Assignment 4: Effects: Control and Storage

15-312: Principles of Programming Languages (Fall 2023)

This assignment will familiarize you with *effects* that are widely present in programming language. *Effects*, or *side effects* are umbrella terms used to describe any computational phenomenon that is not fully captured in the return value of an expression. Common examples of effects includes *control effects*, present in its full form as *continuations*, and *storage evaluation* pertaining to mutation in some form of storage.

We will consider to extensions to the rich expression language that we have developed.

The first extension, called **KPCF**, introduces the explicit manipulation of continuations. For ease and concision of specification and implementation, we will use **KPCF**v, a presentation of **KPCF** that uses a modal separation to distinguish *values* and *computations*. Code you write in **KPCF** will be elaborated to **KPCF**v.

The second extension, called **MA**, adds constructs for storage effects. Again, we will work with a modally-separated presentation, for meta-theoretic and implementation purposes. This will help us to distinguish *expression* and *commands*. The latter are "instructions" that operates on the store.

Make sure to start early and to understand the statics and dynamics of the new languages. There will be plenty of code to write, so don't delay!

Submission

As usual, please submit the written part of this homework as a PDF file to Gradescope. To submit the implementation part, submit a zipfile to Gradescope. To create the zipfile, use the Makefile supplied in the handout. It will ensure that all the relevant files are handed in.

Reference Implementation

As always, we have included the solution to this assignment as a binary heap image. You can load it into SML/NJ by passing in the `@SMLload` flag. Your solutions should behave just like ours.

Typ	τ	::=	$\texttt{cont}(\tau)$	$\tau\ \texttt{cont}$	`cont[tau]`
			$\texttt{unit}$	$\texttt{unit}$	`unit`
			$\texttt{prod}(\tau_1\,;\tau_2)$	$\tau_1 \times \tau_2$	`tau1 * tau2`
			$\texttt{void}$	$\texttt{void}$	`void`
			$\texttt{sum}(\tau_1\,;\tau_2)$	$\tau_1 + \tau_2$	`tau1 + tau2`
			$\texttt{parr}(\tau_1\,;\tau_2)$	$\tau_1 \rightharpoonup \tau_2$	`tau1 -> tau2`
			$\texttt{nat}$	$\texttt{nat}$	`nat`
			$\alpha,\ \beta,\ \ldots$	$\alpha,\ \beta,\ \ldots$	`A`, `B`, ...
Exp	e	::=	x	x	`x`
			$\texttt{let}(e_1\,;x\,.\,e_2)$	$\texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2$	`let x = e1 in e2`
			$\texttt{letcc}[\tau](x\,.\,e)$	$\texttt{letcc}\ x\ \texttt{in}\ e$	`letcc[tau] x in e`
			$\texttt{throw}[\tau](e\,;e_1)$	$\texttt{throw}\ e_1\ \texttt{to}\ e$	`throw[tau](e, e1)`
			$\texttt{triv}$	$\langle\rangle$	`<>`
			$\texttt{pair}(e_1\,;e_2)$	$\langle e_1, e_2\rangle$	`<e1, e2>`
			$\texttt{split}(e\,;x_1,x_2\,.\,e')$	$\texttt{split}\ e\ \texttt{as}\ x_1 \otimes x_2\ \texttt{in}\ e'$	`split e is x1, x2 in e'`
			$\texttt{abort}[\tau](e)$	$\texttt{abort}(e)$	`case[tau] e {}`
			$\texttt{in}[\texttt{l}][\tau_1\,;\tau_2](e)$	$\texttt{l}\cdot e$	`L[tau1, tau2].e`
			$\texttt{in}[\texttt{r}][\tau_1\,;\tau_2](e)$	$\texttt{r}\cdot e$	`R[tau1, tau2].e`
			$\texttt{case}(e\,;x_1\,.\,e_1\,;x_2\,.\,e_2)$	$\texttt{case}\ e\ \{\texttt{l}\cdot x_1 \hookrightarrow e_1 \mid \texttt{r}\cdot x_2 \hookrightarrow e_2\}$	`case e { L.x1 => e1 | R.x2 => e2 }`
			$\texttt{fun}[\tau_1\,;\tau_2](f\,.\,x\,.\,e)$	$\texttt{fun}\ f(x{:}\tau_1){:}\tau_2\ \texttt{is}\ e$	`fun f (x : tau1) : tau2 is e`
			$\lambda[\tau](x\,.\,e)$	$\lambda\,(x:\tau)\,e$	`fn (x : tau) => e`
			$\texttt{ap}(e\,;e_1)$	$e(e_1)$	`e e1`
			$\texttt{z}$	$\texttt{z}$	`z`
			$\texttt{s}(e)$	$\texttt{s}(e)$	`s(e)`
			$\texttt{ifz}(e\,;e_0\,;x\,.\,e_1)$	$\texttt{ifz}(e\,;e_0\,;x\,.\,e_1)$	`ifz e { z => e0 | s(x)=> e1 }`

Figure 1.1: **KPCF** Grammar

1 KPCF

In this section, we will work with **KPCF**, an extension of **PCF** with continuations. To simplify the specification and implementation, though, we will consider **KPCFv**, a presentation of **KPCF** that uses a modal separation to distinguish *values* and *computations*. You will be able to write code in **KPCF** directly, though, which will be elaborated to **KPCFv**. There is no conceptual difference between continuations in **KPCFv** and **KPCF**.

1.1 KPCF

We give the grammar of **KPCF** in Figure 1.1.

Rather than give a statics and dynamics for **KPCF**, we will first elaborate to **KPCFv**, which we will henceforth define.

1.2 KPCFv

First, we define the syntax of **KPCFv** in Figure 1.2. As described, we draw a syntactic distinction between values v and computations e; this leads us to include a few new constructs:

- Expression $\mathtt{ret}(\, v \,)$ treats a value as a trivial computation.
- Type $\mathtt{comp}(\, \tau \,)$ describes computations of type τ.
- Value $\mathtt{comp}(\, e \,)$ suspends a computation.
- Expression $\mathtt{bind}(\, v \,;\, x \,.\, e \,)$ evaluates a suspended computation v, binding the resulting value to x in e.

We also include internal forms (i.e., forms which cannot be written by programmers), such as stacks k, reified stacks $\mathtt{cont}(\, k \,)$, and evaluation states s.

Observe that the types for **KPCFv** contain type variables. This will allow you to write down expressions in an upcoming subsection that do not depend on the choice of any concrete type. To account for type variables, an **KPCFv** term is type-checked under a context Δ of type variables. For the sake of simplicity, in this assignment we will fix our typing context Δ to be

$$\Delta_0 \triangleq \alpha, \beta, \gamma, \delta$$

and omit the Δ from judgments. In the code, the four type variables are `A`, `B`, `C`, and `D`.

1.3 Elaboration

As mentioned, we elaborate **KPCF** expressions (easier for humans to write code in) to **KPCFv** expressions (easier to specify and implement). We give a few cases of the translation $\overline{e}$ here, assuming the trivial translation $\overline{\tau}$:

$$
\begin{aligned}
\overline{x} &= x \\
\overline{\mathtt{let}(\, e_1 \,;\, x \,.\, e_2 \,)} &= \mathtt{bind}(\, \mathtt{comp}(\, e_1 \,) \,;\, x \,.\, e_2 \,) \\
\overline{\mathtt{letcc}[\, \tau \,](\, x \,.\, e \,)} &= \mathtt{letcc}[\, \overline{\tau} \,](\, x \,.\, \overline{e} \,) \\
\overline{\mathtt{throw}[\, \tau \,](\, e \,;\, e_1 \,)} &= \mathtt{bind}(\, \mathtt{comp}(\, \overline{e} \,) \,;\, x_e \,.\, \mathtt{bind}(\, \mathtt{comp}(\, \overline{e_1} \,) \,;\, x_{e_1} \,.\, \mathtt{throw}[\, \overline{\tau} \,](\, x_e \,;\, x_{e_1} \,)\,)\,) \\
\overline{\mathtt{triv}} &= \mathtt{ret}(\, \mathtt{triv} \,) \\
\overline{\mathtt{pair}(\, e_1 \,;\, e_2 \,)} &= \mathtt{bind}(\, \mathtt{comp}(\, \overline{e_1} \,) \,;\, x_{e_1} \,.\, \mathtt{bind}(\, \mathtt{comp}(\, \overline{e_2} \,) \,;\, x_{e_2} \,.\, \mathtt{ret}(\, \mathtt{pair}(\, x_{e_1} \,;\, x_{e_2} \,)\,)\,)\,) \\
\overline{\mathtt{split}(\, e \,;\, x_1, x_2 \,.\, e' \,)} &= \mathtt{bind}(\, \mathtt{comp}(\, \overline{e} \,) \,;\, x_e \,.\, \mathtt{split}(\, x_e \,;\, x_1, x_2 \,.\, \overline{e'} \,)\,)
\end{aligned}
$$

The remaining cases are similar and may be found in `kpcf/elaborator/elaborator.sml`. Observe that **KPCFv** requires us to sequentialize the dynamics such that the code now guarantees a particular evaluation order.

