A Simple Abstract Interpretation Framework for Modular Analysis

JOONHYUP LEE and KWANGKEUN YI

1 SYNTAX AND SEMANTICS

1.1 Abstract Syntax

$$
\begin{array}{rcll}
\text{Identifiers} & x & \in & \text{Var} \\
\text{Expression} & e & \rightarrow & x \mid \lambda x.e \mid e\ e & \lambda\text{-calculus} \\
& & \mid & e \rtimes e & \text{linked expression} \\
& & \mid & \varepsilon & \text{empty module} \\
& & \mid & x = e\ ;\ e & \text{(recursive) binding}
\end{array}
$$

Fig. 1. Abstract syntax of the language.

1.2 Operational Semantics

$$
\begin{array}{rcll}
\text{Environment} & \sigma & \in & \text{Env} \\
\text{Location} & \ell & \in & \text{Loc} \triangleq \{\text{infinite set of locations}\} \\
\text{Value} & v & \in & \text{Val} \triangleq \text{Env} + \text{Var} \times \text{Expr} \times \text{Env} \\
\text{Weak Value} & w & \in & \text{WVal} \triangleq \text{Val} + \text{Loc} \times \text{Val} \\
\text{Environment} & \sigma & \rightarrow & \bullet & \text{empty stack} \\
& & \mid & (x, \ell) :: \sigma & \text{free location binding} \\
& & \mid & (x, w) :: \sigma & \text{weak value binding} \\
\text{Value} & v & \rightarrow & \sigma & \text{exported environment} \\
& & \mid & \langle \lambda x.e, \sigma \rangle & \text{closure} \\
\text{Weak Value} & w & \rightarrow & v & \text{value} \\
& & \mid & \mu\ell.v & \text{recursive value}
\end{array}
$$

Fig. 2. Definition of the semantic domains.

1.3 Reconciling with Conventional Backpatching

The semantics in Figure 3 makes sense due to similarity with a conventional backpatching semantics as presented in Figure 5. We have defined a relation $\sim$ that satisfies:

$$
\sim \subseteq \text{WVal} \times (\text{MVal} \times \text{Mem} \times \mathscr{P}(\text{Loc})) \qquad \bullet \sim (\bullet, \varnothing, \varnothing)
$$

Authors' address: Joonhyup Lee; Kwangkeun Yi.

XXXX-XXXX/2024/2-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$\boxed{\sigma \vdash e \Downarrow v}$$

$$\textsc{Id}\ \frac{\sigma(x) = v}{\sigma \vdash x \Downarrow v} \qquad \textsc{RecId}\ \frac{\sigma(x) = \mu\ell.v}{\sigma \vdash x \Downarrow v[\mu\ell.v/\ell]} \qquad \textsc{Fn}\ \frac{}{\sigma \vdash \lambda x.e \Downarrow \langle \lambda x.e, \sigma \rangle} \qquad \textsc{App}\ \frac{\sigma \vdash e_1 \Downarrow \langle \lambda x.e, \sigma_1 \rangle \quad \sigma \vdash e_2 \Downarrow v_2 \quad (x, v_2) :: \sigma_1 \vdash e \Downarrow v}{\sigma \vdash e_1\ e_2 \Downarrow v}$$

$$\textsc{Link}\ \frac{\sigma \vdash e_1 \Downarrow \sigma_1 \quad \sigma_1 \vdash e_2 \Downarrow v}{\sigma \vdash e_1 \rtimes e_2 \Downarrow v} \qquad \textsc{Empty}\ \frac{}{\sigma \vdash \varepsilon \Downarrow \bullet} \qquad \textsc{Bind}\ \frac{\ell \notin \mathrm{FLoc}(\sigma) \quad (x, \ell) :: \sigma \vdash e_1 \Downarrow v_1 \quad (x, \mu\ell.v_1) :: \sigma \vdash e_2 \Downarrow \sigma_2}{\sigma \vdash x = e_1; e_2 \Downarrow (x, \mu\ell.v_1) :: \sigma_2}$$

Fig. 3. The big-step operational semantics.

