Dependent Type Theory of Stateful Higher-Order Functions

Aleksandar Nanevski
Harvard University

joint with Greg Morrisett and Lars Birkedal

TYPES 2006, Nottingham
April 20, 2006
Dependent type theory

- Type theory is a program logic:
  - types can express and enforce precise program properties

- Doubles up as a programming language.

- Prototypical higher-order language (e.g., polymorphism, inductive/recursive types, subset types, etc.)

- Problem: must be purely functional
  - recursion allowed, if you prove termination
  - effects like state, IO, etc., usually second class
Hoare Logic

• Logic for imperative programs.

• Specifies partial correctness via Hoare triple \( \{ P \} \ E \{ Q \} \):
  – if \( P \) holds, then \( E \) diverges or terminates in a state \( Q \)
  – \( P \): precondition
  – \( Q \): postcondition

• Usually targets first-order languages
  – but recent advances in the higher-order case

• Reasoning about state and aliasing very streamlined
  – Separation Logic by O’Hearn, Pym, Reynolds, Yang...
Why not integrate Hoare Logic into a Type Theory?

Benefits:
- types can enforce correct use of effectful programs
- add effects to type theory
- preserves equational reasoning about pure programs

Idea: follow specifications-as-types principle
- Type of Hoare triples \( \{P\} x : A \{Q\} \)
- precondition \( P \), postcondition \( Q \), return result of type \( A \).
- Dependencies allow \( P \) and \( Q \) to talk about program data.

In this talk: Hoare Type Theory (HTT)
- for reasoning about state and aliasing
Outline

- Introduction ✓
- Assertion logic
- Types and terms
- Typechecking
- Conclusions
• Partial functions, assigning to each natural number at most one value.

• Assertion $\text{select}_\tau(H, M, N)$:
  – In the heap $H$, location $M$ points to $N : \tau$.

• Function $\text{update}_\tau(H, M, N)$:
  – Returns a new heap in which $M$ points to $N : \tau$.

• $\tau$ is a monomorphic type.
Axioms on heaps

- McCarthy’s axioms for functional arrays.
  
  (ax1) \( \text{seleq}_A(\text{upd}_A(H, M, N), M, N) \)
  
  (ax2) \( M_1 \neq M_2 \land \text{seleq}_A(\text{upd}_B(H, M_1, N_1), M_2, N_2) \supset \text{seleq}_A(H, M_2, N_2) \)

- And:
  
  (ax3) \( \text{seleq}_A(\text{empty}, M, N) \supset \bot \)
  
  (ax4) \( \text{seleq}_A(H, M, N_1) \land \text{seleq}_A(H, M, N_2) \supset N_1 = N_2 \)
Assertions

- Classical multi-sorted first-order logic with equality
- Sorts: heaps and all types of HTT
- Plus: type polymorphism (predicative)
- Examples
  - Heap equality can be defined:
    \[ H_1 = H_2 \equiv \forall l:\text{nat}. \forall \alpha. \forall x: \alpha. \]
    \[ \text{seleq}_\alpha(H_1, l, x) \subseteq \text{seleq}_\alpha(H_2, l, x) \]
  - Also definable: disjoint union \( H = H_1 \uplus H_2 \)
Some derived assertions

- We can define propositions from Separation Logic.
  - Variable $\text{mem}$ denotes current heap.

\[
\begin{align*}
\text{emp} & \equiv (\text{mem} = \text{empty}) \\
M \rightarrow_{\tau} N & \equiv (\text{mem} = \text{upd}_{\tau}(\text{empty}, M, N)) \\
M \leftarrow_{\tau} N & \equiv \text{seleq}_{\tau}(\text{mem}, M, N) \\
P \ast Q & \equiv \exists h_1, h_2: \text{heap}. (\text{mem} = h_1 \uplus h_2) \\
& \quad \land [h_1/\text{mem}] P \land [h_2/\text{mem}] Q \\
P \rightarrow* Q & \equiv \forall h_1, h_2: \text{heap}. (h_2 = h_1 \uplus \text{mem}) \\
& \quad \supset [h_1/\text{mem}] P \supset [h_2/\text{mem}] Q \\
\text{this}(H) & \equiv (\text{mem} = H)
\end{align*}
\]
Example: swap

