1 GhostLambda: Lambda-Calculus with Ghost Code

The goal of this section is to present the idea of *ghost* code, its use in program specification and to state some properties about its extraction.

Ghost code is a kind of program annotation that serves as a support for program specification inside logical formulae and assertions.

The main idea about ghost code is that as any other kind of specification it does not affect the program meaning. That is, ghost code does not correspond to any physical entity, and as such can by entirely erased from a program, without altering its formal meaning.

In this section we describe a language where one can annotate programs with such *ghost code*.

We start by formalizing the *ghostλ-calculus*, a tiny language of simply typed λ-calculus enriched with ghost variables and ghost expressions.

We then define ghost code *erasure*, which transforms a well-typed gλ term to a term of standard λ-calculus. Finally we state and proof a few basic preservation properties of such translation.

1.1 gλ-calculus syntax and semantics

The syntax of *ghost-λ* is summarized below.

Syntax

$\mathtt{t}$	$::=$		TERMS:
		$()$	*unit*
		$\mathtt{x}^{\mathfrak{B}}_{\tau}$	*variable*
		$\lambda \mathtt{x}^{\mathfrak{B}}_{\tau}.\mathtt{t}$	*abstraction*
		$\mathtt{tt}$	*application*
		$\mathtt{ghost\ t}$	*ghost term*
$\mathtt{v}$	$::=$		VALUES:
		$()$	*unit*
		$\lambda \mathtt{x}^{\mathfrak{B}}_{\tau}.\mathtt{t}$	*abstraction*
$\mathfrak{B}$	$::=$		GHOST STATUS :
		$\top \ (*true*)$	*ghost code*
		$\bot \ (*false*)$	*physical code*
τ	$::=$		TYPES:
		$\mathtt{unit}$	*unit type*
		$\tau^{\mathfrak{B}} \to \tau$	*function type*

As we can see, gλ-calculus is just an extension of standard simply typed λ-calculus with `arrow` and `unit` enriched with a construction `ghost t` for ghost terms. Moreover, variables and formal arguments in abstraction are annotated with their type and *ghost* mark $\mathfrak{B}$ which indicates

whether it is ghost ($\top$) or not ($\bot$).

The evaluation of gλ-calculus corresponds as well to the standard SOS call-by-value semantics of λ-calculus, except for the rule (E-DeGhost).

Evaluation

$$(\lambda x^{b}_{\tau}.\mathtt{t})v \xrightarrow{\epsilon} \mathtt{t}[x^{b}_{\tau} \leftarrow\!\shortmid v] \qquad \text{(E-AppFun)}$$

$$\mathtt{ghost\ t} \xrightarrow{\epsilon} \mathtt{t} \qquad \text{(E-DeGhost)}$$

$$\frac{\mathtt{t}_1 \rightarrow \mathtt{t}'_1}{\mathtt{t}_1\mathtt{t}_2 \rightarrow \mathtt{t}'_1\mathtt{t}_2} \qquad \text{(T-AppLeft)}$$

$$\frac{\mathtt{t}_2 \rightarrow \mathtt{t}'_2}{\mathtt{v}_1\mathtt{t}_2 \rightarrow \mathtt{v}_1\mathtt{t}'_2} \qquad \text{(T-AppLeft)}$$

Figure 1: ***ghost-*λ syntax and semantics**

Free variables, scope and equivalence of terms

We do not present here the formal definitions of free variables, abstraction scope, α-equivalence and substitution: they are exactly the same that those of λ-calculus.

However, it is important to notice that two variables $x^{\mathfrak{B}_1}_{\tau_1}$ and $y^{\mathfrak{B}_2}_{\tau_2}$ are syntactically equal only if $x = y$, $\mathfrak{B}_1 = \mathfrak{B}_2$ and $\tau_1 = \tau_2$. That is $\lambda x^{\top}_{\tau_1}.x^{\top}_{\tau_1}$ corresponds to τ_1- identity, whilst $\lambda x^{\top}_{\tau_1}.x^{\bot}_{\tau_1}$ corresponds to the constant function that ignores its argument.

