Consider the following simply-typed $\lambda$-calculus extended with mutable references:

- **type**: $A ::= P | A \rightarrow A | \text{unit} | \text{int} | \text{ref } A$
- **expression**: $e ::= x | \lambda x : A. e | e \; e | \text{let } x = e \text{ in } e | \text{ref } e | ! e | e := e | 0 | 1 | \cdots$
- **value**: $v ::= \lambda x : A. e | () | l | 0 | 1 | \cdots$
- **store**: $\psi ::= \cdot | \psi, l \mapsto v$

We use syntactic sugar `let $x = e$ in $e$′ for $(\lambda x : A. e′) e$ for some type $A$.

## 1 Arrays of Integers [50 pts]

We want to represent an array of integers as a function taking an index (of type `int`) and returning a corresponding element of the array. We choose a functional representation of arrays by defining type `iarray` for arrays of integers as follows:

\[
iarray = \text{ref } (\text{int} \rightarrow \text{int})
\]

We need the following constructs for arrays:

- **new** : `unit → iarray` for creating a new array.
  
  `new ()` returns a new array of indefinite size; all elements are initialized as 0.

- **access** : `iarray → int → int` for accessing an array.
  
  `access a i` returns the $i$-th element of array $a$.

- **update** : `iarray → int → int → unit` for updating an array.
  
  `update a i n` updates the $i$-the element of array $a$ with integer $n$.

Implement `update`. Fill in the blank:

\[
\text{new } = \lambda_. \text{unit}. \text{ref } \lambda i : \text{int}. 0
\]

\[
\text{access } = \lambda a : \text{iarray}. \lambda i : \text{int}. (!a) i
\]

\[
\text{update } = \lambda a : \text{iarray}. \lambda i : \text{int}. \lambda n : \text{int}.
\]

\[
\text{let } old = !a \text{ in}
\]

\[
a := \lambda j : \text{int}. \text{if } i = j \text{ then } n \text{ else } old \ j
\]

1
2  Operational Semantics [50 pts]

We write \( \text{dom}(\psi) \) for the domain of \( \psi \), i.e., the set of locations mapped to certain values under \( \psi \) as follows:

\[
\begin{align*}
\text{dom}(\cdot) &= \emptyset \\
\text{dom}(\psi, l \mapsto v) &= \text{dom}(\psi) \cup \{l\}
\end{align*}
\]

We write \([l \mapsto v]\psi\) for the store obtained by updating the contents of \( l \) in \( \psi \) with \( v \). Note that in order for \([l \mapsto v]\psi\) to be defined, \( l \) must be in \( \text{dom}(\psi) \):

\[
[l \mapsto v](\psi', l \mapsto v') = \psi', l \mapsto v
\]

We write \( \psi(l) \) for the value to which \( l \) is mapped under \( \psi \); in order for \( \psi(l) \) to be defined, \( l \) must be in \( \text{dom}(\psi) \):

\[
(\psi', l \mapsto v)(l) = v
\]

Also, we use the following reduction judgment which carries a store along with an expression being reduced:

\[
e | \psi \rightarrow e' | \psi' \Leftrightarrow \text{e with store } \psi \text{ reduces to } e' \text{ with store } \psi'
\]

Using these definitions, complete the reduction rules for references.

\[
\begin{align*}
\frac{e | \psi \rightarrow e' | \psi'}{\text{ref } e | \psi \rightarrow \text{ref } e' | \psi'} & \quad \text{Ref} \\
\frac{l \not\in \text{dom}(\psi)}{\text{ref } v | \psi \rightarrow l | \psi, l \mapsto v} & \quad \text{Ref'} \\
\frac{e | \psi \rightarrow e' | \psi'}{\text{!e } | \psi \rightarrow \text{!e' } | \psi'} & \quad \text{Deref} \\
\frac{\psi(l) = v}{\text{!l } | \psi \rightarrow v | \psi} & \quad \text{Deref'} \\
\frac{e | \psi \rightarrow e'' | \psi'}{e := e' | \psi \rightarrow e'' := e' | \psi'} & \quad \text{Assign} \\
\frac{e | \psi \rightarrow e' | \psi'}{l := e | \psi \rightarrow l := e' | \psi'} & \quad \text{Assign'} \\
\frac{l := v | \psi \rightarrow () | [l \mapsto v]\psi}{l := v | \psi \rightarrow () | [l \mapsto v]\psi} & \quad \text{Assign''}
\end{align*}
\]