A modal foundation for functional programming with effects

Aleksandar Nanevski
Computer Science Department, Carnegie Mellon University

IT University, Copenhagen
September 01, 2003
Functional programming

Defining characteristics:

- Program == mathematical function relating input and output
- Types == specification of domain and range (e.g., \( f: A \rightarrow B \))
- Support for code structuring and reuse:
  - function composition
  - higher-order functions (i.e., take other functions as input)
- Functions == basic building blocks for modular programming (in the small)
- Types == statically enforced interfaces
Impure functional programming

- Partially abandons mathematical nature of function
  - functions not only compute values, but also exact *effects* on the environment
- Justified by tremendous increase in expressiveness; in fact, absolutely essential for practical programming
- Example effects:
  - input/output
  - assignment to state
  - control flow changes:
    - exceptions, catch/throw, continuations
Pure functional programming

If we adhere to the mathematical ideal of function, then:

- Functions *only* compute values
- Prototypical example: $\lambda$-calculus
- Strong connection with logic
  
  programs $\equiv$ proofs in logic of types

- Consequence:
  - types express properties of programs (type safety)
  - easy to reason about programs
  - ordering of program steps is irrelevant
  - flexible program transformation and optimization
Problems with effects

- Adding effects breaks connection with logic
  - types express less accurate program properties
  - reasoning about programs becomes hard
  - as does program transformation and optimization
- There’s more...
Problems with effects

- Nature of effects is not well understood, resulting in:
  - repeated or partial functionality of constructs
    - declaring new exceptions vs. allocating memory vs. declaring destination labels for jumps
    - allocations of only *initialized* memory
  - non-orthogonal and even inconsistent extensions
    - breaks program/type relationship
  - comparing effects very hard
    - how to combine exceptions with continuations?
    - how to relate destructive and non-destructive state update?
Integrating pure and impure

- We require a framework for representing effects, which is:
  - Uniform
    - common mechanisms for common effect features
  - Expressive
    - various aspects of various effects
  - Logical
    - types express effect properties of programs
- Many frameworks were proposed, with various degrees of simplicity, expressiveness and adherence to requirements
- Lot of space for improvements...
A framework for representing effects, with:

- **Uniformity**
  - exceptions, catch/throw, composable continuations, state

- **Expressiveness**
  - distinguish between global and handleable effects
  - dynamic generation of new effect instances

- **Logicality**
  - constructive modal logic $S4 + names$
Outline

- Introduction ✓
- Modal treatment of state
  - names as memory locations
  - modal types for state
- Modal treatment of control effects
- 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
Explicit substitutions

- Reading *uninitialized* location == extend support
- Writing into a location == shrink support
- Locations written using explicit substitutions
  
  Substitutions $\Theta ::= \cdot \mid (\Theta, X \rightarrow e)$
  
  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 $X, Y : int$ be 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>$(X \rightarrow 1)(X^2 + Y^2)$</td>
<td>$Y$</td>
</tr>
<tr>
<td>$(X \rightarrow 1, Y \rightarrow 2)(X^2 + Y^2)$</td>
<td>empty</td>
</tr>
<tr>
<td>$(X \rightarrow 1)((X \rightarrow 2)X^2, X^2)$</td>
<td>empty</td>
</tr>
</tbody>
</table>

Type safety: only initialized locations are read from
• Let $X, Y : int$ == uninitialized 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>$X, Y$</td>
</tr>
<tr>
<td>$(X \rightarrow 1)(X^2 + Y^2)$</td>
<td>$Y$</td>
<td>$Y$</td>
</tr>
<tr>
<td>$(X \rightarrow 1, Y \rightarrow 2)(X^2 + Y^2)$</td>
<td>empty</td>
<td>5</td>
</tr>
<tr>
<td>$(X \rightarrow 1)(\langle X \rightarrow 2 \rangle X^2, X^2)$</td>
<td>empty</td>
<td>$(4,1)$</td>
</tr>
</tbody>
</table>