1.2 Typing Relation

In the simply typed λ-calculus, typing rules out statically some erroneous programs. For instance, it allows to check that when function is applied to some argument, the type of the formal parameter corresponds to that one of actual parameter.

The typing of the programs containing some ghost code should of course assure the same safety with respect of program evaluation. However, in the presence of ghost code, this should also include the *non-interference* of ghost code with the rest of the program. Intuitively, the *non-interference* means that computation of ghost code must be completely disjoint from the computation of the non-ghost code.

To assure that, we define the typing relation as three-argument pred-

icate $\boxed{\vdash_{g\lambda} t : (\tau, \mathfrak{B})}$ which both carries statically the information about type of term t, and indicates whether this term is ghost or not. Formally, we define the typing relation inductively over the structure of terms as follows:

Typing

$$\frac{}{\vdash_{g\lambda} () : (unit, \bot)} \quad \text{(T-Unit)}$$

$$\frac{}{\vdash_{g\lambda} x^{\mathfrak{B}}_{\tau} : (\tau, \mathfrak{B})} \quad \text{(T-Var)}$$

$$\frac{\vdash_{g\lambda} \mathtt{t} : (\tau, \mathfrak{B})}{\vdash_{g\lambda} \mathtt{ghost\ t} : (\tau, \top)} \quad \text{(T-Ghost)}$$

$$\frac{\vdash_{g\lambda} \mathtt{t} : (\tau_2, \mathfrak{B}_2)}{\vdash_{g\lambda} \lambda x^{\mathfrak{B}_1}_{\tau_1}.\mathtt{t} : (\tau_1^{\mathfrak{B}_1} \to \tau_2, \mathfrak{B}_2)} \quad \text{(T-Abs)}$$

$$\frac{\vdash_{g\lambda} \mathtt{t}_1 : (\tau_2^{\mathfrak{B}_2} \to \tau_1, \mathfrak{B}_1) \qquad \vdash_{g\lambda} \mathtt{t}_2 : (\tau_2, \mathfrak{B}'_2) \qquad \mathfrak{B}_2 \Rightarrow \mathfrak{B}'_2}{\vdash_{g\lambda} \mathtt{t}_1\ \mathtt{t}_2 : (\tau_1, \mathfrak{B}_1 \vee (\neg\mathfrak{B}_2 \wedge \mathfrak{B}'_2))} \quad \text{(T-App)}$$

The rules (T-Unit), (T-Var) and (T-Abs) are straightforward.

The rule (T-Ghost) states simply that a term of `ghost t` is a ghost code, whether `t` is ghost or not. That is, if `t` were a well-typed non-ghost term, once we put inside a ghost code, it becomes ghost itself.

The more complex rule (T-App) states that formal argument of $\mathtt{t}_1$ and actual parameter $\mathtt{t}_2$ must be of the same type. Moreover, it forbids the application of normal code to a function whose formal argument is ghost.

However, a function whose formal argument is not ghost is applied to a ghost code, then the application itself becomes ghost, whether function's body is ghost or not. We summarized that is the following table:

$\mathfrak{B}_1$	$\mathfrak{B}_2$	$\mathfrak{B}'_2$	result	
$\bot$	$\bot$	$\bot$	$\bot$	normal code application
$\top$	$\bot$	$\top$	$\bot$	ghost-code passing inside normal code
$\top$	$\top$	$\top$	$\top$	ghost code application
$\bot$	$\top$	$\bot$	$\top$	normal code passing inside ghost code
$\bot$	$\top$	$\top$	$\top$	formal parameter contamination
$\bot$	$\bot$	$\top$	$\top$	function parameter and body contamination
$\top$	$\bot$	$\bot$	$-$	impossible
$\top$	$\top$	$\bot$	$-$	impossible

1.2.1 Properties of typing

Lemma 1.1 [Inversion of typing relation].

