BeePL: Correct-by-compilation kernel extensions

Swarn Priya, Tim Steenvoorden

October 7, 2024

1 Introduction

1.1 Motivation

2 Koka-mini Language

2.1 Syntax and semantics

$e \in \mathsf{Expr} ::=$	x	variable	$c \in \mathsf{Const} ::=$	bool	boolean
$\mid$	c	constant	$\mid$	z	integers
$\mid$	$\lambda\overline{x:T}^n.e$	abstraction	$\mid$	unit	unit
$\mid$	$e\ \overline{e}^n$	application			
$\mid$	$l:T$	mem-address	$b \in \mathsf{bool} :=$	true	
$\mid$	$\&(e:T)$	allocation	$\mid$	false	
$\mid$	$!(e:T)$	dereference	$v \in \mathsf{Value} :=$	c	constant
$\mid$	$e := e$	mem-update	$\mid$	$\lambda\overline{x:T}^n.e$	abstraction
$\mid$	$\mathsf{run}\ e$	isolation	$\mid$	$l:T$	mem-address
$\mid$	$\mathsf{hp}\ \Theta.e$	heap-binding			
$\mid$	$\mathsf{let}\ x := e\ \mathsf{in}\ e$	let-binding			
$\mid$	$\mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$	conditional			

Figure 1: Syntax of Koka-mini IR

$$\Theta := [l_1 \to v_1, ..., l_n \to v_n] \qquad \Pi := [x_1 \to v_1, ..., x_n \to v_n]$$

Figure 2: Memory and Virtual Map

Figure 1 presents the syntax of Koka-mini intermediate language. Expressions are built from variables, constants, abstraction, application, memory location, references, dereferences, assignment, memory isolation, heap binding, let binding and conditional. The semantic uses the judgment of the form $s, e \downarrow s', v$ representing that the evaluation of the expression e in the state s produces an updated state s' and a value v. A state is a pair of a memory Θ (mapping from location to values) and a valuation for variables

Π (mapping from variables to values). $\Pi(x)$ represents the value associated with the variable x in the virtual map Π and $\Theta[l]$ represents the value present as the location l in the heap Θ. The notion $[\_ \leftarrow \_]$ represents updating the memory or the virtual map. The projection s_Θ returns the heap and s_Π returns the virtual map.

Evaluation of a variable SVAR results to the value associated with the variable x in the virtual map Π for the variable x. Allocation rule SALLOC represents the allocation of location e (which is not yet reduced to a value). It produces the new state s', which is obtained during the evaluation of expression e. Allocation rule SALLOCV assigns the location l in the heap with the value v of type T, with the constraint that l needs to be a fresh location. $fresh(l)$ ensures that the location l is never assigned or being used in the heap Θ. It produces the new state s' which contains the updated heap Θ' and the old virtual map s_Π. Dereferencing a location is represented using two rules SDEREF and SDEREFV. SDEREF represents the evaluation of $!(e : T)$ where e has not reduced to a value, but takes step to e'. The rule SDEREFV looks for the value at location l in the heap Θ and reduces to the same value without updating the state.

The memory update is performed using the expression $e_1 := e_2$. The operational semantics consists of three rules. The rule SMEMU1 presents the scenario where the expression e_1 has not reduced to a location and takes further step to expression e_1' producing the updated state s'. Similarly, the rule SMEMU2 presents the scenario where the value to be assigned e_2 to location l has not yet reduced to a value. The rule SMEMUV presents the operational semantics of $l : T := v$, where the location l is updated with the new value v and it returns the updated state.

Evaluation of run hp $\Theta.e$ produces e representing the scenario of safely removing the state effect and proceeding with the expression. This captures the essence of the function whose body can produce stateful effect, but the function itself is a total function. It is also necessary to argue that the stateful effect observed during the computation of the body of the function does not escape and leaks the state. To isolate the state effect, we also need a notion of heap-bindings in our semantics. The expression hp $\Theta.e$ represents the heap-bindings with heap Θ and expression e representing the various operations like assignment and dereferencing. The operational semantics for function application $e\ \overline{e}'\ n$ is pretty-straightforward with the constraint of evaluating the arguments before calling the function. The operational semantics of let − binding and conditional is straightforward.