• Type safety: only initialized locations are read from
Dynamic name generation

- 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 \( e == \nu X:A. \ e \)
- `allocate` fresh name == `choose` \( e \)
- Example: generate new memory cell \( X \)
  
  \[
  \text{choose} (\nu X:\text{int.} \\
  \langle X \rightarrow 10 \rangle X)
  \]
Name declaration and allocation

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

- 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: following term 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) allocates X to be used exclusively in e}
  - type system must prevent X from escaping, otherwise type soundness fails

- Example: following term steps to X and then gets stuck

\[
\text{let val } y = \text{choose (\nu X : \text{int. } X) in } y
\]

- Requirement: X is not in support of e
  - all instances of X in e must be initialized
Outline

- Introduction ✓
- Modal treatment of state
  - names as memory locations ✓
  - modal types for state
- Modal treatment of control effects
- Categorical structure of modalities
- Related and future work
How to treat effects inherited from function arguments?

Hint comes from logic:
– types should specify program behavior

Therefore:

effectful computations should be marked by the type system
Requirements of state operations

- Operations on state:
  - allocation of memory cells
  - reading from a location
  - writing into a location

Writes must be ordered, so that the final value stored into a memory location is well-defined. Reads between two consecutive writes can be executed out of order. If memory not initialized upon allocation:
- a read allowed only on initialized locations
Operations on state:
- allocation of memory cells
- reading from a location
- writing into a location

Writes must be ordered, so that the final value stored into a memory location is well-defined.

Reads between two consecutive writes can be executed out of order.

If memory not initialized upon allocation:
- a read allowed only on initialized locations
Separating the types

- Can we ascribe different types to different state operations?
- How should the types interact?
- Is there a logic behind such a system?
Logical analogy

- 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 $\square$ and $\Diamond$
- $\square$ == modal *necessity*
  
  proposition $\square A$ is true at a node $w$ iff
  $A$ is true at *all* nodes accessible from $w$

- $\Diamond$ == modal *possibility*
  
  proposition $\Diamond A$ is true at a node $w$ iff
  $A$ is true at *some* node accessible from $w$
• type $\ Diamond_c A$
  
  $\equiv$

  \textit{suspended} computation of type $A$ possibly reading locations in $C$

• type $\Box_c A$
  
  $\equiv$

  \textit{suspended} computation of type $A$ writing into locations listed in $C$
Typing necessity

- Terms \( e ::= \ldots | \text{box } e | \text{let box } u = e_1 \text{ in } e_2 \)
- Boxed term is “packaged” to be “shipped” further
- Support lists dereferenced locations
- \( \Box \)-introduction

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

- Notice:
  - boxed expression itself is effect-free
- boxed expressions are not evaluated
• Using a boxed computation: let-box construct
• let-box == unwrap the package, but do not peek into it
• Examples:

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

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

\[
\Sigma; \Delta \vdash e_1 : \square_D A \quad \Sigma; (\Delta, u:A[D]) \vdash e_2 : B
\]

\[
\Sigma; \Delta \vdash \text{let box } u = e_1 \text{ in } e_2 : B
\]

• Notice:
  – context \( \Delta \) types “unwrapped” expressions
  – \( \Delta \) stores the type and support of variables
- □-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
Representation of store

- Recall: $\square_C$ == universal quantification over store of signature $C$
- Explicit substitutions assign values to locations

$$\langle X_1 \to e_1, \ldots, X_n \to e_n \rangle$$

- Application of explicit substitution
  - specialization of a universal quantifier
  - non-destructive update
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)

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
  \(\langle X \rightarrow 0 \rangle\)
  (\(\langle X \rightarrow 1 \rangle u, u\))   == (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 □-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 \( e ::= \ldots \mid \text{dia} \, f \)

- Judgment is internalized using \( \diamond \)

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

- New judgment
  \[ \Sigma; \Delta \vdash f \xrightarrow{D} A[C] \]

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

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

- Judgment is internalized using \( \Diamond \)
  \[ \Sigma; \Delta \vdash f \xrightarrow{D} A[C] \]
  \[ \Sigma; \Delta \vdash \text{dia } f : \Diamond_D A[C] \]
Typing possibility

