An algebraic perspective on behavioral specifications in effectful languages

Danel Ahman
LFCS, University of Edinburgh

PhD work supervised by Gordon Plotkin and Alex Simpson
Examples of reasoning about programs

- **Side-effects** - Hoare Logic, Separation Logic, Hoare Type Theory

- **Network behavior** - session types

- **Permissions** - ownership types

- **Information flow security** - security type systems

- **...** - refinement types, contracts, dependent types
Examples of reasoning about programs

- **Side-effects** - Hoare Logic, Separation Logic, Hoare Type Theory
  - specific to state

- **Network behavior** - session types - specific to i/o

- **Permissions** - ownership types - specific to permissions

- **Information flow security** - security type systems

- **...** - refinement types, contracts, dependent types - often specific to pure values or base types
Our long-term goals

- To develop a general theory of program specifications
  - accommodate various specification areas
    - state side-effects, network behavior, permissions, ...
  - accommodate various notions of computation
    - state, input/output, exceptions, probabilistic computation, handlers, ...
  - accommodate general logical specifications

- The tools we propose to use
  - Algebraic effects and their logics
  - Very fine-grained computational language (e.g., CBPV)
  - Refinement types (both on values and computations)
Computational effects and algebraic theories
Computational effects as monads

- Assume that we work in a cartesian-closed category $\mathcal{C}$
- Usual interpretation of $\lambda$-calculus
  - types as objects, terms as morphisms
- We model pure terms as morphisms $A \rightarrow B$
- But how to model terms that can produce effects?
- Moggi’s ’91 answer to this question: use monads!
  - $T : [\mathcal{C}, \mathcal{C}]$
  - $\eta_X : X \rightarrow TX$
  - $\mu_X : TTX \rightarrow TX$
- or alternatively
  - $T : \text{ob}(\mathcal{C}) \rightarrow \text{ob}(\mathcal{C})$
  - $\eta_X : X \rightarrow TX$
  - $f^* : TX \rightarrow TY$ for every $f : X \rightarrow TY$ in $\mathcal{C}$
Computational effects as monads

- Some of Moggi’s example monads:
  - Global state: \( TX = S \Rightarrow (S \times X) \)
  - Exceptions: \( TX = X + E \)
  - Input/output: \( TX = \mu Y.(V \Rightarrow Y) + (V \times Y) + X \)
  - Non-determinism: \( TX = \mathcal{P}(X) \)
  - Continuations: \( TX = (X \Rightarrow R) \Rightarrow R \)

- But, once we have one such monad, what would be the right effectful program constructs to create elements of \( TX \).

- And, once we have some effectful program constructs, what would be the right monad to model these effects?

- This is where algebraic effects will help us!
Computational effects as monads

- Some of Moggi’s example monads:
  - Global state: $TX = S \Rightarrow (S \times X)$
  - Exceptions: $TX = X + E$
  - Input/output: $TX = \mu Y. (V \Rightarrow Y) + (V \times Y) + X$
  - Non-determinism: $TX = \mathcal{P}(X)$
  - Continuations: $TX = (X \Rightarrow R) \Rightarrow R$

- But, once we have one such monad, what would be the right effectful program constructs to create elements of $TX$.

- And, once we have some effectful program constructs, what would be the right monad to model these effects?

- This is where algebraic effects will help us!
Computational effects as monads

- Some of Moggi’s example monads:
  - Global state: $TX = S \Rightarrow (S \times X)$
  - Exceptions: $TX = X + E$
  - Input/output: $TX = \mu Y. (V \Rightarrow Y) + (V \times Y) + X$
  - Non-determinism: $TX = \mathcal{P}(X)$
  - Continuations: $TX = (X \Rightarrow R) \Rightarrow R$

- But, once we have one such monad, what would be the right effectful program constructs to create elements of $TX$.

- And, once we have some effectful program constructs, what would be the right monad to model these effects?

- This is where algebraic effects will help us!
Algebraic effects

- A research programme started by Gordon Plotkin and John Power
- Key idea: use algebra to describe and represent effects!
- Use algebraic operations \( \text{op} : \vec{\beta}; \vec{\alpha}_1, \ldots, \vec{\alpha}_n \) to present effects
- Use (conditional) equations to specify effectful behavior

- A model of effect theory \( T \) in a suitable category \( \mathbb{C} \):
  - a carrier object \( X \) of \( \mathbb{C} \)
  - a morphism \( \llbracket \text{op} \rrbracket_X : \llbracket \beta \rrbracket \times (\llbracket \alpha \rrbracket \Rightarrow X) \rightarrow X \)
  - for every operation \( \text{op} : \beta; \alpha \) (note: special case)
  - such that the equations are satisfied

- Such models form a category \( \text{Mod}(T, \mathbb{C}) \)

- The monad \( T = UF \) is induced by \( F : \mathbb{C} \rightarrow \text{Mod}(T, \mathbb{C}) \) and \( U : \text{Mod}(T, \mathbb{C}) \rightarrow \mathbb{C} \) with \( F \dashv U \)
A research programme started by Gordon Plotkin and John Power