1. *if* $\vdash_{g\lambda}$ `ghost` $t : (\tau_1, \mathfrak{B}_1)$ *then* $\mathfrak{B}_1 = \top$ *and* $\vdash_{g\lambda} t : (\tau_1, \mathfrak{B}_2)$.
2. *if* $\vdash_{g\lambda} \lambda x^{\mathfrak{B}_2}_{\tau_2}.t : (\tau_1, \mathfrak{B}_1)$ *then* $\tau_1 = \tau_2^{\mathfrak{B}_2} \to \tau_{11}$ *for some* τ_{11} *with* $\vdash_{g\lambda} t : (\tau_{11}, \mathfrak{B}_1)$.
3. *If* $\vdash_{g\lambda} t_1\ t_2 : (\tau_1, \mathfrak{B}_1)$ *then there exist* τ_{11}, τ_2, $\mathfrak{B}_2$ *and* $\mathfrak{B}'_2$ *such that* $\vdash_{g\lambda} t_1 : (\tau_2^{\mathfrak{B}_2} \to \tau_{11}, \mathfrak{B}_1)$ *and* $\vdash_{g\lambda} t_2 : (\tau_2, \mathfrak{B}'_2)$ *with* $\vDash \mathfrak{B}_1 \vee (\neg\mathfrak{B}_2 \wedge \mathfrak{B}'_2) \wedge (\mathfrak{B}_2 \Rightarrow \mathfrak{B}'_2)$.

   In particular, if $\vdash_{g\lambda} t_1\ t_2 : (\tau_1, \bot)$ *then* $\vDash \mathfrak{B}_2 \Leftrightarrow \mathfrak{B}'_2$.

Proof. Straightforward from definition of the typing relation. □

Lemma 1.2 [Progress]. *If* t *is a closed, well-typed term (that is* $\vdash_{g\lambda} t : (\tau, \mathfrak{B})$ *for some* τ *and* $\mathfrak{B}$*, then either* t *is a value or else there is some* t' *such that* $t \to_{g\lambda} t'$.

Proof. Admit. □

Note that the typing preservation works only for well-typed non-ghost terms. Indeed,as the following example shows, the contamination of non-ghost code makes the preservation fail for ghost code: if we evaluate ghost code $\texttt{ghost}\ (\lambda x^{\bot}_{\tau_{unit}}.x^{\bot}_{\tau_{unit}}) \to_{g\lambda} \lambda x^{\bot}_{\tau_{unit}}.x^{\bot}_{\tau_{unit}}$, then $\vdash_{g\lambda} \texttt{ghost}(\lambda x^{\bot}_{\tau_{unit}}.x^{\bot}_{\tau_{unit}}) : (unit \to unit, \top)$ holds, while the typing relation $\vdash_{g\lambda} x^{\bot}_{\tau_{unit}} : (unit, \top)$ fails.

Lemma 1.3 [Non-ghost Code Typing Preservation]. *If* $\vdash_{g\lambda} t : (\tau, \bot)$ *and* $t \to_{g\lambda} t'$*, then* $\vdash_{g\lambda} t' : (\tau, \bot)$

Proof. Admit. □

Theorem 1.1 [Soundness].

Proof. Admit. □

1.3 Ghost Code Erasure

Once we formally defined the simply typed lambda-calculus enriched with ghost expressions, our goal is to show that the computational meaning of non ghost well typed non-ghost $g\lambda$-terms is preserved by their translation to simply typed lambda-calculus.

To achieve this, we start by defining the *type-* and *term- erasure* functions from gλ-calculus to simply typed λ-calculus.

1.3.1 Ghost Erasure

Definition 1.2 (Type-Erasure). We define type-erasure by induction on the structure of types :

$\mathcal{E}_\top(\tau) = \texttt{unit}$
$\mathcal{E}_\bot(\texttt{unit}) = \texttt{unit}$
$\mathcal{E}_\bot(\tau_2^{\mathfrak{B}_2} \to \tau_1) = \mathcal{E}_{\mathfrak{B}_2}(\tau_2) \to \mathcal{E}_\bot(\tau_1)$.

