Verification of Machine Codes Using An Effect Type System

Jaeyoun Chung, Sukyoung Ryu and Kwangkeun Yi

\{jay,puppy,kwang\}@ropas.kaist.ac.kr

July 13, 2000

Abstract

We devise a mechanism to verify the safety of machine codes. We design a stack based machine etySECK whose code part is annotated with types and effects. And we propose an effect type system to verify properties of etySECK programs. Our system analyzes memory effects as a property of programs and we can extend our system to analyze other effects.

1 Effect-Typed Abstract Machine : etySECK

The etySECK machine is a variant of Landin’s SECD [1, 2, 3, 4] machine with type and effect annotations. The syntax and the semantics of the machine is described in Figure 1 and 2, respectively. Since the machine supports functional values, compilation from functional languages to this machine is not difficult. Throughout this paper, we use dot(,) as a list constructing operator and use a single value for a list of length one.

1.1 Syntax

A machine configuration consists of six components – a stack, a heap, an environment, a sequence of commands, a continuation and a set of effects of memory behaviors. The stack and the environment are just lists of values. Each command may consume several values from the stack and produce a result on the stack. The environment represents a lexically enclosing environment and the commands access the values in the environment. The continuation is a list of pairs of an environment and a code. Each pair of the continuation is a frozen state of the machine when the function application appears by the app command. The set of effects is a trace of memory behaviors which have occurred until the machine configuration reaches at the current state.

A memory region is a set of memory locations which are assumed to be allocated to the same block of the memory. Our system analyzes observable side-effects of memory behaviors – init, read, write – for regions.

The values are either primitive values, a location, or function values. A function value is a triple of its type, the function’s body, and an environment for static binding. A polymorphic value is either \texttt{fn} or \texttt{rfn} and it can be instantiated to a concrete monomorphic value.

The type is either an integer type, a boolean type, a unit type, a location type, a function type, a type variable, or a polymorphic type. A location type indicates that a value of this type is a location pointer to a value in the heap. A function type is annotated with a latent effect indicating a possible set of effects during the execution of the function’s body.

\footnote{This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology.}
Machine Configuration: \((S, H, E, C, K, \varphi)\)

\[
\begin{align*}
S & \in \text{Stack} = \text{Value list} & \text{stack} \\
H & \in \text{Heap} = \text{Loc} \rightarrow \text{Value} & \text{heap} \\
E & \in \text{Env} = \text{Value list} & \text{environment} \\
K & \in \text{Cont} = (\text{Env} \times \text{Cmds}) \text{ list} & \text{continuation}
\end{align*}
\]

\[
\begin{align*}
\text{Region} \ni \rho & ::= r_i \quad \text{region constant} \\
& \quad | \rho_i \quad \text{region variable} \\
\text{Effect} \ni \varphi & ::= \epsilon \quad \text{no effect} \\
& \quad | \{\text{init}(\rho)\} \quad \text{memory allocated} \\
& \quad | \{\text{read}(\rho)\} \quad \text{memory read} \\
& \quad | \{\text{write}(\rho)\} \quad \text{memory write} \\
& \quad | \varphi \cup \varphi' \quad \text{effect composition}
\end{align*}
\]

\[
\begin{align*}
\text{Type} \ni \tau & ::= i | b \quad \text{integer, boolean} \\
& \quad | \text{unit} \quad \text{unit type} \\
& \quad | \text{ref}_\rho \tau \quad \text{location type} \\
& \quad | \tau \rightarrow\tau \quad \text{function type} \\
& \quad | \alpha \quad \text{type variable} \\
& \quad | \forall(\vec{\alpha}, \vec{\rho}).\tau \quad \text{polymorphic type}
\end{align*}
\]

\[
\begin{align*}
\text{Prim.Value} \ni p & ::= n | \text{true} | \text{false} | ()
\end{align*}
\]

\[
\begin{align*}
\text{Value} \ni v & ::= p \quad \text{primitive values} \\
& \quad | l_p \quad \text{location} \\
& \quad | \text{fn}(\tau, C, E) \quad \text{function} \\
& \quad | \text{rfn}(\tau, C, E) \quad \text{recursive function} \\
& \quad | \Lambda(\vec{\alpha}, \vec{\rho}).v \quad \text{polymorphic value}
\end{align*}
\]

\[
\begin{align*}
\text{Cmd} \ni C & ::= \text{nil} \\
& \quad | \text{frame}.C \quad | \text{deframe}.C \\
& \quad | \text{fetch}(n).C \quad | \text{quote}(p).C \\
& \quad | \text{add}.C \quad | \text{cond}(C_1, C_2).C \\
& \quad | \text{fn}(\forall \vec{\alpha}, \vec{\rho}.C_f : \tau_1 \rightarrow\tau_2).C \\
& \quad | \text{rfn}(\forall \vec{\alpha}, \vec{\rho}.C_f : \tau_1 \rightarrow\tau_2).C \\
& \quad | \text{app}.C \\
& \quad | \text{tapp}(\vec{\tau}, \vec{\rho}).C \\
& \quad | \text{ref}(\rho).C \quad | \text{get}.C \quad | \text{set}.C
\end{align*}
\]

Figure 1: Machine Configuration of etySECK
(S, H, E, nil, (E’, C), K, φ) ⇒ (S, H, E’, C, K, φ)
(S, H, v₀, ..., vₙ, E, fetch(n), C, K, φ) ⇒ (vₙ, S, H, v₀, ..., vₙ, E, C, K, φ)
(true.S, H, E, cond(C₁, C₂), C, K, φ) ⇒ (S, H, E, C₁, C, K, φ)
(false.S, H, E, cond(C₁, C₂), C, K, φ) ⇒ (S, H, E, C₂, C, K, φ)
(v.S, H, E, ref(p). C, K, φ) ⇒ (lₚ.S, H[lₚ → v]. E, C, K, φ ∪ \{init(p)\}) \ lₚ \ \not \in \ Dom(H)
(v.S, H, E, get.C, K, φ) ⇒ (H(lₚ). S, H, E, C, K, φ ∪ \{read(p)\}) \ lₚ \ \in \ Dom(H)
(v.lₚ.S, H, E, set.C, K, φ) ⇒ ((lₚ.S, H[lₚ → v]. E, C, K, φ ∪ \{write(p)\})) \ lₚ \ \in \ Dom(H)

Figure 2: Machine Transition Rules of etySECK

1.2 Semantics

The semantics of our language is described by the transition relations (⇒) in Figure 2.

If the current sequence of commands is empty (nil), the first element of the continuation replaces the current environment and the commands:


Each instruction may take some values from the top of the stack and put a result value on the stack. The frame moves the stack top value into the environment and the deframe removes the value on the top of the environment:


The fetch(n) gets the n-th value from the environment and puts the value on the stack. A new constant value p is put on the stack by the quote. The add gets two values from the stack and puts the sum on the stack. The cond(C₁, C₂) branches to either C₁ or C₂ according to the truth value of the stack top.
Function values are produced by the \texttt{fn} or the \texttt{rfn}. Let's consider the \texttt{fn} rule:

\[
\begin{align*}
& (S, H, E, \texttt{fn}(\forall \vec{\alpha}, \vec{\rho}. C) : \tau_1 \xrightarrow{\varphi} \tau_2). C, K, \varphi) \\
& \Rightarrow (\Lambda(\vec{\alpha}, \vec{\rho}). \texttt{fn}(\forall \vec{\alpha}, \vec{\rho}. C) : \tau_1 \xrightarrow{\varphi} \tau_2). C, K, \varphi).
\end{align*}
\]

Note that produced values are polymorphic functions which cannot be directly used but should be instantiated to monomorphic ones by the \texttt{tapp}:

\[
\begin{align*}
& (\Lambda(\vec{\alpha}, \vec{\rho}). \texttt{fn}(\forall \vec{\alpha}, \vec{\rho}. C) : \tau_1 \xrightarrow{\varphi} \tau_2). C, K, \varphi) \\
& \Rightarrow (\Lambda(\vec{\alpha}, \vec{\rho}). \texttt{fn}(\forall \vec{\alpha}, \vec{\rho}. C) : \tau_1 \xrightarrow{\varphi} \tau_2). S, H, E, C, K, \varphi).
\end{align*}
\]

When the \texttt{app} applies the argument on the stack top to the function on the second of the stack, the function value is already instantiated to a monomorphic one. The following rule is for applying non-recursive functions:

\[
\begin{align*}
& (v. \texttt{fn}(\forall \vec{\alpha}, \vec{\rho}. C) : \tau_1 \xrightarrow{\varphi} \tau_2). S, H, E, \texttt{app}. C, K, \varphi) \\
& \Rightarrow (S, H, v. E, C, K, \varphi).
\end{align*}
\]

In case of applying recursive functions, the recursive function value is also placed on the environment so that the function value is visible during the evaluation of the function's body.

