A Modal Approach to Functional Programming

Aleksandar Nanevski

Boston University
November 2, 2005
It is becoming increasingly important to execute programs in complex run-time environments.

Examples:
- parallel and mobile programs
- distributed data, often with different owners
- dynamic reconfiguration in face of changing run-time conditions

As environments are becoming more complex, so is programming.

*Programming languages can help manage this complexity through type systems that capture environment properties.*
Types for environments

- Usually, type checking ensures that functions are invoked with matching arguments.

- But if complexity of programming comes from environments, types should also express assumptions and guarantees about environments!
  - Usually *not* the case today.

- A type system for environments should:
  - ensure that expressions execute only in *matching* environments
  - make apparent if/how expressions *depend* on the environment
  - make apparent if/how expressions *change* the environment
Environment: *state of memory*
Assume $X, Y$ are integer memory locations
Consider the expression $e = X^2 + Y^2$

A *matching* environment for $e$ is one where both $X$ and $Y$ are *allocated* and *initialized*.
E.g., environment $\Theta = \langle X \rightarrow 1, Y \rightarrow 2 \rangle$ is matching for $e$.

Program that executes $e$ in environment $\Theta$:

$$\langle X \rightarrow 1, Y \rightarrow 2 \rangle (X^2 + Y^2)$$

- This program typechecks and evaluates to 5.
Types form a logic.

<table>
<thead>
<tr>
<th>Programming</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>function f(x : A) : B</td>
<td>implication A ⇒ B</td>
</tr>
</tbody>
</table>

Programming and logic are related via Curry-Howard isomorphism.

A logic makes it conceptually simple to reason about programs
- easier programming
- easier optimization
- easier language extensions
A type system for environments should be based on a logic for environments.

But which logic should this be?
A type system for environments should be based on a logic for environments.

But which logic should this be?

- Modal logic reasons about truth across various worlds.
- We can take *world* $\equiv$ *environment*.
Outline

- Introduction ✓
- Modal logic for memory
- Modal logic for control effects
- Modal logic for metaprogramming
- Summary and future work
● Consider a language with instructions for load and store

● Notion of environment: *memory*

● Types should discern between:
  – expressions that do not contain reads or writes
  – expressions that only read
  – expressions that may read and write

● Opportunities for optimization
  – e.g., reads may be executed in parallel
Type system for reads

- Associate expression $e$ with its type $A$

  $\Delta \vdash e : A$

- $\Delta$ : types for local variables of $e$
Type system for reads

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

  $\Delta \vdash e : A[C]$

- $\Delta :$ types and supports for local variables of $e$
- $Support$: set of locations that expression may read from.
Type system for reads

- Associate expression \( e \) with its type \( A \) and support \( C \)

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

- \( \Delta \) : types and supports for local variables of \( e \)
- **Support**: set of locations that expression may read from.

- Example: let \( X, Y : \text{int} \) be memory locations. Then

\[ \vdash X^2 + Y^2 : \text{int} [X, Y] \]
Environments

- Environment definition associates values to memory locations
  \[ \Theta = \langle X_1 \rightarrow e_1, \ldots, X_n \rightarrow e_n \rangle \]

- Scope of environment definition is delimited
  \[ \langle X \rightarrow 1 \rangle(\langle X \rightarrow 2 \rangle e_1, e_2) \]
Example

- Let $X, Y : \text{int} =\text{ uninitialized locations}$
- Reading from locations $\Rightarrow$ \textit{extend} support
- Initializing locations $\Rightarrow$ \textit{shrink} support

<table>
<thead>
<tr>
<th>expression</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X^2 + Y^2$</td>
<td>$X, Y$</td>
</tr>
<tr>
<td>$\langle X \rightarrow 1 \rangle (X^2 + Y^2)$</td>
<td>$Y$</td>
</tr>
<tr>
<td>$\langle X \rightarrow 1, Y \rightarrow 2 \rangle (X^2 + Y^2)$</td>
<td>empty</td>
</tr>
<tr>
<td>$\langle X \rightarrow 1 \rangle (\langle X \rightarrow 2 \rangle X^2, X^2)$</td>
<td>empty</td>
</tr>
</tbody>
</table>
Let $X, Y : \text{int} = \text{ uninitialized locations}$

- Reading from locations == \textit{extend} support
- Initializing locations == \textit{shrink} support

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

- Theorem: expressions with empty support do not get stuck.
Dynamic binding

- Environment definitions have delimited scope

\[ \langle X \rightarrow 1 \rangle (\langle X \rightarrow 2 \rangle e_1, e_2) \]

- May be viewed as a non-destructive write into memory.