$$\begin{array}{rcll}
\text{Environment} & \sigma & \in & \text{MEnv} \\
\text{Memory} & m & \in & \text{Mem} \triangleq \text{Loc} \xrightarrow{\text{fin}} \text{MVal} \\
\text{Value} & v & \in & \text{MVal} \triangleq \text{MEnv} + \text{Var} \times \text{Expr} \times \text{MEnv} \\
\text{Environment} & \sigma & \rightarrow & \bullet & \text{empty stack} \\
& & \mid & (x, \ell) :: \sigma & \text{location binding} \\
\text{Value} & v & \rightarrow & \sigma & \text{exported environment} \\
& & \mid & \langle \lambda x.e, \sigma \rangle & \text{closure}
\end{array}$$

Fig. 4. Definition of the semantic domains with memory.

and the following theorem:

Theorem 1.1 (Equivalence of semantics). For all $\sigma \in \text{Env}, \sigma' \in \text{MEnv} \times \text{Mem} \times \mathscr{P}(\text{Loc}), v \in \text{Val}, v' \in \text{MVal} \times \text{Mem} \times \mathscr{P}(\text{Loc})$, we have:

$$\sigma \sim \sigma' \text{ and } \sigma \vdash e \Downarrow v \Rightarrow \exists v' : v \sim v' \text{ and } \sigma' \vdash e \Downarrow v'$$
$$\sigma \sim \sigma' \text{ and } \sigma' \vdash e \Downarrow v' \Rightarrow \exists v : v \sim v' \text{ and } \sigma \vdash e \Downarrow v$$

The actual definition for $\sim$ can be found in the appendix.

$$\boxed{\sigma, m, L \vdash e \Downarrow v, m', L'}$$

$$\textsc{Id}\ \frac{\sigma(x) = \ell \qquad m(\ell) = v}{\sigma, m, L \vdash x \Downarrow v, m, L} \qquad \textsc{Fn}\ \frac{}{\sigma, m, L \vdash \lambda x.e \Downarrow \langle \lambda x.e, \sigma \rangle, m, L}$$

$$\textsc{App}\ \frac{\begin{array}{c}\sigma, m, L \vdash e_1 \Downarrow \langle \lambda x.e, \sigma_1 \rangle, m_1, L_1 \qquad \sigma, m_1, L_1 \vdash e_2 \Downarrow v_2, m_2, L_2 \qquad \ell \notin \mathrm{dom}(m_2) \cup L_2 \\ (x, \ell) :: \sigma_1, m_2[\ell \mapsto v_2], L_2 \vdash e \Downarrow v, m', L'\end{array}}{\sigma, m, L \vdash e_1\ e_2 \Downarrow v, m', L'}$$

$$\textsc{Link}\ \frac{\sigma, m, L \vdash e_1 \Downarrow \sigma_1, m_1, L_1 \qquad \sigma_1, m_1, L_1 \vdash e_2 \Downarrow v, m', L'}{\sigma, m, L \vdash e_1 \rtimes e_2 \Downarrow v, m', L'} \qquad \textsc{Empty}\ \frac{}{\sigma, m, L \vdash \varepsilon \Downarrow \bullet, m, L}$$

$$\textsc{Bind}\ \frac{\begin{array}{c}\ell \notin \mathrm{dom}(m) \cup L \qquad (x, \ell) :: \sigma, m, L \cup \{\ell\} \vdash e_1 \Downarrow v_1, m_1, L_1 \\ (x, \ell) :: \sigma, m_1[\ell \mapsto v_1], L_1 \vdash e_2 \Downarrow \sigma_2, m', L'\end{array}}{\sigma, m, L \vdash x = e_1; e_2 \Downarrow (x, \ell) :: \sigma_2, m', L'}$$

Fig. 5. The big-step operational semantics with memory.

2 GENERATING AND RESOLVING EVENTS

Now we formulate the semantics for generating events.

$$
\begin{array}{rrcll}
\text{Environment} & \sigma & \rightarrow & \cdots & \\
 & & \mid & [E] & \text{answer to an event} \\
\text{Value} & v & \rightarrow & \cdots & \\
 & & \mid & E & \text{answer to an event} \\
\text{Event} & E & \rightarrow & \mathsf{Init} & \text{initial environment} \\
 & & \mid & \mathsf{Read}(E, x) & \text{read event} \\
 & & \mid & \mathsf{Call}(E, v) & \text{call event}
\end{array}
$$

Fig. 6. Definition of the semantic domains with events. All other semantic domains are equal to Figure 2.