- Swap content of locations $x$ and $y$ (here natural numbers).
- Spec with no aliasing between $x$ and $y$:
  - $\alpha, \beta$: type variables

\[
\text{swap} : \forall \alpha. \forall \beta. \forall x: \text{nat}. \forall y: \text{nat}.
\begin{align*}
\{ x \mapsto_{\alpha} m \} & \ast \{ y \mapsto_{\beta} n \} : 1 \\
\{ x \mapsto_{\beta} n \ast y \mapsto_{\alpha} m \}
\end{align*}
\]

- For a spec with aliasing, use $\land$ instead of $\ast$
Example: swap

- Swap content of locations $x$ and $y$ (here natural numbers).
- Spec with no aliasing between $x$ and $y$:
  - $\alpha$, $\beta$: type variables

\[
\text{swap} : \forall \alpha . \forall \beta . \Pi x : \text{nat} . \Pi y : \text{nat} .
\]
\[
m : \alpha . n : \beta . \{ x \mapsto_\alpha m \ast y \mapsto_\beta n \} r : 1
\]
\[
\{ x \mapsto_\beta n \ast y \mapsto_\alpha m \}
\]

- For a spec with aliasing, use $\wedge$ instead of $\ast$
- $m$, $n$: dummy variables
Outline

- Introduction ✓
- Assertion logic ✓
- Types and terms
- Typechecking
- Conclusions
Type structure

- **Primitive types:** nat, bool, 1
- **Dependent functions:** \( \Pi x : A. B \) – standard
- **Polymorphic types:** \( \forall \alpha. A \) – standard
- **Hoare types:** \( \{P\} x : A \{Q\} \)
  - Hoare types are monads
  - encapsulate effectful computations
  - but also formalize reasoning by strongest postconditions
Term structure

- Pure fragment: higher-order functions, polymorphism...

- Impure fragment – first-order imperative language
  - sequence of commands, ending with a return value
  - primitives for allocation, strong update, lookup, deallocation, conditionals, recursion
  - recursive functions must be annotated with a type

- Monadic constructs:
  - \( \text{dia } E \)
    - suspends the effectful computation \( E \)
    - suspension is pure, so it can appear in types
  - \( \text{let dia } x = M \text{ in } E \)
    - run \( M \), then \( E \)
Monadic terms

- Definition and typing of characteristic monadic terms:

  \[ \text{unit} : \ A \rightarrow M(A) = \lambda x. \text{dia} \ x \]

  \[ \text{map} : \ (A \rightarrow B) \rightarrow M(A) \rightarrow M(B) = \lambda f. \lambda x. \text{dia} (\text{let} \ x = y \text{ in } f \ y) \]

  \[ \text{idemp} : \ M(M(A)) \rightarrow M(A) = \lambda x. \text{dia} (\text{let} \ y = x \text{ in } \text{let} \ z = y \text{ in } z) \]
Monadic terms

- Definition and typing of characteristic monadic terms:
  
  \[
  \begin{align*}
  \text{unit} &: A \rightarrow M(A) = \\
  &\quad \lambda x. \text{dia } x \\
  \text{map} &: (A \rightarrow B) \rightarrow M(A) \rightarrow M(B) = \\
  &\quad \lambda f. \lambda x. \text{dia } (\text{let } \text{dia } y = x \text{ in } f \ y) \\
  \text{idemp} &: M(M(A)) \rightarrow M(A) = \\
  &\quad \lambda x. \text{dia } (\text{let } \text{dia } y = x \text{ in } \text{let } \text{dia } z = y \text{ in } z)
  \end{align*}
  \]

- Dependently typed unit:

  \[
  \begin{align*}
  \text{unit'} &: \Pi x:A. \{P\} y:A\{x = y \land P\} = \\
  &\quad \lambda x. \text{dia } x
  \end{align*}
  \]
Example: swap

- Swap content of $x$ and $y$

\[
\text{swap} : \forall \alpha. \forall \beta. \Pi x : \text{nat}. \Pi y : \text{nat}. \\
\quad m : \alpha. n : \beta. \{ x := \alpha \rightarrow m \ast y \leftarrow \beta n \} \ r : \text{unit} \\
\quad \{ x :\leftarrow \beta n \ast y \leftarrow \alpha m \} = \\
\Lambda \alpha. \Lambda \beta. \lambda x. \lambda y. \text{dia} (u = !x; v = !y; \\
\quad y : := u; x := v; \\
\quad ( ))
\]
Example: swap twice

