From Dynamic Binding to State via Modal Possibility

Aleksandar Nanevski
Computer Science Department, Carnegie Mellon University

PPDP 2003, Uppsala
August 27, 2003
Monadic treatment of state

- If \( S \) is the type of state, state monad \( T \) defined as

\[
TA = S \rightarrow (A \times S)
\]

- Term constructors corresponding to \( T \)
  - allocation of new memory
  - reads from memory locations
  - writes into memory locations

- Stateful operations confined to monadic types

- Monadic computation is *single-threaded*
  - total ordering between effects
Requirements of state operations

- Writes must be single-threaded, so that the final value of the written location is well-defined.
- Reads between any two consecutive writes may be performed out of order.
- If assume memory is not initialized upon allocation:
  - a read should be allowed only if a location has been written into.
Can we assign different types to different state operations?
How should the types interact?
Is there a logic behind such a system?
Can we assign different types to different state operations?
How should the types interact?
Is there a logic behind such a system?
In this presentation:

- Modal logic for a system with a type distinction between allocation, reads and writes
- allocation of uninitialized second-class locations
- dereferencing of only initialized locations
- location update can be:
  - destructive (i.e. with global scope)
  - non-destructive (i.e. with delimited scope)
- Introduction ✓
- Names as memory locations
- Modal logic for state
- Categorical structure of modalities
- Related and future work
Names

- A.k.a.: labels, atoms, nonces, symbols, tags, indeterminates
- Names stand for particular locations
- Terms
  \[ e ::= x \mid \lambda x. e \mid e_1 e_2 \mid X \mid \ldots \]
- Important:
  names are generated dynamically
- Type system enforces discipline in name generation and propagation
We associate expression $e$ with its type $A$:

$$\Sigma; \Delta \vdash e : A$$

- $\Sigma$ : types for names
- $\Delta$ : types for variables
Type system

- We associate expression $e$ with its type $A$ and support $C$

  $$\Sigma; \Delta \vdash e : A[C']$$

- $\Sigma$: types for names
- $\Delta$: types for variables
- Support == set of names/locations that $e$ may read
- Pure expressions = empty support
- Type safety: well-typed pure expressions do not get stuck
Explicit substitutions

- Location read == extend support
- Location write == shrink support
- Locations written using explicit substitutions

\[
\text{Substitutions } \Theta ::= \cdot \mid (\Theta, X \rightarrow e) \\
\text{Terms } e ::= \ldots \mid \langle \Theta \rangle e
\]

- Scope of a substitution is delimited

\[
\langle X \rightarrow 1 \rangle (\langle X \rightarrow 2 \rangle X^2, X^2)
\]

- Substitutions model non-destructive update
Let $X, Y : \text{int} \iff$ uninitialized integer locations

<table>
<thead>
<tr>
<th>term</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X^2 + Y^2$</td>
<td>$X, Y$</td>
</tr>
<tr>
<td>$\langle X \to 1 \rangle (X^2 + Y^2)$</td>
<td>$Y$</td>
</tr>
<tr>
<td>$\langle X \to 1, Y \to 2 \rangle (X^2 + Y^2)$</td>
<td>empty</td>
</tr>
<tr>
<td>$\langle X \to 1 \rangle (\langle X \to 2 \rangle X^2, X^2)$</td>
<td>empty</td>
</tr>
</tbody>
</table>

Type-safety: well-typed pure expressions do not get stuck
• Let $X, Y : \texttt{int} \equiv \text{unfinished integer locations}$

<table>
<thead>
<tr>
<th>term</th>
<th>support</th>
<th>evaluates to</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X^2 + Y^2$</td>
<td>$X, Y$</td>
<td></td>
</tr>
<tr>
<td>$\langle X \rightarrow 1 \rangle (X^2 + Y^2)$</td>
<td>$Y$</td>
<td></td>
</tr>
<tr>
<td>$\langle X \rightarrow 1, Y \rightarrow 2 \rangle (X^2 + Y^2)$</td>
<td>empty</td>
<td>5</td>
</tr>
<tr>
<td>$\langle X \rightarrow 1 \rangle (\langle X \rightarrow 2 \rangle X^2, X^2)$</td>
<td>empty</td>
<td>(4,1)</td>
</tr>
</tbody>
</table>