We redefine how to read weak values given an environment.

$$
\begin{array}{rclrcll}
\bullet(x) & \triangleq & \bot & ((x, w) :: \sigma)(x) & \triangleq & w & \\
((x, \ell) :: \sigma)(x) & \triangleq & \bot & ((x', \_) :: \sigma)(x) & \triangleq & \sigma(x) & (x \neq x') \\
{[E]}(x) & \triangleq & \mathsf{Read}(E, x) & & & &
\end{array}
$$

Then we need to add only one rule to the semantics in Figure 3 for the semantics to incorporate events.

$$
\textsc{AppEvent}\ \frac{\sigma \vdash e_1 \Downarrow E \qquad \sigma \vdash e_2 \Downarrow v}{\sigma \vdash e_1\ e_2 \Downarrow \mathsf{Call}(E, v)}
$$

Now we need to formulate the *concrete linking* rules. The concrete linking rule $\sigma_0 \propto w$, given an answer σ_0 to the Init event, resolves all events within w to obtain a set of final results.

Concrete linking makes sense because of the following theorem. First define:

$$
\mathrm{eval}(e, \sigma) \triangleq \{v | \sigma \vdash e \Downarrow v\} \qquad \mathrm{eval}(e, \Sigma) \triangleq \bigcup_{\sigma \in \Sigma} \mathrm{eval}(\sigma, e) \qquad \sigma_0 \propto W \triangleq \bigcup_{w \in W} (\sigma_0 \propto w)
$$

Then the following holds:

Theorem 2.1 (Soundness of concrete linking). *Given $e \in \mathrm{Expr}, \sigma \in \mathrm{Env}, v \in \mathrm{Val}$,*

$$
\forall \sigma_0 \in \mathrm{Env} : \mathrm{eval}(e, \sigma_0 \propto \sigma) \subseteq \sigma_0 \propto \mathrm{eval}(e, \sigma)
$$

$$\boxed{\infty \in \mathrm{Env} \to \mathrm{Event} \to \mathscr{P}(\mathrm{Val})}$$

$$\begin{aligned}
\sigma_0 \infty \mathsf{Init} &\triangleq \{\sigma_0\} \\
\sigma_0 \infty \mathsf{Read}(E, x) &\triangleq \{V | \Sigma \in \sigma_0 \infty E \text{ and } \Sigma(x) = V\} \cup \\
&\quad\ \{V[\mu\ell.V/\ell] | \Sigma \in \sigma_0 \infty E \text{ and } \Sigma(x) = \mu\ell.V\} \\
\sigma_0 \infty \mathsf{Call}(E, v) &\triangleq \{V' | \langle \lambda x.e, \Sigma \rangle \in \sigma_0 \infty E \text{ and } V \in \sigma_0 \infty v \text{ and } (x, V) :: \Sigma \vdash e \Downarrow V'\} \cup \\
&\quad\ \{\mathsf{Call}(E', V) | E' \in \sigma_0 \infty E \text{ and } V \in \sigma_0 \infty v\}
\end{aligned}$$

$$\boxed{\infty \in \mathrm{Env} \to \mathrm{Env} \to \mathscr{P}(\mathrm{Env})}$$

$$\begin{aligned}
\sigma_0 \infty \bullet &\triangleq \{\bullet\} \\
\sigma_0 \infty (x, \ell) :: \sigma &\triangleq \{(x, \ell) :: \Sigma | \Sigma \in \sigma_0 \infty \sigma\} \\
\sigma_0 \infty (x, w) :: \sigma &\triangleq \{(x, W) :: \Sigma | W \in \sigma_0 \infty w \text{ and } \Sigma \in \sigma_0 \infty \sigma\} \\
\sigma_0 \infty [E] &\triangleq \{\Sigma \in \mathrm{Env} | \Sigma \in \sigma_0 \infty E\} \cup \\
&\quad\ \{[E'] | E' \in \sigma_0 \infty E\}
\end{aligned}$$