Definition 1.3 (Term-Erasure). Let $\texttt{t}$ be a term such that $\vdash_{g\lambda} \texttt{t} : (\tau, \mathfrak{B})$ holds. We define term-erasure function by induction on the structure of $\texttt{t}$, parametrizing it by the value of $\mathfrak{B}$.
$\mathcal{E}_\top(\texttt{t}_1) = \texttt{()}$ where $\vdash_{g\lambda} \texttt{t}_1 : (\tau_1, \top)$.
$\mathcal{E}_\bot(\texttt{()}) = \texttt{()}$
$\mathcal{E}_\bot(x_\tau^\bot) = x_{\mathcal{E}_\bot(\tau)}$
$\mathcal{E}_\bot(\lambda x_{\tau_2}^{\mathfrak{B}_2}.t_1) = \lambda x_{\mathcal{E}_{\mathfrak{B}_2}(\tau_2)}.\mathcal{E}_\bot(t_1)$ where $\vdash_{g\lambda} \texttt{t}_1 : (\tau_1, \bot)$.
$\mathcal{E}_\bot(\texttt{t}_1\ \texttt{t}_2) = \mathcal{E}_\bot(\texttt{t}_1)\ \mathcal{E}_{\mathfrak{B}_2}(\texttt{t}_2)$ where $\vdash_{g\lambda} \texttt{t}_1 : (\tau_1, \bot)$ and $\vdash_{g\lambda} \texttt{t}_2 : (\tau_2, \mathfrak{B}_2)$.

As it can be seen, the erasure function is a morphism that preserve the structure of operational (not ghost) terms and their types ($\sim \mathcal{E}_\bot(\star)$), and sends ghost expressions and types to $\texttt{()}$ and $\texttt{unit}$ respectively ($\sim \mathcal{E}_\top(\star)$).

Here is two examples of term erasure:
$\mathcal{E}_\top[(\lambda z_{\tau_{\tau_1}}^\bot.z_{\tau_{\tau_1}}^\bot)(\texttt{ghost ()})] = ()$
$\mathcal{E}\bot[(\lambda x_{\tau_{\tau_1}}^\top.\lambda y_{\tau_{\tau_1}}^\bot.y_{\tau_{\tau_1}}^\bot)(\texttt{ghost ()})] = (\lambda x_{\texttt{unit}}.\lambda y_{\tau_1}.y_{\tau_1})()$

1.4 Properties of ghost erasure

Now that we defined the erasure-translation of $g\lambda$-calculus to λ-calculus, our concern is to show that evaluation result of well-typed operational terms as well as their typing are preserved under erasure. First off we need to state and prove a few basic lemmas.

1.4.1 Evaluation Preservation

Lemma 1.4 [Substitution under erasure].
If $\vdash_{g\lambda} t_1 : (\tau_1, \bot)$ *and* $\vdash_{g\lambda} v_2 : (\tau_2, \mathfrak{B}_2)$ *hold,*
then $\mathcal{E}_\bot(t_1[x_{\tau_2}^{\mathfrak{B}_2} \hookleftarrow v_2]) = \mathcal{E}_\bot(t_1)[x_{\mathcal{E}_{\mathfrak{B}_2}(\tau_2)} \hookleftarrow \mathcal{E}_{\mathfrak{B}_2}(v_2)]$

Proof. By induction on the structure of $\texttt{t}_1$.

Case $\texttt{t}_1 = x_{\tau_2}^{\mathfrak{B}_2}$:
In that case, we can deduce that $\mathfrak{B}_2 = \bot$. Therefore:

$$\begin{aligned}\mathcal{E}_\bot(x_{\tau_2}^\bot[x_{\tau_2}^\bot \hookleftarrow \texttt{v}_2]) = \mathcal{E}_\bot(\texttt{v}_2) &= x_{\mathcal{E}_\bot(\tau_2)}[x_{\mathcal{E}_\bot(\tau_2)} \hookleftarrow \mathcal{E}_\bot(\texttt{v}_2)] \\ &= \mathcal{E}_\bot(x_{\tau_2}^\bot)[x_{\mathcal{E}_\bot(\tau_2)} \hookleftarrow \mathcal{E}_\bot(\texttt{v}_2)]\end{aligned}$$