• Type-safety: well-typed pure expressions do not get stuck
At run time we need to allocate new memory

Two new constructs:

Terms \[ e ::= \ldots | \nu X : A. \ e | \text{choose } e \]

- *declare* fresh name \( X : A \) in \( e \) \( \equiv \nu X : A . \ e \)
- *allocate* fresh name \( \equiv \text{choose } e \)

Example: allocate new memory cell \( X \)

\[
\text{choose } (\nu X : \text{int.} \langle X \to 10 \rangle X)
\]
Outline

- Introduction ✓
- Names as memory locations ✓
- Modal logic for state
- Categorical structure of modalities
- Related and future work
State and higher-order functions

- How to track supports inherited from function arguments?
- Hint comes from Curry-Howard isomorphism:
  - types should specify program behavior
- Therefore:
  
  *stateful computations should be marked by the type system*
Assume

- $E_1$ is a computation reading from location $X:A$ before returning a value of type $B$
- $E_2$ is a computation writing into location $X:A$ before returning a value of type $B$

$E_1$ is a function from store to values
$E_2$ pairs up a store and a value

Conclusion:
- Reading == universal quantification
- Writing == existential quantification
Modal logic

- Reasoning about properties of nodes in a directed graph
- Modal operators □ and ◇
  - □ = modal necessity
    - Proposition □A is true at a node w iff A is true at all nodes accessible from w
  - ◇ = modal possibility
    - Proposition ◇A is true at a node w iff A is true at some node accessible from w
Modal types for state

- type $\square_C A$
  
  $\Rightarrow$

  suspended computation of type $A$ possibly reading locations in $C$

- type $\Diamond_C A$
  
  $\Rightarrow$

  suspended computation of type $A$ writing into locations listed in $C$
• Terms $e ::= \ldots \mid \text{box } e \mid \text{let box } u = e_1 \text{ in } e_2$
• Boxed term is “packaged” to be “shipped” further
• $\Box$-introduction

\[
\Sigma; \Delta \vdash e : A [D] \\
\Sigma; \Delta \vdash \text{box } e : \Box D A [\ ]
\]

• Notice:
  – boxed expression itself is effect-free
• boxed expressions are not evaluated
Typing necessity

- Using a boxed computation: let-box construct
- let-box == unwrap the package, but do not peek into it
- Examples:

  \[
  \text{let box } u = \text{box } e_1 \text{ in } e_2 \mapsto [e_1/u]e_2 \\
  \text{let box } u = \text{box } (1 + 1) \text{ in } (\text{box } u) \mapsto \text{box } (1 + 1) \\
  \text{let box } u = \text{box } (1 + 1) \text{ in } u \mapsto 1 + 1 \mapsto 2
  \]

- Notice: binding to \( u \) is “by-name”
• □-elimination

\[
\Sigma; \Delta \vdash e_1 : \Box_D A \quad \Sigma; (\Delta, u:A[D]) \vdash e_2 : B \\
\Sigma; \Delta \vdash \text{let box } u = e_1 \text{ in } e_2 : B
\]

• Notice:
  – context \( \Delta \) types “unwrapped” expressions
  – \( \Delta \) stores the type and support of variables
Typing necessity

- □-elimination

\[
\Sigma; \Delta \vdash e_1 : \square_D A [C] \quad \Sigma; (\Delta, u : A[D]) \vdash e_2 : B [C] \\
\Sigma; \Delta \vdash \text{let box } u = e_1 \text{ in } e_2 : B [C]
\]

- Notice:
  - context \( \Delta \) types “unwrapped” expressions
  - \( \Delta \) stores the type and support of variables
  - Conclusion inherits support of the premises
Example

let fun f (y : int) : □_X int = box (X + y)
  box u = f(1)

in

  ⟨X → 0⟩
  ⟨⟨X → 1⟩u, u⟩

end

- range type of \( f \) marks the effect of reading \( X \)
- variable \( u \) “unpacked” under two different substitutions
- computation is effectful, but final result is pure
let fun f (y : int) : □X int = box (X + y)
    box u = f(1)   == box (X + 1)
in
    ⟨X → 0⟩
    (⟨X → 1⟩ u, u)
end