$$\boxed{\infty \in \mathrm{Env} \to \mathrm{Val} \to \mathscr{P}(\mathrm{Val})}$$

$$\sigma_0 \infty \langle \lambda x.e, \sigma \rangle \triangleq \{\langle \lambda x.e, \Sigma \rangle | \Sigma \in \sigma_0 \infty \sigma\}$$

$$\boxed{\infty \in \mathrm{Env} \to \mathrm{WVal} \to \mathscr{P}(\mathrm{WVal})}$$

$$\sigma_0 \infty \mu\ell.v \triangleq \{\mu\ell.V | V \in \sigma_0 \infty v\}$$

Fig. 7. Definition for concrete linking.

3 TYPING

The definitions for types are in Figure 8 and the typing rules are in Figure 9. The definitions for subtyping are in Figure 10.

$$\begin{array}{rcll} \text{Types} & \tau & \to & \Gamma & \text{module type} \\ & & \mid & \tau \to \tau & \text{function type} \\ \text{Typing Environment} & \Gamma & \to & \bullet & \text{empty environment} \\ & & \mid & (x,\tau) :: \Gamma & \text{type binding} \end{array}$$

Fig. 8. Definition of types.

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

$$\textsc{T-Id}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{T-Fn}\ \frac{(x,\tau_1) :: \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \textsc{T-App}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_1 \geq \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\textsc{T-Link}\ \frac{\Gamma \vdash e_1 : \Gamma_1 \qquad \Gamma_1 \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \rtimes e_2 : \tau_2} \qquad \textsc{T-Nil}\ \frac{}{\Gamma \vdash \varepsilon : \bullet} \qquad \textsc{T-Bind}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad (x,\tau_1) :: \Gamma \vdash e_2 : \Gamma_2}{\Gamma \vdash x = e_1; e_2 : (x,\tau_1) :: \Gamma_2}$$

Fig. 9. The typing judgment.

$$\boxed{\tau \geq \tau'}$$

$$\textsc{Nil}\ \frac{}{\bullet \geq \bullet} \qquad \textsc{ConsFree}\ \frac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma \geq \Gamma'}{\Gamma \geq (x,\tau) :: \Gamma'} \qquad \textsc{ConsBound}\ \frac{\Gamma(x) \geq \tau \qquad \Gamma - x \geq \Gamma'}{\Gamma \geq (x,\tau) :: \Gamma'} \qquad \textsc{Arrow}\ \frac{\tau_2 \geq \tau_1 \qquad \tau_1' \geq \tau_2'}{\tau_1 \to \tau_1' \geq \tau_2 \to \tau_2'}$$

Fig. 10. The subtype relation.

3.1 Type Safety

Theorem 3.1 (Type Safety). *For all $e \in \text{Expr}$, if $\bullet \vdash e : \tau$ for some τ, then there exists some $v \in \text{Val}$ such that $\bullet \vdash e \Downarrow v$.*

Sketch. We prove this through unary logical relations and induction on the typing judgment.

Value Relation $\boxed{\mathcal{V}[\![\tau]\!]}$

$$\begin{array}{rcl} \mathcal{V}[\![\bullet]\!] & \triangleq & \text{Env} \\ \mathcal{V}[\![(x,\tau) :: \Gamma]\!] & \triangleq & \{\sigma \mid \sigma(x) \in \mathcal{V}[\![\tau]\!] \text{ and } \sigma \in \mathcal{V}[\![\Gamma - x]\!]\} \\ \mathcal{V}[\![\tau_1 \to \tau_2]\!] & \triangleq & \{\langle \lambda x.e, \sigma\rangle \mid \forall v \in \mathcal{V}[\![\tau_1]\!] : (e, (x,v) :: \sigma) \in \mathcal{E}[\![\tau_2]\!]\} \end{array}$$

Expression Relation $\boxed{\mathcal{E}[\![\tau]\!]}$

$$\mathcal{E}[\![\tau]\!] \triangleq \{(e,\sigma) \mid \exists v \in \mathcal{V}[\![\tau]\!] : \sigma \vdash e \Downarrow v\}$$

Semantic Typing $\boxed{\Gamma \vDash e : \tau}$

$$\Gamma \vDash e : \tau \triangleq \forall \sigma \in \mathcal{V}[\![\Gamma]\!] : (e,\sigma) \in \mathcal{E}[\![\tau]\!]$$

We prove

$$\Gamma \vdash e : \tau \Rightarrow \Gamma \vDash e : \tau$$

by induction on $\vdash$. □

3.2 Type Inference

When modules are first-class, type variables can go in the place of type environments.

