A modal foundation for functional programming with effects

Aleksandar Nanevski
Computer Science Department, Carnegie Mellon University

Toyota Technological Institute, Chicago
18 April, 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
If we adhere to the mathematical ideal of function, then:

- Functions *only* compute values
- Prototypical example: \( \lambda \)-calculus
- Strong connection with logic
  
  programs \( == \) 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 does dynamic binding relate to state?
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...
In this presentation

A framework for representing effects, which consolidates and extends many features that were proposed and used before but often in an ad-hoc way

- Uniformity
  - exceptions, catch/throw, composable continuations, dynamic binding, state

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

- Logicality
  - intuitionistic modal logic S4 + names
• Introduction ✓
• Names as markers for effects
• Modal logic for effect scoping
• Related and future work
• Conclusions
Names

- A.K.A.: labels, atoms, nonces, symbols, tags, indeterminates
- Names stand for particular effects
- Example: names of exceptions, names of memory cells, names of destination points for jumps
- Important: *names are generated dynamically*
- Our type system enforces discipline in name introduction and propagation
We associate expression $e$ with its type $A$

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

- $\Sigma$ : types for names
- $\Delta$ : types for variables
We associate expression $e$ with its type $A$ and support $C$

$$\Sigma; \Delta \vdash e : A\lbrack C \rbrack$$

- $\Sigma$ : types for names
- $\Delta$ : types for variables
- Support $== \text{set of names/effects that expression may cause}$
Support of an expression

- Duality:
  - causing effects vs. handling effects
  - expanding support vs. shrinking support
  - introduction vs. elimination
- Execute only expressions with empty support
Let \( X, Y : \text{int} \) == 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) ) &lt;br&gt;handle ( Xx \rightarrow x + 2 )</td>
<td>( Y )</td>
</tr>
<tr>
<td>( (\text{raise}_X 1 + \text{raise}_Y 2) ) &lt;br&gt;handle ( Xx \rightarrow x + 2, Y y \rightarrow y + 3 )</td>
<td>empty</td>
</tr>
</tbody>
</table>

- Soundness: every raised exception is handled
Example: dynamic binding

- Value of dynamic variable determined from context
- Let $X, Y : int$ == names for integer dynamic variables
- effect == reading a name, handler == substituting a name
- $\langle X \rightarrow 10 \rangle$ == substitute 10 for the name $X$

<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>$\langle X \rightarrow 10 \rangle(X^2 + Y^2)$</td>
<td>$Y$</td>
</tr>
<tr>
<td>$\langle X \rightarrow 10, Y \rightarrow 20 \rangle(X^2 + Y^2)$</td>
<td>empty</td>
</tr>
</tbody>
</table>

- Soundness: only initialized names are read from
Dynamic name generation

- At run-time we need to:
  - define new exceptions
  - allocate new dynamic variables, or memory
  - establish labels for jumps
- Fundamental operation: generating new symbols
  - gensym in LISP
- Our type system == disciplined gensym
Name declaration and allocation

- Two new constructs:
  - `declare fresh name X:A in e == \nu X:A. e`
  - `allocate fresh name == choose e`
Name declaration and allocation

- Two new constructs:
  - \( \text{declare} \) fresh name \( X:A \) in \( e \) == \( \nu X:A. \ e \)
  - \( \text{allocate} \) fresh name == \( \text{choose} \ e \)
- Example: allocate new dynamic variable \( X \)
  \( \text{choose} (\nu X:\text{int.} \) \)
  \( \langle X \to 10 \rangle X) \)
Name declaration and allocation

- Two new constructs:
  - *declare* fresh name $X:A$ in $e == \nu X:A. e$
  - *allocate* fresh name $== \text{choose } e$

- Example: allocate new dynamic variable $X$
  
  \[
  \text{choose} (\nu X:\text{int.} \\
  \langle X \rightarrow 10 \rangle X)
  \]

- Example: allocate new exception DIVZERO
  
  \[
  \text{choose} (\nu \text{DIVZERO}:\text{unit.} \\
  \text{if } y = 0 \text{ then raise}_{\text{DIVZERO}} () \text{ else } x/y \\
  \text{handle } \text{DIVZERO} \rightarrow 0)
  \]
Name declaration and allocation