Case $\mathtt{t}_1 = y^{\perp}_{\tau_2'} \neq x^{\mathfrak{B}_2}_{\tau_2}$:

$$\mathcal{E}_\perp(y^{\perp}_{\tau_2'}[x^{\mathfrak{B}_2}_{\tau_2} \leftarrowtail \mathtt{v}_2]) = \mathcal{E}_\perp(y^{\perp}_{\tau_2'}) = y_{\mathcal{E}_\perp(\tau_2')}$$
$$= y_{\mathcal{E}_\perp(\tau_2')}[x_{\mathcal{E}_{\mathfrak{B}_2}(\tau_2)} \leftarrowtail \mathcal{E}_{\mathfrak{B}_2}(\mathtt{v}_2)] = \mathcal{E}_\perp(y^{\perp}_{\tau_2'})[x_{\mathcal{E}_{\mathfrak{B}_2}(\tau_2)} \leftarrowtail \mathcal{E}_{\mathfrak{B}_2}(\mathtt{v}_2)]$$

Case $\mathtt{t}_1 = \lambda y^{\mathfrak{B}_2'}_{\tau_2'}.t_{11}$ with $y^{\mathfrak{B}_2'}_{\tau_2'} \notin \mathtt{FV}(\mathtt{v}_2)$ and $\neq x^{\mathfrak{B}_2}_{\tau_2}$:

$$\mathcal{E}_\perp((\lambda y^{\mathfrak{B}_2'}_{\tau_2'}.t_{11})[x^{\mathfrak{B}_2}_{\tau_2} \leftarrowtail \mathtt{v}_2]) = \mathcal{E}_\perp[\lambda y^{\mathfrak{B}_2'}_{\tau_2'}.(t_{11}[x^{\mathfrak{B}_2}_{\tau_2} \leftarrowtail \mathtt{v}_2])]$$
$$= \lambda y_{\mathcal{E}_{\mathfrak{B}_2'}(\tau_2')}.\mathcal{E}_\perp(\mathtt{t}_{11}[x^{\mathfrak{B}_2}_{\tau_2} \leftarrowtail \mathtt{v}_2])]$$
$$\overset{\textit{Ind.Hyp.}}{=} \lambda y_{\mathcal{E}_{\mathfrak{B}_2'}(\tau_2')}.\mathcal{E}_\perp(\mathtt{t}_{11})[x_{\mathcal{E}_{\mathfrak{B}_2}(\tau_2)} \leftarrowtail \mathcal{E}_{\mathfrak{B}_2}(\mathtt{v}_2)]$$
$$= (\lambda y_{\mathcal{E}_{\mathfrak{B}_2'}(\tau_2')}.\mathcal{E}_\perp(\mathtt{t}_{11}))[x_{\mathcal{E}_{\mathfrak{B}_2}(\tau_2)} \leftarrowtail \mathcal{E}_{\mathfrak{B}_2}(\mathtt{v}_2)]$$
$$= \mathcal{E}_\perp(\lambda y^{\mathfrak{B}_2'}_{\tau_2'}.t_{11})[x_{\mathcal{E}_{\mathfrak{B}_2}(\tau_2)} \leftarrowtail \mathcal{E}_{\mathfrak{B}_2}(\mathtt{v}_2)]$$