- range type of $f$ marks the effect of reading $X$
- variable $u$ “unpacked” under two different substitutions
- computation is effectful, but final result is pure
let fun f (y : int) : □_X int = box (X + y)
  box u = f(1)  == box (X + 1)
in
  ⟨X → 0⟩
  ⟨⟨X → 1⟩u, u⟩  ==  ⟨⟨X → 1⟩X+1, X+1⟩
end

- range type of $f$ marks the effect of reading $X$
- variable $u$ “unpacked” under two different substitutions
- computation is effectful, but final result is pure
let fun f (y : int) : □_X int = box (X + y)
    box u = f(1)  == box (X + 1)
in
    ⟨X → 0⟩
    ⟨⟨X → 1⟩u, u⟩  == (2, 1)
end

- range type of \( f \) marks the effect of reading \( X \)
- variable \( u \) “unpacked” under two different substitutions
- computation is effectful, but final result is pure
Summary of the □-fragment

- □_C == computation reading from store C
  - universal quantification over substitutions for names in C
  - substitution application == specialization of universal quantifier

- Location update by explicit substitutions is non-destructive
- Only initialized locations can be dereferenced
- Type safety: pure terms do not get stuck
- Implements a type-safe calculus of dynamic binding
Typing possibility

- New judgment
  \[ \Sigma; \Delta \vdash f \div_D A \]

- Meaning of the judgment:
  1. \( f \) writes into locations \( D \), and
  2. returns an \( A \) value

- Terms \( e ::= \ldots | \text{dia } f \)

- Judgment is internalized using \( \Diamond \)

\[
\Sigma; \Delta \vdash f \div_D A \\
\Sigma; \Delta \vdash \text{dia } f : \Diamond_D A
\]
• New judgment

\[ \Sigma; \Delta \vdash f \div_D A[C] \]

• Meaning of the judgment:
  1. \( f \) writes into locations \( D \), and
  2. returns an \( A \) value

• Terms \( e ::= \ldots | \text{dia } f \)

• Judgment is internalized using \( \Diamond \)

\[
\begin{align*}
\Sigma; \Delta & \vdash f \div_D A[C] \\
\Sigma; \Delta & \vdash \text{dia } f : \Diamond_D A[C]
\end{align*}
\]
Typing possibility

- Recall: $\Diamond$ is an existential quantifier over store (i.e. over explicit substitutions)
- Frames $f ::= [\Theta, e] \mid \textbf{let dia } x = e \textbf{ in } f$
- Canonical frame is a pair: [explicit substitution, term]
  \[
  \Sigma; \Delta \vdash e_1 : A \quad \Sigma; \Delta \vdash e_2 : B[X] \quad X:A \in \Sigma \\
  \Sigma; \Delta;\vdash [(X \to e_1), e_2] \div_X B
  \]
- Explicit substitution $\langle X \to e_1 \rangle ==$ to be written into the store
- Term $e_2 ==$ computation in the new store
Typing possibility

- Elimination form threads the existential
  
  \[ \Sigma; \Delta \vdash e : \Diamond_C A \quad \Sigma; (\Delta, x:A) \vdash f \Downarrow_D B [C] \]

  \[ \Sigma; \Delta \vdash \text{let dia } x = e \text{ in } f \Downarrow_D B \]

- Pairs and let dia are the only proof-terms for the \( \Downarrow \) judgment

- Consequences:
  - Computations are explicitly single-threaded
  - Explicit substitution embedded in \( e \) has global scope; it can be implemented destructively
• Elimination form threads the existential

\[
\Sigma; \Delta \vdash e : \Diamond_C A [C'] \\
\Sigma; (\Delta, x:A) \vdash f \Downarrow_D B [C]
\]

\[
\Sigma; \Delta \vdash \text{let} \ dia \ x = e \ \text{in} \ f \Downarrow_D B [C']
\]

• Pairs and let dia are the only proof-terms for the $\Downarrow$ judgment

• Consequences:
  – Computations are explicitly single-threaded
  – Explicit substitution embedded in $e$ has global scope; it can be implemented destructively
Example revisited