- Important:
  - choose \((\nu X. \, e)\) allocates \(X\) to be used *exclusively* in \(e\)
  - type system must prevent \(X\) from escaping
- Requirement: \(X\) is not in support of \(e\)
  - all instances of \(X\) in \(e\) must be handled
- For PL specialists:
  - \(\nu\) and choose are introduction and elimination forms for irrelevant implication \(A \nrightarrow B\)
Outline

- Introduction ✓
- Names as markers for effects ✓
- Modal logic for effect scoping
- Related and future work
- Conclusions
Effects and higher-order functions

- How to handle effects inherited from function arguments?
- Hint comes from logic:
  - types should specify program behavior
- Therefore:
  
  effectful arguments should be marked by the type system
Modal logic

- Modal operators □ and ◊
- □A == A is necessarily true
- ◊A == A is possibly true
Names and necessity

- Intuition:
  \[
  \text{type } \Box_C A = \text{suspended computation of type } A \text{ possibly causing effects listed in } C
  \]

- We distinguish between
  \[
  f : A \rightarrow B \text{ applied to } e : A[\Box] \\
  \text{and} \\
  f : \Box_C A \rightarrow B \text{ applied to } e : \Box_C A
  \]
Typing necessity

- Intuition: effectful term is boxed (i.e., packaged) to be “shipped” further

\[
\Sigma; \Delta \vdash e : A [D]
\]

\[
\Sigma; \Delta \vdash \text{box } e : \Box_D A [\cdot]
\]

- Notice:
  - boxed expression itself is effect-free
  - independent of the actual nature of effects

- Boxed expressions are not evaluated

- Analogy: quote in LISP
• Using a suspended computation: let-box construct
• let-box == unwrap the package, but do not peek into it
• Examples:

\[
\text{let box } u = \text{box } e_1 \text{ in } e_2 \longmapsto [e_1/u]e_2
\]

\[
\text{let box } u = \text{box } (1 + 1) \text{ in } (\text{box } u) \longmapsto \text{box } (1 + 1)
\]

\[
\text{let box } u = \text{box } (1 + 1) \text{ in } u \longmapsto 1 + 1 \longmapsto 2
\]

• Notice: binding to \( u \) is “by-name”
Typing necessity

- Using a suspended computation

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

- Notice:
  - context \( \Delta \) types “unwrapped” expressions
Typing necessity

- Using a suspended computation

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

- Notice:
  - new context \( \Delta \) of “unwrapped” expressions

- Also:
  - computing the “package” \( e_1 \) can itself be effectful
  - if you peek into the package, effects are propagated into the environment
Example: dynamic binding

\[
\langle X \rightarrow 0 \rangle
\]

\[
\text{let fun f (y : int) : } X \text{ int } = \text{ box (X + y)}
\]

\[
\text{box u } = f(1)
\]

\[
\text{in}
\]

\[
\langle X \rightarrow 1 \rangle u, u
\]

\[
\text{end}
\]

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

\[ \langle X \rightarrow 0 \rangle \]
let fun f (y : int) : \( X \) int = box \( X + y \)
box u = f(1)  \( == \) box \( X + 1 \)
in
\( \langle X \rightarrow 1 \rangle u, u \) end

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

\[
\langle X \rightarrow 0 \rangle \\
\text{let fun } f (y : \text{int}) : \square X \text{int} = \text{box (} X + y \text{)} \\
\quad \text{box } u = f(1) \quad == \text{ box (} X + 1 \text{)} \\
in \\
\quad (\langle X \rightarrow 1 \rangle u, u) \quad == \ (\langle X \rightarrow 1 \rangle X + 1, X + 1) \\
\text{end}
\]

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

\[ \langle X \rightarrow 0 \rangle \]

let fun f (y : int) : \(\Box_X\) int = box (X + y)

box u = f(1)  \(==\) box (X + 1)

in

(\langle X \rightarrow 1 \rangle u, u)  \(==\) (2, 1)

end

- variable \(u\) “unpacked” in two different environments
- range type of \(f\) marks the effect of reading \(X\)
- computation is effectful, but final result is pure
For PL specialists: Seen this before?