- Swapping twice in a row is identity.

\[
\text{identity} = \Lambda \alpha. \Lambda \beta. \lambda x. \lambda y. \text{dia}(\text{let dia }_e = \text{swap } \alpha \beta x y \\
\text{dia }_e = \text{swap } \beta \alpha x y \\
\text{in} \\
\text{( )} \\
\text{end})
\]

- Heap invariance apparent from the type.

\[
\text{identity} : \forall \alpha. \forall \beta. \Pi x : \text{nat.} \Pi y : \text{nat.} \\
\text{m : } \alpha, \text{n : } \beta, \text{h : heap.} \{ (x \mapsto_\alpha \text{m} * y \mapsto_\beta \text{n}) \land \text{this(h)} \} \ r : 1 \\
\{ \text{this(h)} \}
\]
Outline

- Introduction ✓
- Assertion logic ✓
- Types and terms ✓
- Typechecking
- Conclusions
Typechecking by computing strongest postconditions.

Typechecking is completely syntax-directed.
- effectful programs are (part of) the proofs of their specs
- remaining part of the proof must discharge intermediate assertions
- no whole-program reasoning

Judgment: $\Delta; P \vdash E \Rightarrow x:A. Q$
- $\Delta$: variable context
- $E$: computation
- $P$: what holds before $E$ runs (precondition)
- $A$: return result
- $Q$: how the heap is changed after $E$ (strongest postcondition)
- $Q$ is output
Typechecking deallocation

- `deallocate(M); E`
  - deallocates memory at location `M`, and proceeds to run `E`
Typechecking deallocation

- `dealloc(M); E`
  - deallocates memory at location `M`, and proceeds to run `E`

- Typing rule:

  \[
  \Delta; P \vdash \text{dealloc}(M); E \Rightarrow y:B. Q
  \]
Typechecking deallocation

- `dealloc(M); E`  
  - deallocates memory at location `M`, and proceeds to run `E`

- Typing rule:

\[
\Delta \vdash M : \text{nat}
\]

\[
\Delta; P \vdash \text{dealloc}(M); E \Rightarrow y:B. Q
\]
Typechecking deallocation

- `dealloc(M); E` deallocates memory at location `M`, and proceeds to run `E`

- Typing rule:

\[
\Delta \vdash M : \text{nat} \\
\Delta \vdash P \supset (M \leftrightarrow -) \\
\hline
\Delta; P \vdash \text{dealloc}(M); E \Rightarrow y:B. Q
\]

- Proving `P \supset (M \leftrightarrow -)` can be postponed
Typechecking deallocation

- `dealloc(M); E` deallocates memory at location `M`, and proceeds to run `E`

- Typing rule:

\[
\begin{align*}
\Delta \vdash M : \text{nat} \\
\Delta \vdash P \supset (M \leftrightarrow -) \\
\Delta; P \vdash E \Rightarrow y : B. Q
\end{align*}
\]

\[
\Delta; P \vdash \text{dealloc}(M); E \Rightarrow y : B. Q
\]

- proving `P \supset (M \leftrightarrow -)` can be postponed
Typechecking deallocation

- `dealloc(M); E` 
  - deallocates memory at location `M`, and proceeds to run `E`

- Typing rule:

\[
\Delta \vdash M : \text{nat} \\
\Delta \vdash P \supset (M \leftrightarrow -) \\
\Delta; P \circ ((M \leftrightarrow -) \rightarrow \text{emp}) \vdash E \Rightarrow y:B. Q \\
\Delta; P \vdash \text{dealloc}(M); E \Rightarrow y:B. Q
\]

- proving `P \supset (M \leftrightarrow -)` can be postponed
- `P \circ (R_1 \rightarrow R_2)` is a heap obtained by switching `R_1` with `R_2` in `P`
- connectives `\circ` and `\rightarrow` definable in HTT, but independent of `*` and `→`
Soundness

- In addition to equational theory, we define call-by-value operational semantics.
- Soundness must show that $P \vdash E \Rightarrow x:A. Q$ indeed has the intuitive semantics.
- Soundness requires Preservation and Progress (as usual in type systems) but here much stronger.
- Preservation: evaluation preserves types and canonical forms.
- Progress: well-typed programs do not get stuck.
- Progress depends on the soundness of the assertion logic.
  - assertion logic soundness proved by simple denotational argument.
Related work