**Key idea:** use algebra to describe and represent effects!

- Use **algebraic operations** \( \text{op} : \vec{\beta}; \vec{\alpha}_1, \ldots, \vec{\alpha}_n \) to present effects
- Use **(conditional) equations** to specify effectful behavior

A **model** of effect theory \( \mathbb{T} \) in a suitable category \( \mathbb{C} \):

- a carrier object \( X \) of \( \mathbb{C} \)
- a morphism \( \llbracket \text{op} \rrbracket_X : \llbracket \beta \rrbracket \times (\llbracket \alpha \rrbracket \Rightarrow X) \rightarrow X \)
  for every operation \( \text{op} : \beta; \alpha \) (note: special case)
- such that the equations are satisfied

Such models form a category \( \text{Mod}(\mathbb{T}, \mathbb{C}) \)

The monad \( \mathbb{T} = UF \) is induced by \( F : \mathbb{C} \rightarrow \text{Mod}(\mathbb{T}, \mathbb{C}) \) and \( U : \text{Mod}(\mathbb{T}, \mathbb{C}) \rightarrow \mathbb{C} \) with \( F \dashv U \)
Algebraic effects

- A research programme started by Gordon Plotkin and John Power
- Key idea: use algebra to describe and represent effects!
- Use algebraic operations $\text{op} : \vec{\beta}; \vec{\alpha}_1, \ldots, \vec{\alpha}_n$ to present effects
- Use (conditional) equations to specify effectful behavior
- A model of effect theory $\mathbb{T}$ in a suitable category $\mathbb{C}$:
  - a carrier object $X$ of $\mathbb{C}$
  - a morphism $\lbrack \text{op} \rbrack_X : \lbrack \beta \rbrack \times (\lbrack \alpha \rbrack \Rightarrow X) \rightarrow X$
    for every operation $\text{op} : \beta; \alpha$ (note: special case)
  - such that the equations are satisfied
- Such models form a category $\text{Mod}(\mathbb{T}, \mathbb{C})$
- The monad $T = UF$ is induced by $F : \mathbb{C} \rightarrow \text{Mod}(\mathbb{T}, \mathbb{C})$ and $U : \text{Mod}(\mathbb{T}, \mathbb{C}) \rightarrow \mathbb{C}$ with $F \dashv U$
Algebraic effects examples: State

- Base types: val, loc
- Operations
  - lookup : loc; val
  - update : loc, val; 1
- Equations
  - \( M \equiv \text{lookup}_l((x : \text{val}).\text{update}_{l,x}(M)) \)
  - \( \text{update}_{l,v}(\text{lookup}_l((x : \text{val}).M)) \equiv \text{update}_{l,v}(M[v/x]) \)
  - \( \text{update}_{l,v}(\text{update}_{l,v'}(M)) \equiv \text{update}_{l,v'}(M) \)
  - \( \text{lookup}_l((x : \text{val}).(\text{lookup}_{l'}((y : \text{val}).M))) \equiv \text{lookup}_{l'}((y : \text{val}).(\text{lookup}_l((x : \text{val}).M))) \) (\( l \neq l' \))
  - \( \text{update}_{l,v}(\text{update}_{l',v'}(M)) \equiv \text{update}_{l',v'}(\text{update}_{l,v}(M)) \) (\( l \neq l' \))
  - \( \text{update}_{l,v}(\text{lookup}_{l'}((x : \text{val}).M)) \equiv \text{lookup}_{l'}((x : \text{val}).(\text{update}_{l,v}(M))) \) (\( l \neq l' \))
Algebraic effects examples: Input/output

- Base types: val, chan
- Operations
  - receive : chan; val
  - send : chan, val; 1
- Equations
  - none!
Algebraic effects examples: Non-determinism

- Base types: none
- Operations
  - $\oplus : 1, 1; 1$
- Equations
  - $M \oplus M \equiv M$
  - $M \oplus N \equiv N \oplus M$
  - $(M \oplus N) \oplus P \equiv M \oplus (N \oplus P)$
Logic of algebraic effects (and CBPV)

- Algebraic effects give us straightforward means for equational reasoning about effects
- Plotkin and Pretnar developed a suitable multi-sorted predicate logic

\[ A ::= \alpha | 1 | A_1 \times A_2 | 0 | A_1 + A_2 | U C \]

\[ C ::= FA | C_1 \times C_2 | A \to C \]

\[ V ::= x | f(\vec{V}) | \star | \langle V_1, V_2 \rangle | \text{proj}_i V | \text{thunk} M | ... \]

\[ M ::= \zeta | \text{return} V | M_1 \text{ to } x. M_2 | \text{op}_V(x. M) | ... \]

\[ \phi ::= V_1 \equiv V_2 | M_1 \equiv M_2 | R(\vec{V}) | \pi(\vec{V}, \vec{M}) | \perp \]

| \phi_1 \lor \phi_2 | \exists x : A. \phi | \exists \zeta : C. \phi \]

\[ \pi ::= X | (\vec{x} : \vec{A}, \vec{\zeta} : \vec{C}).\phi | \mu X : (\vec{A}, \vec{C}).\pi | \nu X : (\vec{A}, \vec{C}).\pi \]
Can define various useful (“temporal”) modalities in this logic