- Monads:
  - programming language Haskell
  - effectful expressions must have monadic type
For PL specialists: Seen this before?

- Monads:
  - programming language Haskell
  - effectful expressions must have monadic type
- Monads are crucially different from □

\[ \text{because there is no map } Monad A \rightarrow A, \text{ effects encapsulated by monads cannot be handled} \]
For PL specialists: Seen this before?

- Monads:
  - programming language Haskell
  - effectful expressions must have monadic type
- Monads are crucially different from $\Box$
  
  because there is no map $\text{Monad} A \rightarrow A$, effects encapsulated by monads cannot be handled
- In our setup, $\Box_CA \rightarrow A$ are handlers for effects in $C$; in fact
  $\Box$ is a co-monad
Global effects and handlers

- Rather than handling each effect instance separately, can we install a *global handler* for the effect?
- Example: dynamic binding
  - instead of substituting each occurrence of $X$ explicitly,
  - set default substitution that persists till end of program
• Intuition:

\[ \text{type } \diamond_{C} A \]

\[ =\]

handler for names in \( C \) + \text{suspended computation of type } A \]
Example: dynamic binding

- Use of $\diamond_X A$:

  \[
  \text{let } \text{dia } z = \text{dia } [\langle X \rightarrow 0 \rangle, e_1] \text{ in } e_2
  \]

  - install substitution $\langle X \rightarrow 0 \rangle$
  - evaluate $e_1$ in new environment
  - bind to $z$ and proceed with $e_2$
  - but never exit the scope of the installed substitution

- Consequence:
  - no need to remember the previous value for $X$
  - substitution with $\diamond$ is destructive
• Consequence:

\[
\text{dynamic binding } + \Diamond \\
\quad = \\
\text{destructive assignment} \\
\quad = \\
\text{state}
\]

• The “installed” substitution \(\equiv\) global store
Modal calculus for state

- Intuition:
  
  \[
  \text{type } \diamond_X A \\
  \equiv \\
  \text{assignment for } X + \text{expression of type } A
  \]

- Duality:
  
  - \( \square_X A \equiv \text{computation reading from } X \)
  
  - \( \diamond_X A \equiv \text{computation writing into } X \)

- Destructive and non-destructive update can coexist
Modal calculus for state

- Intuition:
  \[ \text{type } \diamond_X A \]
  \[ == \]
  assignment for \( X \) + expression of type \( A \)

- Duality:
  - \( \square_X A == \) computation \textit{reading from} \( X \)
  - \( \diamond_X A == \) computation \textit{writing into} \( X \)

- Destructive and non-destructive update can coexist

- Dynamic generation of names \( == \) \textit{logical way to allocate uninitialized} memory

- Types guarantee: no uninitialized memory cell will be read
Example: dynamic binding revisited

Dynamic binding

\[ \langle X \rightarrow 0 \rangle \]
let fun f (y) = box (X + y)
    box u = f(1)

in
\[ \langle X \rightarrow 1 \rangle u, u \] end

- Notice: \(u\) “unpacked” in two different environments
Example: dynamic binding revisited

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

- Notice: \(u\) “unpacked” in two different environments
- In state: substitutions are sequenced, and references to \(u\) reordered
### Example: dynamic binding revisited

- **Dynamic binding**

  \[
  \langle X \to 0 \rangle
  \]
  
  let fun \( f \) (\( y \)) = box (\( X + y \))
  
  box \( u \) = f(1) \( \equiv \) box(X+1)

  in

  \((\langle X \to 1 \rangle u, u)\)

  end

- **State**

  - let \( \text{dia dummy} = \text{dia} [\langle X \to 0 \rangle, ()] \)

  fun \( f \) (\( y \)) = box (\( X + y \))

  box \( u \) = f(1) \( \equiv \) box (\( X + 1 \))

  val \( z = u \)

  \( \text{dia w} = \text{dia} [\langle X \to 1 \rangle, u] \)

  in

  \((w, z)\)

  end

- **Notice:** \( u \) “unpacked” in two different environments

- **In state:** substitutions are sequenced, and references to \( u \) reordered
### Example: dynamic binding revisited