$$\textsf{SVAR}\ \frac{s_\Pi = \Pi \wedge \Pi(x) = v}{s, x \downarrow s, v} \qquad \textsf{SCONST}\ \frac{}{s, c \downarrow s, c}$$

$$\textsf{SLOC}\ \frac{}{s, l : T \downarrow s, l : T} \qquad \textsf{SALLOC}\ \frac{s, e \downarrow s,' e'}{s, \&(e : T) \downarrow s,' \&(e' : T)}$$

$$\textsf{SALLOCV}\ \frac{fresh(l) \wedge s_\Theta = \Theta \wedge \Theta[l \leftarrow v] = \Theta' \wedge s' = (\Theta', s_\Pi)}{s, \&(v : T) \downarrow s,' l : T}$$

$$\textsf{SDEREF}\ \frac{s, e \downarrow s,' e'}{s, !(e : T) \downarrow s,' !(e' : T)} \qquad \textsf{SDEREFV}\ \frac{s_\Theta = \Theta \wedge \Theta[l] = v}{s, !((l : T) : T) \downarrow s, v}$$

$$\textsf{SMEMU1}\ \frac{s, e_1 \downarrow s,' e_1'}{s, e_1 := e_2 \downarrow s,' e_1' := e_2} \qquad \textsf{SMEMU2}\ \frac{s, e_2 \downarrow s,' e_2'}{s, l : T := e_2 \downarrow s,' l : T := e_2'}$$

$$\textsf{SMEMUV}\ \frac{s_\Theta = \Theta \wedge \Theta[l \leftarrow v] = \Theta' \wedge s' = (\Theta', s_\Pi)}{s, l : T := v \downarrow s,' \textsf{unit}} \qquad \textsf{SRUN}\ \frac{}{s, \textsf{run hp } \Theta.e \downarrow s, e}$$

$$\textsf{SHEAP1}\ \frac{s, e \downarrow s,' e'}{s, \textsf{hp } \Theta.!(e : T) \downarrow s,' \textsf{hp } \Theta.!(e :')T} \qquad \textsf{SHEAP2}\ \frac{s_\Theta = \Theta \wedge \Theta[l] = v}{s, \textsf{hp } \Theta.!(l : T : T) \downarrow s, v}$$

$$\textsf{SHEAP3}\ \frac{s, e \downarrow s,' e''}{s, \textsf{hp } \Theta.e := e' \downarrow s,' \textsf{hp } \Theta.e'' := e'} \qquad \textsf{SHEAP4}\ \frac{s, e \downarrow s,' e'}{s, \textsf{hp } \Theta.l : T := e' \downarrow s,' \textsf{hp } \Theta.l : T := e'}$$

$$\textsf{SHEAP5}\ \frac{l \in domain(\Theta) \wedge \Theta[l \leftarrow v] = \Theta' \wedge s' = (\Theta', s_\Pi)}{s, \textsf{hp } \Theta.l : T := v \downarrow s,' v} \qquad \textsf{SHEAP6}\ \frac{s_\Theta = \Theta \wedge \Theta[l] = v}{s, \textsf{hp } \Theta.l : T \downarrow s, v}$$

$$\textsf{SAPP1}\ \frac{s, e_1 \downarrow s,' e_1'}{s, e_1\ \overline{e_2}^n \downarrow s,' e_1'\ \overline{e_2}^n} \qquad \textsf{SAPP2}\ \frac{s, e_2 \downarrow s,' e_2'}{s, v\ \overline{e_2}^n \downarrow s,' v\ \overline{e_2'}^n}$$

$$\textsf{SAPPV}\ \frac{n = n'}{s, (\lambda \overline{x : T}^n.e)\ \overline{v}^{n'} \downarrow s, [\overline{x : T}^n \leftarrow \overline{v}^{n'}]e}$$

$$\textsf{SLET}\ \frac{s, e \downarrow s,' e''}{s, \textsf{let } x := e \textsf{ in } e' \downarrow s,' \textsf{let } x := e'' \textsf{ in } e'} \qquad \textsf{SLETV}\ \frac{}{s, \textsf{let } x := v \textsf{ in } e' \downarrow s, [x \leftarrow v]e'}$$

$$\textsf{SCONDT}\ \frac{}{s, \textsf{if true then } e \textsf{ else } e' \downarrow s, e} \qquad \textsf{SCONDF}\ \frac{}{s, \textsf{if false then } e \textsf{ else } e' \downarrow s, e'}$$

$$\textsf{SCOND}\ \frac{s, e_b \downarrow s,' e_b'}{s, \textsf{if } e_b \textsf{ then } e \textsf{ else } e' \downarrow s,' \textsf{if } e_b' \textsf{ then } e \textsf{ else } e'}$$

Figure 3: Semantics for expressions.