- Extended static checking tools: ESC/Java, SPliint, Spec#, Cyclone...
  - Hoare-like annotations verified during type checking
  - but usually no semantic foundations

- Dependent types and effects ([Zhu, Xi’05], [Shao, Trifonov, Saha, Papaspyrou’05])
  - but types cannot depend on effectful programs

- Hoare Logic for higher-order functions ([Schröder, Mossakowski’02], [Honda, Berger, Yoshida’05])
  - simply typed underlying language (with effects)
  - Hoare triples do not integrate into a type system
Conclusions

- HTT is a type-theoretic version of Hoare Logic
  - dually: Hoare Logic for a dependently typed language
  - dually: Type Theory with monadic effects
- Specifications-as-types principle via monad \( \{P\} x : A \{Q\} \)
- Specifications like in Separation Logic.
- Definable connectives \( \ast \) and \( \ast \) from Separation Logic (but new connectives \( \circ \) and \( \circ \) also needed).
- Assertions checked by pushing strongest postconditions
- Proofs-as-programs principle (modulo proofs of assertion) guarantees no need for whole-program reasoning
- Paper available at: http://www.eecs.harvard.edu/~aleks
Future work

- Higher-order assertion logic
- Cook completeness
- Abstract types
- Local state
- Hoare logic for concurrency and runST
Example

- Swapping twice in a row is identity.

\[
\text{identity} : \forall \alpha. \forall \beta. \Pi x : \text{nat}. \Pi y : \text{nat}. \\
\quad \quad m : \alpha, n : \beta, h : \text{heap}. \{(x \mapsto_\alpha m \ast y \mapsto_\beta n) \land \text{this(h)}\} \quad r : 1 \\
\quad \quad \{\text{this(h)}\} = \\
\Lambda \alpha. \Lambda \beta. \lambda x. \lambda y. \text{dia(let dia u = swap } \alpha \beta x y \\
\text{dia v = swap } \beta \alpha x y \\
\text{in} \\
\quad \quad ( ) \\
\text{end)}
\]
Monadic equations

- Equational theory [Pfenning, Davies’99]
- Implements monadic laws, but as $\beta$ and $\eta$ rules.

\[
\text{let dia } x = \text{dia } E \text{ in } F \implies_\beta \langle E/x \rangle F
\]
\[
M : \{ P \} x : A \{ Q \} \implies_\eta \text{dia (let dia } x = M \text{ in } x)\]

- Where $\langle E/x \rangle F$ is monadic linearization

\[
\langle M/x \rangle F = [M/x] F
\]
\[
\langle \text{command; } E''/x \rangle F = \text{command; } \langle E''/x \rangle F
\]
\[
\langle \text{let dia } y = E' \text{ in } E''/x \rangle F = \text{let dia } y = E' \text{ in } \langle E''/x \rangle F
\]
Example: swap

- Swap content of locations \( x \) and \( y \) (here natural numbers).
  - Spec with no aliasing between \( x \) and \( y \):

\[
\text{swap} : \forall \alpha, \beta. \Pi x, y : \text{nat}.
\]
\[
m : \alpha. n : \beta. \{ x \mapsto_\alpha m \ast y \mapsto_\beta n \} r : 1
\]
\[
\{ x \mapsto_\beta n \ast y \mapsto_\alpha m \}
\]

- Spec with aliasing between \( x \) and \( y \):

\[
\text{swap} : \forall \alpha, \beta. \Pi x, y : \text{nat}.
\]
\[
m : \alpha. n : \beta. h : \text{heap}. \{ x \mapsto_\alpha m \wedge y \mapsto_\beta n \wedge \text{this}(h) \} r : 1
\]
\[
\{ \text{this}(\text{upd}_\beta(\text{upd}_\alpha(h, y, m), x, n)) \}
\]

- \( m, n, h \) – dummy variables
Typechecking allocation

- $x = \text{alloc}_\tau(M); E$
  - allocates memory and initializes with $M:\tau$
  - $x$ binds the address of allocated memory
Typechecking allocation

- \( x = \text{alloc}_\tau(M); E \)
  - allocates memory and initializes with \( M:\tau \)
  - \( x \) binds the address of allocated memory