Typ	τ	$::=$	$\mathtt{comp}(\tau)$	$\tau\ \mathtt{comp}$	computation
			$\mathtt{cont}(\tau)$	$\tau\ \mathtt{cont}$	continuation
			$\mathtt{unit}$	$\mathtt{unit}$	unit
			$\mathtt{prod}(\tau_1\,;\tau_2)$	$\tau_1 \times \tau_2$	product
			$\mathtt{void}$	$\mathtt{void}$	void
			$\mathtt{sum}(\tau_1\,;\tau_2)$	$\tau_1 + \tau_2$	sum
			$\mathtt{parr}(\tau_1\,;\tau_2)$	$\tau_1 \rightharpoonup \tau_2$	partial function
			$\mathtt{nat}$	$\mathtt{nat}$	natural number
			$\alpha,\ \beta,\ \ldots$	$\alpha,\ \beta,\ \ldots$	type variable
Stack	k	$::=$		ϵ	empty stack
				$k\,;x\,.\,e$	stack frame
Value	v	$::=$	x	x	variable
			$\mathtt{comp}(e)$	$\mathtt{comp}(e)$	suspended computation
			$\mathtt{cont}(k)$	$\mathtt{cont}(k)$	reified stack
			$\mathtt{triv}$	$\langle\rangle$	unit value
			$\mathtt{pair}(v_1\,;v_2)$	$\langle v_1, v_2\rangle$	pair value
			$\mathtt{in}[\mathtt{l}][\tau_1\,;\tau_2](v)$	$\mathtt{l}\cdot v$	left injection
			$\mathtt{in}[\mathtt{r}][\tau_1\,;\tau_2](v)$	$\mathtt{r}\cdot v$	right injection
			$\mathtt{fun}[\tau_1\,;\tau_2](f\,.\,x\,.\,e)$	$\mathtt{fun}\,f(x{:}\tau_1){:}\tau_2\,\mathtt{is}\,e$	recursive function
			$\lambda[\tau](x\,.\,e)$	$\lambda\,(x:\tau)\,e$	non-recursive function
			$\mathtt{z}$	$\mathtt{z}$	zero
			$\mathtt{s}(v)$	$\mathtt{s}(v)$	successor
Exp	e	$::=$	$\mathtt{ret}(v)$	$\mathtt{ret}(v)$	trivial computation
			$\mathtt{bind}(v\,;x\,.\,e)$	$\mathtt{bind}\ x \leftarrow v\ \mathtt{in}\ e$	sequential evaluation
			$\mathtt{letcc}[\tau](x\,.\,e)$	$\mathtt{letcc}\,x\,\mathtt{in}\,e$	bind current continuation
			$\mathtt{throw}[\tau](v\,;v_1)$	$\mathtt{throw}\,v_1\,\mathtt{to}\,v$	throw to a continuation
			$\mathtt{split}(v\,;x_1,x_2\,.\,e')$	$\mathtt{split}\,v\,\mathtt{as}\,x_1 \otimes x_2\,\mathtt{in}\,e'$	pair split
			$\mathtt{abort}[\tau](v)$	$\mathtt{abort}(v)$	nullary case analysis
			$\mathtt{case}(v\,;x_1\,.\,e_1\,;x_2\,.\,e_2)$	$\mathtt{case}\,v\,\{\mathtt{l}\cdot x_1 \hookrightarrow e_1 \mid \mathtt{r}\cdot x_2 \hookrightarrow e_2\}$	binary case analysis
			$\mathtt{ap}(v\,;v_1)$	$v(v_1)$	function application
			$\mathtt{ifz}(v\,;e_0\,;x\,.\,e_1)$	$\mathtt{ifz}(v\,;e_0\,;x\,.\,e_1)$	zero test
State	s	$::=$		$k \triangleright e$	evaluating e for stack k
				$k \triangleleft v$	returning v to stack k

Figure 1.2: **KPCFv** Grammar

1.4 Statics

We will have a typing judgments for each of our syntactic forms.

$k \div \tau$	stack k accepts a value of type τ
$\Gamma \vdash v : \tau$	value v has type τ
$\Gamma \vdash e \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau$	expression e may evaluate to a value of τ
$s\ \mathsf{ok}$	state s is well-formed

$\boxed{k \div \tau}$

$$\frac{}{\epsilon \div \tau} \qquad \frac{x : \tau \vdash e \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau' \qquad k \div \tau'}{k \,;\, x \,.\, e \div \tau}$$

Notice that typing for e does *not* carry a context Γ with it. This is not a shorthand or a mistake: since we evaluate only closed terms, x should be the only free variable in e. Be sure to implement this correctly.

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

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma \vdash e \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau}{\Gamma \vdash \mathtt{comp}(\,e\,) : \tau\ \mathtt{comp}} \qquad \frac{k \div \tau}{\Gamma \vdash \mathtt{cont}(\,k\,) : \tau\ \mathtt{cont}}$$

$$\frac{}{\Gamma \vdash \mathtt{triv} : \mathtt{unit}} \qquad \frac{\Gamma \vdash v_1 : \tau_1 \qquad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash \mathtt{pair}(\,v_1 \,;\, v_2\,) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash v : \tau_1}{\Gamma \vdash \mathtt{in}[\,\mathtt{l}\,][\,\tau_1 \,;\, \tau_2\,](\,v\,) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash v : \tau_2}{\Gamma \vdash \mathtt{in}[\,\mathtt{r}\,][\,\tau_1 \,;\, \tau_2\,](\,v\,) : \tau_1 + \tau_2}$$

$$\frac{\Gamma, f : \tau_1 \rightharpoonup \tau_2, x : \tau_1 \vdash e \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau_2}{\Gamma \vdash \mathtt{fun}[\,\tau_1 \,;\, \tau_2\,](\,f \,.\, x \,.\, e\,) : \tau_1 \rightharpoonup \tau_2} \qquad \frac{\Gamma, x : \tau_1 \vdash e \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau_2}{\Gamma \vdash \lambda[\,\tau_1\,](\,x \,.\, e\,) : \tau_1 \rightharpoonup \tau_2}$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \qquad \frac{\Gamma \vdash v : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(\,v\,) : \mathtt{nat}}$$

$\boxed{\Gamma \vdash e \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau}$

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \texttt{ret}(\, v \,) \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau} \qquad \frac{\Gamma \vdash v : \tau_1\,\mathsf{comp} \qquad \Gamma, x : \tau_1 \vdash e \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau_2}{\Gamma \vdash \texttt{bind}(\, v \,;\, x \,.\, e \,) \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau_2}$$

$$\frac{\Gamma, x : \tau\,\texttt{cont} \vdash e \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau}{\Gamma \vdash \texttt{letcc}[\, \tau \,](\, x \,.\, e \,) \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau} \qquad \frac{\Gamma \vdash v : \tau\,\texttt{cont} \qquad \Gamma \vdash v_1 : \tau}{\Gamma \vdash \texttt{throw}[\, \rho \,](\, v \,;\, v_1 \,) \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \rho}$$

$$\frac{\Gamma \vdash v : \tau_1 \times \tau_2 \qquad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau}{\Gamma \vdash \texttt{split}(\, v \,;\, x_1, x_2 \,.\, e \,) \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau}$$

$$\frac{\Gamma \vdash v : \texttt{void}}{\Gamma \vdash \texttt{abort}[\, \tau \,](\, v \,) \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau} \qquad \frac{\Gamma \vdash v : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash e_1 \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau \qquad \Gamma, x_2 : \tau_2 \vdash e_2 \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau}{\Gamma \vdash \texttt{case}(\, v \,;\, x_1 \,.\, e_1 \,;\, x_2 \,.\, e_2 \,) \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau}$$

$$\frac{\Gamma \vdash v : \tau_1 \rightharpoonup \tau_2 \qquad \Gamma \vdash v_1 : \tau_1}{\Gamma \vdash \texttt{ap}(\, v \,;\, v_1 \,) \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau_2} \qquad \frac{\Gamma \vdash v : \texttt{nat} \qquad \Gamma \vdash e_0 \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau \qquad \Gamma, x : \texttt{nat} \vdash e_1 \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau}{\Gamma \vdash \texttt{ifz}(\, v \,;\, e_0 \,;\, x \,.\, e_1 \,) \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau}$$

Notice that $\texttt{ret}(v)$ is the only way to elevate a value into a computation.

$\boxed{s\ \mathsf{ok}}$

$$\frac{k \div \tau \qquad \cdot \vdash v : \tau}{k \triangleleft v\ \mathsf{ok}} \qquad \frac{k \div \tau \qquad \cdot \vdash e \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau}{k \triangleright e\ \mathsf{ok}}$$

Task 1.1 (20 pts). Implement the statics for **KPCFv** in `kpcf/language/statics.sml` according to `kpcf/language/statics.sig`.

Remark. The functions `inferTypeValue` and `inferTypeExp` infer (synthesize) a type. However, the function `checkStack` *takes in* a type and checks that the stack can accept values of that type. If an erroneous situation is encountered, you should raise `TypeError` as usual.

1.5 Dynamics

Evaluating a **KPCFv** term using **K** Machines starts in the initial state $\epsilon \triangleright e$. The evaluation terminates when it reaches a final state specified by the dynamics after taking a number of transition steps. The following judgments involved in describing the dynamics using **K** machines:

$s \longmapsto s$	evaluation state taking a step
s final	evaluation state is final

$$\frac{}{k \triangleright \mathtt{ret}(\, v \,) \longmapsto k \triangleleft v} \qquad \frac{}{k \triangleright \mathtt{bind}(\, \mathtt{comp}(\, e \,)\, ; x \,.\, e' \,) \longmapsto k \,;\, x \,.\, e' \triangleright e}$$

$$\frac{}{k \triangleright \mathtt{letcc}[\, \tau \,](\, x \,.\, e \,) \longmapsto k \triangleright \{\mathtt{cont}(\, k \,)/x\}e} \qquad \frac{}{k \triangleright \mathtt{throw}[\, \tau \,](\, \mathtt{cont}(\, k' \,)\, ; v \,) \longmapsto k' \triangleleft v}$$

$$\frac{}{k \triangleright \mathtt{split}(\, \langle v_1, v_2 \rangle \,;\, x_1, x_2 \,.\, e \,) \longmapsto k \triangleright \{v_1, v_2/x_1, x_2\}e}$$

$$\frac{}{k \triangleright \mathtt{case}(\, \mathtt{l} \cdot v \,;\, x \,.\, e_1 \,;\, x_2 \,.\, e_2 \,) \longmapsto k \triangleright \{v/x_1\}e_1}$$

$$\frac{}{k \triangleright \mathtt{case}(\, \mathtt{r} \cdot v \,;\, x_1 \,.\, e_1 \,;\, x_2 \,.\, e_2 \,) \longmapsto k \triangleright \{v/x_2\}e_2}$$

$$\frac{}{k \triangleright \mathtt{ap}(\, \mathtt{fun}[\, \tau_1 \,;\, \tau_2 \,](\, f \,.\, x \,.\, e \,)\, ;\, v_1 \,) \longmapsto k \triangleright \{\mathtt{fun}[\, \tau_1 \,;\, \tau_2 \,](\, f \,.\, x \,.\, e \,), v_1/f, x\}e}$$

$$\frac{}{k \triangleright \mathtt{ap}(\, \lambda[\, \tau \,](\, x \,.\, e \,)\, ;\, v_1 \,) \longmapsto k \triangleright \{v_1/x\}e}$$

$$\frac{}{k \triangleright \mathtt{ifz}(\, \mathtt{z} \,;\, e_0 \,;\, x \,.\, e_1 \,) \longmapsto k \triangleright e_0} \qquad \frac{}{k \triangleright \mathtt{ifz}(\, \mathtt{s}(\, v \,)\, ;\, e_0 \,;\, x \,.\, e_1 \,) \longmapsto k \triangleright \{v/x\}e_1}$$

$$\frac{}{\epsilon \triangleleft v \ \mathsf{final}} \qquad \frac{}{k \,;\, x \,.\, e \triangleleft v \longmapsto k \triangleright \{v/x\}e}$$

In the dynamics, $\mathtt{ret}(v)$ is the only expression that causes the state to change from evaluation mode to return mode. Also, $\mathtt{bind}(\, \mathtt{comp}(\, e \,)\, ; x \,.\, e' \,)$ is the only expression that causes a new stack frame to be generated.

Notice how simple defining dynamics are: you only need to take care for the elimination forms. Introduction forms are naturally taken care of through modal separation. You will also notice how modal separation dramatically simplifies your dynamics implementation.

1.5.1 The **K** Machine

Before we implement the **K** machine, let's run some small **KPCF**v examples on paper.