Case $\mathtt{t}_1 = \mathtt{t}_{11}\mathtt{t}_{12}$

$$\mathcal{E}_\perp(\mathtt{t}_{11}\mathtt{t}_{12}[x^{\mathfrak{B}_2}_{\tau_2} \leftarrowtail \mathtt{v}_2]) = \mathcal{E}_\perp(\mathtt{t}_{11}[x^{\mathfrak{B}_2}_{\tau_2} \leftarrowtail \mathtt{v}_2]\ \mathtt{t}_{12}[x^{\mathfrak{B}_2}_{\tau_2} \leftarrowtail \mathtt{v}_2])$$
$$= \mathcal{E}_\perp(\mathtt{t}_{11}[x^{\mathfrak{B}_2}_{\tau_2} \leftarrowtail \mathtt{v}_2])\ \mathcal{E}_\perp(\mathtt{t}_{12}[x^{\mathfrak{B}_2}_{\tau_2} \leftarrowtail \mathtt{v}_2])$$
$$\overset{\textit{Ind.Hyp.}}{=} (\mathcal{E}_\perp(\mathtt{t}_{11})[x_{\mathcal{E}_{\mathfrak{B}_2}(\tau_2)} \leftarrowtail \mathcal{E}_{\mathfrak{B}_2}(\mathtt{v}_2)])(\mathcal{E}_\perp(\mathtt{t}_{12})[x_{\mathcal{E}_{\mathfrak{B}_2}(\tau_2)} \leftarrowtail \mathcal{E}_{\mathfrak{B}_2}(\mathtt{v}_2)])$$
$$= \mathcal{E}_\perp(\mathtt{t}_{11}\mathtt{t}_{12})[x_{\mathcal{E}_{\mathfrak{B}_2}(\tau_2)} \leftarrowtail \mathcal{E}_{\mathfrak{B}_2}(\mathtt{v}_2)])$$

□

Lemma 1.5 [One-step evaluation under erasure]. *For any closed $g\lambda$-term t such that $\vdash_{g\lambda}\ t : (\tau, \perp)$ holds, if $t \to_{g\lambda}\ t'$ for some term t', then either $\mathcal{E}_\perp(t) \to_\lambda \mathcal{E}_\perp(t')$ or $\mathcal{E}_\perp(t) = \mathcal{E}_\perp(t')$.*

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

Case E-AppAbs: $\mathtt{t} = (\lambda x^{\mathfrak{B}_2}_{\tau_2}.\mathtt{t}_1)\mathtt{v}_1$ with $(\lambda x^{\mathfrak{B}_2}_{\tau_2}.\mathtt{t}_1)\mathtt{v}_1 \xrightarrow{\epsilon}_{g\lambda} \mathtt{t}_1[x^{\mathfrak{B}_2}_{\tau_2} \leftarrowtail \mathtt{v}_1]$
$\vdash_{g\lambda} (\lambda x^{\mathfrak{B}_2}_{\tau_2}.\mathtt{t}_1)\mathtt{v}_1 : (\tau_1, \mathfrak{B}_1)$, $\vdash_{g\lambda} \mathtt{v}_1 : (\tau_2, \mathfrak{B}_2')$,
$\mathfrak{B}_1 = \perp$, $\vDash \mathfrak{B}_2 \Leftrightarrow \mathfrak{B}_2'$

$\mathcal{E}_\perp[(\lambda x^{\mathfrak{B}_2}_{\tau_2}.\mathtt{t}_1)\mathtt{v}_1]$	
$= \lambda x_{\mathcal{E}_{\mathfrak{B}_2}(\tau_2)}.\mathcal{E}_\perp(\mathtt{t}_1))\mathcal{E}_{\mathfrak{B}_2'}(\mathtt{v}_1)$	(as $\mathfrak{B}_1 = \perp$)
$\xrightarrow{\epsilon}_\lambda \mathcal{E}_\perp(\mathtt{t}_1)[x_{\mathcal{E}_{\mathfrak{B}_2}(\tau_2)} \leftarrowtail \mathcal{E}_{\mathfrak{B}_2'}(\mathtt{v}_1)]$	(head red.)
$= \mathcal{E}_\perp(\mathtt{t}_1[x^{\mathfrak{B}_2}_{\tau_2} \leftarrowtail \mathtt{v}_1])$	(by Substitution under erasure lemma)

Case E-DeGhost:
Trivially verified, as for any instance of $\vdash_{g\lambda} (\mathit{ghost}\ \mathtt{t}_1) : (\tau_1, \mathfrak{B}_1)$, $\mathfrak{B}_1 = \top$.