- Typing rule:

\[
\Delta; P \vdash x = \text{alloc}_\tau(M); E \Rightarrow y:B.
\]
Typechecking allocation

- $x = \text{alloc}_\tau(M); \ E$
  - allocates memory and initializes with $M:\tau$
  - $x$ binds the address of allocated memory

- Typing rule:

\[
\Delta \vdash \tau : \text{type}
\]

\[
\Delta; P \vdash x = \text{alloc}_\tau(M); \ E \Rightarrow y:B.
\]
Typechecking allocation

- $x = \text{alloc}_\tau(M) \; ; \; E$
  - allocates memory and initializes with $M:\tau$
  - $x$ binds the address of allocated memory

- Typing rule:

\[
\begin{align*}
\Delta \vdash \tau : \text{type} \\
\Delta \vdash M : \tau
\end{align*}
\]

\[
\Delta; P \vdash x = \text{alloc}_\tau(M) \; ; \; E \Rightarrow y : B.
\]
Typechecking allocation

- \( x = \text{alloc}_\tau(M); E \)
  - allocates memory and initializes with \( M:\tau \)
  - \( x \) binds the address of allocated memory

- Typing rule:

\[
\begin{align*}
\Delta & \vdash \tau : \text{type} \\
\Delta & \vdash M : \tau \\
\Delta, x : \text{nat}; & \quad \vdash E \Rightarrow y : B. \ Q \\
\Delta ; P & \vdash x = \text{alloc}_\tau(M); E \Rightarrow y : B.
\end{align*}
\]
Typechecking allocation

- $x = \text{alloc}_\tau(M); E$
  - allocates memory and initializes with $M:\tau$
  - $x$ binds the address of allocated memory

Typing rule:

\[
\begin{align*}
\Delta & \vdash \tau : \text{type} \\
\Delta & \vdash M : \tau \\
\Delta, x:\text{nat}; P \ast (x \mapsto_\tau M) & \vdash E \Rightarrow y:B. Q
\end{align*}
\]

\[
\Delta; P \vdash x = \text{alloc}_\tau(M); E \Rightarrow y:B. (\exists x:\text{nat}. Q)
\]

- $P \ast (x \mapsto_\tau M)$ means $x$ disjoint from $P$, and hence fresh.
Typechecking letdia

- Typing rule:

\[ \Delta; P \vdash \text{let dia } x = K \text{ in } E \Rightarrow y: B. (\exists x: A. Q) \]
Typechecking `letdia`

- Typing rule:

\[
\Delta \vdash K : \{R_1\} x : A \{R_2\}
\]

\[
\Delta; P \vdash \text{let dia } x = K \text{ in } E \Rightarrow y : B. (\exists x : A. Q)
\]
Typechecking \texttt{letdia}

- Typing rule:

\[
\begin{align*}
\Delta &\vdash K : \{R_1\}x:A\{R_2\} \\
\Delta &\vdash P \triangleright R_1 \ast \top \\
\hline
\Delta; P &\vdash \text{let dia } x = K \text{ in } E \Rightarrow y:B. (\exists x:A. Q)
\end{align*}
\]

- \( P \triangleright R_1 \ast \top \) implements “small footprints”
Typechecking letdia

- Typing rule:

\[
\begin{align*}
\Delta & \vdash K : \{R_1\} x : A \{R_2\} \\
\Delta & \vdash P \supset R_1 \ast \top \\
\Delta, x : A; P \circ (R_1 \rightarrow R_2) & \vdash E \Rightarrow y : B. \ Q \\
\Delta; P & \vdash \text{let dia } x = K \text{ in } E \Rightarrow y : B. (\exists x : A. \ Q)
\end{align*}
\]

- \( P \supset R_1 \ast \top \) implements “small footprints”
Typechecking dia

- Typing rule:

\[
\begin{align*}
\Delta; R_1 \vdash \top & \vdash E \Rightarrow x:A. \ P \\
\Delta & \vdash P \supseteq R_1 \rightarrow R_2 \\
\Delta & \vdash \text{dia } E : \{R_1\}x:A\{R_2\}
\end{align*}
\]

- Precondition \( R_1 \star \top \):
  - \( E \) can run in any heap with a fragment \( R_1 \)

- Strongest postcondition \( P \) must imply \( R_1 \rightarrow R_2 \)
  - the ending heap obtained from initial by swapping \( R_1 \) with \( R_2 \)