Consider the following expression e:

$$
\begin{aligned}
h &\triangleq \lambda[\texttt{nat} \to \texttt{nat}](\, f \,.\, \texttt{bind}(\, \texttt{comp}(\, \texttt{ap}(\, f \,;\, \texttt{z}\,)\,)\,;\, x \,.\, \texttt{bind}(\, \texttt{comp}(\, \texttt{ap}(\, f \,;\, x\,)\,)\,;\, y \,.\, \texttt{ret}(\, \texttt{s}(\, y\,)\,)\,)\,)\,) \\
g[k] &\triangleq \lambda[\texttt{nat}](\, n \,.\, \texttt{ifz}(\, n \,;\, \texttt{ret}(\, \overline{10}\,)\,;\, n' \,.\, \texttt{throw}[\texttt{nat}](\, k \,;\, n'\,)\,)\,) \\
e &\triangleq \texttt{letcc}[\texttt{nat}](\, k \,.\, \texttt{ap}(\, h \,;\, g[k]\,)\,)
\end{aligned}
$$

It calls higher-order function h on function g, which conditionally returns numeral 10 or throws to the top-level continuation.

We can evaluate e on the empty stack ϵ as follows:

$$
\begin{aligned}
& \epsilon \triangleright \texttt{letcc}[\texttt{nat}](\, k \,.\, \texttt{ap}(\, h \,;\, g[k]\,)\,) \\
\longmapsto\ & \epsilon \triangleright \texttt{ap}(\, h \,;\, g[\texttt{cont}(\, \epsilon\,)]\,) \\
\longmapsto\ & \epsilon \triangleright \texttt{bind}(\, \texttt{comp}(\, \texttt{ap}(\, g[\texttt{cont}(\, \epsilon\,)] \,;\, \texttt{z}\,)\,)\,;\, x \,.\, \texttt{bind}(\, \texttt{comp}(\, \texttt{ap}(\, g[\texttt{cont}(\, \epsilon\,)] \,;\, x\,)\,)\,;\, y \,.\, \texttt{ret}(\, \texttt{s}(\, y\,)\,)\,)\,) \\
\longmapsto\ & k_1 \triangleright \texttt{ap}(\, g[\texttt{cont}(\, \epsilon\,)] \,;\, \texttt{z}\,) \\
\longmapsto\ & k_1 \triangleright \texttt{ifz}(\, \texttt{z} \,;\, \texttt{ret}(\, \overline{10}\,)\,;\, n' \,.\, \texttt{throw}[\texttt{nat}](\, \texttt{cont}(\, \epsilon\,)\,;\, n'\,)\,) \\
\longmapsto\ & k_1 \triangleright \texttt{ret}(\, \overline{10}\,) \\
\longmapsto\ & k_1 \triangleleft \overline{10} \\
\longmapsto\ & \epsilon \triangleright \texttt{bind}(\, \texttt{comp}(\, \texttt{ap}(\, g[\texttt{cont}(\, \epsilon\,)] \,;\, \overline{10}\,)\,)\,;\, y \,.\, \texttt{ret}(\, \texttt{s}(\, y\,)\,)\,) \\
\longmapsto\ & \epsilon \,;\, y \,.\, \texttt{ret}(\, \texttt{s}(\, y\,)\,) \triangleright \texttt{ap}(\, g[\texttt{cont}(\, \epsilon\,)] \,;\, \overline{10}\,) \\
\longmapsto\ & \epsilon \,;\, y \,.\, \texttt{ret}(\, \texttt{s}(\, y\,)\,) \triangleright \texttt{ifz}(\, \overline{10} \,;\, \texttt{ret}(\, \overline{10}\,)\,;\, n' \,.\, \texttt{throw}[\texttt{nat}](\, \texttt{cont}(\, \epsilon\,)\,;\, n'\,)\,) \\
\longmapsto\ & \epsilon \,;\, y \,.\, \texttt{ret}(\, \texttt{s}(\, y\,)\,) \triangleright \texttt{throw}[\texttt{nat}](\, \texttt{cont}(\, \epsilon\,)\,;\, \overline{9}\,) \\
\longmapsto\ & \epsilon \triangleleft \overline{9}
\end{aligned}
$$

where we abbreviate:

$$k_1 = \epsilon \,;\, x \,.\, \texttt{bind}(\, \texttt{comp}(\, \texttt{ap}(\, g[\texttt{cont}(\, \epsilon\,)] \,;\, x\,)\,)\,;\, y \,.\, \texttt{ret}(\, \texttt{s}(\, y\,)\,)\,)$$

Observe that $\epsilon \triangleleft \overline{9}$ final, so e evaluates to $\overline{9}$.

Task 1.2 (10 pts). Consider the following expression e:

$$
\begin{aligned}
\textsc{Lem}[\tau] &\triangleq \texttt{letcc}[\, \tau + \tau\ \texttt{cont}\,](\, k \,.\, \texttt{bind}(\, \texttt{comp}(\, \texttt{letcc}[\, \tau\,](\, k' \,.\, \texttt{throw}[\, \tau\,](\, k \,;\, \texttt{r} \cdot k'\,)\,)\,)\,;\, x \,.\, \texttt{ret}(\, \texttt{l} \cdot x\,)\,)\,) \\
e &\triangleq \texttt{bind}(\, \texttt{comp}(\, \textsc{Lem}[\texttt{nat}]\,)\,;\, y \,.\, \texttt{case}(\, y \,;\, n \,.\, \texttt{ret}(\, n\,)\,;\, k \,.\, \texttt{throw}[\texttt{nat}](\, k \,;\, \overline{312}\,)\,)\,)
\end{aligned}
$$

Evaluate $\epsilon \triangleright e$ until it reaches a final state.

1.5.2 Implementing the K Machine

Task 1.3 (20 pts). Implement the structure `Dynamics` in the file `kpcf/language/dynamics.sml`.

Testing A REPL is available through `InterpreterKPCF.repl ()`, in which you can directly input **KPCF** expressions and see the type and the value it evaluates to. Remember your input has to be a modal separated expression, otherwise the parser will reject your input outright. Here is an example interaction with the interpreter:

```
- InterpreterKPCF.repl ();
->z;
Statics: term has type Nat
Zero
->fn (a : A) => fn (b : B) => <a, b>;
Statics: term has type (Arrow (A, (Arrow (B, (Prod (A, B))))))
(Lam ((a17, A) . (Ret (Lam ((b19, B) . (Bind ((Comp (Ret a17)),
   (tuple_e1_21 . (Bind ((Comp (Ret b19)), (tuple_e2_23 . (Ret
   (Tuple (tuple_e1_21, tuple_e2_23))))))))))))))
->letcc[unit] k in throw[unit](k, <>);
Statics: term has type Unit
Unit
```

A Testing harness can be accessed through `TestHarness.runalltests true`. It evaluates files listed in `tests/tests.sml`. These test files also serves as a syntax guide.

1.6 A Continuation of Logic

Earlier in the semester, we observed that types correspond to logical propositions. We considered a language with a type system corresponding to *constructive* propositional logic. Now, we explore this correspondence in the presence of continuations, which we may interpret as *refutations*. Then, the continuation type corresponds to *classical* negation. We recall the earlier correspondence, extending it with continuations:

Connective	Proposition φ	Type $\overline{\varphi}$
trivial truth	$\top$	$\mathtt{unit}$
conjunction	$\varphi_1 \wedge \varphi_2$	$\overline{\varphi_1} \times \overline{\varphi_2}$
trivial falsehood	$\bot$	$\mathtt{void}$
disjunction	$\varphi_1 \vee \varphi_2$	$\overline{\varphi_1} + \overline{\varphi_2}$
implication	$\varphi_1 \supset \varphi_2$	$\overline{\varphi_1} \to \overline{\varphi_2}$
negation	$\neg\varphi$	$\mathtt{cont}(\overline{\varphi})$

This correspondence doesn't quite hold in **KPCF**, since divergence results in an inconsistent proof system. For example, the value

$$\mathtt{fun}[\,\mathtt{unit}\,;\mathtt{void}\,](\,f\,.\,x\,.\,f(x)\,) : \mathtt{unit} \to \mathtt{void}$$

would prove the proposition $\top \supset \bot$, which should not be the case. Therefore, we consider a subset of **KPCF** in which all expressions terminate in this subsection, using $\tau_1 \to \tau_2$ instead of $\tau_1 \rightharpoonup \tau_2$. **For the tasks in this subsection, you may not use recursive functions.**

You should write the following proofs using the concrete syntax of **KPCF**, available in the table in Section 1.1; we will elaborate your code to **KPCFv** for typechecking, so you may interpret $\vdash e : \tau$

in **KPCF** as $\vdash \overline{e} \mathrel{\dot\sim} \overline{\tau}$ in **KPCFv**. Feel free to draw inspiration from sample proofs in `kpcf/tests/`. To test your answers, you can use the function `InterpreterKPCF.checkFile`.

Task 1.4 (10 pts). To refute $A \vee B$, you need to refute both A and B. In `kpcf/refute.kpcf`, exhibit an expression e such that:

$$\vdash e : \texttt{cont}(\,\alpha\,) \times \texttt{cont}(\,\beta\,) \to \texttt{cont}(\,\alpha + \beta\,)$$

Behavior specification: Let k be the continuation which $e(\,\langle k_1, k_2\rangle\,)$ evaluates to. Then, throwing $\texttt{l} \cdot v$ to k throws v to k_1, and throwing $\texttt{r} \cdot v$ to k throws v to k_2.

The use of continuation blurs the difference between constructive negations $A \supset \bot$ and refutations of A. Constructively, a value of $\alpha \to \texttt{void}$ affirms the nonexistence of values of type α, since there is no value of type $\texttt{void}$. However, in (terminating) **KPCFv**, it's possible to come up with a function of type $\alpha \to \texttt{void}$ provided with a continuation of α:

$$\lambda\,(\,k : \alpha\,\texttt{cont}\,)\,\lambda\,(\,x : \alpha\,)\,\texttt{throw}[\,\texttt{void}\,](\,k\,;x\,) : \alpha\,\texttt{cont} \to (\alpha \to \texttt{void})$$

The continuation serves as an escape hatch for the function, saving the term from having to produce a value of type $\texttt{void}$. This corresponds to the tautology $\neg A \supset (A \supset \bot)$.

Task 1.5 (10 pts). Show that the converse also holds. In `kpcf/nothrow.kpcf`, exhibit an expression e such that:

$$\vdash e : (\,\alpha \to \texttt{void}\,) \to \texttt{cont}(\,\alpha\,)$$

Behavior specification: Let k be the continuation which $e(\,f\,)$ evaluates to. Hypothetically, throwing v to k would evaluate $\texttt{abort}[\,\alpha\,](\,v'\,)$, for some v'.

