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...

Type theory for imperative programs

- 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

Heaps

- Partial functions, assigning to each natural number at most one value.
- Assertion $\mathsf{seleq}_\tau(H, M, N)$:
  - In the heap H, location M points to $N : \tau$.
- Function $\mathsf{upd}_\tau(H, M, N)$:
  - Returns a new heap in which M points to $N : \tau$.
- τ is a monomorphic type.

Axioms on heaps

- McCarthy's axioms for functional arrays.

$$\text{(ax1)}\quad \mathsf{seleq}_A(\mathsf{upd}_A(H, M, N), M, N)$$

$$\text{(ax2)}\quad M_1 \neq M_2 \wedge \mathsf{seleq}_A(\mathsf{upd}_B(H, M_1, N_1), M_2, N_2) \supset \mathsf{seleq}_A(H, M_2, N_2)$$

- And:

$$\text{(ax3)}\quad \mathsf{seleq}_A(\mathsf{empty}, M, N) \supset \bot$$

$$\text{(ax4)}\quad \mathsf{seleq}_A(H, M, N_1) \wedge \mathsf{seleq}_A(H, M, N_2) \supset N_1 = N_2$$

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:

$$H_1 = H_2 \quad \equiv \quad \forall l{:}\mathsf{nat}.\forall \alpha.\forall x{:}\alpha. \\ \mathsf{seleq}_\alpha(H_1, l, x) \subset\supset \mathsf{seleq}_\alpha(H_2, l, x)$$

  - Also definable: disjoint union $H = H_1 \uplus H_2$

Some derived assertions

- We can define propositions from Separation Logic.
  - Variable mem denotes current heap.

$$
\begin{array}{rcl}
\mathsf{emp} & \equiv & (\mathsf{mem} = \mathsf{empty}) \\
M \mapsto_\tau N & \equiv & (\mathsf{mem} = \mathsf{upd}_\tau(\mathsf{empty}, M, N)) \\
M \hookrightarrow_\tau N & \equiv & \mathsf{seleq}_\tau(\mathsf{mem}, M, N) \\
P * Q & \equiv & \exists h_1, h_2{:}\mathsf{heap}.(\mathsf{mem} = h_1 \uplus h_2) \\
 & & \wedge [h_1/\mathsf{mem}]P \wedge [h_2/\mathsf{mem}]Q \\
P \mathrel{-\!\!*} Q & \equiv & \forall h_1, h_2{:}\mathsf{heap}.(h_2 = h_1 \uplus \mathsf{mem}) \\
 & & \supset [h_1/\mathsf{mem}]P \supset [h_2/\mathsf{mem}]Q \\
\mathsf{this}(H) & \equiv & (\mathsf{mem} = H)
\end{array}
$$

Example: swap

- Swap content of locations x and y (here natural numbers).
- Spec with no aliasing between x and y:
  - α, β: type variables

$$\begin{aligned}\text{swap}{:}\forall\alpha.\forall\beta.\Pi x{:}\text{nat}.\Pi y{:}\text{nat}.\\ &\{x \mapsto_{\alpha} m * y \mapsto_{\beta} n\}r : 1\\ &\{x \mapsto_{\beta} n * y \mapsto_{\alpha} m\}\end{aligned}$$

- For a spec with aliasing, use $\wedge$ instead of $*$

Example: swap

- Swap content of locations x and y (here natural numbers).
- Spec with no aliasing between x and y:
  - α, β: type variables

$$\text{swap}{:}\forall\alpha.\forall\beta.\Pi x{:}\text{nat}.\Pi y{:}\text{nat}. \\ m{:}\alpha.n{:}\beta.\{x \mapsto_\alpha m * y \mapsto_\beta n\}r : 1 \\ \{x \mapsto_\beta n * y \mapsto_\alpha m\}$$

- For a spec with aliasing, use $\wedge$ instead of $*$
- 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:
  - dia E
    - suspends the effectful computation E
    - suspension is pure, so it can appear in types
  - let dia x = M in E
    - run M, then E

