t_2) : \mathcal{E}(\tau_1)} \quad (\text{T-APP})$$

If $\mathfrak{B}_2 = \mathfrak{B}'_2 = \top$, then by definition of \mathcal{E} we have $\mathcal{E}(\tau_2) = ()$ and $\mathcal{E}_{\mathfrak{B}'_2}(\tau_2) = \text{unit}$. By induction hypothesis on t_1 , $\vdash_{\lambda} \mathcal{E}_{\perp}(t_1) : \text{unit} \rightarrow \mathcal{E}_{\perp}(\tau_1)$. Applying T-APP rule gives us

$$\frac{\frac{}{\vdash_{\lambda} \mathcal{E}_{\perp}(\mathfrak{t}_1 \ \mathfrak{t}_2) : \mathcal{E}(\tau_1)} \quad \frac{}{\vdash_{\lambda} () : \mathbf{unit}} \text{ (T-UNIT)}}{\vdash_{\lambda} \mathcal{E}_{\perp}(\mathfrak{t}_1 \ \mathfrak{t}_2) : \mathcal{E}_{\perp}(\tau_1)} \text{ (T-APP)}$$

The case of (T-GHOST) as well as any other valid derivation where a typed term is marked as ghost do not satisfy lemma's requirement, so these cases are trivially verified. \square

2 logic

In previous section we described how to define formally ghost terms inside ML-like programs.

However, the ghost code becomes useful only within a full-featured specification language.

In this section we define a high-order logic with simple types such as units, integers, boolean. To make our examples more interesting we also provide built-in integer lists and trees.

First off, we describe our logic's syntax, semantics and typing. Then we extend the ghost-ML language with specifications such as assertions and functional pre- and post-conditions.

$\tau ::=$ $\text{unit} \mid \text{int} \mid \text{bool}$ $\text{list int} \mid \text{tree int}$ $\tau \rightarrow \tau$ prop	TYPES <i>built-in simple types</i> <i>built-in recursive types</i> <i>function type</i> <i>proposition</i>	$c ::=$ $0 \mid 1 \mid \dots \mid n$ $\text{true} \mid \text{false}$ $()$ $\mu\text{l. Nil} \mid \text{Cons } n \text{ l}$ $\mu\text{t. Empty} \mid \text{Node } t \text{ n } t$ $\text{True} \mid \text{False}$ $+, -, =, \dots$	CONSTANTS <i>integer</i> <i>boolean</i> <i>unit</i> <i>integer list</i> <i>integer binary tree</i> <i>proposition value</i> <i>built-in operators</i>
$t ::=$ c x_τ $\lambda x_\tau. t$ $t \ t$ $\text{let } x_\tau = t \text{ in } t$ $\text{rec } g \ x_\tau : \tau = t$ f	TERMS <i>constant</i> <i>variable</i> <i>abstraction</i> <i>application</i> <i>local binding</i> <i>recursive function</i> <i>formula</i>	$f ::=$ $\text{True} \mid \text{False}$ $t \wedge t$ $t \vee t$ $\neg t$ $\exists x_\tau^{\mathfrak{B}} x_\tau. f$ $\forall x_\tau^{\mathfrak{B}} x_\tau. f$ $f = f$	FORMULAS <i>logical truth values</i> <i>conjunction</i> <i>disjunction</i> <i>negation</i> <i>existential quantification</i> <i>universal quantification</i> <i>equality</i>

Figure 9: Logic Syntax

3 Inlining

Motivation: instantiate higher-order iterators with previously defined or anonymous functions, in order to obtain a first-order function, whose proof obligation is of the same complexity as p.o. of equivalent loop statement.

Goal: reduce every application of a higher-order function to a term that is not a bound variable.

Input: a higher-order language in *A-normal* form with functions whose codomain is of some base type, and whose formal parameters is partially ordered (the higher order parameters come before lesser order parameters)

Output: a language where the only high-order applications are those where argument of application is a bound variable. That is, a language where higher-order applications can occur only under **inside** high-order functions.

Input Language Syntax:

$$t ::= v \mid vv \mid let$$

Figure 10: Source language in A-normal form