- Recall: $\Diamond$ is an existential quantifier over store (i.e. over explicit substitutions)
- Frames $f ::= [\Theta, e] \mid \text{let } \text{dia } x = e \text{ in } f$
- Canonical frame is a pair: [explicit substitution, term]

$$\Sigma; \Delta \vdash e_1 : A \quad \Sigma; \Delta \vdash e_2 : B[X] \quad X:A \in \Sigma$$

$$\Sigma; \Delta; \vdash [(X \rightarrow e_1), e_2] \div_X B$$

- Explicit substitution $\langle X \rightarrow e_1 \rangle ==$ to be written into the store
- Term $e_2 ==$ computation in the new store
Typing possibility

- Elimination form threads the existential

\[
\Sigma; \Delta \vdash e : \bigtriangleup_C A \quad \Sigma; (\Delta, x:A) \vdash f \div_D B [C]
\]

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

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

- Consequences:
  - Computations are explicitly single-threaded
  - Explicit substitution embedded in \(e\) has global scope; it can be implemented destructively
Typing possibility

- Elimination form threads the existential

\[
\Sigma; \Delta \vdash e : \Box_{C'} A [C'] \quad \Sigma; (\Delta, x:A) \vdash f \Downarrow_D B [C]\\
\Sigma; \Delta \vdash \text{let dia } x = e \text{ in } f \Downarrow_D B [C']
\]

- 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) box u = f(1) in \langle X \rightarrow 0 \rangle (\langle X \rightarrow 1 \rangle u, u) end</td>
<td>let fun f (y) = box (X + y) box u = f(1) in dia dummy = dia [\langle X \rightarrow 0 \rangle, ()] val z = u dia w = dia [\langle X \rightarrow 1 \rangle, u] in [(w, z)] end</td>
</tr>
</tbody>
</table>

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

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)</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>dia dummy = dia [⟨X → 0⟩, ()]</td>
</tr>
<tr>
<td>⟨X → 0⟩</td>
<td>val z = u == X+1</td>
</tr>
<tr>
<td>(⟨X → 1⟩u, u)</td>
<td>dia w = dia [⟨X → 1⟩, u]</td>
</tr>
<tr>
<td>end</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
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>== box (X + 1)</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]</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.
In \( \Box \)-fragment, references to \( u \) reordered to match their substitutions
### 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  
⟨X → 0⟩  
(⟨X → 1⟩u, u)  
end | let fun f (y) = box (X + y)  
box u = f(1)  
== box (X + 1)  
dia dummy = dia [⟨X → 0⟩, ()]  
val z = u  
== 1  
dia w = dia [⟨X → 1⟩, u]  
== [⟨X→1⟩,2]  
in  
[(w, z)]  
end |

- In ◊-fragment, references to $u$ reordered to match their substitutions
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) box u = f(1) in \langle X \rightarrow 0 \rangle \langle X \rightarrow 1 \rangle u, u \rangle end</td>
<td>let fun f (y) = box (X + y) box u = f(1) == box (X + 1) dia dummy = dia [\langle X \rightarrow 0 \rangle, ()] val z = u == 1 dia w = dia [\langle X \rightarrow 1 \rangle, u] == [\langle X \rightarrow 1 \rangle, 2] in [(w, z)] == [(2, 1)] end</td>
</tr>
</tbody>
</table>
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
Introduction ✓

Modal treatment of state
- names as memory locations ✓
- modal types for state ✓

Modal treatment of control effects

Categorical structure of modalities

Related and future work
Control effects

- Perform jumps to predetermined points in the program
- Examples: exceptions, catch/throw, continuations
Control effects

- Perform jumps to predetermined points in the program
- Examples: exceptions, catch/throw, continuations
- Names can be used to label jump destinations
- Example with exceptions:

\[(1 + \text{raise}_x 2)\]

\[\text{handle } X x \rightarrow x + 2\]

\[\rightarrow 4\]
Type system for exceptions

- Associate expression $e$ with its type $A$ and support $C$

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

- Support == set of names/exceptions that $e$ may raise
- Pure expressions == empty support
Exception handling