First we define the syntax for type constraints.

$$
\begin{array}{rclll}
\text{Type Variable} & \alpha & \in & \text{TyVar} & \\
\text{Path} & p & \to & \epsilon & \text{empty string} \\
 & & \mid & px & \text{concatenation with identifier} \\
\text{Types} & \tau & \to & \Gamma \mid \tau \to \tau & \text{module/function types} \\
\text{Type Environment} & \Gamma & \to & \bullet & \text{empty environment} \\
 & & \mid & (x, \tau) :: \Gamma & \text{binding} \\
 & & \mid & \alpha.p & \text{type variable} \\
 & & \mid & [].p & \text{type read from hole} \\
\text{Type Constraint} & u & \to & \tau \doteq \tau & \text{equality constraint} \\
 & & \mid & \tau \mathrel{\dot{\geq}} \tau & \text{subtyping constraint} \\
\text{Set of Constraints} & U & \subseteq & \{u \mid u \text{ type constraint}\} &
\end{array}
$$

Fig. 11. Definition of type constraints.

Next we define the type access operation $\tau(x)$:

$$
\begin{array}{rlcrl}
\bullet(x) & \triangleq \bot & & (\alpha.p)(x) & \triangleq \alpha.px \\
((x, \tau) :: \_)(x) & \triangleq \tau & & ([].p)(x) & \triangleq [].px \\
((x', \_) :: \Gamma)(x) & \triangleq \Gamma(x) & \text{when } x' \neq x & (\_ \to \_)(x) & \triangleq \bot
\end{array}
$$

Now we can define the constraint generation algorithm $V(\Gamma, e, \alpha)$. Note that the **let** U = _ **in** _ notation returns $\bot$ if the right hand side is $\bot$. Also note that we write α for $\alpha.\epsilon$ as well.

$$\boxed{V(\Gamma, e, \alpha) = U}$$

$$
\begin{aligned}
V(\Gamma, \varepsilon, \alpha) &\triangleq \{\alpha \doteq \bullet\} \\
V(\Gamma, x, \alpha) &\triangleq \textbf{let } \tau = \Gamma(x) \textbf{ in} \\
&\quad\ \{\alpha \doteq \tau\} \\
V(\Gamma, \lambda x.e, \alpha) &\triangleq \textbf{let } \alpha_1, \alpha_2 = \textit{fresh} \textbf{ in} \\
&\quad\ \textbf{let } U = V((x, \alpha_1) :: \Gamma, e, \alpha_2) \textbf{ in} \\
&\quad\ \{\alpha \doteq \alpha_1 \to \alpha_2\} \cup U \\
V(\Gamma, e_1\ e_2, \alpha) &\triangleq \textbf{let } \alpha_1, \alpha_2, \alpha_3 = \textit{fresh} \textbf{ in} \\
&\quad\ \textbf{let } U_1 = V(\Gamma, e_1, \alpha_1) \textbf{ in} \\
&\quad\ \textbf{let } U_2 = V(\Gamma, e_2, \alpha_2) \textbf{ in} \\
&\quad\ \{\alpha_1 \doteq \alpha_3 \to \alpha, \alpha_3 \mathrel{\dot{\geq}} \alpha_2\} \cup U_1 \cup U_2
\end{aligned}
\qquad
\begin{aligned}
V(\Gamma, e_1 \rtimes e_2, \alpha) &\triangleq \textbf{let } \alpha_1 = \textit{fresh} \textbf{ in} \\
&\quad\ \textbf{let } U_1 = V(\Gamma, e_1, \alpha_1) \textbf{ in} \\
&\quad\ \textbf{let } U_2 = V(\alpha_1, e_2, \alpha) \textbf{ in} \\
&\quad\ U_1 \cup U_2 \\
V(\Gamma, x = e_1; e_2, \alpha) &\triangleq \textbf{let } \alpha_1, \alpha_2 = \textit{fresh} \textbf{ in} \\
&\quad\ \textbf{let } U_1 = V(\Gamma, e_1, \alpha_1) \textbf{ in} \\
&\quad\ \textbf{let } U_2 = V((x, \alpha_1) :: \Gamma, e_2, \alpha_2) \textbf{ in} \\
&\quad\ \{\alpha \doteq (x, \alpha_1) :: \alpha_2\} \cup U_1 \cup U_2
\end{aligned}
$$

We want to prove that the constraint generation algorithm is correct.