**Pureness modalities:**
- \([\downarrow](\pi) \overset{\text{def}}{=} (\zeta : FA). \forall x : A. \zeta \equiv \text{return } x \implies \pi(x)\]
- \([\langle \downarrow \rangle](\pi) \overset{\text{def}}{=} (\zeta : FA). \exists x : A. \zeta \equiv \text{return } x \land \pi(x)\]

**Operation modalities:**
- \([\text{op}](\pi) \overset{\text{def}}{=} (\zeta : C). \ \forall y : \beta, \zeta' : \alpha \to C. \zeta \equiv \text{op}_y((x : \alpha).\zeta'(x)) \implies \pi(y, \zeta')\]
- \([\langle \text{op} \rangle](\pi) \overset{\text{def}}{=} (\zeta : C). \ \exists y : \beta, \zeta' : \alpha \to C. \zeta \equiv \text{op}_y((x : \alpha).\zeta'(x)) \land \pi(y, \zeta')\]

Can extend \([\text{op}]\) and \([\langle \text{op} \rangle]\) to \([-]\) and \([\langle - \rangle]\) by taking conjunctions and disjunctions over all operations \text{op}

Can use \(\mu\) and \(\nu\) to extend the local modalities \([-]\) and \([\langle - \rangle]\) to global modalities
Value and computation refinement types
The general picture

Value and computation refinement types

\[
\uparrow
\]

CBPV

+ Algebraic effects

+ The logic of algebraic effects
Refinement types in general

- We use standard notation for refinement types: \( \{ x : \sigma \mid \varphi \} \)
  - \( \sigma \) is the type we are refining
  - \( \varphi \) is the refinement proposition (i.e., logical specification)
  - \( x \) might appear free in \( \varphi \)

- As we use the CBPV paradigm, we can define two notions of refinement types:
  - **Value refinement types:** \( \vdash \{ x : \sigma \mid \varphi \} : \text{Ref}(A) \)  
    (these describe properties on values)
  - **Computation refinement types:** \( \vdash \{ \zeta : \tau \mid \varphi \} : \text{Ref}(C) \)  
    (these describe properties of effectful behavior)
Value and computation refinement types

<table>
<thead>
<tr>
<th>⊢ α : Ref(α)</th>
<th>⊢ 1 : Ref(1)</th>
<th>⊢ 0 : Ref(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢ τ : Ref(C)</td>
<td>⊢ σ : Ref(A)</td>
<td></td>
</tr>
<tr>
<td>⊢ Uτ : Ref(UC)</td>
<td>⊢ Fσ : Ref(FA)</td>
<td></td>
</tr>
<tr>
<td>⊢ σ_i : Ref(A_i) (i ∈ {1, 2})</td>
<td>⊢ τ_i : Ref(C_i) (i ∈ {1, 2})</td>
<td></td>
</tr>
<tr>
<td>⊢ σ_1 + σ_2 : Ref(A_1 + A_2)</td>
<td>⊢ τ_1 × τ_2 : Ref(C_1 × C_2)</td>
<td></td>
</tr>
<tr>
<td>⊢ σ_1 : Ref(A_1) ⊢ σ_2 : Ref(A_2)</td>
<td>⊢ σ : Ref(A) ⊢ τ : Ref(C)</td>
<td></td>
</tr>
<tr>
<td>⊢ σ_1 × σ_2 : Ref(A_1 × A_2)</td>
<td>⊢ σ → τ : Ref(A → C)</td>
<td></td>
</tr>
<tr>
<td>⊢ σ : Ref(A) x : A ⊢ ϕ : prop</td>
<td>⊢ τ : Ref(C) ζ : C ⊢ ϕ : prop</td>
<td></td>
</tr>
<tr>
<td>⊢ {x : σ</td>
<td>ϕ} : Ref(A)</td>
<td>⊢ {ζ : τ</td>
</tr>
</tbody>
</table>
Translating to underlying CBPV and the logic

- We can easily define a "forgetful" operation \([\_\_]\)

  - On value refinement types: \([\vdash \sigma : \text{Ref}(A)] \overset{\text{def}}{=} A\)
  - On computation refinement types: \([\vdash \tau : \text{Ref}(C)] \overset{\text{def}}{=} C\)
  - On terms, for example: \([\text{proj}_i V] \overset{\text{def}}{=} \text{proj}_i [V]\)

- There is also a translation \((\_\_)^\bullet\) of refinement types to the logic of algebraic effects. For example:

  - \(x : [\{x' : \sigma | \varphi\}] \vdash (\{x' : \sigma | \varphi\})^\bullet \overset{\text{def}}{=} \varphi[x/x'] \land \sigma^\bullet\)
  - \(\zeta : [F\sigma] \vdash (F\sigma)^\bullet \overset{\text{def}}{=} \left(\mu X.((\zeta : [F\sigma])).(\exists x : [\sigma].\zeta \equiv \text{return } x \land \sigma^\bullet(x)) \lor (\left<\_\right>(X)(\zeta)))\right)(\zeta)\)
Introduction and elimination of refinements

\[
\begin{align*}
\Gamma \vdash V : \sigma & \quad [\Gamma] \mid \Gamma^\bullet \vdash \varphi[[V]/x] \\
\Gamma \vdash V : \{x : \sigma \mid \varphi\} \\
\Gamma \vdash V : \{x : \sigma \mid \varphi\} & \quad [\Gamma] \mid \Gamma^\bullet \vdash \varphi[[V]/x]
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash M : \tau & \quad [\Gamma] \mid \Gamma^\bullet \vdash \varphi[[M]/\zeta] \\
\Gamma \vdash M : \{\zeta : \tau \mid \varphi\} \\
\Gamma \vdash M : \{\zeta : \tau \mid \varphi\} & \quad [\Gamma] \mid \Gamma^\bullet \vdash \varphi[[M]/\zeta]
\end{align*}
\]
Some typing rules for value terms

- These typing rules resemble the standard CBPV rules

\[
\begin{align*}
\Gamma, x : \sigma, \Gamma' & \vdash \text{wf} \\
\Gamma, x : \sigma, \Gamma' & \vdash x : \sigma \\
\Gamma & \vdash \text{wf} \\
\Gamma & \vdash \star : 1
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash V_1 : \beta_1 \quad \cdots \quad \Gamma & \vdash V_n : \beta_n \\
\Gamma & \vdash f(V_1, \ldots, V_n) : \beta \\
\Gamma & \vdash V : \sigma_1 \quad \Gamma & \vdash W : \sigma_2 \\
\Gamma & \vdash \langle V, W \rangle : \sigma_1 \times \sigma_2 \\
\Gamma & \vdash V : \sigma_1 \times \sigma_2 \\
\Gamma & \vdash \text{proj}_i V : \sigma_i \\
\Gamma & \vdash M : \tau \\
\Gamma & \vdash \text{thunk} M : U\tau
\end{align*}
\]
Some typing rules for computation terms

These typing rules resemble the standard CBPV rules:

\[
\begin{align*}
\Gamma \vdash V : \sigma & \quad \Rightarrow \quad \Gamma \vdash \text{return } V : F\sigma \\
\Gamma, x : \sigma \vdash M : \tau & \quad \Rightarrow \quad \Gamma \vdash \lambda x : \sigma. M : \sigma \rightarrow \tau \\
\Gamma \vdash M_1 : \tau_1 \quad \Gamma \vdash M_2 : \tau_2 & \quad \Rightarrow \quad \Gamma \vdash \langle M_1, M_2 \rangle : \tau_1 \times \tau_2 \\
\Gamma \vdash M : \tau_1 \times \tau_2 & \quad \Rightarrow \quad \Gamma \vdash \text{proj}_i M : \tau_i
\end{align*}
\]
Some typing rules for effect operations

- These typing rules differ from the standard CBPV rules

\[
\begin{align*}
\Gamma \vdash V : \beta & \quad \Gamma, x : \alpha \vdash M : F\sigma \\
\Rightarrow \quad \Gamma \vdash \op_V((x : \alpha).M) : F\sigma \\
\Gamma, y : \sigma \vdash \op_V((x : \alpha).(M y)) : \tau \\
\Rightarrow \quad \Gamma \vdash \op_V((x : \alpha).M) : \sigma \rightarrow \tau \\
\Gamma \vdash \op_V((x : \alpha).(\text{fst } M)) : \tau_1 \\
\Rightarrow \quad \Gamma \vdash \op_V((x : \alpha).(\text{snd } M)) : \tau_2 \\
\Gamma \vdash \op_V((x : \alpha).M) : \tau_1 \times \tau_2
\end{align*}
\]

- Compare these rules to the standard CBPV typing rule

\[
\begin{align*}
\Gamma \vdash V : \beta & \quad \Gamma, x : \alpha \vdash M : C \\
\Rightarrow \quad \Gamma \vdash \op_V((x : \alpha).M) : C
\end{align*}
\]
Some typing rules for sequencing

- These typing rules differ from the standard CBPV rules

\[
\frac{\Gamma \vdash M : F\sigma_1 \quad \Gamma, x : \sigma_1 \vdash N : F\sigma_2}{\Gamma \vdash M \text{ to } x. N : F\sigma_2}
\]

\[
\frac{\Gamma \vdash M \text{ to } x. (N y) : \tau \quad \Gamma, y : \sigma \vdash M \text{ to } x. (N y) : \tau}{\Gamma \vdash M \text{ to } x. N : \sigma \to \tau}
\]

\[
\frac{\Gamma \vdash M \text{ to } x. (\text{fst } N) : \tau_1 \quad \Gamma \vdash M \text{ to } x. (\text{snd } N) : \tau_2}{\Gamma \vdash M \text{ to } x. N : \tau_1 \times \tau_2}
\]

- Compare these rules to the standard CBPV typing rule

\[
\frac{\Gamma \vdash M : FA \quad \Gamma, x : A \vdash N : C}{\Gamma \vdash M \text{ to } x. N : C}
\]
We can give a straightforward structural definition for refinement relations

\[ \Gamma \vdash \sigma_2 \sqsubseteq \sigma_1 \quad \Gamma \vdash \tau_2 \sqsubseteq \tau_1 \]

