1 Typed $\lambda_{\mathbf{lvar}}$ calculus

Given a set D, let $\mathbb{D}$ be a 4-tuple $(D, \sqcup_D, \bot_D, \top_D)$, and let J be threshold sets, where $J \subseteq D$ such that $\forall d, d' \in J. d \sqcup d' = \top_D$.

1.1 Syntax

Types and environments

$$
\begin{array}{llll}
T, U & := & \mathbf{1} & \text{unit} \\
 & \mid & T \times U & \text{product} \\
 & \mid & T \to U & \lambda \text{ abstraction} \\
 & \mid & \mathcal{J} & \text{threshold sets} \\
 & \mid & \mathcal{D}^d & \text{elements } d \text{ in } D \text{ such that } \bigsqcup D^d = d \\
 & \mid & \mathcal{L}_{\mathcal{D}}^d & \text{locations (indexed by } d\text{) of elements in } \mathcal{D}^d
\end{array}
$$

$$
\begin{array}{llll}
\Gamma & := & \cdot & \text{empty environment} \\
 & \mid & x : T & \text{environment extension}
\end{array}
$$

Term-level syntax

terms L, M, N	$:=$	x	variables
	$\mid$	$l \mid J \mid d \mid \lambda x.M$	values
	$\mid$	K	constants
	$\mid$	$M\ N$	application
	$\mid$	$()$	unit introduction
	$\mid$	$\textbf{let}\ () = M\ \textbf{in}\ N$	unit elimination
	$\mid$	(V, W)	product introduction
	$\mid$	$\textbf{let}\ (x, y) = M\ \textbf{in}\ N$	product elimination
status bits B	$:=$	1	frozen LVar
	$\mid$	0	non-frozen LVar
state s	$:=$	$\langle B, d\rangle$	states
stores S	$:=$	$\cdot$	empty store
	$\mid$	$S, l \mapsto s$	store extension
	$\mid$	$\top$	top
configurations C	$:=$	$\langle M \mid S\rangle$	programs are composed of terms and store
	$\mid$	$\textbf{error}$	runtime crash
values V, W	$:=$	l	locations
	$\mid$	J	threshold set
	$\mid$	s	states
	$\mid$	$\lambda x.M$	λ abstraction
	$\mid$	$()$	unit
	$\mid$	(V, W)	product
constants K	$:=$	$\textbf{new}$	allocate new LVar
	$\mid$	$\textbf{freeze}$	freeze LVar
	$\mid$	$\textbf{get}$	read threshold from LVar
	$\mid$	$\textbf{put}$	add value to LVar
evaluation contexts E	$:=$	$\square$	
	$\mid$	$V\ E$	
	$\mid$	$E\ V$	
	$\mid$	(V, E)	
	$\mid$	(E, V)	
	$\mid$	$\textbf{let}\ () = E\ \textbf{in}\ M$	
	$\mid$	$\textbf{let}\ (x, y) = E\ \textbf{in}\ M$	

1.2 Type System

Definition 1. $\sqsubseteq_J (d) = \exists d' \in \mathcal{J}.d \sqsubseteq d'$

$$\textsc{T-Var}\quad \frac{}{x : T \vdash x : T}$$

$$\textsc{T-Lam}\quad \frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x.M : T \to U}$$

$$\textsc{T-App}\quad \frac{\Gamma \vdash M : T \to U \qquad \Gamma \vdash N : T}{\Gamma \vdash M\ N : U}$$

$$\textsc{T-Unit}\quad \frac{}{\cdot \vdash () : \mathbf{1}}$$

$$\textsc{T-LetUnit}\quad \frac{\Gamma \vdash M : \mathbf{1} \qquad \Gamma \vdash N : T}{\Gamma \vdash \mathbf{let}\ () = M\ \mathbf{in}\ N : T}$$

$$\textsc{T-Pair}\quad \frac{\Gamma \vdash M : T \qquad \Gamma \vdash N : U}{\Gamma \vdash (M, N) : T \times U}$$

$$\textsc{T-LetPair}\quad \frac{\Gamma \vdash M : T \times T' \qquad \Gamma, x : T, y : T' \vdash N : U}{\Gamma \vdash \mathbf{let}\ (x, y) = M\ \mathbf{in}\ N : U}$$

$$\textsc{T-New}\quad \frac{}{\Gamma \vdash \mathbf{new} : \mathcal{L}_{\mathcal{D}}^{\perp}}$$

$$\textsc{T-Freeze}\quad \frac{\Gamma \vdash M : \mathcal{L}_{\mathcal{D}}^{d}}{\Gamma \vdash \mathbf{freeze}\ M : \mathcal{D}^{d}}$$

$$\textsc{T-Get}\quad \frac{\Gamma \vdash M : \mathcal{L}_{\mathcal{D}}^{d} \qquad \Gamma \vdash N : \mathcal{J} \qquad \sqsubseteq_J (d)}{\Gamma \vdash \mathbf{get}\ M\ N : \mathcal{D}^{d}}$$

$$\textsc{T-Put}\quad \frac{\Gamma \vdash M : \mathcal{L}_{\mathcal{D}}^{d} \qquad \Gamma \vdash N : \mathcal{D}^{d'}}{\Gamma \vdash \mathbf{put}\ M\ N : 1}$$