<table>
<thead>
<tr>
<th>□-fragment</th>
<th>◊-fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>let fun f (y) = box (X + y)</td>
<td>let fun f (y) = box (X + y)</td>
</tr>
<tr>
<td>box u = f(1)</td>
<td>box u = f(1)</td>
</tr>
<tr>
<td>in</td>
<td></td>
</tr>
<tr>
<td>⟨X → 0⟩</td>
<td>dia dummy = dia [⟨X → 0⟩, ()]</td>
</tr>
<tr>
<td>⟨⟨X → 1⟩u, u⟩</td>
<td>val z = u</td>
</tr>
<tr>
<td>end</td>
<td>dia w = dia [⟨X → 1⟩, u]</td>
</tr>
<tr>
<td></td>
<td>in</td>
</tr>
<tr>
<td></td>
<td>[(w, z)]</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>

- In ◊-fragment, references to u reordered to match their substitutions
<table>
<thead>
<tr>
<th>□-fragment</th>
<th>◊-fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>let fun f (y) = box (X + y)</td>
<td>let fun f (y) = box (X + y)</td>
</tr>
<tr>
<td>box u = f(1)</td>
<td>box u = f(1) == box (X + 1)</td>
</tr>
<tr>
<td>in</td>
<td></td>
</tr>
<tr>
<td>(\langle X \rightarrow 0\rangle)</td>
<td>dia dummy = dia [(\langle X \rightarrow 0\rangle, ())]</td>
</tr>
<tr>
<td>((\langle X \rightarrow 1\rangle u, u))</td>
<td>val z = u</td>
</tr>
<tr>
<td>end</td>
<td>dia w = dia [(\langle X \rightarrow 1\rangle, u)]</td>
</tr>
<tr>
<td></td>
<td>in [(w, z)]</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>

- In ◊-fragment, references to \(u\) reordered to match their substitutions
Example revisited

- In $\square$-fragment, references to $u$ reordered to match their substitutions.

<table>
<thead>
<tr>
<th>$\square$-fragment</th>
<th>$\Diamond$-fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>let fun f (y) = box (X + y)</td>
<td></td>
</tr>
<tr>
<td>box u = f(1)</td>
<td></td>
</tr>
<tr>
<td>in</td>
<td></td>
</tr>
<tr>
<td>$\langle X \rightarrow 0 \rangle$</td>
<td></td>
</tr>
<tr>
<td>$(\langle X \rightarrow 1 \rangle u, u)$</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>let fun f (y) = box (X + y)</td>
<td></td>
</tr>
<tr>
<td>box u = f(1) $\Rightarrow$ box (X + 1)</td>
<td></td>
</tr>
<tr>
<td>dia dummy = dia [\langle X \rightarrow 0 \rangle, ()]</td>
<td></td>
</tr>
<tr>
<td>val z = u $\Rightarrow$ X+1</td>
<td></td>
</tr>
<tr>
<td>dia w = dia [\langle X \rightarrow 1 \rangle, u]</td>
<td></td>
</tr>
<tr>
<td>in</td>
<td></td>
</tr>
<tr>
<td>[(w, z)]</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>
Example revisited

-fragment | ◇-fragment

let fun f (y) = box (X + y)  
  box u = f(1)  
  in  
  \langle X \to 0 \rangle  
  (\langle X \to 1 \rangle u, u)  
end

let fun f (y) = box (X + y)  
  box u = f(1)  
  == box (X + 1)  
  dia dummy = dia [\langle X \to 0 \rangle, ()]  
  val z = u  
  == 1  
  dia w = dia [\langle X \to 1 \rangle, u]  
  in  
  [(w, z)]  
end

- In ◇-fragment, references to \( u \) reordered to match their substitutions
Example revisited

- In $\Box$-fragment, references to $u$ reordered to match their substitutions.