- **Warning**: not to be confused with ordinary writes into memory, which are destructive.

- Environment definitions correspond to dynamic binding in LISP.

- Connection with logic will suggest interesting extensions.
Say $e$ is an expression \textit{reading} from locations in $C$.

If locations in $C$ are not initialized, $e$ cannot execute.

We must \textit{suspend} the execution of $e$ until later.

New programming construct: \textbf{box}

\begin{itemize}
  \item if $e$ is an expression, then \textbf{box} $e$ is a suspended expression.
\end{itemize}
Activating suspended expressions

- Say $e_1$ is a suspended expression *reading* from locations in $C$.
- We may want to activate $e_1$ once locations in $C$ are initialized.
- New programming construct: \textbf{let box} $u = e_1 \textbf{ in } e_2$
  - $u$ is bound to the expression suspended by $e_1$
  - $e_2$ may use $u$ under many different environments
Example

- Assume $X$ : int is an allocated but not initialized memory location

- Consider the program:

  let box $u = box (X + 1)$
  in
  $\langle X \rightarrow 0 \rangle (\langle X \rightarrow 1 \rangle u, u)$
  end

- In this program:
  - $X + 1$ is substituted for $u$
  - and then executed in two different environments:
    - once in environment $\langle X \rightarrow 1 \rangle$,
      and again in the environment $\langle X \rightarrow 0 \rangle$
  - to produce result (2, 1)
Type system for writes

- Associate writing expression $f$ with its type $A$
  \[ \Delta \vdash f : A \]

- $\Delta$: types and supports for variables
Type system for writes

- Associate writing expression $f$ with its type $A$ and set of locations $C$
  \[ \Delta \vdash f \div C \ A \]

- $\Delta$: types and supports for variables
- $C$: set of locations that $f$ writes into
Type system for writes

- Associate writing expression \( f \) with its type \( A \) and set of locations \( C \)

\[
\Delta \vdash f \div C \ A[D]
\]

- \( \Delta \): types and supports for variables
- \( C \): set of locations that \( f \) writes into
- \textit{Support} \( D \): locations that \( f \) may first read from.
• Say $f$ is an expression that *writes* into locations $C$.

• We may want to *suspend* $f$, and *activate* it later when needed.

• New programming construct **dia**:
  – if $f$ is a writing expression, then
    **dia** $f$ is a suspended expression.

• New programming construct: **let dia** $x = e$ **in** $f$
  – executes the writes suspended by $e$
  – computes the value of $e$ in the changed environment
  – binds that value to $x$, to be used in $f$

• *Note*: choice of construct names will become clear soon.
Types for suspensions

- Assume
  - $e_1$ is a suspended expression that *reads* from location $X:B$ before returning a value of type $A$
  - $e_2$ is a suspended expression that *writes* into location $X:B$ before returning a value of type $A$

- Reformulation:
  - $e_1$ computes a value of type $A$ in *all* environments in which $X$ is initialized
  - there *exists* an environment (the one obtained after $e_2$ writes into $X$) in which $e_2$ computes a value of type $A$

- Type for $e_1$: *universal* quantification over environments
- Type for $e_2$: *existential* quantification over environments
Modal logic

- Reasoning about truth across various worlds
- New propositions: \( \square A \) and \( \Diamond A \)
- \( \square A \equiv \text{necessarily } A \)
  \( \square A \) is true iff \( A \) is true in all worlds
- \( \Diamond A \equiv \text{possibly } A \)
  \( \Diamond A \) is true iff \( A \) is true in some world
Partial modal logic

- Introduce a condition $C$ that may or may not be satisfied at any given world.

- New propositions: $\Box_C A$ and $\Diamond_C A$

- $\Box_C A$ is true iff $A$ is true in all worlds that satisfy $C$.