Operations on the heap are \texttt{ref}, \texttt{get}, and \texttt{set}. The \texttt{ref} allocates a memory cell, stores the stack top value to it, and places the location of the cell on the stack:

\[
\begin{align*}
& (v. \texttt{fn}(\forall \vec{\alpha}, \vec{\rho}. C) : \tau_1 \xrightarrow{\varphi} \tau_2). S, H, E, \texttt{ref}. \rho. C, K, \varphi) \\
& \Rightarrow (l_\rho. S, H[l_\rho \rightarrow v], E, C, K, \varphi).
\end{align*}
\]

The \texttt{get} fetches a value from the heap and the \texttt{set} updates a value in the heap. Note that each operation on the heap produces a corresponding effect.

The etySECK machine starts from the initial configuration with all components except the commands are empty. The machine stops when both the commands and the continuation are empty or there is no transition rule to apply. The former case is when the machine terminates normally and the latter case is when the machine is in a stuck configuration. We define these cases as follows:

**Definition 1 (Normal Termination)** Program $C$ terminates normally if

\[
(n, C, \texttt{nil}, \texttt{nil}, \varphi, \texttt{nil}) \Rightarrow (S, H, E, \texttt{nil}, \texttt{nil}, \varphi).
\]

**Definition 2 (Stuck Configuration)** The etySECK machine configuration $(S, H, E, C, K, \varphi)$ $(C \neq \texttt{nil}, K \neq \texttt{nil})$ is in a stuck configuration if there is no $(S', H', E', C', K', \varphi')$ such that $(S, H, E, C, K, \varphi) \Rightarrow (S', H', E', C', K', \varphi')$.

### 1.3 Type System

The type of a machine configuration is defined by the following judgement:

\[
(S, H, E, C, K, \varphi) \rightsquigarrow (s_o, e_o, \varphi_o).
\]

This judgement means that if a given machine configuration continues its execution to a final configuration then the stack, the environment, and the effects of the final configuration have the types $s_o$, $e_o$, and the effects $\varphi_o$, respectively. Figure 3 defines these typing rules.

In order to determine the type of a given machine configuration, we need to execute the configuration abstractly – that is, we need to execute it based on the type system. Typing
\[ (S, H, E, C, K, \varphi) \rightsquigarrow (s_0, e_0, \varphi_0) \]

\[
H \vdash S : s_1 \quad H \vdash E : e_1 \quad H \vdash K : k_1 \quad (s_1, e_1, k_1, \varphi) \vdash C \rightsquigarrow (s_0, e_0, \varphi_0) \\
(S, H, E, C, K, \varphi) \rightsquigarrow (s_0, e_0, \varphi_0)
\]

\[
H \vdash S : s, H \vdash E : e, H \vdash K : k
\]

\[
H \vdash v : \tau \quad H \vdash S : s \\
\quad \quad \quad \quad H \vdash v.S : \tau.s \\
H \vdash v : \tau \quad H \vdash E : e \\
\quad \quad \quad \quad H \vdash E.C : (e, C).k \\
H \vdash v.E : \tau.e
\]

\[
H \vdash v : \tau
\]

\[
H \vdash p : \text{type_of}(p) \quad l_p \in \text{Dom}(H) \quad H \vdash H(l_p) : \tau \\
\quad \quad \quad \quad H \vdash l_p : \text{ref}_{\rho \tau} \\
H \vdash E_f : e \quad \langle \epsilon, \tau_1.e, \epsilon, \epsilon \rangle \vdash C_f \rightsquigarrow \langle \tau_2, \tau_1.e, \varphi' \rangle \quad \varphi' \subseteq \varphi_f \\
\quad \quad \quad \quad H \vdash \text{fn}(\langle \tau_1 \rightarrow \tau_2, C_f, E_f \rangle : \tau_1 \rightarrow \tau_2) \\
H \vdash E_f : e \quad \langle \epsilon, \tau_1.\tau_1 \rightarrow \tau_2.e, \epsilon, \epsilon \rangle \vdash C_f \rightsquigarrow \langle \tau_2, \tau_1.\tau_1 \rightarrow \tau_2.e, \varphi' \rangle \quad \varphi' \subseteq \varphi_f \\
\quad \quad \quad \quad H \vdash \text{fn}(\langle \tau_1 \rightarrow \tau_2, C_f, E_f \rangle : \tau_1 \rightarrow \tau_2) \\
H \vdash \text{fn}(\langle \tau_1 \rightarrow \tau_2, C_f, E_f \rangle : \tau_1 \rightarrow \tau_2) \\
H \vdash E_f : e \quad \alpha_i, \rho_i \notin \text{FTV}(e) \\
\quad \quad \quad \quad H \vdash \Lambda(\alpha, \rho).\text{fn}(\langle \tau_1 \rightarrow \tau_2, C_f, E_f \rangle : \forall(\alpha, \rho).\tau_1 \rightarrow \tau_2) \\
H \vdash \text{fn}(\langle \tau_1 \rightarrow \tau_2, C_f, E_f \rangle : \tau_1 \rightarrow \tau_2) \\
H \vdash E_f : e \quad \alpha_i, \rho_i \notin \text{FTV}(e) \\
\quad \quad \quad \quad H \vdash \Lambda(\alpha, \rho).\text{fn}(\langle \tau_1 \rightarrow \tau_2, C_f, E_f \rangle : \forall(\alpha, \rho).\tau_1 \rightarrow \tau_2)
\]

Figure 3: Typing Rules for etySECK (1)
a machine configuration consists of typing a stack, an environment, a continuation, and a sequence of commands:

\[
H \models S : s_1 \quad H \models E : e_1 \quad H \models K : k_1 \quad (s_1, e_1, k_1, \varphi) \vdash C \leadsto (s_o, e_o, \varphi_o)
\]

\[
(S, H, E, C, K, \varphi) \leadsto (s_o, e_o, \varphi_o).
\]

The judgements \(H \models S : s, H \models E : e, H \models K : k\) mean that for a given heap \(H\), a stack \(S\), an environment \(E\), and a continuation \(K\) have types \(s, e, \) and \(k\), respectively. The type of a stack is a list of the types whose values are in the stack:

\[
H \models v : \tau \quad H \models S : s
\]

\[
H \models v.S : \tau.s.
\]

Types of an environment and a continuation are also lists of the types whose values are in those components.

The judgement \(H \models v : \tau\) means that for a given heap \(H\) a value \(v\) has a type \(\tau\). Since the type of a location \(l_{\rho}\) is determined by the value stored in the heap, we need \(H\) for the value typing:

\[
l_{\rho} \in \text{Dom}(H) \quad H \models H(l_{\rho}) : \tau
\]

\[
H \models l_{\rho} : \text{ref}_{\rho} \tau.
\]

A location value \(l_{\rho}\) has a type \(\text{ref}_{\rho} \tau\) if a value stored in the location has a type \(\tau\). We assume that there is a function \(\text{type}\) of which is a mapping from primitive values to their types. The type of a function value is determined by checking whether the annotated type is correct:

\[
H \models E_f : e \quad (e, \tau_1, e, e, e) \vdash C_f \leadsto (\tau_2, \tau_1, e, \varphi'_f) \quad \varphi'_f \subseteq \varphi_f
\]

\[
H \models \text{fn}(\tau_1 \xrightarrow{\varphi_f} \tau_2, C_f, E_f) : \tau_1 \xrightarrow{\varphi_f} \tau_2.
\]

Note that the actual effects of executing the function’s body may be less than the annotated effects.

We define the type of a sequence of commands as follows:

\[
\langle s, e, k, \varphi \rangle \vdash C \leadsto \langle s_o, e_o, \varphi_o \rangle.
\]

This judgement means that if a given machine configuration which has a stack, an environment, a continuation of types \(s, e, k\), respectively, effects \(\varphi\), and a sequence of commands \(C\) continues its execution to a final configuration, then the types of the stack, the environment, and the effects of the final configuration are \(s_o, e_o, \varphi_o\), respectively. Figure 4 defines the typing rules.

If the sequence of commands is empty, there are two possible typing rules. Rule \text{nil} describes that if both the commands and the continuation are empty, this configuration is a final state. Rule \text{nilk} states that if only the commands are empty but the continuation is not empty, the machine replaces the environment and the commands by the first element of the continuation and keeps going on:

\[
\langle s, e', k, \varphi \rangle \vdash C' \leadsto \langle s_o, e_o, \varphi_o \rangle
\]

\[
\langle s, e, (e', C'), k, \varphi \rangle \vdash \text{nil} \leadsto \langle s_o, e_o, \varphi_o \rangle \quad \text{nilk}.
\]