1.3 Operational semantics

E-Lam	$\langle (\lambda x.M)\ V \mid S\rangle$	$\longrightarrow$	$\langle M\{V/x\} \mid S\rangle$
E-Unit	$\langle \textbf{let}\ () = ()\ \textbf{in}\ M \mid S\rangle$	$\longrightarrow$	$\langle M \mid S\rangle$
E-Pair	$\langle \textbf{let}\ (x, y) = (V, W)\ \textbf{in}\ M \mid S\rangle$	$\longrightarrow$	$\langle M\{V/x\}\{W/y\} \mid S\rangle$
E-New	$\langle \textbf{new} \mid S\rangle$	$\longrightarrow$	$\langle l \mid S, l \mapsto (0, \bot)\rangle$
E-Freeze	$\langle \textbf{freeze} \mid S, l \mapsto (b, d)\rangle$	$\longrightarrow$	$\langle d \mid S, l \mapsto (1, d)\rangle$

E-Put
$$\frac{d \sqcup d' \neq \top}{\langle \textbf{put}\ l\ d' \mid S, l \mapsto \langle 0,\ d\rangle\rangle \longrightarrow \langle () \mid S, l \mapsto \langle 0,\ d'\rangle\rangle}$$

E-Put-Err
$$\frac{d \sqcup d' = \top}{\langle \textbf{put}\ l\ d' \mid S, l \mapsto s\rangle \longrightarrow \textbf{error}}$$

E-Get
$$\frac{d' \in J \qquad d' \sqsubseteq d}{\langle \textbf{get}\ l\ J \mid S, l \mapsto \langle b,\ d\rangle\rangle \longrightarrow \langle d' \mid S, l \mapsto \langle b,\ d\rangle\rangle}$$

E-Pair'
$$\frac{\langle M \mid S\rangle \longrightarrow \langle M' \mid S'\rangle \qquad \langle N \mid S\rangle \longrightarrow \langle N' \mid S''\rangle}{\langle (M, N) \mid S\rangle \longrightarrow \langle (M', N') \mid S' \sqcup S''\rangle}$$

E-App
$$\frac{\langle M \mid S\rangle \longrightarrow \langle M' \mid S'\rangle \qquad \langle N \mid S\rangle \longrightarrow \langle N' \mid S''\rangle}{\langle M\ N \mid S\rangle \longrightarrow \langle M'\ N' \mid S' \sqcup S''\rangle}$$

E-Lift
$$\frac{\langle M \mid S\rangle \longrightarrow \langle N \mid S'\rangle}{\langle E[M] \mid S\rangle \longrightarrow_E \langle E[N] \mid S'\rangle}$$

1.4 Syntatic sugar

T-RunLVar
$$\frac{\Gamma \vdash M : \mathcal{L}^d_{\mathcal{D}} \to ()}{\Gamma \vdash \textbf{runLVar}\ M : \mathcal{D}^d}$$

E-RunLVar $\quad \textbf{runLVar}\ M \quad \coloneqq \quad (\lambda l.\textbf{let}\ () = M\ l\ \textbf{in freeze}\ l)\ \textbf{new}$

2 Metatheory of Typed $\lambda_{\textbf{lvar}}$ calculus

2.1 Translation to $\lambda_{\textbf{LVar}}$ from Typed $\lambda_{\textbf{lvar}}$

Definition 2. A translation is a function $\zeta : C \to \sigma$, such that:

Add partial-order rules for state s, where $s = (b, d)$.

- it should maintain the same number of steps in C when translated into σ;
- it should not introduce synchronisation.