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

- Notice: \( u \) “unpacked” in two different environments
- In state: substitutions are sequenced, and references to \( u \) reordered
### Example: dynamic binding revisited

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

- Notice: \( u \) “unpacked” in two different environments
- In state: substitutions are sequenced, and references to \( u \) reordered
the operator ◊ is a monad

- Just like the uses of monads in previous work:
  - ◊ serves to *globalize* effect scope...
  - ... or rather, handler scope
- But unlike in previous work:
  - no need to put all effects under ◊
  - only those that do not admit handling
• Introduction ✓
• Names as markers for effects ✓
• Modal logic for effect scoping ✓
• Related and future work
• Conclusions
Other contributions

Meta-programming

- Run-time code generation \textit{and inspection} (Nanevski ’01)
- Necessitation fragment of presented system
- Reinterpreted modality

\[ \Box_X A \]

\[ == \]

\textit{syntactic (i.e., source)} code with “hole” \( X \)

- non-modal types \( == \) \textit{compiled} code
- names \( == \) variables \textit{with identity}
- Source code can be compiled but also syntactically analyzed and destructed at run-time
Meta-programming

- Applications:
  - hygienic macros
  - type-safe grafting of user-written optimizations on a compiler
  - symbolic computation (in the style of Mathematica, Maple)

- Related work
  - MetaML (Sheard, Taha, Moggi, Calcagno)
  - Nominal Logic and FreshML (Pitts, Gabbay ’00)
  - Run-time code generation (Wickline, Lee, Pfenning, Davies ’98)
Natural extensions

- Integrating nominal modal types into a realistic language
  - how about printing on the screen, or opening a file?
  - type and support inference
  - universal and existential abstraction over supports
- Equational theory
  - also for a call-by-name/need variant
- Combining effects
  - composable continuations require ordered supports
  - this indicates problems in combining with, say, exceptions
  - but the framework makes orderings explicit
Dynamic generation of symbols is a fundamental operation:

- parallel and distributed computing
  - $\pi$-calculus
    - $\Box_X A$ == process reading from channel $X$
    - $\Diamond_X A$ == process writing to channel $X$
  - modal logic for grid computing
- SML module systems
  - run-time generation of new abstract types
  - names and supports for recursive modules
- Use of names in modal logic?
Modal logic with names is excellent system for effects

- uniform
  - exceptions, catch/throw, composable continuations, dynamic binding, state
- expressive
  - global vs. handleable, dynamic effect generation
- simple
  - names, □, ◊
- extensible
  - meta-programming
Thank you!
Conclusions

- Important aspects of effects:
  - does it admit a notion of handling?
  - can instances be dynamically generated?
- Modalities $\Box$ and $\Diamond$ distinguish handleable vs. global
- Names provide dynamic generation of new instances
- Therefore:

  Modal logic with names is a framework for representing both aspects

- And it is uniform, expressive, simple and extendable
- Companion papers:
  http://www.cs.cmu.edu/~aleks
Pure functional programming

- Program == *mathematical* function relating inputs to outputs
Pure functional programming

- Program == *mathematical* function relating inputs to outputs
- Types == fix domain and range of functions
Pure functional programming

- Program == mathematical function relating inputs to outputs
- Types == fix domain and range of functions
- Higher-order functions == basic building blocks for modular programming (in the small) and code reuse
Pure functional programming

- Program == mathematical function relating inputs to outputs
- Types == fix domain and range of functions
- Higher-order functions == basic building blocks for modular programming (in the small) and code reuse
- Types == statically enforced interfaces between modular blocks
Pure functional programming

- No effects!
- Mathematical functions *only* compute values
- λ-calculus
  + Logical connection: programs == proofs in logic of types
  + Easy to reason about programs
  + Evaluation order is irrelevant
  + Flexible program transformation and optimization
- But, effects are necessary for realistic applications
Many languages have effects, but do not treat them in a uniform manner, resulting in:

- repeated functionality
  - declaring new exceptions vs. allocating memory vs. declaring destination labels for jumps
- non-orthogonal language design
  - hard to extend, reason about and program
- hard to compare effects
  - how to combine exceptions with continuations?
  - how does dynamic binding relate to state?