Task 1.6 (10 pts). Now consider the proposition $(A \supset B) \supset (B \vee \neg A)$. The *law of excluded middle* (LEM), $A \vee \neg A$, is directly derivable from this proposition if we substitute A for B. In fact, $(A \supset B) \supset (B \vee \neg A)$ is equivalent to $A \vee \neg A$.

In `kpcf/derivable.kpcf`, exhibit an expression e of type:

$$\vdash e : (\alpha \to \beta) \to (\beta + \texttt{cont}(\,\alpha\,))$$

Behavior specification: Suppose v is the result of evaluating $e(\,f\,)$ on stack k. If $v = \texttt{r} \cdot k'$ for some continuation k', then throwing some v' to k' throws $\texttt{l} \cdot f(\,v'\,)$ to k.

In the lecture, we have observed it's possible to prove the law of excluded middle (LEM) by exhibiting a term of type $\tau + \texttt{cont}(\,\tau\,)$:

$$\texttt{letcc}[\,\tau + \texttt{cont}(\,\tau\,)\,](\,r\,.\,\texttt{bind}(\,\texttt{comp}(\,\texttt{letcc}[\,\tau_1\,](\,r'\,.\,\texttt{throw}\,(\texttt{r} \cdot r')\,\texttt{to}\,r\,)\,)\,; x\,.\,\texttt{l} \cdot x\,)\,)$$

Task 1.7 (10 pts). Another proposition that is equivalent to LEM is known as Peirce's law. Although it's equivalent to LEM, it does not involve negation in the formula. In `kpcf/peirce.kpcf`, exhibit an expression e of the following type that corresponds to Peirce's law.

$$\vdash e : ((\alpha \to \beta) \to \alpha) \to \alpha$$

Behavior specification: When applied to some function f, it returns a value of type α.

Hint: Consider how f may behave: it either returns a value of α, or it activates its argument with a value of α. Either way, f knows a proof of α.

2 Twice Upon the **MA**

In the previous section we have seen a language that manipulates the control flow. The modal separated language separates values, which does not step, from expressions, which steps and therefore induce a notion of control flow. This separation enables us to reify the control flow with a stack machine, then further reify the stack as a value, enabling programs to directly program their control flow.

In this section we will deploy the idea of modal separation once more for *storage effects*. Imperative languages are known for their ability to allocate, store and mutate contents in some storage, identified colloquially by variables, but here precisely as *assignables*. Observe that the result of a program is sensitive to the order of storage effects, prompting us to consider a modal separated language between expressions, which are effect-free computations, and commands, which engenders, among possibly others, storage effects.

In this section we will consider storage effects in two flavours: the *free* assignables and the *scoped* assignables. The scoped assignables are assignables whose existence are tied to a *scope*, i.e. tied to command they are declared in. They model the idea of a "local assignable" commonly found in imperative languages. In contrast, free assignables are assignables that, once allocated, exists independent of their scope of declaration. They model the idea of "heap allocation" commonly found in languages.

In this assignment, you will explore both setups. We will start with modernized algo with free assignables. You will implement type checking and dynamics for this calculus, then implement a few commonly found syntax extensions for this language. You will then explore scoped assignables, presented with an explicit control stack, borrowing ideas from **KPCF**. You will see that it is possible to further extend the language to support another commonly found non-local transfer of control, namely the `exit` command.

Typ	τ	$::=$	$\cdots$	$\cdots$	everything else
			$\tau\;\texttt{cmd}$	$\tau\;\texttt{cmd}$	command types
Exp	e	$::=$	$\cdots$	$\cdots$	everything else
			$\texttt{cmd}(\,m\,)$	$\texttt{cmd}\,m$	encapsulated commands
Cmd	m	$::=$	$\texttt{ret}(\,e\,)$	$\texttt{ret}\,e$	return
			$\texttt{bnd}(\,e\,;x\,.\,m\,)$	$\texttt{bnd}\,x \leftarrow e\,;m$	bind command
			$\texttt{dcl}[\,\tau\,](\,e\,;a\,.\,m\,)$	$\texttt{dcl}\,a := e\;\texttt{in}\;m$	declare new assignable
			$\texttt{get}[\,a\,]$	$@\,a$	get variable
			$\texttt{set}[\,a\,](\,e\,)$	$a := e$	set variable

Figure 2.1: **MA** Grammar

2.1 MA with *free* assignables

The (reduced) syntax chart for **MA** with free assignables is presented in Figure 2.1. The syntax chart as presented focuses on the introduction of commands.

The language **MA** with free assignable has two program syntax sort: that of expressions and that of commands. The top-level user program will be a command.

On the expression level, expressions for sums, products, (recursive) functions, booleans and recursive types are all available and standard. They are evaluated eagerly. It additionally supports encapsulated commands (i.e. expressions containing "unevaluated" commands). This provides the language the ability to "stage" commands in the expression layer so that they can be effected down the line. Unlike **KPCFv**, values of expressions are part of expressions, as the modality distinguishes between commands and expressions, not values and non-values (within expressions).

On the command layer, in addition to commands arising from the modal distinction, three additional commands are present: $\texttt{dcl}[\,\tau\,](\,e\,;a\,.\,m\,)$ declares the *assignable* a and make it available within m. It is important to understand that assignables themselves are in a different syntactic then variables, in particular the command m is under an assignable binder, in other words, a is *not* a variable within m. The command $\texttt{get}[\,a\,]$ and $a := e$ reads from the assignable a and assigns to it with expression e respectively.

"Runtime" incarnations of commands are machines of syntax $\nu\,\Sigma\,\{\,m \parallel \mu\,\}$, where m is executing command, Σ is the signature for already-allocated *free* assignables, and μ is memory mapping assignables to expressions. The empty memory is denoted with $\emptyset$ (so is the empty signature). Entries in the memory is syntactically delimited with the symbol "$\otimes$"

$$\mu ::= \emptyset \mid \mu \otimes a \hookrightarrow e_a$$

For example, the following memory μ_0 maps assignable a to $\texttt{z}$ and b to $\texttt{true}$

$$\mu_0 \triangleq a \hookrightarrow \texttt{z} \otimes b \hookrightarrow \texttt{true}$$

Statics Similar to **KPCFv**, we will have two judgments in our static semantics: one for expressions $\boxed{\Gamma \vdash_\Sigma e : \tau}$ and one for commands $\boxed{\Gamma \vdash_\Sigma m \mathrel{\dot\sim} \tau}$. Both typing judgments are indexed by the signature of assignable, mapping the name of the assignable to the type of its (expression) content.

The expression typing judgment states that expression e has type τ under the signature Σ. For the expression forms that we have warmed up to, the signature is simply threaded through. To typed encapsulated commands, one simply that encapsulated command is well-typed as a command of type τ, which we will explain shortly after.

$\boxed{\Gamma \vdash_\Sigma e : \tau}$

$$\frac{\Gamma \vdash_\Sigma m \mathrel{\dot\sim} \tau}{\Gamma \vdash_\Sigma \mathtt{cmd}(\,m\,) : \tau\,\mathtt{cmd}}\ \mathtt{cmd}\text{-I}$$

The command typing judgment states that command m, when executed under signature Σ, (if terminates) produces a value of type τ.

$\boxed{\Gamma \vdash_\Sigma m \mathrel{\dot\sim} \tau}$

$$\frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \mathtt{ret}(\,e\,) \mathrel{\dot\sim} \tau}\ \mathtt{ret}\text{-C} \qquad \frac{\Gamma \vdash_\Sigma e : \tau'\,\mathtt{cmd} \qquad \Gamma, x : \tau' \vdash_\Sigma m \mathrel{\dot\sim} \tau}{\Gamma \vdash_\Sigma \mathtt{bnd}(\,e\,;x\,.\,m\,) \mathrel{\dot\sim} \tau}\ \mathtt{bnd}\text{-C}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a\sim\tau} \mathtt{get}[\,a\,] \mathrel{\dot\sim} \tau}\ \mathtt{get}\text{-C} \qquad \frac{\Gamma \vdash_{\Sigma, a\sim\tau} e : \tau}{\Gamma \vdash_{\Sigma, a\sim\tau} \mathtt{set}[\,a\,](\,e\,) \mathrel{\dot\sim} \mathtt{unit}}\ \mathtt{set}\text{-C}$$

$$\frac{\Gamma \vdash_{\Sigma, a\sim\tau} m \mathrel{\dot\sim} \tau' \qquad \Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \mathtt{dcl}[\,\tau\,](\,e\,;a\,.\,m\,) \mathrel{\dot\sim} \tau'}\ \mathtt{dcl}\text{-C}$$

By executing a command under signature Σ, we meant the command is executed along with a memory μ that adheres to Σ, i.e. all assignables in Σ are allocated and contains properly typed contents. This intuition is captured in the following "machine" typing judgment.

$\boxed{\vdash_\Sigma \nu\,\Sigma\,\{\,m \parallel \mu\,\}\ \mathsf{ok}}$

$$\frac{\vdash_\Sigma m \mathrel{\dot\sim} \tau \qquad \forall_{a\in\Sigma}(\ \vdash_\Sigma \mu(a) : \Sigma(a))}{\nu\,\Sigma\,\{\,m \parallel \mu\,\}\ \mathsf{ok}}\ M-\text{OK}$$

The rules state that a machine is well-behaving if the command is well-typed under the signature of the currently allocated assignables, and that for every entry a of the memory μ, the content of the cell $\mu(a)$ is well-typed, again under the current signature Σ, with type specified by the signature $\Sigma(a)$.

Dynamics When provided with user program m, the machine executes with the initial state $\nu\,\emptyset\,\{\,m \parallel \emptyset\,\}$, i.e. with an empty signature and memory. The final state of the machine consists of single $\mathtt{ret}(e)$ where e val, regardless of the status of Σ and μ. For the machine to be final, not only all commands are processed, the expression must also have reached a value.

$\boxed{\nu\,\Sigma\,\{\,m \parallel \mu\,\}\ \mathsf{initial}}$ $\boxed{\nu\,\Sigma\,\{\,m \parallel \mu\,\}\ \mathsf{final}}$

$$\frac{}{\nu\,\emptyset\,\{\,m \parallel \emptyset\,\}\ \mathsf{initial}}\ \text{D-}\mathtt{init} \qquad \frac{e\ \mathsf{val}}{\nu\,\Sigma\,\{\,\mathtt{ret}(e) \parallel \mu\,\}\ \mathsf{final}}\ \text{D-}\mathtt{fin}$$

Execution of the machine are defined case-by-case for each command. Commands that work on expressions, such as $\mathtt{ret}(e)$ and $a := e$, must first evaluate their expression fully.