And then show underlying type equality

\[ \Gamma \vdash \sigma_2 \sqsubseteq \sigma_1 \implies \llbracket \sigma_1 \rrbracket = \llbracket \sigma_2 \rrbracket \]
\[ \Gamma \vdash \tau_2 \sqsubseteq \tau_1 \implies \llbracket \tau_1 \rrbracket = \llbracket \tau_2 \rrbracket \]

Using refinement relations, we also define weakening principles

\[ \begin{align*}
\Gamma \vdash v : \sigma_2 & \quad \Gamma \vdash \sigma_2 \sqsubseteq \sigma_1 \\
\hline \Gamma \vdash v : \sigma_1
\end{align*} \]
\[ \begin{align*}
\Gamma \vdash c : \tau_2 & \quad \Gamma \vdash \tau_2 \sqsubseteq \tau_1 \\
\hline \Gamma \vdash c : \tau_1
\end{align*} \]
Some elements of our denotational semantics
Categorical semantics of CBPV and logic

- Semantics of CBPV:
  - Based on adjunction $F \dashv U$
  - We use a specific adjunction suitable for alg. effects: $F : \text{Set} \to \text{Mod}(\mathbb{T}, \text{Set}), U : \text{Mod}(\mathbb{T}, \text{Set}) \to \text{Set}$
  - Value types $A$ are interpreted as objects $[A]$ in $\text{Set}$
  - Computation types $C$ are interpreted as $[C]$ in $\text{Mod}(\mathbb{T}, \text{Set})$
  - Terms are interpreted as morphisms in $\text{Set}$
    - $[\Gamma \vdash V : A] : [\Gamma] \to [A]$
    - $[\Gamma \vdash M : C] : [\Gamma] \to U[C]$

- Semantics of the logic of algebraic effects:
  - Propositions as: $[\Gamma \mid \Delta \mid \Theta \vdash \varphi] \subseteq [\Gamma] \times [\Delta] \times [\Theta]$
  - Predicates as:
    $[\Gamma \mid \Delta \mid \Theta \vdash \pi] : [\Gamma] \times [\Delta] \times [\Theta] \rightarrow \mathcal{P}([\tilde{A}] \times U[\tilde{C}])$
  - Sequents as: $\Gamma \mid \Delta \mid \Theta \mid \varphi \vdash \varphi$ as $[[\varphi]] \subseteq [[\varphi]]$
Categorical semantics of our type system

- Our current concrete categorical semantics is based on the $\text{Set}$-based semantics of CBPV and the logic.
- We abstract a little bit and work explicitly with the subobject fibration.

$$\text{Sub}(\text{Set}) \quad \Gamma \mid \Delta \mid \Theta \vdash \varphi$$

- Base category $\text{Set}$ models contexts (and the language).
- Fibres $\text{Sub}(\text{Set})(\Gamma \mid \Delta \mid \Theta)$ model the logic.
- We give our semantics in the total category $\text{Sub}(\text{Set})$. 
Categorical semantics of our type system ctd.

- So we start with \( p : \text{Sub}(\text{Set}) \to \text{Set} \)

- **Value refinement types** \( \vdash \sigma : \text{Ref}(A) \) are interpreted as objects \([\sigma] \hookrightarrow [A]\) in \(\text{Sub}(\text{Set})\)

- **Computation refinement types** \( \vdash \tau : \text{Ref}(C) \) are interpreted as objects \([\tau] \hookrightarrow U[\!\! C \!\!]\) in \(\text{Sub}(\text{Set})\)
  - The interpretation of \( \vdash F\sigma : \text{Ref}(FA) \) relies on the fact that all monads on \(\text{Set}\) preserve subobjects.

- Value and computation terms will be interpreted as morphisms in \(\text{Sub}(\text{Set})\)
  - **Value terms** \( \Gamma \vdash^V V : \sigma \) as morphisms from \([\Gamma] \hookrightarrow [[\Gamma]]\) to \([\sigma] \hookrightarrow [[\sigma]]\)
  - **Computation terms** \( \Gamma \vdash^C M : \tau \) as morphisms from \([\Gamma] \hookrightarrow [[\Gamma]]\) to \([\tau] \hookrightarrow U[[\tau]]\)
Example: interpreting refinement introduction

\[ \Gamma \vdash V : \sigma \quad [\Gamma] \mid \Gamma^* \vdash \varphi[[V]/x] \]

\[ \Gamma \vdash V : \{x : \sigma \mid \varphi\} \]
Some examples of specifications
Pre- and post-condition specifications on state

- Based on the theory of state and follows ideas from:
  - Hoare triples \( \{ P \} C \{ Q \} \)
  - Hoare types \( \{ P \} x : A \{ Q \} \)

- Pre- and post-conditions as computation refinement types:
  - \( \vdash \{ \zeta : FA \ | \ P \triangleright_{x:A} Q \} : \text{Ref}(FA) \)