- Raising exceptions == extend support
- Handle exceptions == shrink support

\[
\begin{align*}
\text{Handlers} \quad \Theta & ::= \quad \cdot \mid (\Theta, Xx \to e) \\
\text{Terms} \quad e & ::= \quad \ldots \mid e \ \text{handle} \ \Theta
\end{align*}
\]

- Scope of a handler is delimited

\[
((\text{raise}_x 0) \ \text{handle} \ Xx \to (x + 2), \text{raise}_x 0) \\
\text{handle} \ Xx \to (x, x + 1)
\]
Example: exceptions

- Let $X, Y : int$ == names of integer exceptions

<table>
<thead>
<tr>
<th>term</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{raise}_X 1 + \text{raise}_Y 2$</td>
<td>$X, Y$</td>
</tr>
<tr>
<td>$(\text{raise}_X 1 + \text{raise}_Y 2)$</td>
<td>$Y$</td>
</tr>
<tr>
<td>handle $Xx \rightarrow x + 2$</td>
<td>empty</td>
</tr>
<tr>
<td>$(\text{raise}_X 1 + \text{raise}_Y 2)$</td>
<td></td>
</tr>
<tr>
<td>handle $Xx \rightarrow x + 2, Yy \rightarrow y + 3$</td>
<td>empty</td>
</tr>
<tr>
<td>$((\text{raise}_X 0) \text{ handle } Xx \rightarrow (x + 2),$</td>
<td>empty</td>
</tr>
<tr>
<td>$\text{raise}_X 0)$</td>
<td></td>
</tr>
<tr>
<td>handle $Xx \rightarrow (x, x + 1)$</td>
<td></td>
</tr>
</tbody>
</table>

- Type soundness: every raised exception is handled
Example: exceptions

- Let \( X, Y : int \) == names of integer exceptions

<table>
<thead>
<tr>
<th>term</th>
<th>support</th>
<th>evaluates to</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{raise}_X 1 + \text{raise}_Y 2 )</td>
<td></td>
<td>( X, Y )</td>
</tr>
<tr>
<td>( \text{raise}_X 1 + \text{raise}_Y 2 )</td>
<td></td>
<td>( Y )</td>
</tr>
<tr>
<td>handle ( Xx \rightarrow x + 2 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{raise}_X 1 + \text{raise}_Y 2 )</td>
<td></td>
<td>empty 3</td>
</tr>
<tr>
<td>handle ( Xx \rightarrow x + 2, )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( Yy \rightarrow y + 3 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ((\text{raise}_X 0) \text{ handle } Xx \rightarrow (x + 2)) )</td>
<td></td>
<td>empty (0, 1)</td>
</tr>
<tr>
<td>( \text{raise}_X 0 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>handle ( Xx \rightarrow (x, x + 1) )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Type soundness: every raised exception is handled
Exception declaration and allocation

- Same constructs as for state:
  - `declare` fresh exception $X:A$ in $e$ == $\nu X:A.\ e$
  - `allocate` fresh exceptions == `choose` $e$
Exception declaration and allocation

- Same constructs as for state:
  - `declare` fresh exception $X : A$ in $e \equiv \nu X : A. \ e$
  - `allocate` fresh exceptions $\equiv$ `choose` $e$

- Example: generate new exception DIVZERO

  ```
  choose (\nu\text{DIVZERO}:\text{unit}. \\
  \quad \text{if } y = 0 \text{ then } \text{raise}_{\text{DIVZERO}} () \text{ else } x/y \\
  \quad \text{handle } \text{DIVZERO} \rightarrow 0)
  ```
• Just like with state, we want to internalize exceptional computation
• But, should we use $\Box$ or $\Diamond$?
Logical analogy

- Just like with state, we want to internalize exceptional computation
- But, should we use $\square$ or $\diamond$?
- Reformulation:
  
  computation of type $A$ raising exceptions from the set $C$
  
  $\square \Rightarrow$

  \textit{for every} handler for exceptions in $C$, return value of type $A$
Modal necessity for exceptions

\[ \square_C A \]

\[ == \]

computation of type \( A \) possibly raising exceptions in \( C \)