In typed languages, types often do not say anything about effects
In this presentation

- A simple, unified framework for defining effects
- Capable of:
  - discerning between global effects (e.g. state) and handleable effects (e.g. exceptions)
  - dynamic allocation of new effect instances
- Examples: exceptions, dynamic binding, catch/throw, composable continuations, state
- Logical: orthogonal design, simple to extend, reason about and use
- Based on two simple concepts:
  - Names
  - Modal Logic
- Assign type $A$ and support $C$ to an expression $e$

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

- $\Sigma$ == context of names (i.e., effects)
- $\Delta$ == context of variables
- Only pure terms considered for evaluation
- Important: support $C$ may be ordered
Typing exceptions

- Performed action must be in the support

\[
\Sigma; \Delta \vdash e : A [C, X] \quad X : A \in \Sigma
\]

\[
\Sigma; \Delta \vdash \text{raise}_X e : B [C, X]
\]

- Performing response shrinks the support

\[
\Sigma; \Delta \vdash e : B [C, X] \quad \Sigma; (\Delta, x : A) \vdash e' : B [C'] \quad X : A \in \Sigma
\]

\[
\Sigma; \Delta \vdash e \text{ handle } X x \rightarrow e' : B [C']
\]

- Typing guarantee: every raised exception will be handled
Typing dynamic binding

- Performed action must be in the support

\[ X : A \in \Sigma \]
\[ \Sigma; \Delta \vdash X : A [C, X] \]

- Performing response shrinks the support

\[ \Sigma; \Delta \vdash e : B [C, X] \]
\[ \Sigma; \Delta \vdash e' : A [C] \]
\[ X : A \in \Sigma \]
\[ \Sigma; \Delta \vdash \langle X \rightarrow e' \rangle e : B [C] \]

- Typing guarantee: only initialized dynamic variables are dereferenced
Name declaration

- Construct $\nu X : A. \ e =\ textit{declares} \ fresh \ name X : A$ in $e$
- Typing discipline:

$$\Gamma \vdash \nu X : A. \ e [C]$$
Name declaration

- Construct $\nu X : A. \ e$ == declares fresh name $X : A$ in $e$
- Typing discipline:

$$
(\Sigma, X : A) \vdash e [C] \quad X \notin C
$$

$$
\Sigma \vdash \nu X : A. \ e [C]
$$

- context $\Sigma$ == list of names and their types
- $X$ must be irrelevant for $e$
Name declaration

- Construct $\nu X : A. \ e \Rightarrow \text{declares} \ \text{fresh name} \ X : A \ \text{in} \ e$
- Typing discipline:

$$\frac{(\Sigma, X : A) \vdash e : B [C] \quad X \notin C}{\Sigma \vdash \nu X : A. \ e : A \Rightarrow B [C]}$$

- context $\Sigma \Rightarrow \text{list of names and their types}$
- $X \ \text{must be irrelevant} \ \text{for} \ e$
- new type $A \Rightarrow B \Rightarrow \text{irrelevant implication}$
Name allocation

- Term constructor `choose e == allocate` fresh name according to the declaration in `e`
- Typing discipline

\[ \Sigma \vdash e : A \leadsto B [C] \]

\[ \Sigma \vdash \text{choose } e : B [C] \]
Name allocation

- Term constructor `choose e == allocate` fresh name according to the declaration in `e`
- Typing discipline

\[
\Sigma \vdash e : A \Rightarrow B \left[C\right] \\
\Sigma \vdash \text{choose } e : B \left[C\right]
\]

- Example: allocate dynamic variable `X`, and `Y`, but “may garbage collect” them at the end

```
choose (\nu X : \text{int.})
  choose (\nu Y : \text{int.})
    let box u = box (X + 1)
    box v = box (Y + 1)
  in
  \langle X \rightarrow 0 \rangle u
```
Typing for name generation

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

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

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

\[
\Sigma; \Delta \vdash e : A \leftrightarrow B [C] \\
\Sigma; \Delta \vdash \text{choose} \ e : B [C]
\]
How do monads fit in?