Monadic terms

- Definition and typing of characteristic monadic terms:

$$
\begin{array}{rcl}
\text{unit} & : & A \to M(A) = \\
& & \lambda x.\ \text{dia}\ x \\
\text{map} & : & (A \to B) \to M(A) \to M(B) = \\
& & \lambda f.\ \lambda x.\ \text{dia}\ \big(\text{let dia}\ y = x\ \text{in}\ f\ y\big) \\
\text{idemp} & : & M(M(A)) \to M(A) = \\
& & \lambda x.\ \text{dia}\ \big(\text{let dia}\ y = x\ \text{in let dia}\ z = y\ \text{in}\ z\big)
\end{array}
$$

Monadic terms

- Definition and typing of characteristic monadic terms:

$$
\begin{aligned}
\mathsf{unit} &: A \longrightarrow M(A) = \\
&\lambda x.\ \mathsf{dia}\ x \\
\mathsf{map} &: (A \longrightarrow B) \longrightarrow M(A) \longrightarrow M(B) = \\
&\lambda f.\ \lambda x.\ \mathsf{dia}\ (\mathsf{let\ dia}\ y\ \mathsf{=}\ x\ \mathsf{in}\ f\ y) \\
\mathsf{idemp} &: M(M(A)) \longrightarrow M(A) = \\
&\lambda x.\ \mathsf{dia}\ (\mathsf{let\ dia}\ y\ \mathsf{=}\ x\ \mathsf{in\ let\ dia}\ z\ \mathsf{=}\ y\ \mathsf{in}\ z)
\end{aligned}
$$

- Dependently typed unit:

$$
\begin{aligned}
\mathsf{unit'} &: \Pi x{:}A.\ \{P\}y{:}A\{x = y \wedge P\} = \\
&\lambda x.\ \mathsf{dia}\ x
\end{aligned}
$$

Example: swap

- Swap content of x and y

$$\text{swap} : \forall\alpha.\forall\beta.\ \Pi\text{x:nat}.\ \Pi\text{y:nat}.$$
$$\text{m:}\alpha.\ \text{n:}\beta.\ \{\text{x} \mapsto_\alpha \text{m} * \text{y} \mapsto_\beta \text{n}\}\ \text{r : unit}$$
$$\{\text{x} \mapsto_\beta \text{n} * \text{y} \mapsto_\alpha \text{m}\} =$$

$$\Lambda\alpha.\Lambda\beta.\ \lambda\text{x}.\ \lambda\text{y}.\ \text{dia (u = !x; v = !y;}$$
$$\text{y := u; x := v;}$$
$$\text{( ))}$$

Example: swap twice

- Swapping twice in a row is identity.

identity = $\Lambda\alpha.\Lambda\beta.\lambda$x.$\lambda$y. dia(let dia _ = swap α β x y
dia _ = swap β α x y
in
( )
end)

  - Heap invariance apparent from the type.

identity : $\forall\alpha.\forall\beta.\Pi$x:nat.$\Pi$y:nat.
m:α,n:β,h:heap.$\{$(x $\mapsto_\alpha$ m * y $\mapsto_\beta$ n) $\wedge$ this(h)$\}$ r : 1
$\{$this(h)$\}$

Outline

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

Judgments

- 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: $\Delta; P \vdash E \Rightarrow x{:}A.\ Q$
  - Δ: 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

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

Typechecking deallocation

- $\text{dealloc}(M); E$
  - deallocates memory at location M, and proceeds to run E
- Typing rule:

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

Typechecking deallocation

- dealloc$(M); E$
  - deallocates memory at location M, and proceeds to run E
- Typing rule:

$$\frac{\Delta \vdash M : \mathsf{nat}}{\Delta; P \vdash \mathsf{dealloc}(M); E \Rightarrow y{:}B.\ Q}$$