- Handling exceptions == specialization of universal quantifier
- Scope of a handler is delimited
Example

```
let fun f (y : int) : □_X int = box (raise X y)
   box u = f(1)
in
   (u handle X x → 0, u)
end
handle X x → (x + 1, x + 2)
```

- variable `u` “unpacked” in two different environments
- range type of `f` marks the effect of raising `X`
- computation is effectful, but final result is pure
Example

```
let fun f (y : int) : □_X int = box (raise X y)
    box u = f(1)    == box (raise X 1)
in
    (u handle X x → 0, u)
end
handle X x → (x + 1, x + 2)
```

- variable \(u\) “unpacked” in two different environments
- range type of \(f\) marks the effect of raising \(X\)
- computation is effectful, but final result is pure
let fun f (y : int) : □_X int = box (raise X y)
   box u = f(1)  == box (raise X 1)
in
   (u handle X x → 0, u)  == ((raise X 1) handle X x → 0, raise X 1)
end
handle X x → (x + 1, x + 2)

- variable u “unpacked” in two different environments
- range type of f marks the effect of raising X
- computation is effectful, but final result is pure
let fun f (y : int) : \(X\) int = box (raise X y)

box u = f(1) == box (raise X 1)

in

(u handle X x \(\rightarrow\) 0, u) == (0, raise X 1)

end

handle X x \(\rightarrow\) (x + 1, x + 2)

- variable \(u\) “unpacked” in two different environments
- range type of \(f\) marks the effect of raising \(X\)
- computation is effectful, but final result is pure
Example

let fun f (y : int) : □_X int = box (raise X y)
  box u = f(1)       == box (raise X 1)
in
  (u handle X x → 0, u) == (0, raise X 1)
end
handle X x → (x + 1, x + 2) == (2, 3)

• variable u “unpacked” in two different environments
• range type of f marks the effect of raising X
• computation is effectful, but final result is pure
Summary of exceptions

- Exceptions can be modeled by names
- Exceptions are effects with delimited scope
- Handler restores a purity of an exceptional computation
- Consequence:
  - □-fragment of a modal calculus is sufficient for internalizing exceptional computation
- Similar development for catch/throw calculus and for composable continuations
Outline

- Introduction ✓
- Modal treatment of state
  - names as memory locations ✓
  - modal types for state ✓
- Modal treatment of control effects ✓
- Categorical structure of modalities
- Related and future work
Categorical structure of modalities

- \( \Box \) is a comonad

\[
\begin{align*}
    f_1 : & \Box A \rightarrow A = \\
         & \lambda x. \text{let } \text{box } u = x \text{ in } u \\
    f_2 : & \Box A \rightarrow \Box \Box A = \\
         & \lambda x. \text{let } \text{box } u = x \text{ in } \text{box } (\text{box } u) \\
    f_3 : & (A \rightarrow B) \rightarrow \Box A \rightarrow \Box B = \\
         & \lambda x. \lambda y. \text{let } \text{box } u = x \text{ in } \text{let } \text{box } v = y \text{ in } \text{box } (u \; v)
\end{align*}
\]
Categorical structure of modalities

- $\Box$ is a comonad

\[
\begin{align*}
  f_1 : & \Box A \to A = \\
  & \lambda x. \text{let box } u = x \text{ in } u \\
  f_2 : & \Box C A \to \Box \Box C A = \\
  & \lambda x. \text{let box } u = x \text{ in box (box u)} \\
  f_3 : & \Box C (A \to B) \to \Box D A \to \Box C, D B = \\
  & \lambda x. \lambda y. \text{let box } u = x \text{ in let box } v = y \text{ in box (u v)}
\end{align*}
\]