- Monads are used for globalizing scope of effects via sequencing of code
- Modal operator $\Diamond$ for possibility is a monad
- Which effects are global and need to be sequenced?
- One example: state update
Case study: dynamic binding (cont’d)

- But ♦ makes handlers global!
  - once substitution for $X$ is “installed” as a global handler, old value of $X$ will never be required
  - destructive assignment
Case study: dynamic binding (cont’d)

- But ♦ makes handlers global!
  - once substitution for $X$ is “installed” as a global handler, old value of $X$ will never be required
  - destructive assignment

- Conclusion:

\[ \diamond_C A \]
\[ == \]
\[ \text{global substitution + dynamic binding} \]
\[ == \]
\[ \text{State} \]

- The “installed” substitution $==$ global store
Stateful terms constructors

- \texttt{dia} \left[ \langle X \to e_1 \rangle, e_2 \right]

  suspended computation that writes \( e_1 \) into \( X \) and proceeds with \( e_2 \)

- \texttt{let dia } x = e_1 \texttt{ in } e_2

  force \( e_1 \) to change the state, then proceed with \( e_2 \)
Example

- Assume $X : int$ == uninitialized store location
- The program writes 0 into $X$, and then computes with it

  ```
  let dia x = dia [\langle X \to 0 \rangle, X + 1] in
  [x + X]
  end
  ```

  ```
  val it = [\langle X \to 0 \rangle, 1 + X] \div X \ \text{int}
  ```

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

- Closure = substitution + expression

\[
\Sigma; \Delta \vdash e : A[C, X] \quad \Sigma; \Delta \vdash e' : A[C] \quad X : A \in \Sigma
\]

\[
\Sigma; \Delta \vdash [\langle X \to e' \rangle, e] \vdash_X A[C]
\]

- Coercion: closures → terms

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

\[
\Sigma; \Delta \vdash \text{dia } f : \Diamond_D A[C]
\]
- Changing global store

\[
\Sigma;\Delta \vdash e : \diamond D_1 A [C] \quad \Sigma; (\Delta, x:A) \vdash f \parallel_{D_2} B [D_1] \\
\Sigma;\Delta \vdash \text{let } \text{dia } x = e \text{ in } f \parallel_{D_2} B [C]
\]

- Notice: if \( e \) initializes store locations \( D_1 \), then \( f \) can freely use them
- Store location $X$ below dereferenced in two different environments
- Updates are single-threaded and destructive

```ocaml
- let dia dummy = dia [\langle X \to 0 \rangle, ()]
  fun f (y : int) : \Box X \text{int} = box (X + y)
  box u = f 1
  val z = u
  dia w = dia [\langle X \to 1 \rangle, u]
  in
    [(w, z)]
  end

val it = [(2, 1)] : int * int
```
Example

- Store location $X$ below dereferenced in two different environments
- Updates are single-threaded and destructive

```ml
- let dia dummy = dia [⟨X → 0⟩, ()]  set $X$ → 0
  fun f (y : int) : ⊤$X$ int = box ($X + y$
  box u = f 1
  val z = u
  dia w = dia [⟨X → 1⟩, u]
  in
    [(w, z)]
  end

val it = [(2, 1)] ÷ int * int
```

A modal foundation for functional programming with effects – p.58
• Store location $X$ below dereferenced in two different environments

• Updates are single-threaded and destructive

```plaintext
- let dia dummy = dia [⟨X → 0⟩, ()]  set X → 0
  fun f (y : int) : □X int = box (X + y)
  box u = f 1   == box (X + 1)
  val z = u
  dia w = dia [⟨X → 1⟩, u]
  in
    [(w, z)]
  end

val it = [(2, 1)] ÷ int * int
```