Each typing rule for commands is an abstraction of the corresponding machine transition rule. The \text{frame} moves the stack top type into the environment and the \text{deframe} removes the
\[ \begin{align*}
\langle s, e, k, \varphi \rangle & \vdash C \leadsto \langle s_o, e_o, \varphi_o \rangle \\
\langle s, e, \varphi \rangle & \vdash \text{nil} \leadsto \langle s, e, \varphi \rangle & \text{nil} \\
\langle s, e, \varphi \rangle & \vdash \text{frame} \leadsto \langle s, e, \varphi \rangle & \text{frame} \\
\langle r, s, e, k, \varphi \rangle & \vdash \text{framer} \leadsto \langle s_o, e_o, \varphi_o \rangle & \text{deframe} \\
\langle r, s, e, k, \varphi \rangle & \vdash \text{fetch} \leadsto \langle s_o, e_o, \varphi_o \rangle & \text{fetch} \\
\tau & = \text{typeof}(p) & \tau = \text{typeof}(p) \\
\langle s, e, k, \varphi \rangle & \vdash \text{quote}(p).C \leadsto \langle s_o, e_o, \varphi_o \rangle & \text{quote} \\
\langle s, e, k, \varphi \rangle & \vdash \text{add} \leadsto \langle s, e, k, \varphi \rangle & \text{add} \\
\langle s, e, k, \varphi \rangle & \vdash \text{cond} \leadsto \langle s, e, k, \varphi \rangle & \text{cond} \\
\langle s, e, k, \varphi \rangle & \vdash \text{fn} \leadsto \langle s, e, k, \varphi \rangle & \text{fn} \\
\langle s, e, k, \varphi \rangle & \vdash \text{rfn} \leadsto \langle s, e, k, \varphi \rangle & \text{rfn} \\
\langle s, e, k, \varphi \rangle & \vdash \text{app} \leadsto \langle s, e, k, \varphi \rangle & \text{app} \\
\langle s, e, k, \varphi \rangle & \vdash \text{tapp} \leadsto \langle s, e, k, \varphi \rangle & \text{tapp} \\
\langle s, e, k, \varphi \rangle & \vdash \text{ref} \leadsto \langle s, e, k, \varphi \rangle & \text{ref} \\
\langle s, e, k, \varphi \rangle & \vdash \text{get} \leadsto \langle s, e, k, \varphi \rangle & \text{get} \\
\langle s, e, k, \varphi \rangle & \vdash \text{set} \leadsto \langle s, e, k, \varphi \rangle & \text{set}
\end{align*} \]

Figure 4: Typing Rules for etySECK (2)
type on the top of the environment:

\[
\begin{align*}
\langle s, \tau, e, k, \varphi \rangle &\vdash C \rightsquigarrow \langle s_0, e_0, \varphi_0 \rangle \quad \text{frame} \\
\langle \tau, s, e, k, \varphi \rangle &\vdash \text{frame}.C \rightsquigarrow \langle s_0, e_0, \varphi_0 \rangle \quad \text{frame} \\
\langle s, \tau, e, k, \varphi \rangle &\vdash \text{deframe}.C \rightsquigarrow \langle s_0, e_0, \varphi_0 \rangle \quad \text{deframe}.
\end{align*}
\]

Typing rules for functions, \( \text{fn} \) and \( \text{rfn} \), check if the annotated types are correct and put polymorphic types on the stack. The following rule describes the typing of non-recursive functions:

\[
\begin{align*}
\langle e, \tau_1, e, e, \rangle &\vdash C_f \rightsquigarrow \langle \tau_2, \tau_1, e, \varphi_f \rangle \\
\langle \forall (\alpha, \rho).\tau_1 \rightarrow \tau_2 \rangle, s, e, k, \varphi &\vdash C \rightsquigarrow \langle s_0, e_0, \varphi_0 \rangle
\end{align*}
\]

\( \text{fn} \)

Rule \( \text{app} \) does not distinguish applications of non-recursive functions and recursive functions:

\[
\begin{align*}
\langle \tau_2, s, e, k, \varphi \rangle &\vdash C \rightsquigarrow \langle s_0, e_0, \varphi_0 \rangle \quad \text{app} \\
\langle \tau_1, \tau_1 \rightarrow \tau_2, s, e, k, \varphi \rangle &\vdash \text{app}.C \rightsquigarrow \langle s_0, e_0, \varphi_0 \rangle \quad .
\end{align*}
\]

Rule \( \text{tapp} \) also does not distinguish them and instantiates polymorphic types to monomorphic ones:

\[
\begin{align*}
\langle \tau_1, \alpha_i, r_i, r_i \rangle \tau, s, e, k, \varphi &\vdash C \rightsquigarrow \langle s_0, e_0, \varphi_0 \rangle \\
\langle \forall (\alpha, \rho).\tau_1 \rightarrow \tau_2 \rangle, s, e, k, \varphi &\vdash \text{tapp}(\tau, \tau).C \rightsquigarrow \langle s_0, e_0, \varphi_0 \rangle
\end{align*}
\]

Rules \( \text{ref}, \text{get}, \) and \( \text{set} \) introduce corresponding memory effects. For example, rule \( \text{ref} \) introduces a \( \{ \text{init}(\rho) \} \) effect:

\[
\begin{align*}
\langle \text{ref}_\rho \tau, s, e, k, \varphi \cup \{ \text{init}(\rho) \} \rangle &\vdash C \rightsquigarrow \langle s_0, e_0, \varphi_0 \rangle \\
\tau, s, e, k, \varphi &\vdash \text{ref}(\rho).C \rightsquigarrow \langle s_0, e_0, \varphi_0 \rangle \quad \text{ref}.
\end{align*}
\]

2 Soundness

In this section, we prove the soundness of our typing rules by following the standard formalism of Wright and Felleisen [6]. Before presenting the main theorem, we need a few properties of our typing rules.

The first property is that the typing judgement \( \langle s, e, k, \varphi \rangle \vdash C \rightsquigarrow \langle s_0, e_0, \varphi_0 \rangle \) can be instantiated to a more concrete judgement by substituting the free variables occurring in the judgement. Lemma 1 formalizes this property.

**Lemma 1 (Substitution)** If \( \langle s, e, k, \varphi \rangle \vdash C \rightsquigarrow \langle s_0, e_0, \varphi_0 \rangle \), then for any substitution \( S \), 
\( \langle Ss, Sc, Sk, S\varphi \rangle \vdash SC \rightsquigarrow \langle Ss_0, Sc_0, Sk_0, S\varphi_0 \rangle \).

**Proof** We prove by the case analysis of \( \langle s, e, k, \varphi \rangle \vdash C \rightsquigarrow \langle s_0, e_0, \varphi_0 \rangle \). Since most cases are obvious, we prove just two nontrivial cases and omit the rest of the proofs.

- **case** \( C = \text{fn}(\forall (\alpha, \rho).\tau_1 \rightarrow \tau_2).C' \)

Suppose \( \langle s, e, k, \varphi \rangle \vdash \text{fn}(\forall (\alpha, \rho).\tau_1 \rightarrow \tau_2).C' \rightsquigarrow \langle s_0, e_0, \varphi_0 \rangle \). Then by \( \text{fn} \),

\[
\langle e, \tau_1, e, e, \rangle \vdash C_f \rightsquigarrow \langle \tau_2, \tau_1, e, \varphi_f \rangle \varphi_f \subseteq \varphi_f \quad \alpha_i, \rho_i \not\in \text{FTV}(e) (1)
\]
\[(\forall(a,\bar{p},r_1)_{\tau_1}.(\exists\bar{r},\tilde{\tau}_2).s,e,k,\varphi) \vdash C' \sim \langle s_o,e_o,\varphi_o \rangle.\]  
(2)

Let \( S' = S[\beta_i/\alpha_i, \eta_i/\rho_i] \) for new \( \beta_i, \eta_i. \) Since \( \alpha_i, \rho_i \notin \text{FTV}(e), S'e = S e. \) Then by I.H., (1) implies

\[\langle e, (S'\tau_1), S e, e, e \rangle \vdash S' C_f \sim \langle S'\tau_2, (S'\tau_1), S e, S'\varphi' \rangle \]
\[S'\varphi' \subseteq S'\varphi \beta_i, \eta_i \notin \text{FTV}(S e).\]  
(3)

By I.H. and applying \( S \) to (2),

\[\langle S(\forall(a,\bar{p},r_1)_{\tau_1}.(\exists\bar{r},\tilde{\tau}_2)).s,e,Sk,S\varphi \rangle \vdash SC' \sim \langle S s_o, S e_o, S \varphi_o \rangle.\]

Since \( S(\forall(a,\bar{p},\tilde{r}_1)_{\tilde{\tau}_2}) \equiv (\forall(\tilde{\beta}, \tilde{\eta}).S'_{\tau_1} \supseteq S'_{\tau_2}, S e, Sk, S\varphi \rangle \vdash SC' \sim \langle S s_o, S e_o, S \varphi_o \rangle.\]  
(4)

Now, by applying \( \text{fn} \) to (3) and (4),

\[\langle S s, S e, Sk, S\varphi \rangle \vdash \text{fn}(\forall(\tilde{\beta}, \tilde{\eta}).S'_{\tau_1} \supseteq S'_{\tau_2}), SC' \sim \langle S s_o, S e_o, S \varphi_o \rangle\]

and this means

\[\langle S s, S e, Sk, S\varphi \rangle \vdash S(\text{fn}(\forall(a,\bar{p},r_1)_{\tau_1} : \tau_1 \supseteq \tilde{\tau}_2), C') \sim \langle S s_o, S e_o, S \varphi_o \rangle.\]

\textbf{case} \( C = \text{tapp}(\bar{r}, \tilde{r}).C' \)