- $\Diamond_C A$ is true iff there exists a world which satisfies $C$ and in which $A$ is true.
Modal types for memory

- **Worlds == Environments == States of memory**
- **Support C**: set of allocated memory locations
- **type □_C A**: 
  - computation that may *read* from locations in C before returning a value of type A
  - may be executed in *any* environment in which locations C are initialized
  - C is a *precondition*
- **type ◊_C A**: 
  - computation that *writes* into locations in C before computing a value of type A
  - there *exists* an environment (i.e. the one obtained after the write) in which a value may be computed
- **C** is a *postcondition*
Characteristic properties

- Emphasis on environments.

- Related ideas:
  - type distinction between pure and effectful computation
  - Monads [Moggi’91, Wadler’95, Wadler’98]
  - Effect systems [Gifford, Lucassen’86, Talpin, Jouvelot’92]

- In contrast to monads and effect systems, in modal logic:
  - More than one type operator. Even families of operators.
  - Operators not necessarily monadic in nature:
    - \( \diamond \) is a monad
    - \( \Box \) is a comonad [Kobayashi’97, Bierman, de Paiva’00]
  - Monads give rise to lax logic, which is a simple modal logic.
Type system for writes

- Type assignment for suspended writes:

\[
\Delta \vdash f \div_{C} A \\
\Delta \vdash \text{dia } f : \Diamond_{C} A
\]

Conclusion: writes are serialized. Expressions may depend on \( C_2 \) and \( D \).
Type system for writes

- Type assignment for suspended writes:
  \[
  \Delta \vdash f \vdash_C A \\
  \Delta \vdash \text{dia } f : \Diamond_C A
  \]

- Type assignment for activating writes:
  \[
  \Delta \vdash e_1 : \Diamond_C A \\
  (\Delta, x:A) \vdash f_2 \vdash B [C]
  \]
  \[
  \Delta \vdash \text{let dia } x = e_1 \text{ in } f_2 \vdash B
  \]
  - Note: \( f_2 \) may read from the locations \( C \) that \( e_1 \) wrote
  - Conclusion: writes are \textit{serialized}
Type system for writes

- Type assignment for suspended writes:

\[ \Delta \vdash f \vdash_C A[D] \]
\[ \Delta \vdash \text{dia } f : \diamond_C A[D] \]

- Type assignment for activating writes:

\[ \Delta \vdash e_1 : \diamond_C A[D] \quad (\Delta, x:A) \vdash f_2 \vdash_{C_2} B[C] \]
\[ \Delta \vdash \text{let dia } x = e_1 \text{ in } f_2 \vdash_{C_2} B[D] \]

- Note: \( f_2 \) may read from the locations \( C \) that \( e_1 \) wrote
- Conclusion: writes are *serialized*
- Expressions may depend on \( C_2 \) and \( D \)
Type system for reads

- Type assignment for suspended reads:

\[
\Delta \vdash e : A[C] \\
\Delta \vdash \textbf{box} \ e : \Box_C A
\]

- Type assignment for activating reads:

\[
\Delta \vdash e_1 : \Box_C A \\
(\Delta, u : A[C]) \vdash e_2 : B \\
\Delta \vdash \textbf{let} \ \textbf{box} \ u = e_1 \ \textbf{in} \ e_2 : B
\]

- Note: the reads of \(e_1\) may be \textit{duplicated} by \(e_2\)
- Conclusion: reads are not serialized
Type system for reads

- Type assignment for suspended reads:
  \[
  \Delta \vdash e : A [C] \\
  \Delta \vdash \text{box } e : \Box_C A [D]
  \]

- Type assignment for activating reads:
  \[
  \Delta \vdash e_1 : \Box_C A [D] \quad (\Delta, u : A [C]) \vdash e_2 : B [D] \\
  \Delta \vdash \text{let box } u = e_1 \text{ in } e_2 : B [D]
  \]

- Note: the reads of \( e_1 \) may be duplicated by \( e_2 \)
- Conclusion: reads are not serialized
- Expressions may depend on support \( D \)
What is the *basic* expression that writes into memory locations $C$ before computing a value?