$$
\begin{array}{lcll}
\zeta(\mathbf{error}) & = & \mathbf{error} & \\
\zeta(\langle \mathbf{get}\ l\ J \mid S\rangle) & = & \langle S\ ;\ \mathbf{get}\ l\ P\rangle & \text{where } p_1 \cong s \text{ and } P \cong J \\
\zeta(\langle \mathbf{put}\ l\ d' \mid S\rangle) & = & \langle S\ ;\ \mathbf{put}_i\ l\rangle & \text{where } u_{p_i} \coloneqq \lambda d_i.d \sqcup d_i \\
\zeta(\langle \mathbf{new} \mid S\rangle) & = & \langle S\ ;\ \mathbf{new}\rangle & \\
\zeta(\langle \mathbf{freeze}\ l \mid S\rangle) & = & \langle S\ ;\ \mathbf{freeze}\ l\rangle & \\
\zeta(\langle \lambda x.M \mid S\rangle) & = & \langle S\ ;\ \lambda x.e\rangle & \\
\zeta(\langle M\ N \mid S\rangle) & = & \langle S\ ;\ e\ e'\rangle & \\
\zeta(\langle () \mid S\rangle) & = & \langle S\ ;\ ()\rangle & \\
\zeta(\langle \mathbf{let}\ () = M\ \mathbf{in}\ N \mid S\rangle) & = & \langle S\ ;\ \lambda().e\rangle & \\
\zeta(\langle (M, N) \mid S\rangle) & = & \langle S\ ;\ (\lambda x.\lambda y.\lambda f.fxy)\ e\ e'\rangle & \\
\zeta(\langle \mathbf{let}\ (x, y) = M\ \mathbf{in}\ N \mid S\rangle) & = & \langle S\ ;\ e\ (\lambda x.\lambda y.e')\rangle & \\
\zeta(\langle M \mid S, l \mapsto (0, d)\rangle) & = & \langle S[l \mapsto (d, \mathtt{false})]\ ;\ e\rangle & \\
\zeta(\langle M \mid S, l \mapsto (1, d)\rangle) & = & \langle S[l \mapsto (d, \mathtt{true})]\ ;\ e\rangle &
\end{array}
$$

Lemma 1 (Translation, Typed $\lambda_{\text{lvar}} \rightsquigarrow\ \lambda_{\text{LVar}}$)**.**
For any translation ζ,

- *if $C \longrightarrow C'$ and $\sigma \hookrightarrow \sigma'$ and $\zeta(C) = \sigma$, then $\zeta(C') = \sigma'$;*
- *if $C \longrightarrow_E C'$ and $\sigma \mapsto \sigma'$ and $\zeta(C) = \sigma$, then $\zeta(C') = \sigma'$.*

Proof. By induction on the structure of C. All cases are straight-forward, except for the introduction and elimination of pairs.

Case. $C = \langle \mathbf{error} \mid S\rangle$, $\sigma = \langle S\ ;\ \mathbf{error}\rangle$.
C and σ cannot step. Hence, the translation is vacuously valid.

Case. $C = \langle \mathbf{get}\ l\ J \mid S\rangle$, $\sigma = \langle S\ ;\ \mathbf{get}\ l\ P\rangle$.
Given the operational semantics, C steps to $C' = \langle s' \mid S, l \mapsto (b, d)\rangle$. And given λ_{LVar}'s operational semantics, σ steps to $\sigma' = \langle S\ ;\ p_2\rangle$. Applying $\zeta(C')$, we get $\langle S\ ;\ p_2\rangle$. Typed λ_{lvar}maintains the same number of steps and there is no unwanted synchronisation introduced. Hence, this translation is valid.

Case. $C = \langle \mathbf{put}\ l\ d' \mid S\rangle$, $\sigma = \langle S\ ;\ \mathbf{put}_i\ l\rangle$.
Given the operational semantics, C can either error or take a step.

Sub-case. $C' = \langle s' \mid S, l \mapsto s'\rangle$
Given λ_{LVar}'s operational semantics, σ steps to $\sigma' = \langle S\ ;\ p_2\rangle$ if $d \sqcup d_i \neq \top$, which is exactly the same as applying ζ to C'. Typed λ_{lvar}maintains the same number of steps and there is no unwanted synchronisation introduced.

Sub-case. $C' = \textbf{error}$
Given λ_{LVar}'s operational semantics, σ steps to $\sigma' = \textbf{error}$ if $d \sqcup d_i = \top$, which is exactly the same as applying ζ to C'. Typed λ_{lvar}maintains the same number of steps and there is no unwanted synchronisation introduced.

Hence, this translation is valid.

Case. $C = \langle \textbf{new} \mid S \rangle$, $\sigma = \langle S \ ; \ \textbf{new} \rangle$
C' steps to $\langle l \mid S, l \mapsto (0, \bot) \rangle$ which is equivalent to $\langle S[l \mapsto (\bot, false)] \ ; \ l \rangle$, as showed in the last two cases of the proof. Typed λ_{lvar}maintains the same number of steps and there is no unwanted synchronisation introduced. Hence, this translation is valid.

Case. $C = \langle \textbf{freeze} \mid S \rangle$, $\sigma = \langle S \ ; \ \textbf{freeze} \rangle$
C' steps to $\langle d \mid S, l \mapsto (1, d) \rangle$ which is equivalent to $\langle S[l \mapsto (p, true)] \ ; \ p \rangle$, as showed in the last two cases of the proof. Typed λ_{lvar}maintains the same number of steps and there is no unwanted synchronisation introduced. Hence, this translation is valid.