Typechecking deallocation

- $\mathsf{dealloc}(M); E$
  - deallocates memory at location M, and proceeds to run E
- Typing rule:

$$\frac{\begin{array}{c}\Delta \vdash M : \mathsf{nat} \\ \Delta \vdash P \supset (M \hookrightarrow -)\end{array}}{\Delta; P \vdash \mathsf{dealloc}(M); E \Rightarrow y{:}B.\ Q}$$

- proving $P \supset (M \hookrightarrow -)$ can be postponed

Typechecking deallocation

- dealloc$(M); E$
  - deallocates memory at location M, and proceeds to run E
- Typing rule:

$$\frac{\Delta \vdash M : \mathsf{nat} \quad \Delta \vdash P \supset (M \hookrightarrow -) \quad \Delta; \vdash E \Rightarrow y{:}B.\ Q}{\Delta; P \vdash \mathsf{dealloc}(M); E \Rightarrow y{:}B.\ Q}$$

- proving $P \supset (M \hookrightarrow -)$ can be postponed

Typechecking deallocation

- $\mathsf{dealloc}(M); E$
  - deallocates memory at location M, and proceeds to run E
- Typing rule:

$$\frac{\Delta \vdash M : \mathsf{nat} \qquad \Delta \vdash P \supset (M \hookrightarrow -) \qquad \Delta; P \circ ((M \mapsto -) \multimap \mathsf{emp}) \vdash E \Rightarrow y{:}B.\ Q}{\Delta; P \vdash \mathsf{dealloc}(M); E \Rightarrow y{:}B.\ Q}$$

- proving $P \supset (M \hookrightarrow -)$ can be postponed
- $P \circ (R_1 \multimap R_2)$ is a heap obtained by switching R_1 with R_2 in P
- connectives $\circ$ and $\multimap$ 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, SPlint, Spec#, Cyclone...
  - Hoare-like annotations verifi ed 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 $*$ and $-\!\!*$ 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.

$$\text{identity} : \forall\alpha.\forall\beta.\Pi\text{x:nat}.\Pi\text{y:nat}.$$
$$\text{m:}\alpha,\text{n:}\beta,\text{h:heap}.\{(\text{x} \mapsto_\alpha \text{m} * \text{y} \mapsto_\beta \text{n}) \wedge \text{this(h)}\}\ \text{r} : 1$$
$$\{\text{this(h)}\} =$$

$\Lambda\alpha.\Lambda\beta.\lambda$x.$\lambda$y. dia(let dia u = swap α β x y
dia v = swap β α x y
in
( )
end)

Monadic equations

- Equational theory [Pfenning,Davies'99]
- Implements monadic laws, but as β and η rules.

$$\begin{array}{rcl} \text{let dia } x = \text{dia } E \text{ in } F & \Longrightarrow_\beta & \langle E/x\rangle F \\ M : \{P\}x{:}A\{Q\} & \Longrightarrow_\eta & \text{dia } (\text{let dia } x = M \text{ in } x) \end{array}$$

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

$$\begin{array}{rcl} \langle M/x\rangle F & = & [M/x]F \\ \langle \text{command}; E''/x\rangle F & = & \text{command}; \langle E''/x\rangle F \\ \langle \text{let dia } y = E' \text{ in } E''/x\rangle F & = & \text{let dia } y = E' \text{ in } \langle E''/x\rangle F \end{array}$$

Example: swap

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

$$
\begin{aligned}
&\mathsf{swap}{:}\forall \alpha, \beta. \Pi x, y{:}\mathsf{nat}. \\
&\qquad m{:}\alpha. n{:}\beta. \{x \mapsto_\alpha m * y \mapsto_\beta n\} r : 1 \\
&\qquad\qquad \{x \mapsto_\beta n * y \mapsto_\alpha m\}
\end{aligned}
$$