Case E-AppLeft: $\mathtt{t} = \mathtt{t}_1\mathtt{t}_2$, $\mathtt{t}' = \mathtt{t}_1'\mathtt{t}_2$, with $\mathtt{t}_1 \to_{g\lambda} \mathtt{t}_1'$
$\vdash_{g\lambda} \mathtt{t}_1 : (\tau_2^{\mathfrak{B}_2} \to \tau_1, \mathfrak{B}_1)$, $\vdash_{g\lambda} \mathtt{t}_2 : (\tau_2, \mathfrak{B}_2')$,

$$\mathfrak{B}_1 = \bot, \quad \vDash \mathfrak{B}_2 \Leftrightarrow \mathfrak{B}'_2$$

As $\mathfrak{B}_1 = \bot$, we can apply induction hypothesis on t_1 which gives $\mathcal{E}_\bot(t_1) \to_\lambda \mathcal{E}_\bot(t_1')$. Then, applying E-AppRight rule, we obtain:

$$\mathcal{E}_\bot(t) = \mathcal{E}_\bot(t_1)\mathcal{E}_\bot(t_2) \to_\lambda \mathcal{E}_\bot(t_1')\mathcal{E}_\bot(t_2) = \mathcal{E}_\bot(t').$$

Case E-AppRight: $t = v_1 t_2, \quad t' = v_1 t_2', \text{ with } t_2 \to_{g\lambda} t_2'$
$\vdash_{g\lambda} v_1 : (\tau_2^{\mathfrak{B}_2} \to \tau_1, \mathfrak{B}_1), \quad \vdash_{g\lambda} t_2 : (\tau_2, \mathfrak{B}'_2),$
$\mathfrak{B}_1 = \bot, \quad \vDash \mathfrak{B}_2 \Leftrightarrow \mathfrak{B}'_2$

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

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

Otherwise, $\mathfrak{B}_2 = \mathfrak{B}'_2 = \bot$. By induction hypothesis, $\mathcal{E}_\bot(t_2) \to_\lambda \mathcal{E}_\bot(t_2')$. Then, applying E-AppRight rule, we obtain:

$$\mathcal{E}_\bot(t) = \mathcal{E}_\bot(v_1)\mathcal{E}_\bot(t_2) \to_\lambda \mathcal{E}_\bot(v_1)\mathcal{E}_\bot(t_2') = \mathcal{E}_\bot(t').$$

□

Now we can prove the main theorem.

Theorem 1.4 [Value preservation under erasure]. *For any closed $g\lambda$-term t such that $\vdash_{g\lambda} t : (\tau, \bot)$ holds, if $t \to^*_{g\lambda} v$ for some value v, then $\mathcal{E}(t) \to^*_\lambda \mathcal{E}(v)$.*

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

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

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

1.4.2 Typing Erasure

Lemma 1.6 [Typing relation under erasure].
If $\vdash_{g\lambda} t : (\tau, \bot)$ then $\vdash_\lambda \mathcal{E}_\bot(t) : \mathcal{E}_\bot(\tau)$.

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

Case T-Unit: $\vdash_{g\lambda} () : (\mathtt{unit}, \bot)$

Immediately by definition of $\mathcal{E}_\bot$.

Case T-Var: $\vdash_{g\lambda} x^\bot_\tau : (\tau, \bot)$

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

Case T-Abs: $\vdash_{g\lambda} \lambda x^{\mathfrak{B}_2}_{\tau_2}.\mathtt{t}_1 : (\tau_2^2 \to \tau_1, \bot)$ with $\vdash_{g\lambda} \mathtt{t}_1 : (\tau_1, \bot)$

By induction hypothesis $\vdash_\lambda \mathcal{E}_\bot(\mathtt{t}_1) : \mathcal{E}_\bot(\tau_1)$. There are two cases to consider, depending on whether the parameter of the abstraction is ghost or not. If $\mathfrak{B}_2 = \top$ then $\mathcal{E}_\bot(\lambda x^{\top}_{\tau_2}.\mathtt{t}_1) = \lambda x_{\mathtt{unit}}.\mathcal{E}(\mathtt{t}_1)$ and therefore

$$\frac{\vdash_\lambda \mathcal{E}_\bot(\mathtt{t}_1) : \mathcal{E}_\bot(\tau_2)}{\vdash_\lambda \lambda x_{\mathtt{unit}}.\mathcal{E}_\bot(\mathtt{t}_1) : \mathtt{unit} \to \mathcal{E}_\bot(\tau_1)} \quad \text{(T-Abs)}$$