Case. $C = \langle \lambda x.M \mid S \rangle$, $\sigma = \langle S \ ; \ \lambda x.e \rangle$
C and σ do not step since lambda abstractions are values. Also, C and σ are immediately equivalent up to α-equivalence.

Case. $\langle M \ N \mid S \rangle$, $\sigma = \langle S \ ; \ e \ e' \rangle$
The application case is simple, where expressions take one step each in parallel in both languages. Typed λ_{lvar}maintains the same number of steps and there is no unwanted synchronisation introduced. Hence, this translation is valid.

Case. $C = \langle () \mid S \rangle$, $\sigma = \langle S \ ; \ () \rangle$
C and σ do not step since unit is a value. Also, C and σ are immediately equivalent.

Case. $C = \langle \textbf{let} \ () = M \ \textbf{in} \ N \mid S \rangle$, $\sigma = \langle S \ ; \ (\lambda().e') \ e \rangle$
The λ_{LVar}calculus does not provide an elimination rule for unit, since it introduces an explicit synchronisation construct. However, such construct is easily defined by forcing e to evaluate before e' via the introduction and elimination a lambda abstraction. Kuper'15 informally uses a generalised version of this construct. Both C and σ steps to the outermost expression, N and e', respectively. Explicit unit elimination here is used to synchronise and necessary for `runLVars to work. No extra steps are taken in the translation, hence, this translation is valid.`

Case. $C = \langle (M, N) \mid S \rangle$, $\sigma = \langle S \ ; \ (\lambda x.\lambda y.\lambda f.fxy) \ e \ e' \rangle$
In Typed λ_{lvar}, pair components are evaluated in parallel and the next step will be blocked until both components are evaluated to a value. The λ_{LVar}calculus

does not provide pairs, therefore we encode them using lambda abstractions. In our encoding, a function takes two values and returns function that takes both values. According to the semantics of the λ_{LVar} calculus, when those two expressions are passed to a function, they are evaluated in parallel. Hence, our encoding does not introduce synchronisation, blocking the next step unnecessarily. The translation maintains the same number of steps and, hence, is valid.

Case. $C = \langle \textbf{let } (x, y) = M \textbf{ in } N \mid S\rangle$, $\sigma = \langle S \,;\, e' \; (\lambda x.\lambda y.e)\rangle$
In Typed λ_{lvar}, the elimination rule for pairs require that both components are values, and those are substituted within the next computation by using two fresh variables. Given that we encoded pairs as a function that takes a function with two arguments, we need to create said function in order to eliminate pairs. The eliminating function has to make the values within the pair available to the next computation, in this case e'. According to the λ_{LVar}calculus, parameters must be fully evaluated before being passed on to a lambda abstraction - fact easily verifiable since λ_{LVar} has call-by-value semantics. Therefore, our encoding does not introduce synchronisation, blocking the next step unnecessarily, and maintains the same number of steps. Hence, this translation is valid.

Add case for lifted environment contexts.

Case. $C = \langle M \mid S, l \mapsto (0, d)\rangle$, $\sigma = \langle S[l \mapsto (d, \texttt{false})] \,;\, e\rangle$
The encoding of LVar's writeability status in Typed λ_{lvar}uses 0 and 1, while in λ_{LVar}, they are encoded as regular booleans. Irregardless of the most common representation of booleans as numbers, the LVar should be initialised with one status and switch to a different one once frozen, which happens in both $\lambda_{\text{LVar}}$and Typed λ_{lvar}.

Case. $C = \langle M \mid S, l \mapsto (1, d)\rangle$, $\sigma = \langle S[l \mapsto (d, \texttt{true})] \,;\, e\rangle$
Follows an analogous argument as previous case.

□

2.2 Determinism

Given the translation of Typed λ_{lvar} into λ_{LVar} is valid, we can infer that Typed λ_{lvar} is quasi-deterministic as well. Here, we restate all proofs and definitions leading to the quasi-determinism proof as stated in Kuper'15.

Definition 3. Permutation

Definition 4. Permutation of an expression

Definition 5. Permutation of a store

Definition 6. Permutation of configurations

Lemma 2.
Permutability

Lemma 3.
Internal Determinism

Lemma 4.
Strong Confluence

Lemma 5.
Confluence

Theorem 1.
Quasi-Determinism

2.3 Type safety

Theorem 2.
Progress

Theorem 3.
Preservation

Corollary 1.
Type Safety

2.4 Fully deterministic programming with LVars

In this section, we prove that Typed λ_{lvar} *is deterministic, which is the main contribution of this work. We proceed by proving that all* **error** *states within the Typed* λ_{lvar}*calculus are not typeable given our type rules.*

Theorem 4.
Untypeable **error***s*

Corollary 2.
Full Determinism