Suppose \( \langle (\forall(a,\bar{p},r_1)_{\tau_1}).s,e,k,\varphi \rangle \vdash \text{tapp}(\bar{r}, \tilde{r}).C' \sim \langle s_o,e_o,\varphi_o \rangle \) then for new \( \beta_i, \eta_i,\)

\[\langle (\forall(\tilde{\beta}, \tilde{\eta}).[\beta_i/\alpha_i, \eta_i/\rho_i])_{\tau_1}).s,e,k,\varphi \rangle \vdash \text{tapp}(\bar{r}, \tilde{r}).C' \sim \langle s_o,e_o,\varphi_o \rangle.\]

Then by \( \text{tapp},\)

\[\langle [r_i/\beta_i, e_i/\eta_i][[\beta_i/\alpha_i, \eta_i/\rho_i]]_{\tau_1}).s,e,k,\varphi \rangle \vdash C' \sim \langle s_o,e_o,\varphi_o \rangle.\]  
(5)

By I.H. and applying \( S \) to (5),

\[\langle S([r_i/\beta_i, e_i/\eta_i][[\beta_i/\alpha_i, \eta_i/\rho_i]]_{\tau_1}).s,e,Sk,S\varphi \rangle \vdash SC' \sim \langle S s_o, S e_o, S \varphi_o \rangle.\]

Since \( \beta_i, \eta_i \)’s are new type variables,

\[S([r_i/\beta_i, e_i/\eta_i][[\beta_i/\alpha_i, \eta_i/\rho_i]]_{\tau_1}).s \equiv \]
\[[S(r_i)]/\beta_i, (S r_i)]/\eta_i[[S[\beta_i/\alpha_i, \eta_i/\rho_i]]_{\tau_1}).s.\]

Thus we have

\[\langle ([S r_i)]/\beta_i, (S r_i)]/\eta_i[[S[\beta_i/\alpha_i, \eta_i/\rho_i]]_{\tau_1}).s, e, S e, Sk, S\varphi \rangle \vdash SC' \sim \langle S s_o, S e_o, S \varphi_o \rangle.\]  
(6)

By applying \( \text{tapp} \) to (6),

\[\langle (\forall(\tilde{\beta}, \tilde{\eta}).[\beta_i/\alpha_i, \eta_i/\rho_i])_{\tilde{\tau}}).s, e, S e, Sk, S\varphi \rangle \vdash \text{tapp}(S \bar{r}, \bar{\tilde{r}}).SC' \sim \langle S s_o, S e_o, S \varphi_o \rangle\]

and this implies

\[\langle (S(\forall(a,\bar{p},r_1)_{\tau_1}).s, e, Sk, S\varphi \rangle \vdash S(\text{tapp}(\bar{r}, \tilde{r}).SC') \sim \langle S s_o, S e_o, S \varphi_o \rangle.\]
Lemma 2 states that the quantified type variables or region variables can be instantiated to arbitrary types or regions, respectively.

**Lemma 2 (Instantiation)** If \( H \models \Lambda(\overline{a}, \overline{p}).\text{fn}(\tau_1 \mapsto \tau_2, C_f, E_f) : \forall (\overline{a}, \overline{p}).\tau_1 \mapsto \tau_2 \), then \( H \models [\tau_i/\alpha_i, r_i/\rho_i] \text{fn}(\tau_1 \mapsto \tau_2, C_f, E_f) : [\tau_i/\alpha_i, r_i/\rho_i] (\tau_1 \mapsto \tau_2) \).

**Proof** By \( H \models \Lambda(\overline{a}, \overline{p}).\text{fn}(\tau_1 \mapsto \tau_2, C_f, E_f) : \forall (\overline{a}, \overline{p}).\tau_1 \mapsto \tau_2 \), we have \( H \models E_f : e \) and \( H \models \text{fn}(\tau_1 \mapsto \tau_2, C_f, E_f) : \tau_1 \mapsto \tau_2 \) for any \( \alpha_i, \rho_i \not\in \text{FTV}(e) \); that is,

\[
\langle e, r_i/e, \epsilon, \epsilon \rangle \vdash C_f \leadsto \langle r_2, r_1/e, \phi_f \rangle \quad \phi_f \subseteq \phi_f.
\]

(7)

Let \( S = [\tau_i/\alpha_i, r_i/\rho_i] \). Since \( \alpha_i, \rho_i \not\in \text{FTV}(e) \), we have \( \text{Se} = e \) and \( H \models SE_f : e \). By (7) and Lemma 1, \( (e, S(\tau_1/e), \epsilon, \epsilon) \vdash SC_f \leadsto (S_{\tau_2}, S(\tau_1/e), S\phi_f) \) and \( S\phi_f \subseteq S\phi_f \).

Since \( H \models \text{fn}(S(\tau_1 \mapsto \tau_2), SC_f, SE_f) : S(\tau_1 \mapsto \tau_2), H \models \text{fn}(\tau_1 \mapsto \tau_2, C_f, E_f) : S(\tau_1 \mapsto \tau_2) \). \( \square \)

Lemma 3 states that any extension of the stack bottom does not affect the typability of a given machine configuration and Lemma 4 states the monotonicity of effects. Since the proofs of Lemma 3 and 4 are trivial we omit them.

**Lemma 3 (Stack Bottom Extensibility)** If \( \langle s, e, k, \phi \rangle \vdash C \leadsto \langle s_o, e_0, \phi_0 \rangle \), then for any \( s' \), \( \langle s', e, k, \phi \rangle \vdash C \leadsto \langle s_o, e_0, \phi_0 \rangle \).

**Lemma 4 (Effect monotonicity)**

1. If \( \langle s, e, k, \phi \rangle \vdash C \leadsto \langle s_o, e_0, \phi_0, \phi' \rangle \), then for any \( \phi' \), \( \langle s, e, k, \phi \cup \phi' \rangle \vdash C \leadsto \langle s_o, e_0, \phi_0 \cup \phi' \rangle \).
2. If \( \langle s, e, k, \phi \rangle \vdash C \leadsto \langle s_o, e_0, \phi_0 \rangle \) and \( \phi' \subseteq \phi \), then \( \langle s, e, k, \phi' \rangle \vdash C \leadsto \langle s_o, e_0, \phi_0' \rangle \) and \( \phi'_o \subseteq \phi_0 \).

Lemma 5 describes the typing of commands under the configuration with a non-nil continuation: after typing the current code, our typing mechanism gets a pair of an environment and commands from the continuation and keeps typing the new commands under the new environment.

**Lemma 5 (Continuation)**

If \( \langle s_1, e_1, \epsilon, \phi_1 \rangle \vdash C_1 \leadsto \langle s_2, e', \varphi_2 \rangle \) and \( \langle s_2, s_3, e_2, k_2, \varphi_2 \cup \varphi' \rangle \vdash C_2 \leadsto \langle s, e, \varphi \rangle \), then \( \langle s_1, s_3, e_1, (e_2, C_2).k_2, \varphi_1 \cup \varphi' \rangle \vdash C_1 \leadsto \langle s, e, \varphi \rangle \).

**Proof** We prove by induction on the length of \( C_1 \).

- **case** \( C_1 = \text{nil} \)

  By assumption, \( s_2 = s_1, e' = e_1, \varphi_2 = \varphi_1 \), and \( \langle s_2, s_3, e_2, k_2, \varphi_2 \cup \varphi' \rangle \vdash C_2 \leadsto \langle s, e, \varphi \rangle \).

  Then by nilk,

  \[
  \langle s_1, s_3, (e_2, C_2).k_2, \varphi_1 \cup \varphi' \rangle \vdash \text{nil} \leadsto \langle s, e, \varphi \rangle.
  \]

- **case** \( C_1 = \text{frame}.C' \)

  By assumption, \( s_1 = \tau.s'_1 \) for some \( s'_1 \) such that

  \[
  \langle \tau.s'_1, e_1, \epsilon, \varphi_1 \rangle \vdash \text{frame}.C' \leadsto \langle s_2, e', \varphi_2 \rangle
  \]

  (8)

  Then by (8) and frame,

  \[
  \langle s_1.s'_1, \tau.e_1, \epsilon, \varphi_1 \rangle \vdash C' \leadsto \langle s_2, e', \varphi_2 \rangle
  \]

  (10)
and by (10), (9), and I.H.,
\[ \langle s'_1.s_3, \tau.e, (e_2, C_2), k_2, \varphi_1 \cup \varphi' \rangle \vdash C' \leadsto (s, e, \varphi). \quad (11) \]

Thus by (11) and frame,
\[ \langle \tau.s'_1.s_3, e, (e_2, C_2), k_2, \varphi_1 \cup \varphi' \rangle \vdash \text{frame}.C' \leadsto (s, e, \varphi). \]

• case \( C_1 = \text{deframe}.C' \)

By assumption, \( e_1 = \tau.e' \) such that
\[ \langle s_1, \tau.e', \epsilon, \varphi_1 \rangle \vdash \text{deframe}.C' \leadsto (s_2, e', \varphi_2) \quad (12) \]

\[ (s_2.s_3, e_2, k_2, \varphi_2 \cup \varphi') \vdash C_2 \leadsto (s, e, \varphi). \quad (13) \]