\[
\begin{align*}
\Box\text{-fragment} & \quad \Diamond\text{-fragment} \\
\text{let fun f (y) = box (X + y)} & \quad \text{let fun f (y) = box (X + y)} \\
\text{  box u = f(1)} & \quad \text{  box u = f(1) == box (X + 1)} \\
\text{in} & \quad \text{dia dummy = dia [⟨X → 0⟩, ()]} \\
\langle X → 0 \rangle & \quad \text{val z = u == 1} \\
\langle X → 1 \rangle u, u) & \quad \text{dia w = dia [⟨X → 1⟩, u] == [⟨X→1⟩,X+1]} \\
\text{end} & \quad \text{in} \\
\text{[(w, z)]} & \quad \text{end}
\end{align*}
\]
### Example revisited

<table>
<thead>
<tr>
<th>□-fragment</th>
<th>◊-fragment</th>
</tr>
</thead>
</table>
| let fun f (y) = box (X + y)  
  box u = f(1)  
  in  
  \[\langle X \rightarrow 0 \rangle\]  
  \(\langle X \rightarrow 1 \rangle u, u\)  
  end | let fun f (y) = box (X + y)  
  box u = f(1)  
  \[\text{val } z = u \quad \text{== 1}\]  
  \[\text{dia } w = \text{dia } [\langle X \rightarrow 1 \rangle, u] \quad \text{==[}\langle X\rightarrow1\rangle,2]\]  
  \(\text{in}\)  
  \([w, z]\)  
  end |

- In ◊-fragment, references to \(u\) reordered to match their substitutions
Example revisited

<table>
<thead>
<tr>
<th>□-fragment</th>
<th>◊-fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>let fun f (y) = box (X + y) box u = f(1)</td>
<td>let fun f (y) = box (X + y) box u = f(1) == box (X + 1)</td>
</tr>
<tr>
<td>in</td>
<td></td>
</tr>
<tr>
<td>⟨X → 0⟩</td>
<td>dia dummy = dia [⟨X → 0⟩, ()]</td>
</tr>
<tr>
<td>⟨X → 1⟩ u, u</td>
<td>val z = u == 1</td>
</tr>
<tr>
<td>end</td>
<td>dia w = dia [⟨X → 1⟩, u] == [⟨X→1⟩,2]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(w, z)] == [(2, 1)]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>

- In ◊-fragment, references to $u$ reordered to match their substitutions

From Dynamic Binding to State via Modal Possibility – p.24
Summary of $\Diamond$-fragment

- $\Diamond_C$ == computation writing into store $C$
  - existential quantification over substitutions
- Because of monadic character of $\Diamond$, the substitutions can be implemented destructively
- When combined with the $\Box$-fragment, destructive and non-destructive state update coexist
Outline

- Introduction ✓
- Names as memory locations ✓
- Modal logic for state ✓
- Categorical structure of modalities
- Related and future work

From Dynamic Binding to State via Modal Possibility – p.26
Categorical structure of modalities

- □ is a comonad

\[ f_1 : \square A \to A = \lambda x. \text{let box } u = x \text{ in } u \]

\[ f_2 : \square A \to \square \square A = \lambda x. \text{let box } u = x \text{ in box (box } u) \]

\[ f_3 : \square (A \to B) \to \square A \to \square B = \lambda x. \lambda y. \text{let box } u = x \text{ in let box } v = y \text{ in box } (u v) \]
Categorical structure of modalities

- □ is a comonad
  
  \[ f_1 : \Box A \to A = \lambda x. \text{let } \text{box } u = x \text{ in } u \]

  \[ f_2 : \Box_C A \to \Box_C \Box C A = \lambda x. \text{let } \text{box } u = x \text{ in } \text{box } (\text{box } u) \]

  \[ f_3 : \Box_C (A \to B) \to \Box_D A \to \Box_{C,D} B = \lambda x. \lambda y. \text{let } \text{box } u = x \text{ in } \text{let } \text{box } v = y \text{ in } \text{box } (u v) \]

- \[ \Box_C A \to \Box_D A \quad C \subseteq D \]

- and others...
Categorical structure of modalities

- Need $A \to \Box A$ to coerce values into computations

- Consider fragment where function arguments are boxed

  \[
  (A \to B)^+ = \Box A^+ \to B^+
  \]

  \[
  (\Box A)^+ = \Box A^+
  \]

  \[
  (\Diamond A)^+ = \Diamond \Box A^+
  \]