First, for $\tau \in$ Type, define the path access operation $\tau(p)$:

$$\tau(\epsilon) \triangleq \tau \qquad\qquad \tau(px) \triangleq \tau(p)(x)$$

and define the injection operation $\tau[\Gamma_0]$:

$$(\bullet)[\Gamma_0] \triangleq \bullet \qquad ((x,\tau) :: \Gamma)[\Gamma_0] \triangleq (x, \tau[\Gamma_0]) :: \Gamma[\Gamma_0]$$
$$(\alpha.p)[\Gamma_0] \triangleq \alpha.p \qquad ([].p)[\Gamma_0] \triangleq \Gamma_0(p)$$
$$(\tau_1 \to \tau_2)[\Gamma_0] \triangleq \tau_1[\Gamma_0] \to \tau_2[\Gamma_0]$$

Let Subst $\triangleq$ TyVar $\xrightarrow{\text{fin}}$ Type be the set of substitutions. For $S \in$ Subst, define:

$$S\bullet \triangleq \bullet \qquad S(\tau_1 \to \tau_2) \triangleq S\tau_1 \to S\tau_2$$
$$S(\alpha.p) \triangleq \alpha.p \quad \text{when } \alpha \notin \mathrm{dom}(S) \qquad S(\alpha.p) \triangleq \tau(p) \quad \text{when } \alpha \mapsto \tau \in S$$
$$S([].p) \triangleq [].p$$

Define:

$$(S, \Gamma_0) \vDash U \triangleq \forall(\tau_1 \doteq \tau_2) \in U : (S\tau_1)[\Gamma_0] = (S\tau_2)[\Gamma_0] \text{ and}$$
$$\forall(\tau_1 \dot{\geq} \tau_2) \in U : (S\tau_1)[\Gamma_0] \geq (S\tau_2)[\Gamma_0]$$

where subtyping rules are the same as Figure 10 and subtyping between type variables are not defined.

Then we can show that:

Theorem 3.2 (Correctness of V). For $e \in$ Expr, $\Gamma, \Gamma_0 \in$ TyEnv, $\alpha \in$ TyVar, $S \in$ Subst:

$$(S, \Gamma_0) \vDash V(\Gamma, e, \alpha) \Leftrightarrow (S\Gamma)[\Gamma_0] \vdash e : (S\alpha)[\Gamma_0]$$

Sketch. Structural induction on e. □

Note that by including $[].p$ in type environments, we can naturally generate constraints about the external environment $[]$. Also, by injection, we can utilize constraints generated *in advance* to obtain constraints generated from a more informed environment. We extend injection to the output of the constraint-generating algorithm:

$$\bot[\Gamma_0] \triangleq \bot$$
$$U[\Gamma_0] \triangleq \{\tau_1[\Gamma_0] \doteq \tau_2[\Gamma_0] | (\tau_1 \doteq \tau_2) \in U\} \cup$$
$$\{\tau_1[\Gamma_0] \dot{\geq} \tau_2[\Gamma_0] | (\tau_1 \dot{\geq} \tau_2) \in U\} \qquad \text{when all injections succeed}$$
$$U[\Gamma_0] \triangleq \bot \qquad \text{when injection fails}$$

Then we can prove:

Theorem 3.3 (Advance). For $e \in$ Expr, $\Gamma, \Gamma_0 \in$ TyEnv, $\alpha \in$ TyVar:

$$V(\Gamma[\Gamma_0], e, \alpha) = V(\Gamma, e, \alpha)[\Gamma_0]$$

Sketch. Structural induction on Γ. □

REFERENCES