Then by (12) and deframe,
\[ \langle s_1, e'_1, \epsilon, \varphi_1 \rangle \vdash C' \leadsto (s_2, e', \varphi_2) \quad (14) \]

and by (13), (14), and I.H.,
\[ \langle s_1.s_3, e'_1, (e_2, C_2), k_2, \varphi' \cup \varphi_1 \rangle \vdash C' \leadsto (s, e, \varphi). \quad (15) \]

Thus by (15) and deframe,
\[ \langle s_1.s_3, \tau.e'_1, (e_2, C_2), k_2, \varphi' \cup \varphi_1 \rangle \vdash \text{deframe}.C' \leadsto (s, e, \varphi). \]

• case \( C_1 = \text{fetch}(n).C' \)

By assumption, \( e_1 = \tau_0, \cdots \tau_n.e'_1 \) such that
\[ \langle s_1, \tau_0, \cdots \tau_n.e', \epsilon, \varphi_1 \rangle \vdash \text{fetch}(n).C' \leadsto (s_2, e', \varphi_2) \quad (16) \]

\[ (s_2.s_3, e_2, k_2, \varphi_2 \cup \varphi') \vdash C_2 \leadsto (s, e, \varphi). \quad (17) \]

Then by (16) and fetch,
\[ \langle \tau_n.s_1.\tau_0, \cdots \tau_n.e'_1, \epsilon, \varphi_1 \rangle \vdash C' \leadsto (s_2, e', \varphi_2) \quad (18) \]

and by (18), (17), and I.H.,
\[ \langle \tau_n.s_1.s_3, \tau_0, \cdots \tau_n.e'_1, (e_2, C_2), k_2, \varphi' \cup \varphi_1 \rangle \vdash C' \leadsto (s, e, \varphi). \quad (19) \]

Thus by (19) and fetch,
\[ \langle \tau_n.s_1.s_3, \tau_0, \cdots \tau_n.e'_1, (e_2, C_2), k_2, \varphi' \cup \varphi_1 \rangle \vdash \text{fetch}(n).C' \leadsto (s, e, \varphi). \]

• case \( C_1 = \text{quote}(p).C' \)

Let \( \tau = \text{type.of}(p) \) then by assumption,
\[ \langle s_1, e_1, \epsilon, \varphi_1 \rangle \vdash \text{quote}(p).C' \leadsto (s_2, e', \varphi_2) \quad (20) \]
\((s_2.s_3.e_2,k_2,\varphi_2 \cup \varphi') \vdash C_2 \leadsto \langle s, e, \varphi \rangle\).

(21)

Then by (20) and quote,

\(\langle \tau.s_1.e_1,\epsilon,\varphi_1 \rangle \vdash C' \leadsto \langle s_2,e',\varphi_2 \rangle\)

(22)

and by (22), (21), and I.H.,

\(\langle \tau.s_1.s_3.e_1, (e_2,C_2).k_2, \varphi' \cup \varphi_1 \rangle \vdash C' \leadsto \langle s,e,\varphi \rangle\).

(23)

Thus by (23) and quote,

\(\langle s_1.s_3.e_1, (e_2,C_2).k_2, \varphi' \cup \varphi_1 \rangle \vdash \text{quote}(p).C' \leadsto \langle s,e,\varphi \rangle\).

- **case** \(C_1 = \text{add}.C'\)

By assumption, \(s_1 = i.i.s'_1\) such that

\(\langle i.i.s'_1.e_1,\epsilon,\varphi_1 \rangle \vdash \text{add}.C' \leadsto \langle s_2,e',\varphi_2 \rangle\)

(24)

\((s_2.s_3.e_2,k_2,\varphi_2 \cup \varphi') \vdash C_2 \leadsto \langle s,e,\varphi \rangle\).

(25)

Then by (24) and add,

\(\langle i.s'_1.e_1,\epsilon,\varphi_1 \rangle \vdash C' \leadsto \langle s_2,e',\varphi_2 \rangle\)

(26)

and by (26), (25), and I.H.,

\(\langle i.s'_1.s_3.e_1, (e_2,C_2).k_2, \varphi' \cup \varphi_1 \rangle \vdash C' \leadsto \langle s,e,\varphi \rangle\).

(27)

Thus by (27) and add,

\(\langle i.i.s'_1.s_3.e_1, (e_2,C_2).k_2, \varphi' \cup \varphi_1 \rangle \vdash \text{add}.C' \leadsto \langle s,e,\varphi \rangle\).

- **case** \(C_1 = \text{cond}(C_t,C_f).C'\)

By assumption, \(s_1 = b.s'_1\) such that

\(\langle b.s'_1.e_1,\epsilon,\varphi_1 \rangle \vdash \text{cond}(C_t,C_f).C' \leadsto \langle s_2,e',\varphi_2 \rangle\)

(28)

\((s_2.s_3.e_2,k_2,\varphi_2 \cup \varphi') \vdash C_2 \leadsto \langle s,e,\varphi \rangle\).

(29)

Then by (28) and cond,

\(\langle s'_1,e_1,\epsilon,\epsilon \rangle \vdash C_t \leadsto \langle s_b,e_b,\varphi_{b_1} \rangle\)

\(\langle s'_1,e_1,\epsilon,\epsilon \rangle \vdash C_f \leadsto \langle s_b,e_b,\varphi_{b_2} \rangle\)

(30)

\((s_b,e_b,\epsilon,\varphi_{b_1} \cup \varphi_{b_2} \cup \varphi_1) \vdash C' \leadsto \langle s_2,e',\varphi_2 \rangle\).

(31)

and by applying Lemma 3 to (30),

\(\langle s'_1.s_3,e_1,\epsilon,\epsilon \rangle \vdash C_t \leadsto \langle s_b.s_3,e_b,\varphi_{b_1} \rangle\)

\(\langle s'_1.s_3,e_1,\epsilon,\epsilon \rangle \vdash C_f \leadsto \langle s_b.s_3,e_b,\varphi_{b_2} \rangle\).

(32)

By applying I.H. to (31) and (29),

\((s_b.s_3,e_b, (e_2,C_2).k_2, \varphi_1 \cup \varphi_{b_2} \cup \varphi_1 \cup \varphi') \vdash C' \leadsto \langle s,e,\varphi \rangle\)

(33)

and by (32), (33), and cond,

\(\langle b.s'_1.s_3.e_1, (e_2,C_2).k_2, \varphi_1 \cup \varphi' \rangle \vdash \text{cond}(C_t,C_f).C' \leadsto \langle s,e,\varphi \rangle\).
July 13, 2000

• case \( C_1 = \text{fn}(\forall \bar{\alpha}, \bar{p}, C_f : \tau_1 \rightarrow^f \tau_2), C' \)

By assumption,
\[
\langle s_1, e_1, e, \varphi_1 \rangle \vdash \text{fn}(\forall \bar{\alpha}, \bar{p}, C_f : \tau_1 \rightarrow^f \tau_2), C' \leadsto (s_2, e', \varphi_2) (34)
\]
\[
\langle s_2, s_3, e_2, k_2, \varphi_2 \cup \varphi' \rangle \vdash C_2 \leadsto (s, e, \varphi). (35)
\]

Then by (34) and fn,
\[
\langle e, \tau_1, e_1, e, e \rangle \vdash C_f \leadsto \langle \tau_2, \tau_1, e_1, \varphi'_f \rangle \quad \varphi'_f \subseteq \varphi_f \quad \alpha_i, \rho_i \notin \text{FTV}(e_1) (36)
\]
\[
\langle (\forall (\bar{\alpha}, \bar{p}). \tau_1 \rightarrow^f \tau_2), s_1, s_1, e_1, (e_2, C_2), k_2, \varphi' \cup \varphi_1 \rangle \vdash C' \leadsto (s_2, e', \varphi_2) (37)
\]

and by (37), (35), and I.H.,
\[
\langle (\forall (\bar{\alpha}, \bar{p}). \tau_1 \rightarrow^f \tau_2), s_1, s_3, e_1, (e_2, C_2), k_2, \varphi' \cup \varphi_1 \rangle \vdash C' \leadsto (s, e, \varphi). (38)
\]

Thus by (38), (36), and fn,
\[
\langle s_1, s_3, e_1, (e_2, C_2), k_2, \varphi' \cup \varphi_1 \rangle \vdash \text{fn}(\forall \bar{\alpha}, \bar{p}, C_f : \tau_1 \rightarrow^f \tau_2), C' \leadsto (s, e, \varphi). (39)
\]

• case \( C_1 = \text{rfn}(\forall \bar{\alpha}, \bar{p}, C_f : \tau_1 \rightarrow^f \tau_2), C' \)

The proof of this case is same for the case \( C_1 = \text{fn}(\forall \bar{\alpha}, \bar{p}, C_f : \tau_1 \rightarrow^f \tau_2), C' \).