- $\Box$ still a comonad
Categorical structure of modalities

- $\Diamond$ is a monad

\[
g_1 : A \to \Diamond A = \lambda x. \text{dia } [x]
g_2 : \Diamond \Diamond A \to \Diamond A = \lambda x. \text{dia } (\text{let } \text{dia } y = x \text{ in let } \text{dia } z = y \text{ in } [z])
g_3 : (A \to B) \to \Diamond A \to \Diamond B = \lambda e_1. \lambda e_2. \text{dia } (\text{let } \text{val } u = e_1 \text{ in let } \text{dia } x = e_2 \text{ in } [u \ x])
\]
Categorical structure of modalities

- ◊ is a monad

\[ g_1 : A \rightarrow ◊A = \lambda x. \text{dia } [x] \]

\[ g_2 : ◊C ◊D A \rightarrow ◊D A = \lambda x. \text{dia } (\text{let dia } y = x \text{ in let dia } z = y \text{ in } [z]) \]

\[ g_3 : □C (A \rightarrow B) \rightarrow ◊D A \rightarrow ◊D B = (\text{where } C \subseteq D) \]

\[ \lambda e_1. \lambda e_2. \text{dia } (\text{let } \text{box } u = e_1 \text{ in let dia } x = e_2 \text{ in } [u x]) \]

- ◊D A \rightarrow ◊C A \quad C \subseteq D

- and others...
Representation of state in constructive modal logic

Split the state monad $TA = C \rightarrow (A \times C)$ into
  
  - modality $\Box_C$ for reading
  - modality $\Diamond_C$ for writing

Names == dynamically allocated memory

Typing discipline: only initialized locations are read from

$\Box$-fragment == dynamic binding; locations updated non-destructively by explicit substitutions

$\Diamond$-fragment adds destructive update

$\Diamond$ is a monad, and $\Box$ is a comonad
Future work

- Integrating nominal modal types into a realistic language
  - other effects, like control effects and I/O
  - type and support inference
  - universal and existential abstraction over supports
- First-class locations
- Equational theory
  - also for a call-by-name variant
Some related work

- Monads [Moggi ’91, Wadler ’95]
- Natural deduction for constructive S4 [Pfenning, Davies’99]
- Embedding of monads into modal logic [Kobayashi’97], [Pfenning, Davies’99]
- Categorical models for modal logic [Kobayashi’97], [Bierman, de Paiva’00], [Alechina,Mendler,de Paiva,Ritter’01]
- Nominal logic, FreshML [Pitts,Gabbay’00]
Representation of state in constructive modal logic

- Split the state monad \( T \ A = C \rightarrow (A \times C) \) into
  - modality \( \boxdot_C \) for reading
  - modality \( \Diamond_C \) for writing

- Names == dynamically allocated memory
- Typing discipline: only initialized locations are read from
- \( \square \)-fragment == dynamic binding; locations updated non-destructively by explicit substitutions
- \( \Diamond \)-fragment adds destructive update
- \( \Diamond \) is a monad, and \( \square \) is a comonad
Categorical structure of modalities

- In constructive S4, □ is a monoidal comonad and ◊ is a □-strong monad

\[
\begin{align*}
\square A & \Rightarrow A \\
\square A & \Rightarrow \boxdot \square A \\
\square (A \Rightarrow B) & \Rightarrow \square A \Rightarrow \square B
\end{align*}
\]

\[
\begin{align*}
A & \Rightarrow ◊ A \\
◊◊ A & \Rightarrow ◊ A \\
\square (A \Rightarrow B) & \Rightarrow ◊ A \Rightarrow ◊ B
\end{align*}
\]

- There is no map \( A \Rightarrow □ A \) from values into comonadic computations
Categorical structure of modalities