It is a pair $[\Theta, e]$ consisting of:
- a part $\Theta$ that writes into locations $C$
- a part $e$ that computes a value

Typing assignment:

$$
\Delta \vdash \langle \Theta \rangle : C \quad \Delta \vdash e : A [C] \\
\therefore \Delta \vdash [\Theta, e] \div C A
$$

$\Theta$ is an *environment definition* which is used to change the current state of memory.
Logical correspondence

- Relationship to logic: for each axiom of modal logic, there is a corresponding program in the language.
- Examples in $\lambda$-calculus notation.

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

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

\[ f_3 : \square (A \rightarrow B) \rightarrow \square A \rightarrow \square B = \lambda x. \lambda y. \text{let } box\ u = x \text{ in } \text{let } box\ v = y \text{ in } box\ (u\ v) \]
Logical correspondence

- Relationship to logic: for each axiom of modal logic, there is a corresponding program in the language.
- Examples in $\lambda$-calculus notation.

\[
f_1 : \Box A \to A = \\
\quad \lambda x. \ let \ box \ u = x \ in \ u
\]

\[
f_2 : \Box C A \to \Box \Box C A = \\
\quad \lambda x. \ let \ box \ u = x \ in \ box (box \ u)
\]

\[
f_3 : \Box C (A \to B) \to \Box D A \to \Box C,D B = \\
\quad \lambda x. \ \lambda y. \ let \ box \ u = x \ in \ let \ box \ v = y \ in \ box (u \ v)
\]
Logical correspondence

- Relationship to logic: for each axiom of modal logic, there is a corresponding program in the language.
- Examples in $\lambda$-calculus notation.

$$g_1 : A \rightarrow \diamond A = \lambda x. \text{dia } x$$

$$g_2 : \diamond \diamond A \rightarrow \diamond A = \lambda x. \text{dia} \ (\text{let dia } y = x \text{ in let dia } z = y \text{ in } z)$$

$$g_3 : \Box (A \rightarrow B) \rightarrow \diamond A \rightarrow \diamond B = \lambda e_1. \lambda e_2. \text{dia} \ (\text{let box } u = e_1 \text{ in let dia } x = e_2 \text{ in } u \ x)$$
Logical correspondence

- Relationship to logic: for each axiom of modal logic, there is a corresponding program in the language.
- Examples in $\lambda$-calculus notation.

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

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

\[ g_3 : \Box_C (A \rightarrow B) \rightarrow \Diamond_D A \rightarrow \Diamond_D B = \lambda e_1. \, \lambda e_2. \, \text{dia} (\text{let box } u = e_1 \text{ in let dia } x = e_2 \text{ in } u \, x) \]
Summary of the calculus for memory

- Type $\square_C$: computation *reading* from locations $C$
- Type $\Diamond_C$: computation *writing* into locations $C$
  - universal vs. existential quantification over states of memory
  - direct correspondence to logic
  - can be viewed as computations with pre- and postconditions

- Writes into memory:
  - *non-destructive* if done by environment definition
  - *destructive* if done by modal possibility

- Theorem: programs with empty support do not get stuck.

- Part of the language containing only $\square$: 
  - a type-safe version of *dynamic binding*. 
Outline

- Introduction ✓
- Modal logic for memory ✓
- Modal logic for control effects
- Modal logic for metaprogramming
- Summary and future work
Type system for exceptions

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

  $\Delta \vdash e : A[C]$

- Support: set of exceptions that $e$ may raise
Exception handling

- Environment definition specifies exception handling
  \[ \Theta = \langle X_1 w_1 \to e_1, \ldots, X_n w_n \to e_n \rangle \]

- Notation: To execute expression \( e \) in environment \( \Theta \), write
  \[ e \text{ handle } \Theta \]

- Scope of environment definition is delimited
  \[ (e_1 \text{ handle } \Theta_1, e_2) \text{ handle } \Theta_2 \]

- Raising exceptions == \textit{extend} support
- Handling exceptions == \textit{shrink} support
Example: exceptions

- Let \( X, Y : \text{int} \equiv \text{integer exceptions} \)

<table>
<thead>
<tr>
<th>expression</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>( X )</td>
</tr>
<tr>
<td>handle ( Y y \rightarrow y + 2 )</td>
<td></td>
</tr>
<tr>
<td>(( \text{raise}_X 1 + \text{raise}_Y 2 ))</td>
<td></td>
</tr>
<tr>
<td>handle ( Y y \rightarrow y + 2 ), empty</td>
<td></td>
</tr>
<tr>
<td>( X x \rightarrow x + 3 )</td>
<td></td>
</tr>
</tbody>
</table>
### Example: exceptions

- Let $X, Y : \text{int} \Rightarrow \text{integer exceptions}$

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

- Theorem: expressions with empty support do not get stuck.
Just like with memory, we want to have a modal type for suspended exceptional computations.