Example

- Store location $X$ below dereferenced in two different environments
- Updates are single-threaded and destructive

```
fun f (y : int) : \Box int = box (X + y)

box u = f 1  == box (X + 1)
val z = u  == X + 1

val it = [(2, 1)] : int * int
```

```
- let dia dummy = dia [\langle X \rightarrow 0 \rangle, ()]  set X \rightarrow 0

in

(dia w = dia [\langle X \rightarrow 1 \rangle, u]

in

[(w, z)]

end

end

val it = [(2, 1)] : int * int
```
- Store location $X$ below dereferenced in two different environments
- Updates are single-threaded and destructive

```ml
fun f (y : int) : int = box (X + y)

val z = u == 1
dia w = dia [X → 1], u

val it = [(w, z)]
val it = [(2, 1)] → int * int
```

```
- let dia dummy = dia [X → 0], ()

set $X → 0$
```

```
box u = f 1 == box (X + 1)
```

```
val z = u == 1
dia w = dia [X → 1], u

end
```

```
[(w, z)]
```

```
val it = [(2, 1)] → int * int
```
Example

- Store location $X$ below dereferenced in two different environments
- Updates are single-threaded and destructive

```ocaml
- let dia dummy = dia [\langle X \rightarrow 0 \rangle, ()]  set $X \rightarrow 0$
  fun f (y : int) : $\Box X \text{int} = \text{box } (X + y)$
  box u = f 1  == box (X + 1)
  val z = u  == 1
  dia w = dia [\langle X \rightarrow 1 \rangle, u]  set $X \rightarrow 1$ and $w == 2$
  in
  [(w, z)]
  end

val it = [(2, 1)] : int * int
```

A modal foundation for functional programming with effects – p.58
Q: How to make substitution have global scope?
Q: How to make substitution have global scope?
A: Attach it to a monad ◊. Then the attached substitution plays the role of global store!
Modal calculus for state

- Q: How to make substitution have global scope?
- A: Attach it to a monad $\Diamond$. Then the attached substitution plays the role of global store!
- Intuition:

\[
\text{type } \Diamond_X A \\
\text{== substitution for a name } X + \text{ expression of type } A
\]
Modal calculus for state

- Q: How to make substitution have global scope?
- A: Attach it to a monad ◊. Then the attached substitution plays the role of global store!
- Intuition:
  \[
  \text{type } \diamond_X A \\
  \equiv \\
  \text{substitution for a name } X + \text{expression of type } A
  \]
- Duality:
  - $\Box_X A \equiv \text{computation reading from } X$
  - $\diamond_X A \equiv \text{computation writing into } X$
- Curiosity: ◊ marks a handler, not an effect
- Destructive and non-destructive update can coexist
Applications of names and modalities (cont’d)

- Relationship with dynamic dispatch and object-oriented programming
- Names and modalities for parallel computing
- Names on the level of types
- Names for higher-order modal logic
- Run-time code generation (Wickline, Lee, Pfenning and Davies ’98)
- Adaptive and incremental computing (Acar, Blelloch and Harper ’02)
- Modal types for logical frameworks (Pientka and Pfenning ’03)
- Modal types for parallel computation (Moody and Pfenning ’03)
When defining effects, it is important to distinguish following:
- does the effect admit a notion of handling?
- can effect instances be dynamically generated

For some effects, both answers are “No”
- non-termination, non-determinism

Some effects cannot be handled
- Example: writing into memory

But for many effects (in fact, probably most), both answers are “Yes”
- Examples: exceptions, dynamic binding, jumps
Conclusions (cont’d)

- We *need* an effect system capable of
  - discerning between global vs. handleable effects
  - representing handlers
  - allocating new instances
- Modal logic combined with names provides one
- Modal types in effect analysis:
  - co-monad $\square$ == represents handleable effects
  - monad $\Diamond$ == represents global effects
- Names are symbols which:
  - encode specific effects
  - *are dynamically introduced*
Conclusions (cont’d)

- Typing guarantees that all effects are handled
- Flexible framework with many possible extensions and applications
- In the papers:
  - catch-and-throw calculus (i.e., jumps)
  - composable continuations (i.e., prompts)
    - supports are ordered sets
  - theory required for proving progress and preservation
Conclusions (cont’d)

- Typing guarantees that all effects are handled
- Flexible framework with many possible extensions and applications
- In the papers:
  - catch-and-throw calculus (i.e., jumps)
  - composable continuations (i.e., prompts)
    - supports are ordered sets
  - theory required for proving progress and preservation