  - Spec with aliasing between x and y:

$$
\begin{aligned}
&\mathsf{swap}{:}\forall \alpha, \beta. \Pi x, y{:}\mathsf{nat}. \\
&\qquad m{:}\alpha. n{:}\beta. h{:}\mathsf{heap}. \{x \hookrightarrow_\alpha m \wedge y \hookrightarrow_\beta n \wedge \mathsf{this}(h)\} r : \\
&\qquad\qquad \{\mathsf{this}(\mathsf{upd}_\beta(\mathsf{upd}_\alpha(h, y, m), x, n))\}
\end{aligned}
$$

- m, n, h – *dummy variables*

Typechecking allocation

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

Typechecking allocation

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

$$\frac{}{\Delta; P \vdash x = \mathsf{alloc}_\tau(M); E \Rightarrow y{:}B.}$$

Typechecking allocation

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

$$\frac{\Delta \vdash \tau : \mathsf{type}}{\Delta; P \vdash x = \mathsf{alloc}_\tau(M); E \Rightarrow y{:}B.}$$

Typechecking allocation

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

$$\frac{\begin{array}{c}\Delta \vdash \tau : \mathsf{type} \\ \Delta \vdash M : \tau\end{array}}{\Delta; P \vdash x = \mathsf{alloc}_\tau(M); E \Rightarrow y{:}B.}$$

Typechecking allocation

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

$$\frac{\Delta \vdash \tau : \textsf{type} \quad \Delta \vdash M : \tau \quad \Delta, x{:}\textsf{nat}; \vdash E \Rightarrow y{:}B.\ Q}{\Delta; P \vdash x = \textsf{alloc}_\tau(M); E \Rightarrow y{:}B.}$$

Typechecking allocation

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

$$\frac{\Delta \vdash \tau : \mathsf{type} \qquad \Delta \vdash M : \tau \qquad \Delta, x{:}\mathsf{nat}; P * (x \mapsto_\tau M) \vdash E \Rightarrow y{:}B.\ Q}{\Delta; P \vdash x = \mathsf{alloc}_\tau(M); E \Rightarrow y{:}B.\ (\exists x{:}\mathsf{nat}.Q)}$$

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

Typechecking letdia

- Typing rule:

$$\frac{}{\Delta; P \vdash \textsf{let dia } x = K \textsf{ in } E \Rightarrow y{:}B.\ (\exists x{:}A.\ Q)}$$

Typechecking letdia

- Typing rule:

$$\frac{\Delta \vdash K : \{R_1\}x{:}A\{R_2\}}{\Delta; P \vdash \mathsf{let\ dia}\ x = K\ \mathsf{in}\ E \Rightarrow y{:}B.\ (\exists x{:}A.\ Q)}$$

Typechecking letdia

- Typing rule:

$$\frac{\Delta \vdash K : \{R_1\}x{:}A\{R_2\} \qquad \Delta \vdash P \supset R_1 * \top}{\Delta; P \vdash \text{let dia } x = K \text{ in } E \Rightarrow y{:}B.\ (\exists x{:}A.\ Q)}$$

- $P \supset R_1 * \top$ implements "small footprints"

Typechecking letdia

- Typing rule:

$$\frac{\Delta \vdash K : \{R_1\}x{:}A\{R_2\} \qquad \Delta \vdash P \supset R_1 * \top \qquad \Delta, x{:}A; P \circ (R_1 \multimap R_2) \vdash E \Rightarrow y{:}B.\ Q}{\Delta; P \vdash \text{let dia } x = K \text{ in } E \Rightarrow y{:}B.\ (\exists x{:}A.\ Q)}$$

- $P \supset R_1 * \top$ implements "small footprints"

Typechecking dia

- Typing rule:

$$\frac{\Delta; R_1 * \top \vdash E \Rightarrow x{:}A.\ P \quad \Delta \vdash P \supset R_1 \multimap R_2}{\Delta \vdash \mathsf{dia}\ E : \{R_1\}x{:}A\{R_2\}}$$

- Precondition $R_1 * \top$:
  - E can run in any heap with a fragment R_1
- Strongest postcondition P must imply $R_1 \multimap R_2$
  - the ending heap obtained from initial by swapping R_1 with R_2