• case \( C_1 = \text{app}.C' \)

By assumption, \( s_1 = \tau_1 \rightarrow^f \tau_2, s'_1 \) such that
\[
\langle \tau_1 \rightarrow^f \tau_2, s'_1, e_1, e, \varphi_1 \rangle \vdash \text{app}.C' \leadsto (s_2, e', \varphi_2) (39)
\]
\[
\langle s_2, s_3, e_2, k_2, \varphi_2 \cup \varphi' \rangle \vdash C_2 \leadsto (s, e, \varphi). (40)
\]

Then by (39) and app,
\[
\langle \tau_2, s'_1, e_1, e, \varphi_1 \cup \varphi_f \rangle \vdash C' \leadsto (s_2, e', \varphi_2) (41)
\]

and by (40), (41), and I.H.,
\[
\langle \tau_2, s'_1, s_3, e_1, (e_2, C_2), k_2, \varphi' \cup \varphi_1 \cup \varphi_f \rangle \vdash C' \leadsto (s, e, \varphi). (42)
\]

Thus by (42) and app,
\[
\langle \tau_1 \rightarrow^f \tau_2, s'_1, s_3, e_1, (e_2, C_2), k_2, \varphi' \cup \varphi_1 \rangle \vdash \text{app}.C' \leadsto (s, e, \varphi). (43)
\]

• case \( C_1 = \text{tapp}(\bar{\tau}, \bar{r}).C' \)

By assumption, \( s_1 = (\forall (\bar{\alpha}, \bar{p}). \tau), s'_1 \) such that
\[
\langle (\forall (\bar{\alpha}, \bar{p}). \tau), s'_1, e_1, e, \varphi_1 \rangle \vdash \text{tapp}(\bar{\tau}, \bar{r}).C' \leadsto (s_2, e', \varphi_2) (43)
\]
\[
(s_2, s_3, e_2, k_2, \varphi_2 \cup \varphi') \vdash C_2 \rightsquigarrow (s, e, \varphi).
\]  

(44)

Let \( S = [\tau_i/\alpha_i, \tau_i/\rho_i] \) then by \texttt{tapp},
\[
(S \tau, s_3', e_1, e, \varphi_1) \vdash \varphi \rightsquigarrow (s_2, e', \varphi_2)
\]  

(45)

and by (44), (45), and I.H.,
\[
(S \tau, s_3', s_3, e_1, (e_2, C_2), k_2, \varphi' \cup \varphi_1) \vdash C' \rightsquigarrow (s, e, \varphi).
\]  

(46)

Thus by (46) and \texttt{tapp},
\[
\langle (\forall (\alpha, \rho). \tau). s_3', s_3, e_1, (e_2, C_2), k_2, \varphi' \cup \varphi_1 \rangle \vdash \texttt{tapp}(\tau, \tau). C' \rightsquigarrow (s, e, \varphi).
\]

• case \( C_1 = \text{ref}(\rho). C' \)

By assumption, \( s_1 = \tau.s_3' \) such that
\[
\langle \tau, s_3', e_1, e, \varphi_1 \rangle \vdash \text{ref}(\rho). C' \rightsquigarrow (s_2, e', \varphi_2)
\]  

(47)

\[
(s_2, s_3, e_2, k_2, \varphi_2 \cup \varphi') \vdash C_2 \rightsquigarrow (s, e, \varphi).
\]  

(48)

Then by (47) and \texttt{ref},
\[
\langle \text{ref}(\rho). \tau, s_3', e_1, e, \varphi_1 \cup \{\text{init}(\rho)\} \rangle \vdash C' \rightsquigarrow (s_2, e', \varphi_2)
\]  

(49)

and by (48), (49), and I.H.,
\[
\langle \text{ref}(\rho). \tau, s_3', s_3, e_1, (e_2, C_2), k_2, \varphi' \cup \varphi_1 \cup \{\text{init}(\rho)\} \rangle \vdash C_2 \rightsquigarrow (s, e, \varphi).
\]  

(50)

Thus by (50) and \texttt{ref},
\[
\langle \tau, s_3', s_3, e_1, (e_2, C_2), k_2, \varphi' \cup \varphi_1 \rangle \vdash \text{ref}(\rho). C_1 \rightsquigarrow (s, e, \varphi).
\]

• case \( C_1 = \text{get}. C' \)

By assumption, \( s_1 = \text{ref}(\rho). s_3' \) such that
\[
\langle \text{ref}(\rho). \tau, s_3', e_1, e, \varphi_1 \rangle \vdash \text{get}. C' \rightsquigarrow (s_2, e', \varphi_2)
\]  

(51)

\[
(s_2, s_3, e_2, k_2, \varphi_2 \cup \varphi') \vdash C_2 \rightsquigarrow (s, e, \varphi).
\]  

(52)

Then by (51) and \texttt{get},
\[
\langle \text{ref}(\rho). \tau, s_3', e_1, e, \varphi_1 \cup \{\text{read}(\rho)\} \rangle \vdash C' \rightsquigarrow (s_2, e', \varphi_2)
\]  

(53)

and by (52), (53), and I.H.,
\[
\langle \text{ref}(\rho). \tau, s_3', s_3, e_1, (e_2, C_2), k_2, \varphi' \cup \varphi_1 \cup \{\text{read}(\rho)\} \rangle \vdash C_2 \rightsquigarrow (s, e, \varphi).
\]  

(54)

Thus by (54) and \texttt{get},
\[
\langle \text{ref}(\rho). \tau, s_3', s_3, e_1, (e_2, C_2), k_2, \varphi' \cup \varphi_1 \rangle \vdash \text{get}. C' \rightsquigarrow (s, e, \varphi).
July 13, 2000

ROPAS-2000-6

Theorem 6 states that if a machine configuration has a type, its next configuration, if any, will have the same type. Since effects annotated on types may not occur, effects can be reduced after one step transition.

Lemma 6 (Subject Reduction) If \((S, H, E, C, K, \varphi)~\leadsto~(s_o, e_o, \varphi_o)\) and \((S, H, E, C, K, \varphi)\Rightarrow M\), then \(M~\leadsto~(s_o, e_o, \varphi_o)\) and \(\varphi_o\subseteq\varphi_o\).

Proof We prove by the case analysis of the transition relation(\(\Rightarrow\)).

- **Case** \((S, H, E, \text{nil}, (E', C), K, \varphi)\Rightarrow(S, H, E', C, K, \varphi)\)
  
  By assumption, there exist \(s, e, k\), and \(e'\) such that \(H \models s : s, \ H \models E : e, \ H \models K : k, \ H \models E' : e', \ H \models (E', C).K : (e', C).k,\) and

  \[
  \langle s, e, (e', C), k, \varphi \rangle \Rightarrow \text{nil} \leadsto \langle s_o, e_o, \varphi_o \rangle.
  \]

  Then by nilk, \(\langle s, e', k, \varphi \rangle \Rightarrow C \leadsto \langle s_o, e_o, \varphi_o \rangle\) and \((S, H, E', C, K, \varphi) \leadsto (s_o, e_o, \varphi_o)\).

- **Case** \((v, S, H, E, \text{frame}, C, K, \varphi)\Rightarrow(S, H, v, E, C, K, \varphi)\)
  
  Since by assumption \((v, S, H, E, \text{frame}, C, K, \varphi) \leadsto (s_o, e_o, \varphi_o)\), there exist \(\tau, s, e\) and \(k\) such that \(H \models v : \tau, \ H \models S : s, \ H \models E : e, \ H \models K : k,\) and

  \[
  \langle \tau, s, e, k, \varphi \rangle \Rightarrow \text{frame} \leadsto \langle s_o, e_o, \varphi_o \rangle.
  \]

  By frame, \(\langle s, \tau, e, k, \varphi \rangle \Rightarrow C \leadsto \langle s_o, e_o, \varphi_o \rangle\). Since \(H \models v.E : \tau.e,\)

  \[(S, H, v, E, C, K, \varphi) \leadsto (s_o, e_o, \varphi_o)\) and \(\varphi_o \subseteq \varphi_o\).