But, should we use □ or ◊?
• Just like with memory, we want to have a modal type for suspended exceptional computations.

• But, should we use $\Box$ or $\Diamond$?

• Exceptional computations have a precondition, but no postcondition.

• Reformulate in terms of interaction with environments:

  computation of type $A$ raising exceptions from the set $C$

  $\equiv$

  *for every* handler for exceptions in $C$, return value of type $A$
Modal necessity for exceptions

- Type $\square_C A$: suspended expression of type $A$ possibly raising exceptions from the set $C$.

- The language constructs identical to the ones for memory reads.
  - box suspends exceptional expressions
  - let box activates suspended expressions
  - typing rules are same as before
Example

- Assume $E$\text{:int} is an exception.
- Function $\text{mult} : \text{intlist} \rightarrow \Box E \text{int}$ multiplies elements of a list, raising exception $E$ if encounters 0.

\[
\text{fun mult (nil) = box (1)} \\
\quad | \quad \text{mult (hd :: tl) =} \\
\quad \quad | \quad \text{if hd = 0 then box (raise E 0)} \\
\quad \quad \quad \text{else} \\
\quad \quad \quad \quad | \quad \text{let box u = mult tl in box (hd * u)}
\]

- Results of $\text{mult}$ must be \textit{activated} e.g. $\text{mult [2, 1, 0]}$ is itself suspended, but

\[
(\text{let box u = mult [2,1,0] in u}) \text{ handle } E x \rightarrow x
\]

evaluates to 0.
Outline

- Introduction ✓
- Modal logic for memory ✓
- Modal logic for control effects ✓
- Modal logic for metaprogramming
- Summary and future work
Metaprogramming

• Writing code that generates/compiles/inspects other code.

• Main applications:
  – dynamic reconfiguration in face of changing run-time conditions
  – representation of abstract syntax

• Assume here: generated code is *source code*, i.e.
it is a syntactic entity.

• Type safety:
  – *Non-interference*: source code and compiled code are separated
  – Well-typed compiled programs generate well-typed source code

• Related work: ModalML [Wickline, Lee, Pfenning, Davies’98],
  MetaML [Taha, Sheard ’97], FreshML [Pitts, Gabbay’00], \( \lambda_{\text{code}} \)
  [Chen, Xi’03]
Metaprogramming operations

- Source code is generated by substitution into larger context.

- This substitution incurs capture of free variables of $e$.

- The substitution may be viewed as interaction with environments.
  - environments are syntactic contexts.
  - notion of interaction is capture of free variables.
• Capture-admitting variables are different from bound variables, and are represented by names.

• Assume $X$:int is a name.

• Type $\square_X A$:
  – source code of type $A$ possibly depending on the name $X$
  – can be substituted into any context capable of capturing $X$

• The language constructs identical to the ones for memory reads.
  – box encapsulates source code
  – let box substitutes source code into a larger context
  – typing rules very similar to before
Example: exponentiation

- How to create a function exp

  \[ \text{exp} : \text{int} \rightarrow \Box(\text{int} \rightarrow \text{int}) \]

  so that \( \text{exp}(n) \) generates source code for exponentiation by \( n \)?