- $\Box C A \to \Box D A \quad C \subseteq D$

- and others...
Categorical structure of modalities

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

- Consider fragment where function arguments are boxed

  $$ (A \rightarrow B)^+ = \Box A^+ \rightarrow B^+ $$

  $$ (\Box A)^+ = \Box A^+ $$

  $$ (\Diamond A)^+ = \Diamond \Box A^+ $$

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

- ♦ is a monad

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

- $\Diamond$ is a monad

\[
g_1 : A \to \Diamond A = \\
\lambda x. \text{dia } [x]
\]

\[
g_2 : \Diamond C \Diamond_D A \to \Diamond_D A = \\
\lambda x. \text{dia (let dia } y = x \text{ in let dia } z = y \text{ in } [z])
\]

\[
g_3 : \Box_C (A \to B) \to \Diamond_D A \to \Diamond_D B = \quad \text{(where } C \subseteq D) \\
\lambda e_1. \lambda e_2. \text{dia (let box } u = e_1 \text{ in let dia } x = e_2 \text{ in } [u x])
\]

- $\Diamond_D A \to \Diamond_C A \quad C \subseteq D$

- and others...
Modal logic with names is a very good system for effects

- uniform
  - state, control effects
- expressive
  - effects with delimited scope == universal quantification == comonad □
  - effects with global scope == existential quantification == monad ◊
  - dynamic effect generation
- simple
  - names, □, ◊
Some related work

- Monads [Moggi ’91, Wadler ’95]
  - formulation of state with no type distinction between reads and writes
  - exceptional computations always tested before use
- 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]
Future work

- Integrating modal types and names into a realistic language
  - I/O effects
  - type and support inference
  - universal and existential abstraction over supports
- First-class effects
- Equational theory
  - also for a call-by-name variant
- Combining effects
  - supports need not be simply sets (e.g. in case of composable continuations, supports are lists)
  - but the framework makes support and its ordering explicit
Modal logic with names is a very good system for effects

- uniform
  - state, control effects
- expressive
  - effects with delimited scope == universal quantification == comonad  □
  - effects with global scope == existential quantification == monad  ◇
  - dynamic effect generation
- simple
  - names,  □,  ◇
Categorical structure of modalities

- In constructive S4, □ is a \textit{monoidal comonad} and ◊ is a □-\textit{strong monad}

\[
\begin{align*}
\Diamond A & \Rightarrow A \\
\Box A & \Rightarrow \Box \Box A \\
\Box (A \Rightarrow B) & \Rightarrow \Box A \Rightarrow \Box B
\end{align*}
\]

\[
\begin{align*}
A & \Rightarrow \Diamond A \\
\Diamond \Diamond A & \Rightarrow \Diamond A \\
\Box (A \Rightarrow B) & \Rightarrow \Diamond A \Rightarrow \Diamond B
\end{align*}
\]

- There is no map \( A \Rightarrow \Box A \) 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 \( \lambda \)

- \( \Box \) 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
  \[ \Delta; \Gamma \vdash e : \Diamond A \quad \Delta; x : A \vdash f \div B \]
  \[ \Delta; \Gamma \vdash \text{let dia } x = e \text{ in } f \div B \]
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  
  \[(A \rightarrow B)^+ = \Box A^+ \Rightarrow B^+\]
  \[(\Box A)^+ = \Box A^+\]
  \[(\Diamond A)^+ = \Diamond \Box A^+\]
- 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
- Reinterpretation eliminates the need for \(\Gamma\)

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

\[
\Delta \vdash \text{box } e : \Box A \quad \Delta \vdash \text{let } \text{dia } x = e \text{ in } f : B
\]
Typing for name generation

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

\[
(\Sigma, X:A); \Delta \vdash e : B \ [C] \ \ X \notin \text{fn}(A, B, C, \Delta) \\
\Sigma; \Delta \vdash \nu X : A. \ e : A \imp B \ [C]
\]

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

\[
\Sigma; \Delta \vdash e : A \imp B \ [C] \\
\Sigma; \Delta \vdash \text{choose} \ e : B \ [C]
\]
Example

- Assume $X : int == \text{ uninitialized store location}$
- The program writes 0 into $X$, and then computes with it
  
  ```
  let dia x = dia [\langle X \rightarrow 0 \rangle, X + 1]
  in
  [x + X]
  end
  ```

  $$\text{val it = } [\langle X \rightarrow 0 \rangle, 1 + X] \div X \hspace{1em} \text{int}$$

- Notice: the result is a closure remembering used store
- Closures are typed with a new judgment