2.2 Type system

$t \in \mathsf{T} ::=$	t_b	basic-type	$e_l \in \mathsf{effect_l} ::=$	exn	exception
$\mid$	$\overline{t}^n \to \mathsf{e_f}, t$	fun-type	$\mid$	div	divergence
$\mid$	$\mathsf{ref}(h, t_b)$	ref-type	$\mid$	$st < \xi >$	heap
$\mid$	$\overline{\mathsf{t}}^{\mathsf{n}}$	row-type			
			$e_f \in \mathsf{effect} :=$	$\phi_{\mathsf{e_f}}$	empty
$t_p \in \mathsf{T_p} :=$	tunit	unit	$\mid$	$\mathsf{e_l}$	single
$\mid$	tint	int	$\mid$	$\overline{\mathsf{e_f}}$	row effect
$\mid$	tbool	bool	$\mid$	μ	var
$t_b \in \mathsf{T_b} :=$	$\mathsf{t_p}$	prim-type			
$\mid$	$\overline{\mathsf{t_p}}^{\mathsf{n}}$	row-base-type			

Figure 4: Types of Koka-mini IR

$$\Gamma := [x_1 \to T_1, ..., x_n \to T_n] \qquad \Sigma := [l_1 \to T_1, ..., l_n \to T_n]$$

Figure 5: Type and store environment

Figure 4 presents the available types in the language. The type of the language does not only capture the type of the computations involved, but also captures the side-effect that can be produced during the computation.

2.3 Effects

To capture the notion of side-effects that might arise during a computation, a new type called effect is introduced. The basic effects represented as e_l in the Figure 4 can be exn, div and $st < \xi >$. When a function can throw an exception, it gets an exn effect. A non-terminating function gets div effect. The effect $st <¿$ represents an identifier that captures the stateful effects over the heap. For example, allocation or dereferencing a pointer issues a stateful effect as it performs operation related to heap. A function can also produce more than one effect, which is represented as row of effects $\overline{e_f}$. For example, the effect of a pure function is represented as $[\mathsf{exn}, \mathsf{div}]$. A total function does not produce any side-effect, which is represented by ϕ_{e_f}.

The types are made from basic and primitive types. The primitive types supported in the language are tunit, int and bool. The basic types are made from primitive types. The types T are made from basic types, function types, reference type and collection of types. The function type is represented as $\overline{t}^n \to \mathsf{e_f}, t$, where $\overline{t}$ represents the type of the arguments, n represents the number of arguments, $\mathsf{e_f}$ captures the effect and t represents the return type of the function.

$$\textsf{VAR}\ \frac{\Gamma(x) = T}{\Gamma, \Sigma \vdash x : T, \epsilon} \quad \textsf{BOOL}\ \frac{}{\Gamma, \Sigma \vdash b : \mathit{tbool}, \epsilon} \quad \textsf{INT}\ \frac{}{\Gamma, \Sigma \vdash i : \mathit{tint}, \epsilon}$$

$$\textsf{UNIT}\ \frac{}{\Gamma, \Sigma \vdash \mathsf{unit} : \mathit{tunit}, \epsilon} \quad \textsf{LOC}\ \frac{\Sigma(l) = T}{\Gamma, \Sigma \vdash l : T : \mathsf{ref}(h, T), \epsilon}$$

$$\textsf{ALLOC}\ \frac{\Gamma, \Sigma \vdash e : t_b, \epsilon}{\Gamma, \Sigma \vdash \&(e : t_b) : \mathsf{ref}(h, t_b), [st < h >; \epsilon]} \quad \textsf{DEREF}\ \frac{\Gamma, \Sigma \vdash e : \mathsf{ref}(h, t_b), \epsilon}{\Gamma, \Sigma \vdash !(e : t_b) : t_b, [st < h >; \epsilon]}$$