- Typing judgment for constructive S4 has two variable contexts:
  \[ \Delta; \Gamma \vdash e : A \]
- \( \Gamma \) variables introduced by \( \lambda \)
- \( \Delta \) variables introduced by let box
- Box-introduction rule erases \( \Gamma \) in the premise
  \[ \Delta; \cdot \vdash e : A \]
  \[ \Delta; \Gamma \vdash \text{box } e : \Box A \]
- Diamond-elimination rule erases \( \Gamma \) in the premise
  \[ \Delta; \Gamma \vdash e : \Diamond A \]
  \[ \Delta; x:A \vdash f \triangleright B \]
  \[ \Delta; \Gamma \vdash \text{let dia } x = e \text{ in } f \triangleright B \]
Categorical structure of modalities

- We used a fragment where function arguments are assumed boxed.
- Interpretation \((A \rightarrow B)^+ = \Box A^+ \Rightarrow B^+\)
  \[ (\Box A)^+ = \Box A^+ \]
  \[ (\Diamond A)^+ = \Diamond \Box A^+ \]
- Provides coercion \(A \rightarrow \Box A\) (i.e., \(\Box A^+ \Rightarrow \Box A^+\))
- \(\Box\) is equal to \(\Box\) (i.e., still a comonad); \(\Diamond\) is a strong monad
Categorical structure of modalities

- We used a fragment where function arguments are assumed boxed.
- Interpretation: 
  
  \[(A \to B)^+ = \Box A^+ \Rightarrow B^+
  
  (\Box A)^+ = \Box A^n
  
  (\Diamond A)^+ = \Diamond \Box A^+
  
- Provides coercion \( A \to \Box A \) (i.e., \( \Box A^+ \Rightarrow \Box A^+ \))
- \( \Box \) is equal to \( \Box \) (i.e., still a comonad); \( \Diamond \) is a strong monad.
- Reinterpretation eliminates the need for \( \Gamma \)

\[
\Delta \vdash e : A \\
\Delta \vdash \text{box } e : \Box A
\]

\[
\Delta \vdash e : \Diamond A \\
\Delta, x : A \vdash f : B
\]

\[
\Delta \vdash \text{let } \text{dia } x = e \text{ in } f : B
\]
Dynamic name generation

- At run time we need to allocate new memory
- Two new constructs:

  Terms \( e ::= \ldots \mid \nu X : A. \ e \mid \text{choose } e \)

- \textit{declare} fresh name \( X : A \) in \( e \Rightarrow \nu X : A. \ e \)
- \textit{allocate} fresh name \( \Rightarrow \text{choose } e \)
- Example: allocate new memory cell \( X \)

  \[
  \text{choose } (\nu X : \text{int.} \langle X \rightarrow 10 \rangle X)
  \]
Name declaration and allocation

- Important:
  - `choose (νX. e)` allocates `X` to be used *exclusively* in `e`
  - type system must prevent `X` from escaping, otherwise type soundness fails
Important:
- `choose (\nu X. e)` allocates `X` to be used *exclusively* in `e`
- type system must prevent `X` from escaping, otherwise type soundness fails

Example: the following steps to `X` and then gets stuck

```plaintext
let val y = choose (\nu X : int. X) in y
```
Name declaration and allocation

- Important:
  - \texttt{choose } (\nu X. \ e) \texttt{ allocates } X \texttt{ to be used } exclusively \texttt{ in } e
  - type system must prevent \( X \) from escaping, otherwise type soundness fails

- Example: the following steps to \( X \) and then gets stuck
  \[
  \texttt{let val } y = \texttt{choose } (\nu X:\text{int. } X) \texttt{ in } y
  \]

- Requirement: \( X \) is not in support of \( e \)
  - all instances of \( X \) in \( e \) must be substituted away
Typing for name generation

- **Irrelevant** implication $A \rightsquigarrow B$
- $\rightsquigarrow$ introduction

\[
\begin{align*}
(\Sigma, X:A); \Delta \vdash e : B [C] & \quad X \not\in \text{fn}(A, B, C, \Delta) \\
\Sigma; \Delta \vdash \nu X:A. e : A \rightsquigarrow B [C]
\end{align*}
\]

- Side-condition ensures all uses of $X$ in $e$ are handled or dead code
- $\rightsquigarrow$ elimination

\[
\begin{align*}
\Sigma; \Delta \vdash e : A \rightsquigarrow B [C] \\
\Sigma; \Delta \vdash \text{choose } e : B [C]
\end{align*}
\]