- The Hoare refinement has the following definition

\[
\zeta : FA \vdash P \triangleright_{x:A} Q \overset{\text{def}}{=} \\
\forall x_{l_1} : \text{int}, \ldots, x_{l_n} : \text{int}, y_{l_1} : \text{int}, \ldots, y_{l_n} : \text{int}, z : A . \\
P \triangleright [x_{l_1}/x_1, \ldots, x_{l_n}/x_n] \land \\
\exists \zeta' : F1 . \text{update}_{l_1,x_{l_1}} ( \ldots (\text{update}_{l_n,x_{l_n}} (\zeta)) ) \equiv \\
\zeta' \text{ to } x . \text{update}_{l_1,y_{l_1}} ( \ldots (\text{update}_{l_n,y_{l_n}} (\text{return } z)) ) \\
\implies Q \triangleright [y_{l_1}/x_1, \ldots, y_{l_n}/x_n, z/x]
\]
**Pre- and post-condition specifications on state**

- Based on the theory of state and follows ideas from:
  - Hoare triples \( \{P\} C \{Q\} \)
  - Hoare types \( \{P\}x : A\{Q\} \)

- Pre- and post-conditions as computation refinement types:
  - \( \vdash \{\zeta : FA \mid P \triangleright_{x:A} Q\} : \text{Ref}(FA) \)

- The Hoare refinement has the following definition

\[
\zeta : FA \vdash P \triangleright_{x:A} Q \quad \text{def} \quad \forall x_{l_1} : \text{int}, \ldots, x_{l_n} : \text{int}, y_{l_1} : \text{int}, \ldots, y_{l_n} : \text{int}, z : A.
\]

\[
P \triangleright [x_{l_1}/x_1, \ldots, x_{l_n}/x_n] \land
\exists \zeta' : F1.\text{update}_{l_1,x_{l_1}}(\ldots(\text{update}_{l_n,x_{l_n}}(\zeta')))) \equiv
\zeta' \text{ to } x.\text{update}_{l_1,y_{l_1}}(\ldots(\text{update}_{l_n,y_{l_n}}(\text{return } z)))
\]

\[
\implies Q \triangleright [y_{l_1}/x_1, \ldots, y_{l_n}/x_n, z/x]
\]
Pre- and post-condition specifications on state

- Based on the theory of state and follows ideas from:
  - Hoare triples \( \{ P \} C \{ Q \} \)
  - Hoare types \( \{ P \} x : A \{ Q \} \)

- Pre- and post-conditions as computation refinement types:
  - \( \vdash \{ \zeta : FA \mid P \triangleright_{x : A} Q \} : \text{Ref}(FA) \)

- The Hoare refinement has the following definition

\[
\zeta : FA \vdash P \triangleright_{x : A} Q \overset{\text{def}}{=} \\
\forall x_{l_1} : \text{int}, ..., x_{l_n} : \text{int}, y_{l_1} : \text{int}, ..., y_{l_n} : \text{int}, z : A . \\
P \triangleright [x_{l_1}/x_1, ..., x_{l_n}/x_n] \land \\
\exists \zeta' : F1 . \text{update}_{l_1,x_{l_1}} ( ... (\text{update}_{l_n,x_{l_n}} (\zeta))) \equiv \\
\zeta' \text{ to } x . \text{update}_{l_1,y_{l_1}} ( ... (\text{update}_{l_n,y_{l_n}} (\text{return } z))) \\
\implies Q \triangleright [y_{l_1}/x_1, ..., y_{l_n}/x_n, z/x]
\]
Pre- and post-condition specifications on state

- Based on the theory of state and follows ideas from:
  - Hoare triples \( \{P\} C \{Q\} \)
  - Hoare types \( \{P\}x : A\{Q\} \)

- Pre- and post-conditions as computation refinement types:
  - \( \vdash \{\zeta : FA \mid P \triangleright_{x:A} Q\} : \text{Ref}(FA) \)

- The **Hoare refinement** has the following definition

  \[
  \zeta : FA \vdash P \triangleright_{x:A} Q \overset{\text{def}}{=} \\
  \forall x_{l_1} : \text{int}, \ldots, x_{l_n} : \text{int}, y_{l_1} : \text{int}, \ldots, y_{l_n} : \text{int}, z : A.
  \]

  \[
  P \triangleright_{x_1/x, \ldots, x_n/x} \wedge \\
  \exists \zeta' : F1. \text{update}_{l_1,x_1} (\ldots (\text{update}_{l_n,x_{l_n}}(\zeta))) \equiv \\
  \zeta' \text{ to } x. \text{update}_{l_1,y_1} (\ldots (\text{update}_{l_n,y_{l_n}}(\text{return } z)))
  \]

  \[
  \implies Q \triangleright_{y_1/x_1, \ldots, y_{l_n}/x_n, z/x}
  \]
Pre- and post-condition specifications on state

- Based on the theory of state and follows ideas from:
  - Hoare triples \( \{P\} \ C \ {Q}\)
  - Hoare types \( \{P\} x : A\{Q\}\)

- Pre- and post-conditions as computation refinement types:
  - \( \vdash \{\zeta : FA \mid P \triangleright x : A \ Q\} : \text{Ref}(FA)\)