$\boxed{\nu\,\Sigma\,\{\,m \parallel \mu\,\} \longmapsto \nu\,\Sigma'\,\{\,m' \parallel \mu'\,\}}$

$$\frac{e \longmapsto e'}{\nu\,\Sigma\,\{\,\mathtt{ret}(\,e\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{ret}(\,e'\,) \parallel \mu\,\}}\ \text{D}_1\text{-}\mathtt{ret}$$

$$\frac{\nu\,\Sigma\,\{\,m \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,m' \parallel \mu'\,\}}{\nu\,\Sigma\,\{\,\mathtt{bnd}(\,\mathtt{cmd}(\,m\,)\,;x\,.\,m_1\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{bnd}(\,\mathtt{cmd}(\,m'\,)\,;x\,.\,m_1\,) \parallel \mu'\,\}}\ \text{D}_1\text{-}\mathtt{bnd}$$

$$\frac{e\ \mathsf{val}}{\nu\,\Sigma\,\{\,\mathtt{bnd}(\,\mathtt{cmd}(\,\mathtt{ret}(\,e\,)\,)\,;x\,.\,m_1\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{[e/x]m_1 \parallel \mu\,\}}\ \text{D}_2\text{-}\mathtt{bnd}$$

$$\frac{e \longmapsto e'}{\nu\,\Sigma\,\{\,\mathtt{bnd}(\,e\,;x\,.\,m''\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{bnd}(\,e'\,;x\,.\,m''\,) \parallel \mu\,\}}\ \text{D}_3\text{-}\mathtt{bnd}$$

$$\frac{}{\nu\,\Sigma, a \sim \tau\,\{\,\mathtt{get}[\,a\,] \parallel \mu \otimes a \hookrightarrow e\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,\mathtt{ret}(\,e\,) \parallel \mu \otimes a \hookrightarrow e\,\}}\ \text{D}_1\text{-}\mathtt{get}$$

$$\frac{e \longmapsto e'}{\nu\,\Sigma, a \sim \tau\,\{\,\mathtt{set}[\,a\,](\,e\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,\mathtt{set}[\,a\,](\,e'\,) \parallel \mu\,\}}\ \text{D}_1\text{-}\mathtt{set}$$

$$\frac{e\ \mathsf{val}}{\nu\,\Sigma, a \sim \tau\,\{\,\mathtt{set}[\,a\,](\,e\,) \parallel \mu \otimes a \hookrightarrow e'\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,\mathtt{ret}(\,\langle\rangle\,) \parallel \mu \otimes a \hookrightarrow e\,\}}\ \text{D}_2\text{-}\mathtt{set}$$

$$\frac{e \longmapsto e'}{\nu\,\Sigma\,\{\,\mathtt{dcl}[\,\tau\,](\,e\,;a\,.\,m\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,\mathtt{dcl}[\,\tau\,](\,e'\,;a\,.\,m\,) \parallel \mu\,\}}\ \text{D}_1\text{-}\mathtt{dcl}$$

$$\frac{e\ \mathsf{val}}{\nu\,\Sigma\,\{\,\mathtt{dcl}[\,\tau\,](\,e\,;a\,.\,m\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,m \parallel \mu \otimes a \hookrightarrow e\,\}}\ \text{D}_2\text{-}\mathtt{dcl}$$

One notable command is that of $\texttt{dcl}\, a := e \,\texttt{in}\, m$. This commands after evaluating the principal expression to a value, allocates a new assignable in the signature and memory. Once allocated, the assignable exists "permanently", even when the encapsulated command returns. This allows references to the allocated assignable to escape it's intended scope, setting the assignable *free*.

Implementation In this part you will complete a partial implementation of the language. We have provided type checker and stepping for expressions. You will need to implement the statics and dynamics of the language

Task 2.1 (20 pts). Implement the typechecker for **MA** with free assignables in the structure `StaticsMA` in `ma/statics_ma.sml` according to `ma/statics_ma.sig`. In particular we have implemented the statics for expressions, so you only need to implement that of commands.

Programming in plain **MA** quickly becomes unwieldy. To ease the pain, the implementation provides the following syntactical conveniences:

- $\texttt{if}_{\texttt{m}}\,(\,m\,)\,\texttt{then}\,\{m_1\}\,\texttt{else}\,\{m_2\}$: Conditional command, which executes m for `bool` and branch.
- $\texttt{while}\,(\,m\,)\{m_1\}$: Loop command, which executes m_1 until test in m fails.
- $m_1\,;m_2$: Sequencing command, which execute m_1 and m_2 sequentially.
- $\texttt{ignore}\,(\,m\,)$: "ignore" command, which runs the command and throws away the result.

Typing judgments for these short-hands are provided as follows. The loop command has type `unit` because it is possible that the loop body never executes.

$$\frac{\Gamma \vdash_\Sigma m \mathrel{\dot\sim} \texttt{bool} \qquad \Gamma \vdash_\Sigma m_1 \mathrel{\dot\sim} \tau \qquad \Gamma \vdash_\Sigma m_2 \mathrel{\dot\sim} \tau}{\Gamma \vdash_\Sigma \texttt{if}_{\texttt{m}}\,(\,m\,)\,\texttt{then}\,\{m_1\}\,\texttt{else}\,\{m_2\} \mathrel{\dot\sim} \tau} \qquad \frac{\Gamma \vdash_\Sigma m \mathrel{\dot\sim} \texttt{bool} \qquad \Gamma \vdash_\Sigma m_1 \mathrel{\dot\sim} \texttt{unit}}{\Gamma \vdash_\Sigma \texttt{while}\,(\,m\,)\{m_1\} \mathrel{\dot\sim} \texttt{unit}}$$

$$\frac{\Gamma \vdash_\Sigma m_1 \mathrel{\dot\sim} \texttt{unit} \qquad \Gamma \vdash_\Sigma m_2 \mathrel{\dot\sim} \tau}{\Gamma \vdash_\Sigma m_1\,;m_2 \mathrel{\dot\sim} \tau} \qquad \frac{\Gamma \vdash_\Sigma m \mathrel{\dot\sim} \tau}{\Gamma \vdash_\Sigma \texttt{ignore}\,(\,m\,) \mathrel{\dot\sim} \texttt{unit}}$$

Task 2.2 (20 pts). Desugar these syntax conveniences into commands of our language in `ma/parser/desugar.sml` according to `ma/parser/desugar.sig`. Implement the command dynamics for **MA** with free assignables in the structure `DynamicsMA` in `ma/dynamics_ma.sml`.

The parser will use your version of `Desugar` to parse the concrete syntax, so you will not be able to run **MA** code until you have implemented the desugaring functions.

To help you program in **MA**, we provide some other syntactic sugars, which are explained in Appendix A. Concrete syntax is also explained in Appendix A. We also provide you a set of test cases in `ma/tests/basic` and `ma/tests/large`.

Task 2.3 (10 pts). Implement the following functions in **MA** using concrete syntax. Your solution shouldn't be recursive and should use **MA** features.

1. In `ma/tests/tasks/collatz.ma`, implement `collatz : nat -> cmd[nat]` such that `collatz n` returns the number of steps it takes to reach 1 when starting from `n` and repeatedly applying

the following rule: if `n` is even, divide it by 2. If `n` is odd, multiply it by 3 and add 1. For example, `collatz 27` should return 111.[1]

2. In `ma/tests/tasks/gcd.ma`, implement `gcd : nat -> nat -> cmd[nat + unit]` such that `gcd m n` returns the greatest common divisor of `m` and `n` when defined. Greatest common divisor of 0 and 0 is undefined and should return right of unit. For example, `gcd 4 6` should return 2 injected to the left.

You might recall similar implementation of those functions in language like C as 2.2.

```
int collatz(int n) {
  int count = 0;
  while (n != 1) {
    if (n % 2 == 0) {
      n = n / 2;
    } else {
      n = 3 * n + 1;
    }
    count = count + 1;
  }
  return count;
}

int gcd(int m, int n) {
  if (m == 0 && n == 0) {
    // error
  }
  while (n != 0) {
    int r = m % n;
    m = n;
    n = r;
  }
  return m;
}
```


Figure 2.2: `collatz` and `gcd` in C

Testing As usual, you can test your code using `InterpreterMA`. We provide a set of test cases for `collatz` and `gcd` in `ma/tests/test.ma`. For example:

```
- InterpreterMA.repl ();
->ret(1);
(Num 1)
->local a := 1 in { set a := 2 , get a };
```

[1]https://en.wikipedia.org/wiki/Collatz_conjecture

```
(Num 2)

- InterpreterMA.evalFile "large/sum.ma";
(Num 10000)
val it = () : unit
```

2.2 MA with *scoped* assignables

Alternative to assignables being *free*, they can be alternatively be *scoped*. The existence of a scoped assignable is tied to the "block" that the assignable is declared in. The assignable is created when we enter the scope, and it is de-allocated when we exit it. The number of assignables allocated at a time increases as we drill down into sub-commands, and decreases when we come back up.

Therefore, it is a common design to tie the scoped assignable to the control stack, as exemplified by the C-family languages. Scoped assignables are colloquially referred to as *stack-allocated* assignable for this reason. In principle, the assignables do not have to be allocated literally "on the stack", as in occupying control stack spaces.

In this exercise, we will examine this common design choice, by first taking inspiration from **KPCF** and make explicit the control stack of **MA** with scoped assignables. We will see how this design presents us intricate proofs of safety. We will then extend the language with a new command that resembles a `throw` expression, performing a non-local transfer of control. However, this new command must respect the scoped assignable allocation scheme, which leads to an interesting design.

Typ	τ	$::=$	$\cdots$	$\cdots$	everything else
Exp	e	$::=$	$\cdots$	$\cdots$	everything else
Cmd	m	$::=$	$\cdots$	$\cdots$	everything else
Stack	k	$::=$	ϵ	ϵ	empty stack
			$k\,;\mathtt{bnd}(\,-\,;x\,.\,m\,)$	$k\,;\mathtt{bnd}(\,-\,;x\,.\,m\,)$	sequenced command
			$k\,;\mathtt{dcl}[\,a\,]$	$k\,;\mathtt{dcl}[\,a\,]$	allocation frame
Mach	$\mathcal{M}$	$::=$	$\nu\,\Sigma\,\{\,k \triangleright m \parallel \mu\,\}$	$\nu\,\Sigma\,\{\,k \triangleright m \parallel \mu\,\}$	machine evaluates
			$\nu\,\Sigma\,\{\,k \triangleleft e \parallel \mu\,\}$	$\nu\,\Sigma\,\{\,k \triangleleft e \parallel \mu\,\}$	machine returns

Figure 2.3: **MA** with a stack machine Grammar

Fig. 2.3 presents the syntax for the stack frame and machine. The command level is exactly like before. The expression level contains sums, products (recursive functions) and a few base types, including `nat`, and is also completely standard otherwise.

The runtime machine is again indexed by an allocated assignable signature Σ. Associated with the signature is the memory μ. It now additionally contains a command stack k, recording subsequent commands along with scope of the allocated assignables.

The machine can either be an evaluation state, in which case it contains the currently executing command m, or it can be in a returning state with an expression to be returned.