- **Case** \((S, H, v, E, \text{deframe}, C, K, \varphi)\Rightarrow(S, H, E, C, K, \varphi)\)
  
  There exist \(s, e, \tau,\) and \(k\) such that \(H \models S : s, \ H \models E : e, \ H \models v : \tau, \ H \models K : k,\) and

  \[
  \langle s, \tau, e, k, \varphi \rangle \Rightarrow \text{deframe} \leadsto \langle s_o, e_o, \varphi_o \rangle,\] and then by deframe, \(\langle s, e, k, \varphi \rangle \Rightarrow C \leadsto \langle s_o, e_o, \varphi_o \rangle,\) and \((S, H, E, C, K, \varphi) \leadsto (s_o, e_o, \varphi_o)\).
• case \((S, H, v_0, \ldots, v_n, E, \text{fetch}(n))C, K, \varphi)\)\(\Rightarrow\)(\(v_n S, H, v_0, \ldots, v_n, E, C, K, \varphi\))

There exist \(s, e, \tau_i (i = 0, \ldots, n)\) and \(k\) such that \(H \models S : s, H \models E : e, H \models v_0 : \tau_0, \ldots, H \models v_n : \tau_n, H \models K : k\) and \((s, \tau_0, \ldots, \tau_n, e, k, \varphi) \models \text{fetch}(n)C \Rightarrow (s_0, e_0, \varphi_0)\). Then by \text{fetch},

\[
\langle \tau_n, s, \tau_0, \ldots, \tau_n, e, k, \varphi \rangle \models C \Rightarrow (s_0, e_0, \varphi_0)
\]

and

\[
(v_n S, H, E, C, K, \varphi) \Rightarrow (s_0, v_0, \varphi_0).
\]

• case \((S, H, E, \text{quote}(p))C, K, \varphi)\(\Rightarrow\)(\(p S, H, E, C, K, \varphi\))

There exist \(s, e, k\) such that \(H \models S : s, H \models E : e, H \models K : k\) and

\[
(s, e, k, \varphi) \vdash \text{quote}(p)C \Rightarrow (s_0, e_0, \varphi_0).
\]

Let \(\tau = \text{type_of}(p)\) then by \text{quote}, \((\tau, s, e, k, \varphi) \vdash C \Rightarrow (s_0, e_0, \varphi_0)\) and

\[
(p S, H, E, C, K, \varphi) \Rightarrow (s_0, e_0, \varphi_0).
\]

• case \((i_2, i_1 S, H, E, \text{add.C})C, K, \varphi)\(\Rightarrow\)(\(i_1 + i_2 S, H, E, C, K, \varphi\))

There exist \(s, e, k\) such that \(H \models S : s, H \models i_j : i (j = 1, 2), H \models E : e, H \models K : k\) and \((i, i_s, e, k, \varphi) \vdash \text{add.C} \Rightarrow (s_0, e_0, \varphi_0)\). Then by \text{add}, \((i, s, e, k, \varphi) \vdash C \Rightarrow (s_0, e_0, \varphi_0)\) and \(H \models i_1 + i_2 : i\). Thus,

\[
(i_1 + i_2 S, H, E, C, K, \varphi) \Rightarrow (s_0, e_0, \varphi_0).
\]

• case \((\text{true.S}, H, E, \text{cond}(C_1, C_2))C, K, \varphi)\(\Rightarrow\)(\(S, H, E, C_1 C, K, \varphi\))

By the same reasoning, there exist \(s, e, k\) such that \(H \models S : s, H \models E : e, H \models K : k\) and \((b, s, e, k, \varphi) \vdash \text{cond}(C_1, C_2)C \Rightarrow (s_0, e_0, \varphi_0)\). Then by \text{cond},

\[
(s, e, e, e) \vdash C_1 \Rightarrow (s_b, e_b, \varphi_b_1)
\]

\[
(s, e, e, e) \vdash C_2 \Rightarrow (s_b, e_b, \varphi_b_2)
\]

\[
(s_b, e_b, k, \varphi_b_1 \cup \varphi_b_2 \cup \varphi) \vdash C \Rightarrow (s_0, e_0, \varphi_0).
\]

By applying Lemma 4 to (60),

\[
(s_b, e_b, k, \varphi_b_1 \cup \varphi) \vdash C \Rightarrow (s_0, e_0, \varphi'_0)
\]

for some \(\varphi'_0\) and by applying Lemma 4 to (59),

\[
(s, e, k, \varphi) \vdash C_1 \Rightarrow (s_b, e_b, \varphi_b_1 \cup \varphi)\]

Thus by (62) and (61),

\[
(s, e, k, \varphi) \vdash C_1 \Rightarrow (s_0, e_0, \varphi'_0)\]

\[
\text{and } \varphi'_0 \subseteq \varphi_0.
\]

• case \((\text{false.S}, H, E, \text{cond}(C_1, C_2))C, K, \varphi)\(\Rightarrow\)(\(S, H, E, C_1 C, K, \varphi\))

The proof of this case is same for the case

\[
(\text{true.S}, H, E, \text{cond}(C_1, C_2))C, K, \varphi)\(\Rightarrow\)(\(S, H, E, C_1 C, K, \varphi\)).
\]
• case (S, H, E, fn(∀\vec{\alpha}, \vec{\rho}, C_f : \tau_1 \xrightarrow{\phi_f} \tau_2), C, K, \phi) \Rightarrow (\Lambda(\vec{\alpha}, \vec{\rho}).fn(\tau_1 \xrightarrow{\phi_f} \tau_2), C, E).S, H, E, C, K, \phi)

There exist s, e, and k such that H \models S : s, H \models E : e, H \models K : k and

\langle s, e, k, \phi \rangle \vdash fn(∀\vec{\alpha}, \vec{\rho}, C_f : \tau_1 \xrightarrow{\phi_f} \tau_2).C \rightsquigarrow (s_o, e_o, \phi_o).

Then by fn,

\langle e, \tau_1.e, e, e \rangle \vdash C_f \rightsquigarrow \langle \tau_2, \tau_1.e, \phi'_f \rangle \quad \phi'_f \subseteq \phi_f \quad \alpha_i, \beta_i \notin FTV(e) \tag{63}

\langle ∀(\vec{\alpha}, \vec{\rho}).\tau_1 \xrightarrow{\phi_f} \tau_2.s, e, k, \phi \rangle \vdash C \rightsquigarrow (s_o, e_o, \phi_o). \tag{64}

By (63) and H \models E : e,

H \models fn(\tau_1 \xrightarrow{\phi_f} \tau_2, C_f, E) : \tau_1 \xrightarrow{\phi_f} \tau_2 \tag{65}

and by (65), (63), and H \models E : e,

H \models \Lambda(\vec{\alpha}, \vec{\rho}).fn(\tau_1 \xrightarrow{\phi_f} \tau_2, C_f, E) : ∀(\vec{\alpha}, \vec{\rho}).\tau_1 \xrightarrow{\phi_f} \tau_2. \tag{66}

Thus by (64) and (66),

(\Lambda(\vec{\alpha}, \vec{\rho}).fn(\tau_1 \xrightarrow{\phi_f} \tau_2, C_f, E).S, H, E, C, K, \phi) \rightsquigarrow (s_o, e_o, \phi_o).

• case (S, H, E, rfn(∀\vec{\alpha}, \vec{\rho}, C_f : \tau_1 \xrightarrow{\phi_f} \tau_2), C, K, \phi) \Rightarrow (\Lambda(\vec{\alpha}, \vec{\rho}).rfn(\tau_1 \xrightarrow{\phi_f} \tau_2, C_f, E).S, H, E, C, K, \phi)

The proof of this case is same for the case whose code is fn(∀\vec{\alpha}, \vec{\rho}, C_f : \tau_1 \xrightarrow{\phi_f} \tau_2).C.

• case (v.fn(\tau_1 \xrightarrow{\phi_f} \tau_2, C_f, E_f).S, H, E, app.C, K, \phi) \Rightarrow (S, H, v.E_f, C_f, (E, C).K, \phi)

There exist s, e, and k such that H \models v : \tau_1, H \models fn(\tau_1 \xrightarrow{\phi_f} \tau_2, C_f, E_f) : \tau_1 \xrightarrow{\phi_f} \tau_2, H \models S : s, H \models E : e, H \models K : k and

\langle \tau_1, \tau_1 \xrightarrow{\phi_f} \tau_2.s, e, k, \phi \rangle \vdash app.C \rightsquigarrow (s_o, e_o, \phi_o). \tag{67}

Since H \models fn(\tau_1 \xrightarrow{\phi_f} \tau_2, C_f, E_f) : \tau_1 \xrightarrow{\phi_f} \tau_2, H \models E_f : e_f and

\langle e, \tau_1.e_f, e, e \rangle \vdash C_f \rightsquigarrow \langle \tau_2, \tau_1.e_f, \phi'_f \rangle \quad \phi'_f \subseteq \phi_f. \tag{68}

By (67) and app,

\langle \tau_2.s, e, k, \phi \cup \phi'_f \rangle \vdash C \rightsquigarrow (s_o, e_o, \phi_o) \tag{69}

and by (69) and Lemma 4,

\langle \tau_2.s, e, k, \phi \cup \phi'_f \rangle \vdash C \rightsquigarrow (s_o, e_o, \phi'_o) \quad \phi'_o \subseteq \phi_o \tag{70}

for some \phi'_o. By (68), (70), and Lemma 5,

\langle s, \tau_1.e_f, (e, C).k, \phi \rangle \vdash C_f \rightsquigarrow (s_o, e_o, \phi'_o)

which means

(S, H, v.E_f, C_f, (E, C).K, \phi) \rightsquigarrow (s_o, e_o, \phi'_o) \quad \phi'_o \subseteq \phi_o.
• case \((v.\text{rfn}(\tau_1^{\tilde{v}_1}t_2,C_f,E_f)).S,H,E,\text{app}.C,K,\varphi)\rightarrow
(S,H,v.\text{rfn}(\tau_1^{\tilde{v}_1}t_2,C_f,E_f).E_f,C_f,(E,C).K,\varphi)\)

The proof of this case is same for the case

\((v.\text{rfn}(\tau_1^{\tilde{v}_1}t_2,C_f,E_f)).S,H,E,\text{app}.C,K,\varphi)\rightarrow(S,H,v.E_f,C_f,(E,C).K,\varphi)\).