- The Hoare refinement has the following definition

\[
\begin{align*}
\zeta : FA \vdash P \triangleright x : A \ Q \ \text{def} & \equiv \\
\forall x_{l_1} : \text{int}, ..., x_{l_n} : \text{int}, y_{l_1} : \text{int}, ..., y_{l_n} : \text{int}, z : A . \\
P \triangleright [x_{l_1}/x_1, ..., x_{l_n}/x_n] \land \\
\exists \zeta' : F1 . \text{update}_{l_1,x_1} ( \ ... \ (\text{update}_{l_n,x_n} (\zeta))) \equiv \\
\zeta' \ \text{to} \ x . \text{update}_{l_1,y_1} ( \ ... \ (\text{update}_{l_n,y_n} (\text{return} \ z))) \\
\implies Q \triangleright [y_{l_1}/x_1, ..., y_{l_n}/x_n, z/x]
\end{align*}
\]
Session refinements on network communication

Based on the theory of I/O and follows ideas from:

- **Session types:** $S ::= \text{end} \mid !A.S \mid ?A.S$

We consider simple communication of bit-valued data over some fixed set of global channels $i \in \{a, b, c\}$

We can define a grammar for session refinements

- $S_i ::= \text{end}_i \mid \text{!bit}.S_i \mid \text{?bit}.S_i$

  - $\text{end}_i \overset{\text{def}}{=} \zeta.\neg(\diamond(\zeta;\tau).\exists\zeta'.(\zeta \equiv \text{send}_i,0(\zeta') \lor \zeta \equiv \text{send}_i,1(\zeta')) \lor \exists\zeta',\zeta''.\zeta \equiv \text{receive}_i(\zeta',\zeta''))(\zeta)$

  - $\text{!bit}.S_i \overset{\text{def}}{=} \zeta.\Box(\zeta;\tau).\forall\zeta'.(\zeta \equiv \text{send}_i,0(\zeta') \implies S_i(\zeta') \land \zeta \equiv \text{send}_i,1(\zeta') \implies S_i(\zeta')) \land \forall\zeta',\zeta''.\zeta \equiv \text{receive}_i(\zeta',\zeta'') \implies \bot(\zeta)$

  - $\text{?bit}.S_i \overset{\text{def}}{=} \zeta.\Box(\forall\zeta',\zeta''.\zeta \equiv \text{receive}_i(\zeta',\zeta'') \implies (S_i(\zeta') \land S_i(\zeta'')) \land (\forall\zeta'.\zeta \equiv \text{send}_i,0(\zeta') \implies \bot \land \zeta \equiv \text{send}_i,1(\zeta') \implies \bot)(\zeta)$

Unfortunately no true concurrency in algebraic effects yet

- Plotkin’s CONCUR’12 work on CCS-style parallel interleaving
Session refinements on network communication

- Based on the theory of I/O and follows ideas from:
  - Session types: \( S ::= \text{end} | !A.S | ?A.S \)

- We consider simple communication of bit-valued data over some fixed set of global channels \( i \in \{a, b, c\} \)

- We can define a grammar for session refinements
  - \( S_i ::= \text{end}_i | !\text{bit}.S_i | ?\text{bit}.S_i \)
  - \( \text{end}_i \overset{\text{def}}{=} \zeta.\neg(\Diamond(\zeta: \tau).(\exists \zeta'.(\zeta \equiv \text{send}_i,0(\zeta') \lor \zeta \equiv \text{send}_i,1(\zeta'))) \lor (\exists \zeta', \zeta''.\zeta \equiv \text{receive}_i(\zeta', \zeta'')))(\zeta) \)
  - \( !\text{bit}.S_i \overset{\text{def}}{=} \zeta.\Box(\zeta: \tau).(\forall \zeta'.(\zeta \equiv \text{send}_i,0(\zeta') \Rightarrow S_i(\zeta') \land \zeta \equiv \text{send}_i,1(\zeta') \Rightarrow S_i(\zeta')) \land (\forall \zeta', \zeta''.\zeta \equiv \text{receive}_i(\zeta', \zeta'') \Rightarrow \bot))(\zeta) \)
  - \( ?\text{bit}.S_i \overset{\text{def}}{=} \zeta.\Box(\forall \zeta', \zeta''.\zeta \equiv \text{receive}_i(\zeta', \zeta'') \Rightarrow (S_i(\zeta') \land S_i(\zeta'')) \land (\forall \zeta'.\zeta \equiv \text{send}_i,0(\zeta') \Rightarrow \bot \land \zeta \equiv \text{send}_i,1(\zeta') \Rightarrow \bot))(\zeta) \)

- Unfortunately no true concurrency in algebraic effects yet

- Plotkin’s CONCUR’12 work on CCS-style parallel interleaving
Session refinements on network communication

Based on the theory of I/O and follows ideas from:

- Session types: $S ::= \text{end} | !A.S | ?A.S$

We consider simple communication of bit-valued data over some fixed set of global channels $i \in \{a, b, c\}$