Statics We will derive the statics of the language from the previous part. The expressions of the language are typed exactly the same. On the command level, the declaration command now demands the return type and the type of the assignable to be *mobile*. Intuitively, a type is mobile if its values does not depend on assignables. This is to prevent the declared assignable to leave the scope of declaration and break safety.

$\boxed{\Gamma \vdash_\Sigma m \mathrel{\dot{\sim}} \tau}$

$$\frac{\Gamma \vdash_\Sigma e : \tau \qquad \Gamma \vdash_{\Sigma, a \sim \tau} m \mathrel{\dot{\sim}} \tau' \qquad \tau\ \mathsf{mobile} \qquad \tau'\ \mathsf{mobile}}{\Gamma \vdash_\Sigma \mathtt{dcl}[\,\tau\,](\,e\,;a\,.\,m\,) \mathrel{\dot{\sim}} \tau'}\ \texttt{dcl-C}$$

The mobility judgment is defined inductively on the structure of τ. Notably, command type is not mobile as it can contain encapsulated accesses to the assignable. In addition, function types are not mobile, as the expression in the body of the function may contain encapsulated commands.

$\boxed{\tau\ \mathsf{mobile}}$

$$\frac{}{\mathtt{nat}\ \mathsf{mobile}}\ M_{\mathtt{nat}} \qquad \frac{}{\mathtt{unit}\ \mathsf{mobile}}\ M_{\mathtt{unit}} \qquad \frac{}{\mathtt{void}\ \mathsf{mobile}}\ M_{\mathtt{void}}$$

$$\frac{\tau_1\ \mathsf{mobile} \qquad \tau_2\ \mathsf{mobile}}{\tau_1 + \tau_2\ \mathsf{mobile}}\ M_{+} \qquad \frac{\tau_1\ \mathsf{mobile} \qquad \tau_2\ \mathsf{mobile}}{\tau_1 \times \tau_2\ \mathsf{mobile}}\ M_{\times}$$

Stack Machine Dynamics Given program command m, the initial configuration executes m against the empty stack, in the empty signature with an empty memory. This is exactly the same as before. Unlike **MA** with free assignables, the terminal state of the machine must be a return of value to the empty stack, with an *empty* assignable signature and memory. This is because the assignables are scoped.

$\boxed{\mathcal{M}\ \mathsf{initial}}$ $\boxed{\mathcal{M}\ \mathsf{final}}$

$$\frac{}{\nu\,\emptyset\,\{\,\epsilon \triangleright m \parallel \emptyset\,\}\ \mathsf{initial}}\ \texttt{D-init} \qquad \frac{e\ \mathsf{val}}{\nu\,\emptyset\,\{\,\epsilon \triangleleft e \parallel \emptyset\,\}\ \mathsf{final}}\ \texttt{D-fin}$$

Dynamics for execution of commands are provided in Fig. 2.4. Rules again insist that principal expression be evaluated before the commands themselves. The sequencing command pushes a sequencing frame onto the stack. The declaration command pushes a declaration frame onto the stack, and extends the signature along with the memory.

The dynamics is much more interesting when the machine transitions into returning. The machine makes such transitions as it executes a $\mathtt{ret}\,e$ command (where e val). When it returns, it examines

$$\boxed{\mathcal{M} \longmapsto \mathcal{M}'}$$

$$\frac{e \longmapsto e'}{\nu\,\Sigma\,\{\,k \triangleright \texttt{ret}(\,e\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,k \triangleright \texttt{ret}(\,e'\,) \parallel \mu\,\}}\ \text{D-}\texttt{ret}$$

$$\frac{e \longmapsto e'}{\nu\,\Sigma\,\{\,k \triangleright \texttt{bnd}(\,e\,;x\,.\,m\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,k \triangleright \texttt{bnd}(\,e'\,;x\,.\,m\,) \parallel \mu\,\}}\ \text{D}_1\text{-}\texttt{bnd}$$

$$\frac{}{\nu\,\Sigma\,\{\,k \triangleright \texttt{bnd}(\,\texttt{cmd}(\,m\,)\,;x\,.\,m'\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,k\,;\texttt{bnd}(\,-\,;x\,.\,m'\,) \triangleright m \parallel \mu\,\}}\ \text{D}_2\text{-}\texttt{bnd}$$

$$\frac{e\ \mathsf{val}}{\nu\,\Sigma, a \sim \tau\,\{\,k \triangleright \texttt{get}[\,a\,] \parallel \mu \otimes a \hookrightarrow e\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,k \triangleleft e \parallel \mu \otimes a \hookrightarrow e\,\}}\ \text{D}_1\text{-}\texttt{get}$$

$$\frac{e \longmapsto e'}{\nu\,\Sigma, a \sim \tau\,\{\,k \triangleright \texttt{set}[\,a\,](\,e\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,k \triangleright \texttt{set}[\,a\,](\,e'\,) \parallel \mu\,\}}\ \text{D}_1\text{-}\texttt{set}$$

$$\frac{e\ \mathsf{val}}{\nu\,\Sigma, a \sim \tau\,\{\,k \triangleright \texttt{set}[\,a\,](\,e\,) \parallel \mu \otimes a \hookrightarrow e'\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,k \triangleleft \langle\rangle \parallel \mu \otimes a \hookrightarrow e\,\}}\ \text{D}_2\text{-}\texttt{set}$$

$$\frac{e \longmapsto e'}{\nu\,\Sigma\,\{\,k \triangleright \texttt{dcl}[\,\tau\,](\,e\,;a\,.\,m\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,k \triangleright \texttt{dcl}[\,\tau\,](\,e'\,;a\,.\,m\,) \parallel \mu\,\}}\ \text{D}_1\text{-}\texttt{dcl}$$

$$\frac{e\ \mathsf{val}}{\nu\,\Sigma\,\{\,k \triangleright \texttt{dcl}[\,\tau\,](\,e\,;a\,.\,m\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,\{\,k\,;\texttt{dcl}[\,a\,] \triangleright m \parallel \mu \otimes a \hookrightarrow e\,\}}\ \text{D}_2\text{-}\texttt{dcl}$$

Figure 2.4: **MA** with a stack machine dynamics

the top level stack. If the top stack represents a sequenced a command, it then transitions back to normal execution. If the top stack represents a declaration, then the associated assignable is de-allocated, and the machine continues to return. Such de-allocation is often called *stack unwinding*.

$\boxed{\mathcal{M} \longmapsto \mathcal{M}'}$

$$\frac{e\ \mathsf{val}}{\nu\,\Sigma\,\{\,k \triangleright \mathtt{ret}(\,e\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,k \triangleleft e \parallel \mu\,\}}\ \mathrm{D}_1\text{-}\mathtt{mret}$$

$$\frac{e\ \mathsf{val}}{\nu\,\Sigma\,\{\,k\,;\mathtt{bnd}(\,-\,;x\,.\,m\,) \triangleleft e \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,k \triangleright \{e/x\}m \parallel \mu\,\}}\ \mathrm{D}_2\text{-}\mathtt{mret}$$

$$\frac{e\ \mathsf{val}}{\nu\,\Sigma, a \sim \tau\,\{\,k\,;\mathtt{dcl}[\,a\,] \triangleleft e \parallel \mu \otimes a \hookrightarrow e'\,\} \longmapsto \nu\,\Sigma\,\{\,k \triangleleft e \parallel \mu\,\}}\ \mathrm{D}_3\text{-}\mathtt{mret}$$

Type Safety The intermingling of language designs poses challenges to the safety of the language. It is not immediately obvious allocations are handled properly. Therefore, we will briefly consider the argument for type safety.

The judgment $\boxed{\mathcal{M}\ \mathsf{ok}}$ asserts that the machine is well-formed. To define this judgment, we further define $\boxed{\mu : \Sigma}$, stating the memory is well-formed with respect to the signature Σ, and $\boxed{\vdash_\Sigma k \div \tau}$, which states that under signature Σ, stack k refutes τ.

The machine is well-formed if the command (in normal execution) or the expression (when returning) has the type refuted by a well-formed stack. The memory is well-formed, if it corresponds to the signature one-to-one, and every cell of contains an expression of the appropriate mobile type, typed in the *empty* signature. The signature is empty as types of the cells are all mobile, and the judgment records this fact.

$\boxed{\mathcal{M}\ \mathsf{ok}}$ $\boxed{\mu : \Sigma}$ $\boxed{\vdash_\Sigma k \div \tau}$

$$\frac{\vdash_\Sigma k \div \tau \qquad \vdash_\Sigma m \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau \qquad \mu : \Sigma}{\nu\,\Sigma\,\{\,k \triangleright m \parallel \mu\,\}\ \mathsf{ok}}\ \mathcal{M}_1 \qquad \frac{\vdash_\Sigma k \div \tau \qquad \vdash_\Sigma e : \tau \qquad \mu : \Sigma \qquad e\ \mathsf{val}}{\nu\,\Sigma\,\{\,k \triangleleft e \parallel \mu\,\}\ \mathsf{ok}}\ \mathcal{M}_2$$

$$\frac{\forall_{a \in \Sigma}(\ \vdash_\emptyset \mu(a) : \Sigma(a) \qquad \mu(a)\ \mathsf{val} \qquad \Sigma(a)\ \mathsf{mobile}\ ) \qquad \forall_{a \in \mu}(\ a \in \Sigma)}{\mu : \Sigma}\ \mathcal{M}_3$$

$$\frac{}{\vdash_\emptyset \epsilon \div \tau}\ K_1 \qquad \frac{\vdash_\Sigma k \div \tau \qquad \tau\ \mathsf{mobile}}{\vdash_{\Sigma, a \sim \tau'} k\,;\mathtt{dcl}[\,a\,] \div \tau}\ K_2 \qquad \frac{x : \tau \vdash_\Sigma m \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau' \qquad \vdash_\Sigma k \div \tau'}{\vdash_\Sigma k\,;\mathtt{bnd}(\,-\,;x\,.\,m\,) \div \tau}\ K_3$$

The role of the mobility judgment is further clarified with the following lemma, which states that closed values of mobile type cannot contain any assignables, therefore is well typed in the empty signature. The proof is a straightforward induction on both premises.

Lemma 2.1 (Validity of mobile values)**.** *If e val, $\cdot \vdash_\Sigma e : \tau$ and τ mobile, then $\cdot \vdash_\emptyset e : \tau$.*

We are now ready to state the safety theorems:

Theorem 2.2 (Preservation)**.** *Preservation is stated for expression and commands*

- *(Expression) If $e \longmapsto e'$ and $\vdash_\Sigma e : \tau$ then $\vdash_\Sigma e' : \tau$.*
- *(Command) If $\mathcal{M} \longmapsto \mathcal{M}'$ and $\mathcal{M}$ ok then $\mathcal{M}'$ ok.*

Task 2.4 (15 pts). Prove preservation for rules $D_3 - \texttt{mret}$, $D_1 - \texttt{get}$, $D_2 - \texttt{set}$ and $D_2 - \texttt{dcl}$. That is show preservation for (main rules of) `dcl`, `set` and `get`. Your proof only need to show important steps.