• case \((\Lambda(\vec{\alpha},\vec{\rho}).\text{rfn}(\tau_1^{\tilde{v}_1}t_2,C_f,E_f)).S,H,E,\text{tapp}(\vec{\tau},\vec{r}).C,K,\varphi)\rightarrow
([\tau_1/\alpha_1,r_1/\rho_1]\text{rfn}(\tau_1^{\tilde{v}_1}t_2,C_f,E_f)).S,H,E,C,K,\varphi)\)

There exist \(s,e,\) and \(k\) such that

\[H \models \Lambda(\vec{\alpha},\vec{\rho}).\text{fn}(\tau_1^{\tilde{v}_1}t_2,C_f,E_f) : \forall (\vec{\alpha},\vec{\rho}).\tau_1^{\tilde{v}_1}t_2, \quad (71)\]

\[H \models S : s, \quad H \models E : e, \quad H \models K : k\]

\[\langle \forall (\vec{\alpha},\vec{\rho}).\tau_1^{\tilde{v}_1}t_2,s,e,k,\varphi \rangle \vdash \text{tapp}(\vec{\tau},\vec{r}).C \leadsto \langle s_o,e_o,\varphi_o \rangle.\]  

Then by \text{tapp},

\[\langle [\tau_1/\alpha_1,r_1/\rho_1] (\tau_1^{\tilde{v}_1}t_2).s,e,k,\varphi \rangle \vdash C \leadsto \langle s_o,e_o,\varphi_o \rangle. \quad (72)\]

Let \(S \equiv [\tau_1/\alpha_1,r_1/\rho_1]\) then by (71) and Lemma 2,

\[H \models S \text{ fn}((\tau_1^{\tilde{v}_1}t_2),C_f,E_f) : S(\tau_1^{\tilde{v}_1}t_2). \quad (73)\]

Thus by (72) and (73),

\[\langle S \text{ fn}((\tau_1^{\tilde{v}_1}t_2),C_f,E_f).S,H,E,C,K,\varphi \rangle \leadsto \langle s_o,e_o,\varphi_o \rangle.\]

• case \((\Lambda(\vec{\alpha},\vec{\rho}).\text{rfn}(\tau_1^{\tilde{v}_1}t_2,C_f,E_f)).S,H,E,\text{tapp}(\vec{\tau},\vec{r}).C,K,\varphi)\rightarrow
([\tau_1/\alpha_1,r_1/\rho_1]\text{rfn}(\tau_1^{\tilde{v}_1}t_2,C_f,E_f)).S,H,E,C,K,\varphi)\)

The proof of this case is same for the case

\((\Lambda(\vec{\alpha},\vec{\rho}).\text{fn}(\tau_1^{\tilde{v}_1}t_2,C_f,E_f)).S,H,E,\text{tapp}(\vec{\tau},\vec{r}).C,K,\varphi)\rightarrow
([\tau_1/\alpha_1,r_1/\rho_1]\text{fn}(\tau_1^{\tilde{v}_1}t_2,C_f,E_f)).S,H,E,C,K,\varphi)\).

• case \((v.S,H,E,\text{ref}(\rho)).C,K,\varphi)\rightarrow(l_p.S,H[l_p \rightarrow v],E,C,K,\varphi \cup \{\text{init}(\rho)\})\quad l_p \notin \text{Dom}(H)\)

There exist \(s,e,\) and \(k\) such that \(H \models v : \tau, \quad H \models s : s, \quad H \models E : e, \quad H \models K : k\) and \(\langle \tau,s,e,k,\varphi \rangle \vdash \text{ref}(\rho).C \leadsto \langle s_o,e_o,\varphi_o \rangle.\) Then by \text{ref},

\[\langle \text{ref}_\rho \tau,s,e,k,\varphi \cup \{\text{init}(\rho)\} \rangle \vdash C \leadsto \langle s_o,e_o,\varphi_o \rangle. \quad (74)\]

Let \(H' = H[l_p \mapsto v].\) Since \(H'(l_p) = v\) and \(H \models v : \tau, \quad H' \models v : \tau.\) Therefore

\[H' \models l_p : \text{ref}_\rho \tau, H' \models S : s, H' \models E : e, H' \models K : k. \quad (75)\]

By (74) and (75),

\[\langle l_p.S,H[l_p \rightarrow v],E,C,K,\varphi \cup \{\text{init}(\rho)\} \rangle \leadsto \langle s_o,e_o,\varphi_o \rangle.\]
Lemma 7 (Progress) If \((S, H, E, C, K, \varphi) \sim (s_o, e_o, \varphi_o)\) then either \(C\) and \(K\) are nil or there exists \(M\) such that \((S, H, E, C, K, \varphi) \Rightarrow M\).

Proof If the configuration is typable, there exist \(s, e\) and \(k\) such that \(H \models s: s, H \models E: e, H \models K: k\) and \(\langle \tau, r, s, e, k, \varphi \rangle \vdash \text{set.C} \sim (s_o, e_o, \varphi_o)\). Since the code \(C\) is typable, there exists a rule last used during the typing of \(C\). If the used rule is nil then \(C\) and \(K\) are nil. Otherwise we prove by the case analysis of the typing rules in Figure 4 that there exists a machine transition rule for each last used typing rule. Since the proof is obvious we omit the details.

Now we prove the soundness theorem of our typing rules. Theorem 1 states that a machine configuration which has a type would not go wrong.

Theorem 1 (Soundness) Let \(C\) be a program. If \((\text{nil}, \phi, \text{nil}, C, \text{nil}, \phi) \sim (s_o, e_o, \varphi_o)\) then either the machine transition sequence from \((\text{nil}, \phi, \text{nil}, C, \text{nil}, \phi)\) is infinite or it stops with a final configuration \(M\) such that \(M \sim (s_o, e_o, \varphi'_o)\), and \(\varphi'_o \subseteq \varphi_o\).

Proof By Lemma 6 and 7.
\[ \lambda \text{initial.} \]
\[ \text{let counter = new initial in} \]
\[ \lambda \text{inc.} \ (\text{counter} := !\text{counter} + \text{inc}; \]
\[ !\text{counter}) \]

\[ \Rightarrow \] compiled to etySECK

\[ \text{fn } (\forall \rho . \]
\[ \text{fetch } (0); \text{ref } \rho; \text{frame;} \]
\[ \text{fn } (\text{fetch } (1); \text{fetch } (1); \text{get}; \text{fetch } (0); \]
\[ \text{add}; \text{set}; \text{frame}; \text{deframe}; \text{fetch } (1); \]
\[ \text{get} \]
\[ \quad : \text{int} \rightarrow \text{int}); \text{deframe} \]
\[ \quad : \text{int} \rightarrow \text{int}]) \]

\[ ; ; \]

\[ \Rightarrow \] the result of type checking

\[ (\forall \rho . \text{int} \rightarrow \text{int}, \text{read } \rho \rightarrow \text{int}, \text{write } \rho \rightarrow \text{int}, \text{nil, nil}) \]

\[ \Rightarrow \] the result of actual run

\[ (\forall \rho . \text{fn } (\text{fetch } (0); \text{ref } \rho; \text{frame;} \]
\[ \text{fn } (\text{fetch } (1); \text{fetch } (1); \text{get}; \text{fetch } (0); \]
\[ \text{add}; \text{set}; \text{frame}; \text{deframe}; \text{fetch } (1); \]
\[ \text{get} \]
\[ \quad : \text{int} \rightarrow \text{int}); \text{deframe} \]
\[ \quad : \text{int} \rightarrow \text{int})), \]
\[ \text{nil, nil} \]

Figure 5: Example program

3 Examples

Since our typing rules in Figure 3 and 4 are deterministic, implementation of a type checker is straightforward. Thus we omit a checking algorithm in this paper.

We present an example program from Talpin’s paper[5] and the type checking result of the program in Figure 5. The figure shows that our type checker verifies that annotated types and effects are valid and the result of the execution actually has the same type.

In order to show what happens to the code annotated with wrong types and effects, we present a modified code in Figure 6. The figure shows that since the return type of the outer function mismatches with the type the type checker expects, the type checker reports a type error.

Compilation from mini-ML to etySECK is trivial and we omit the detailed description of it. The key idea is to instantiate each polymorphic value when it is used by inserting a `tapp` instruction. In this case, we obtain the argument of the `tapp` instruction by unifying the
fn (∀ρ. 
    fetch (0); ref ρ; frame;
    fn (fetch (1); fetch (1); get; fetch (0); 
        add; set; frame; deframe; fetch (1); 
        get 
          read(ρ).write(ρ) 
            : int → int); deframe 
          init(ρ) (int → int))
  ;;
===> the result of type checking

Type Error: Function return type mismatch
  (nil, nil, nil, fn (∀ρ. 
      fetch (0); ref ρ; frame;
      fn (fetch (1); fetch (1); get; fetch (0); 
          add; set; frame; deframe; fetch (1); 
          get 
            read(ρ).write(ρ) 
              : int → int); deframe 
            init(ρ) (int → int)), nil)
DECLARED : int → int,
ACTUALLY : int → int,

Figure 6: Example program Modified

polymorphic type and the type of the value at that point.

References