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

$$\frac{\vdash_\lambda \mathcal{E}_\bot(\mathtt{t}_1) : \mathcal{E}_\bot(\tau_1)}{\vdash_\lambda \lambda x_{\mathcal{E}_\bot(\tau_2)}.\mathcal{E}_\bot(\mathtt{t}_1) : \mathcal{E}_\bot(\tau_2) \to \mathcal{E}_\bot(\tau_1)} \quad \text{(T-Abs)}$$

Case T-App: $\vdash_{g\lambda} \mathtt{t}_1\ \mathtt{t}_2 : (\tau_1, \bot)$ with sub-derivations:
$\vdash_{g\lambda} \mathtt{t}_1 : (\tau_2^{\mathfrak{B}_2} \to \tau_1, \mathfrak{B}_1)$
$\vdash_{g\lambda} \mathtt{t}_2 : (\tau_2, \mathfrak{B}'_2)$,

As $\vdash_{g\lambda} \mathtt{t}_1\ \mathtt{t}_2 : (\tau_1, \bot)$, the inversion lemma gives as By inversion that $\vDash \mathfrak{B}_2 \Leftrightarrow \mathfrak{B}'_2$. That is, we have two cases to consider.

If $\mathfrak{B}_2 = \mathfrak{B}'_2 = \bot$ then by induction hypotheses
$\vdash_\lambda \mathcal{E}_\bot(\mathtt{t}_1) : \mathcal{E}_\bot(\tau_2) \to \mathcal{E}_\bot(\tau_1)$ and $\vdash_\lambda \mathcal{E}_\bot(\mathtt{t}_2) : \mathcal{E}_\bot(\tau_2)$. By T-App rule,

$$\frac{\vdash_\lambda \mathcal{E}_\bot(\mathtt{t}_1) : \mathcal{E}_\bot(\tau_2) \to \mathcal{E}_\bot(\tau_1) \qquad \vdash_\lambda \mathcal{E}_\bot(\mathtt{t}_2) : \mathcal{E}_\bot(\tau_2)}{\vdash_\lambda \mathcal{E}_\bot(\mathtt{t}_1\ \mathtt{t}_2) : \mathcal{E}(\tau_1)} \quad \text{(T-App)}$$

If $\mathfrak{B}_2 = \mathfrak{B}'_2 = \top$, then by definition of $\mathcal{E}$ we have $\mathcal{E}(\mathtt{t}_2) = \mathtt{()}$ and $\mathcal{E}_{\mathfrak{B}'_2}(\tau_2) = \mathtt{unit}$. By induction hypothesis on $\mathtt{t}_1$, $\vdash_\lambda \mathcal{E}_\bot(\mathtt{t}_1) : \mathtt{unit} \to \mathcal{E}_\bot(\tau_1)$. Applying T-App rule gives us

$$\frac{\dfrac{}{\vdash_\lambda \mathcal{E}_\bot(\mathtt{t}_1\ \mathtt{t}_2) : \mathcal{E}(\tau_1)} \quad \dfrac{}{\vdash_\lambda \mathtt{()} : \mathtt{unit}}\ \text{(T-Unit)}}{\vdash_\lambda \mathcal{E}_\bot(\mathtt{t}_1\ \mathtt{t}_2) : \mathcal{E}_\bot(\tau_1)}\ \text{(T-App)}$$

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. □

2 Extensions

So far we have only showed how to give a formal meaning to ghost code and how to erase it from the programs safely with respect to evaluation.

From now, we move towards more realistic examples, where we could show how ghost code can be used for program specification.

In this section we show how to extend the $g\lambda$-calculus with some basic constructions such as conditionals and local bindings, recursive definitions, and access and assignment of global references. In the same time, we extend the typing relation and erase of terms and type, and reprove the preservation properties of the previous section.

2.1 Simple Extensions

2.2 Global References

2.3 Properties of Ghost Erasure