Theorem 2.3 (Progress)**.** *Progress is also stated for expression and commands*

- *(Expression) If $\vdash_\Sigma e : \tau$ then e val or $e \longmapsto e'$.*
- *(Command) If $\mathcal{M}$ ok then $\mathcal{M}$ final or $\mathcal{M} \longmapsto \mathcal{M}'$.*

Task 2.5 (15 pts). Prove the progress theorem for rule $\mathcal{M}_2$, and $\mathcal{M}_1$ when m is `get` or `set`.

2.3 An `exit` from traditions

With the introduction of an explicit control stack it is only natural to further consider other commands that influences the control flow. The difficulty is that any such command must be made to play-well with the scoped assignables design: that is when they redirect control flow, outstanding allocation must be taken care of.

The intermixing of control and assignable stack provides us a surprisingly simple solution. One trick that works is to piggy-back the transfer of control on the declaration command. The idea is that assignables, when declared, serves as a "label" that marks the block of code it is declared in. Later on we can choose to "exit from" the block, or we can "retry" the block but perhaps with the assignable initialized to a different value.

The changes to syntax is listed below. Upon declaring an assignable for a scope, the signature now additionally records the return type of the command within that scope.

$$\Sigma, a \sim \tau \,;\, \tau'$$

This signature contains the assignable a, and marks the command that a is declared in expects to produce a value of type τ'. In other words, its *continuation* expects τ'.

Two new commands are introduced

- The command $\texttt{exit}[\,\tau\,][\,a\,](\,e\,)$ transfers the control out of the scope where a is declared in, with a value e. It behaves likes a forward `goto`.

- The command $\texttt{retry}[\,\tau\,][\,a\,](\,e\,)$ transfer the control to be beginning of the command where a is declared in and re-initialize a with e. It behaves like a backward `goto`.

New machine form $\nu\,\Sigma\,\{\,k \blacktriangleleft^{\texttt{xt}}_{a} e \parallel \mu\,\}$ and $\nu\,\Sigma\,\{\,k \blacktriangleleft^{\texttt{rt}}_{a} e \parallel \mu\,\}$ states that the machine is in the process of transferring control towards (or out of) a with expression e.

In addition, be advised that the declaration frame in addition records the associated command.

Cmd	m	$::=$	$\cdots$	$\cdots$	other commands
			$\texttt{dcl}[\,\tau\,;\tau'\,](\,e\,;a\,.\,m\,)$	$\texttt{dcl}\; a := e \;\textbf{in}\; m$	declare
			$\texttt{exit}[\,\tau\,][\,a\,](\,e\,)$	$\texttt{exit}[\,\tau\,][\,a\,](\,e\,)$	exit from scope
			$\texttt{retry}[\,\tau\,][\,a\,](\,e\,)$	$\texttt{retry}[\,\tau\,][\,a\,](\,e\,)$	retry in scope
Stack	k	$::=$	$\cdots$	$\cdots$	as before
			$k\,;\texttt{dcl}[\,a\,](\,m\,)$	$k\,;\texttt{dcl}[\,a\,](\,m\,)$	allocation frame
Mach	$\mathcal{M}$	$::=$	$\cdots$	$\cdots$	as before
			$\nu\,\Sigma\,\{\,k \blacktriangleleft^{\texttt{xt}}_{a} e \parallel \mu\,\}$	$\nu\,\Sigma\,\{\,k \blacktriangleleft^{\texttt{xt}}_{a} e \parallel \mu\,\}$	exit from a with e
			$\nu\,\Sigma\,\{\,k \blacktriangleleft^{\texttt{rt}}_{a} e \parallel \mu\,\}$	$\nu\,\Sigma\,\{\,k \blacktriangleleft^{\texttt{rt}}_{a} e \parallel \mu\,\}$	retry with $a := e$

Why are these useful? For example in Fig. 2.5, this allows us to borrow the concrete syntax from last part, to code up the following program to test primality of a natural number. The $\texttt{exit}[\,\tau\,][\,a\,](\,e\,)$ allow us to exit early from the loop the moment we have found a divisor.


```
/* If n is not prime, returns a divisor */
fun is_prime (n : nat) : cmd[unit + nat] =
  cmd {
    local i := 1 in {
      while ([i](i < n)) {
        if [i](n mod i = 0) then {
          exit[i](inr[unit + nat](i))
        } else {
          [i] { set i := i + 1 }
        }
      },
      ret (inl[unit + nat](<>))
    }
  }
```

Figure 2.5: Primality test with early exit

In Fig. 2.6, the code simplifies proper fractions, given by a pair of natural numbers, without using `gcd`. Given a pair p, the code test divides p with ever-increasing integers. If a divisor is found then it goes back to where p is initially declared by $\texttt{retry}[\,\tau\,][\,p\,](\,\cdots\,)$ with now simplified fraction. If the divisor reaches the numerator, the fraction is then at its simplest.

```
fun simplify (p0 : nat * nat)  : cmd[nat * nat] =
  cmd{
      local p := p0 in
      local i := 2 in {
        cmdlet n = [p](p.l) in
        cmdlet d = [p](p.r) in
        if [i](i = n) then {
          ret (<n, d>)
        } else {
          if [i](i mod n = 0 && i mod d = 0) then {
            [i] { retry[p](<n / i, d / i>)}
          } else {
            [i] { set i := i + 1 }
          }
        }
      }
  }
```

Figure 2.6: Simplifying proper fractions

Statics The mobility requirement in `dcl` ensures that both `exit` and `continue` command cannot be abused to smuggle assignables out of their scope. Given $a \sim \tau\,;\tau'$, The $\texttt{exit}[\,\tau\,][\,a\,](\,e\,)$ exits the scope of a with $e : \tau'$, and $\texttt{retry}[\,\tau\,][\,a\,](\,e\,)$ re-initializes a with $e : \tau$. Both commands abandon their current continuation, therefore itself is a command of arbitrary type τ_c.

$\boxed{\Gamma \vdash_\Sigma m \mathrel{\dot\sim} \tau}$

$$\frac{}{\Gamma \vdash_{\Sigma, a\sim\tau;\tau'} \texttt{get}[\,a\,] \mathrel{\dot\sim} \tau}\ \textsf{get-C} \qquad \frac{\Gamma \vdash_{\Sigma, a\sim\tau;\tau'} e : \tau}{\Gamma \vdash_{\Sigma, a\sim\tau;\tau'} \texttt{set}[\,a\,](\,e\,) \mathrel{\dot\sim} \texttt{unit}}\ \textsf{set-C}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau \qquad \Gamma \vdash_{\Sigma, a\sim\tau;\tau'} m \mathrel{\dot\sim} \tau' \qquad \tau\ \textsf{mobile} \qquad \tau'\ \textsf{mobile}}{\Gamma \vdash_\Sigma \texttt{dcl}[\,\tau\,;\tau'\,](\,e\,;a\,.\,m\,) \mathrel{\dot\sim} \tau'}\ \textsf{dcl-C}$$

$$\frac{\Gamma \vdash_{\Sigma, a\sim\tau;\tau'} e : \tau'}{\Gamma \vdash_{\Sigma, a\sim\tau;\tau'} \texttt{exit}[\,\tau_c\,][\,a\,](\,e\,) \mathrel{\dot\sim} \tau_c}\ \textsf{exit-C} \qquad \frac{\Gamma \vdash_{\Sigma, a\sim\tau;\tau'} e : \tau}{\Gamma \vdash_{\Sigma, a\sim\tau;\tau'} \texttt{retry}[\,\tau_c\,][\,a\,](\,e\,) \mathrel{\dot\sim} \tau_c}\ \textsf{continue-C}$$

Dynamics The dynamics of the language are updated to support the two new commands. The declaration frame is updated to include the associated command m.

Executing either new commands puts the machine into a back-tracking mode. When in back-tracking, the machine examines the top stack frame. For both commands, it throws away the sequencing frame. These are captured by the following rules.

$$\boxed{\mathcal{M} \longmapsto \mathcal{M}'}$$

$$\frac{e\ \mathsf{val}}{\nu\,\Sigma\,\{\,k \triangleright \mathtt{dcl}[\,\tau\,;\tau'\,](\,e\,;a\,.\,m\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma, a \sim \tau\,;\tau'\,\{\,k\,;\mathtt{dcl}[\,a\,](\,m\,) \triangleright m \parallel \mu \otimes a \hookrightarrow e\,\}}\ \mathrm{D}_2\text{-}\mathtt{dcl}$$

$$\frac{e\ \mathsf{val}}{\nu\,\Sigma\,\{\,k \triangleright \mathtt{exit}[\,\tau\,][\,a\,](\,e\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,k \blacktriangleleft^{\mathtt{xt}}_{a}\ e \parallel \mu\,\}}\ \mathrm{D}_1\text{-}\mathtt{exit}$$

$$\frac{e \longmapsto e'}{\nu\,\Sigma\,\{\,k \triangleright \mathtt{exit}[\,\tau\,][\,a\,](\,e\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,k \triangleright \mathtt{exit}[\,\tau\,][\,a\,](\,e'\,) \parallel \mu\,\}}\ \mathrm{D}_2\text{-}\mathtt{exit}$$

$$\frac{e\ \mathsf{val}}{\nu\,\Sigma\,\{\,k \triangleright \mathtt{retry}[\,\tau\,][\,a\,](\,e\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,k \blacktriangleleft^{\mathtt{rt}}_{a}\ e \parallel \mu\,\}}\ \mathrm{D}_1\text{-}\mathtt{retry}$$

$$\frac{e \longmapsto e'}{\nu\,\Sigma\,\{\,k \triangleright \mathtt{retry}[\,\tau\,][\,a\,](\,e\,) \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,k \triangleright \mathtt{retry}[\,\tau\,][\,a\,](\,e'\,) \parallel \mu\,\}}\ \mathrm{D}_2\text{-}\mathtt{retry}$$

$$\frac{}{\nu\,\Sigma\,\{\,k\,;\mathtt{bnd}(\ -\ ;x\,.\,m\,) \blacktriangleleft^{\mathtt{xt}}_{a}\ e \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,k \blacktriangleleft^{\mathtt{xt}}_{a}\ e \parallel \mu\,\}}\ \mathrm{D}_2\text{-}\mathtt{exit}$$

$$\frac{}{\nu\,\Sigma\,\{\,k\,;\mathtt{bnd}(\ -\ ;x\,.\,m\,) \blacktriangleleft^{\mathtt{rt}}_{a}\ e \parallel \mu\,\} \longmapsto \nu\,\Sigma\,\{\,k \blacktriangleleft^{\mathtt{rt}}_{a}\ e \parallel \mu\,\}}\ \mathrm{D}_2\text{-}\mathtt{retry}$$

$$\frac{}{\nu\,\Sigma, b \sim \tau'\,;\tau\,\{\,k\,;\mathtt{dcl}[\,b\,](\,m\,) \blacktriangleleft^{\mathtt{xt}}_{a}\ e \parallel \mu \otimes b \hookrightarrow e'\,\} \longmapsto \nu\,\Sigma\,\{\,k \blacktriangleleft^{\mathtt{xt}}_{a}\ e \parallel \mu\,\}}\ \mathrm{D}_4\text{-}\mathtt{exit}$$

$$\frac{}{\nu\,\Sigma, b \sim \tau'\,;\tau\,\{\,k\,;\mathtt{dcl}[\,b\,](\,m\,) \blacktriangleleft^{\mathtt{rt}}_{a}\ e \parallel \mu \otimes b \hookrightarrow e'\,\} \longmapsto \nu\,\Sigma\,\{\,k \blacktriangleleft^{\mathtt{rt}}_{a}\ e \parallel \mu\,\}}\ \mathrm{D}_4\text{-}\mathtt{retry}$$

For a non-matching declaration frame, both commands de-allocates the assignable and remains in backtracking.