$$\textsf{MEMU}\ \frac{\Gamma, \Sigma \vdash e_1 : \mathsf{ref}(h, t_b), \epsilon \wedge \Gamma, \Sigma \vdash e_2 : t_b, \epsilon}{\Gamma, \Sigma \vdash e_1 := e_2 : \mathsf{tunit}, [st < h >; \epsilon]}$$

$$\textsf{RUN}\ \frac{\Gamma, \Sigma \vdash e : t, [st < h >; \epsilon] \wedge h \notin ftv(\Gamma, \Sigma, t, \epsilon)}{\Gamma, \Sigma \vdash \mathsf{run}\ e : t, \epsilon} \quad \textsf{HEAPB}\ \frac{\Gamma, \Sigma \vdash e : t, \epsilon}{\Gamma, \Sigma \vdash \mathsf{hp}\ \Theta.e : t, [st < h >; \epsilon]}$$

$$\textsf{LET}\ \frac{\Gamma, \Sigma \vdash e : t, \phi_{e_f} \wedge \Gamma, \Sigma \vdash e' : t', \epsilon}{\Gamma, x : t, \Sigma \vdash \mathsf{let}\ x := e\ \mathsf{in}\ e' : t', \epsilon}$$

$$\textsf{COND}\ \frac{\Gamma, \Sigma \vdash e_1 : \mathit{tbool}, \epsilon_1 \wedge \Gamma, x : t, \Sigma \vdash e_2 : t, \epsilon_2 \wedge \Gamma, \Sigma \vdash e_3 : t, \epsilon_2}{\Gamma, \Sigma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : t, [\epsilon_1; \epsilon_2]}$$

$$\textsf{ABS}\ \frac{\Gamma, \overline{x : T}^n, \Sigma \vdash e : t_2, \epsilon_2}{\Gamma, \Sigma \vdash \lambda \overline{x : T}^n.e : (\overline{T}^n \rightarrow \epsilon_2, t_2), \epsilon} \quad \textsf{APP}\ \frac{\Gamma, \Sigma \vdash e : (\overline{T}^n \rightarrow \epsilon, t), \epsilon \wedge \Gamma, \Sigma \vdash \overline{e'} : \overline{T}, \epsilon}{\Gamma, \Sigma \vdash e\ \overline{e'}^n : t, \epsilon}$$

Figure 6: Typing rules for expressions.

The typing rules are stated under the typing environment Γ and store typing environment Σ. Γ represents the typing environment that maps variables to their types. Since our language deals with memory, we need to take into account the types associated with the memory locations. Σ represents the store typing environment that maps locations to their types. The typing judgment is of the form $\Gamma, \Sigma \vdash e : T, e_f$, which states that in the typing environment Γ and store typing environment Σ, the expression e has the type T and effect e_f. The environment can be extended to add new pair of variables or locations and types. It is represented as $\Gamma, x : T$

The typing rule VAR for variable involves looking for the type of the variable x in the environment Γ. The derived effect for variable is chosen as any arbitrary effect ϵ as the evaluation of the variable does not produce any side-effect. The typing rules for constants are derived in the similar manner. The typing rule LOC derives the type $\mathsf{ref}(h, T)$ for the location l by looking at the type associated with location l in the store typing environment Σ. Whenever a location is created, it is initialised with some value of some type, which in turns determines the type of the location. The rule ALLOC presents the typing rule for allocation $\&(e : t_b)$. It ensures that the type associated with allocation is always a basic-type t_b (it avoids having a

situation of pointer containing another pointer). The effect type captures the stateful effect ($st < h >$) and also the effect that might arise during the computation of e. The typing rule DEREF for dereferencing is similar to allocation; it returns basic-type to ensure that the value store at location e is not a reference type. The typing rule MEMU presents the typing derivation for memory update $e_1 := e_2$. It ensures that the type of value assigned to the location e_1 is of basic-type and the location e_1 is of reference type. The rule RUN presents the typing rule for memory isolation. As we are aware of the fact that run e isolates the state and makes sure that the state h is not observed or modified from outside the expression. Hence the effect of expression e is $[st < h >; \epsilon]$, but the result effect is only ϵ. To safely eliminate the state effect, we need to ensure that h is not a free variable in the context [SP] Not sure what to write about to free variables. The rule HEAPB presents the typing rule for heap binding. The heap binding hp $\Theta.e$ get the stateful effect $[st < h >; \epsilon]$ and type of the expression e. The typing rule for let-binding and conditional expression is straight-forward.