- Example: \( \text{exp}(3) \) should produce \( \text{box}(\lambda y. y \ast y \ast y) \).
Example: exponentiation

- Assume $X:\text{int}$ is a name
- Create helper function $\text{exp'} : \text{int} \rightarrow \Box X \text{ int}$ so that $\text{exp'}(n) = \text{box} (\underbrace{X \ast \cdots \ast X}_n)$.

```
fun exp' (n : int) : \Box X \text{ int} =
  if n = 1 then box (X)
  else
    let box u = exp' (n - 1)
    in
    box (X \ast u)
  end
```
Once we have \( \text{exp}'(n) = \text{box } \left( \underbrace{X \times \cdots \times X}_n \right) \),

we can \textit{capture} \( X \) with a bound variable.

\[
\text{fun exp } (n) = \\
\text{let box } u = \text{exp'} (n) \\
\text{in} \\
\text{box } (\lambda y. \langle X \rightarrow y \rangle u) \\
\text{end}
\]

Example: \( \text{exp}(3) \) evaluates to \( \text{box } (\lambda y. \, y \times y \times y) \)
Logical frameworks and unification

- Metaprogramming system extendable to dependent types [Nanevski,Pfenning,Pientka’05].

- Precisely captures important invariants of unification in Twelf.
  - Existential variables introduced during proof search depend on a local context of ordinary variables.
  - Existential variable depending on the local context $C$ corresponds to a source code variable with support $C$.
  - Logical explanation of many implementation strategies that have previously been justified algorithmically (e.g. lowering, raising, grafting).
Outline

- Introduction ✓
- Modal logic for memory ✓
- Modal logic for control effects ✓
- Modal logic for metaprogramming ✓
- Summary and future work
Summary

- A type system that can *uniformly* represent:
  - memory reads and writes
  - exception raising and handling
  - source code with free variables and substitution with capture

- Expression $e : \Box_X A$:
  - reads from memory location $X$
  - raises exception $X$
  - is a source code depending on name $X$

- Expression $e : \Diamond_X A$:
  - writes into memory location $X$
Summary

- The language is based on logic:
  - hence, it is very general and will likely be very extendable
  - the logic in question is a version of modal logic

- Main idea: emphasis on interaction between programs and environments.

- Related work:
  - type distinction between pure and effectful computation
  - Monads [Moggi’91, Wadler’95, Wadler’98]
  - Effect systems [Gifford, Lucassen’86, Talpin, Jouvelot’92]
Future work

- Can we use modal logic to capture more complex environments?
- Example:
  - parallel and mobile programs with distributed data
  - security

- Type $\square_x A$ may stand for expressions that:

- Type $\diamond_x A$ may stand for expressions that:
Future work

- Can we use modal logic to capture more complex environments?
- Example:
  - parallel and mobile programs with distributed data
  - security
- Type $\square_X A$ may stand for expressions that:
  - can execute on *every* machine with resource $X$
- Type $\diamond_X A$ may stand for expressions that:
  - can execute on *some* machine with resource $X$
Future work

- Can we use modal logic to capture more complex environments?
- Example:
  - parallel and mobile programs with distributed data
  - security

- Type $\Box_X A$ may stand for expressions that:
  - can execute on *every* machine with resource $X$
  - are *encrypted* by the key $X$

- Type $\Diamond_X A$ may stand for expressions that:
  - can execute on *some* machine with resource $X$
  - *carry* the key $X$ and a computation encrypted by $X$
Future work

- Can we use modal logic to capture more complex environments?
- Example:
  - parallel and mobile programs with distributed data
  - security

- Type $\Box_X A$ may stand for expressions that:
  - can execute on *every* machine with resource $X$
  - are *encrypted* by the key $X$
  - can execute only with *permission* for security level $X$

- Type $\Diamond_X A$ may stand for expressions that:
  - can execute on *some* machine with resource $X$
  - *carry* the key $X$ and a computation encrypted by $X$
  - *carry* permission for security level $X$ and a computation at level $X$
Future work

- Supports may be *general propositions* about environments.

- Then the type $\Box_P \Diamond_Q A$ can be viewed as a *Hoare triple*.
  - It’s a computation with precondition $P$, postcondition $Q$, returning a value of type $A$

- When combined with dependent types, results with a type-theoretic analog of Hoare logic for higher-order functions.

- From the programming standpoint, it’s a dependently typed monadic calculus.

- Ongoing work with Greg Morrisett.
• Programming becomes more complex as run-time environments become more diverse, dynamic, distributed.

• Imposing a typing discipline on environments can help manage this complexity.
  – And it requires a logic that can capture diverse environment properties.

• Modal logic(s) capture environment properties.
  – Modal framework is diverse; there are many modal logics.
  – It is applicable to numerous challenges in language design.