Task 2.6 (10 pts). Complete the dynamics by defining the rules for the 2 remaining case where the top frame is a matching declaration frame.

Recovering the loop Another common feature in imperative languages is that (`while`) loops often supports two commands that affects control flow within the loop body

- `continue`: Skips over the remaining commands in the current iteration, and

- $\texttt{break}$: Exits the loop immediately.

In some languages these commands carries an extra operand, usually in the form of a label, signaling which layer of loop to break from (or continue with) in the case of being used in a nested loop.

With the mechanism available we can implement both of these commands. You will define three commands: $\texttt{while}(\,m\,;x\,.\,m_1\,)$, $\texttt{continue}[\,\tau\,](\,x\,)$ and $\texttt{break}[\,\tau\,](\,x\,)$, where $x : \texttt{lp}$ is an expression that identifies the runtime incarnation of the loop.

$$\frac{\Gamma \vdash_\Sigma m \mathrel{\dot{\sim}} \texttt{bool} \qquad \Gamma, x : \texttt{lp} \vdash_\Sigma m_1 \mathrel{\dot{\sim}} \texttt{unit}}{\Gamma \vdash_\Sigma \texttt{while}(\,m\,;x\,.\,m_1\,) \mathrel{\dot{\sim}} \texttt{unit}}$$

$$\frac{}{\Gamma, x : \texttt{lp} \vdash_\Sigma \texttt{continue}[\,\tau\,](\,x\,) \mathrel{\dot{\sim}} \tau} \qquad \frac{}{\Gamma, x : \texttt{lp} \vdash_\Sigma \texttt{break}[\,\tau\,](\,x\,) \mathrel{\dot{\sim}} \tau}$$

For example, in the following code, the `break` would break from the outer loop (i.e. completely break out of both loops), as the argument x comes from the outer loop, therefore identifies it.

$$\texttt{while}(\,m_1\,;x\,.\,\texttt{while}(\,m_2\,;y\,.\,(\,\cdots\,;\texttt{break}[\,\texttt{unit}\,](\,x\,)\,)\,)\,)$$

Similarly, here the `continue` would skip to the next iteration of the inner loop

$$\texttt{while}(\,m_1\,;x\,.\,\texttt{while}(\,m_2\,;y\,.\,(\,\cdots\,;\texttt{continue}[\,\texttt{unit}\,](\,y\,)\,;\cdots)\,)\,)$$

Having the loop identified by an expression (which is a runtime value, contrary to “static” labels) means that it is possible to pass the break point as an argument to functions, providing more flexibility than afforded by languages in practice.

Task 2.7 (20 pts). Define `lp` and the three commands with the `exit` and `retry` commands.

A Concrete Syntax of **MA**

A subset of concrete syntax of **MA** is given below.

Typ	τ	::=	$\cdots$	$\cdots$	everything else
			τ `cmd`	`cmd[tau]`	command types
Exp	e	::=	$\cdots$	$\cdots$	everything else
			`cmd`(m)	`cmd m`	encapsulated commands
Cmd	m	::=	`ret`(e)	`ret (e)`	return
			`bnd`(e ; x . m)	`bndcmdexp x = e in m`	bind command
			`dcl`[τ](e ; a . m)	`local a := e in m`	declare new assignable
			`get`[a]	`get a`	get variable
			`set`[a](e)	`set a := e`	set variable

Figure A.1: **MA** Concrete Syntax

We also provide you some syntactic sugar, some of which you need to implement.

1. `do (e)` is defined as `bnd`(e ; x . `ret`(x)).
2. `cmdlet x = m1 in m2` is defined as `bnd`(`cmd`(m_1) ; x . m_2).
3. Concrete syntax for `if`$_{\mathtt{m}}$ (m) `then` { m_1 } `else` { m_2 } is `if m then m1 else m2`, which is the same concrete syntax for the expression level `if`.
4. Concrete syntax for `while` (m){ m_1 } is `while (m){ m1 }`.
5. Concrete syntax for m_1 ; m_2 is `m1, m2`.
6. Concrete syntax for `ignore` (m) is `ignore m`.

If you have an assignable and you want to use its content to perform some computation, in our core language you need to do

```
local a := 1 in
local b := 2 in {
  bndcmdexp x = cmd { get a } in
    bndcmdexp y = cmd { get b } in
      ret (x + y)
}
```

This can be cumbersome to write once we have more assignables, so we provide a syntactic sugar for this.

1. `[x, y] ( e )` where `x` and `y` are assignables, is expanded into

$$\mathtt{bnd}(\,\mathtt{cmd}(\,\mathtt{get}[\,x\,]\,)\,;\,x\,.\,\mathtt{bnd}(\,\mathtt{cmd}(\,\mathtt{get}[\,y\,]\,)\,;\,y\,.\,\mathtt{ret}(\,e\,)\,)\,)$$

More precisely, in the bracket is a list of assignables, and we bind them sequentially to variables (!) of the same name, and then later on we can use the variables as expressions.

2. `[x, y] { m }` where `x` and `y` are assignables, is expanded into

$$\texttt{bnd}(\,\texttt{cmd}(\,\texttt{get}[\,x\,]\,)\,;x\,.\,\texttt{bnd}(\,\texttt{cmd}(\,\texttt{get}[\,y\,]\,)\,;y\,.\,m\,)\,)$$

.

So the previous example can be written as

```
local a := 1 in
local b := 2 in
  [a, b] (a + b)
```

B Statics for **MA** with *scoped* assignables

The expression level is completely standard

$\boxed{\Gamma \vdash_\Sigma e : \tau}$

$$\ldots \qquad \frac{\Gamma \vdash_\Sigma m \mathrel{\dot\sim} \tau}{\Gamma \vdash_\Sigma \texttt{cmd}(\, m \,) : \tau\ \texttt{cmd}}\ \texttt{cmd-I}$$

$\boxed{\Gamma \vdash_\Sigma m \mathrel{\dot\sim} \tau}$

$$\frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \texttt{ret}(\, e \,) \mathrel{\dot\sim} \tau}\ \texttt{ret-C} \qquad \frac{\Gamma \vdash_\Sigma e : \tau'\ \texttt{cmd} \qquad \Gamma, x : \tau' \vdash_\Sigma m \mathrel{\dot\sim} \tau}{\Gamma \vdash_\Sigma \texttt{bnd}(\, e \,; x \,.\, m \,) \mathrel{\dot\sim} \tau}\ \texttt{bnd-C}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a\sim\tau} \texttt{get}[\, a \,] \mathrel{\dot\sim} \tau}\ \texttt{get-C} \qquad \frac{\Gamma \vdash_{\Sigma, a\sim\tau} e : \tau}{\Gamma \vdash_{\Sigma, a\sim\tau} \texttt{set}[\, a \,](\, e \,) \mathrel{\dot\sim} \texttt{unit}}\ \texttt{set-C}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau \qquad \Gamma \vdash_{\Sigma, a\sim\tau} m \mathrel{\dot\sim} \tau' \qquad \tau\ \mathsf{mobile} \qquad \tau'\ \mathsf{mobile}}{\Gamma \vdash_\Sigma \texttt{dcl}[\, \tau \,](\, e \,; a \,.\, m \,) \mathrel{\dot\sim} \tau'}\ \texttt{dcl-C}$$

$\boxed{\tau\ \mathsf{mobile}}$

$$\frac{}{\texttt{nat}\ \mathsf{mobile}}\ M_{\texttt{nat}} \qquad \frac{}{\texttt{unit}\ \mathsf{mobile}}\ M_{\texttt{unit}} \qquad \frac{}{\texttt{void}\ \mathsf{mobile}}\ M_{\texttt{void}}$$

$$\frac{\tau_1\ \mathsf{mobile} \qquad \tau_2\ \mathsf{mobile}}{\tau_1 + \tau_2\ \mathsf{mobile}}\ M_+ \qquad \frac{\tau_1\ \mathsf{mobile} \qquad \tau_2\ \mathsf{mobile}}{\tau_1 \times \tau_2\ \mathsf{mobile}}\ M_\times$$

$\boxed{\mathcal{M}\ \mathsf{ok}}$ $\boxed{\mu : \Sigma}$ $\boxed{\vdash_\Sigma k \div \tau}$

$$\frac{\vdash_\Sigma k \div \tau \qquad \vdash_\Sigma m \mathrel{\dot\sim} \tau \qquad \mu : \Sigma}{\nu\, \Sigma\, \{\, k \triangleright m \parallel \mu \,\}\ \mathsf{ok}}\ \mathcal{M}_1 \qquad \frac{\vdash_\Sigma k \div \tau \qquad \vdash_\Sigma e : \tau \qquad \mu : \Sigma \qquad e\ \mathsf{val}}{\nu\, \Sigma\, \{\, k \triangleleft e \parallel \mu \,\}\ \mathsf{ok}}\ \mathcal{M}_2$$

$$\frac{\forall_{a \in \Sigma}(\ \vdash_\emptyset \mu(a) : \Sigma(a) \qquad \mu(a)\ \mathsf{val} \qquad \Sigma(a)\ \mathsf{mobile}\ ) \qquad \forall_{a \in \mu}(\ a \in \Sigma)}{\mu : \Sigma}\ \mathcal{M}_3$$

$$\frac{}{\vdash_\emptyset \epsilon \div \tau}\ K_1 \qquad \frac{\vdash_\Sigma k \div \tau \qquad \tau\ \mathsf{mobile}}{\vdash_{\Sigma, a\sim\tau'} k \,; \texttt{dcl}[\, a \,] \div \tau}\ K_2 \qquad \frac{x : \tau \vdash_\Sigma m \mathrel{\dot\sim} \tau' \qquad \vdash_\Sigma k \div \tau'}{\vdash_\Sigma k \,; \texttt{bnd}(\ - \,; x \,.\, m \,) \div \tau}\ K_3$$