We can define a grammar for session refinements

- $S_i ::= \text{end}_i | !\text{bit}.S_i | ?\text{bit}.S_i$
- $\text{end}_i \overset{\text{def}}{=} \zeta.\neg(\Diamond (\zeta:\tau). (\exists \zeta'. (\zeta \equiv \text{send}_{i, 0}(\zeta') \lor \zeta \equiv \text{send}_{i, 1}(\zeta'))) \lor (\exists \zeta', \zeta''. \zeta \equiv \text{receive}_i(\zeta', \zeta''))) (\zeta)$
- $!\text{bit}.S_i \overset{\text{def}}{=} \zeta.\Box (\zeta:\tau). (\forall \zeta'. (\zeta \equiv \text{send}_{i, 0}(\zeta')) \implies S_i(\zeta') \land \zeta \equiv \text{send}_{i, 1}(\zeta') \implies S_i(\zeta'))^\land$
  $$(\forall \zeta', \zeta''. \zeta \equiv \text{receive}_i(\zeta', \zeta'') \implies \bot)(\zeta)$$
- $?\text{bit}.S_i \overset{\text{def}}{=} \zeta.\Box (\forall \zeta', \zeta''. \zeta \equiv \text{receive}_i(\zeta', \zeta'') \implies (S_i(\zeta') \land S_i(\zeta''))) ^\land$
  $$((\forall \zeta'. \zeta \equiv \text{send}_{i, 0}(\zeta') \implies \bot \land \zeta \equiv \text{send}_{i, 1}(\zeta') \implies \bot)(\zeta)$$

Unfortunately no true concurrency in algebraic effects yet

- Plotkin’s CONCUR’12 work on CCS-style parallel interleaving
Based on the theory of I/O and follows ideas from:

- Session types: \( S ::= end \mid !A.S \mid ?A.S \)

We consider simple communication of bit-valued data over some fixed set of global channels \( i \in \{a, b, c\} \)

We can define a grammar for session refinements

- \( S_i ::= end_i \mid !\text{bit}.S_i \mid ?\text{bit}.S_i \)
- \( \text{end}_i \overset{\text{def}}{=} \zeta.\neg(\Diamond(\zeta;\exists).(\exists \zeta'.(\zeta'\equiv \text{send}_i,0(\zeta') \lor \zeta'\equiv \text{send}_i,1(\zeta'))) \lor (\exists \zeta', \zeta''.\zeta'\equiv \text{receive}_i(\zeta', \zeta''))(\zeta) \)
- \( !\text{bit}.S_i \overset{\text{def}}{=} \zeta.\Box(\zeta;\forall).(\exists \zeta'.(\zeta'\equiv \text{send}_i,0(\zeta')) \Rightarrow S_i(\zeta') \land \zeta'\equiv \text{send}_i,1(\zeta') \Rightarrow S_i(\zeta')) \land ((\forall \zeta', \zeta''.\zeta'\equiv \text{receive}_i(\zeta', \zeta'')) \Rightarrow \bot)(\zeta) \)
- \( ?\text{bit}.S_i \overset{\text{def}}{=} \zeta.\Box(\forall \zeta', \zeta''.\zeta'\equiv \text{receive}_i(\zeta', \zeta'') \Rightarrow (S_i(\zeta') \land S_i(\zeta''))) \land ((\forall \zeta'.\zeta'\equiv \text{send}_i,0(\zeta') \Rightarrow \bot \land \zeta\equiv \text{send}_i,1(\zeta') \Rightarrow \bot)(\zeta) \)

Unfortunately no true concurrency in algebraic effects yet

Plotkin’s CONCUR’12 work on CCS-style parallel interleaving
Session refinements on network communication

- Based on the theory of I/O and follows ideas from:
  - Session types: $S ::= \text{end} | !A.S | ?A.S$

- We consider simple communication of bit-valued data over some fixed set of global channels $i \in \{a, b, c\}$

- We can define a grammar for session refinements
  - $S_i ::= \text{end}_i | !\text{bit}.S_i | ?\text{bit}.S_i$
  - $\text{end}_i \overset{\text{def}}{=} \zeta.\neg(\Diamond((\zeta;\tau).\exists\zeta'.(\zeta\equiv\text{send}_i,0(\zeta') \lor \zeta\equiv\text{send}_i,1(\zeta')) \lor (\exists\zeta',\zeta''.\zeta\equiv\text{receive}_i(\zeta',\zeta'')))(\zeta)$
  - $!\text{bit}.S_i \overset{\text{def}}{=} \zeta.\Box((\zeta;\tau).\forall\zeta'.(\zeta\equiv\text{send}_i,0(\zeta') \Rightarrow S_i(\zeta') \land \zeta\equiv\text{send}_i,1(\zeta') \Rightarrow S_i(\zeta')) \land$
    $(\forall\zeta',\zeta''.\zeta \equiv \text{receive}_i(\zeta',\zeta'') \Rightarrow \bot)(\zeta)$
  - $?\text{bit}.S_i \overset{\text{def}}{=} \zeta.\Box((\forall\zeta',\zeta''.\zeta\equiv\text{receive}_i(\zeta',\zeta'') \Rightarrow (S_i(\zeta') \land S_i(\zeta''))) \land$
    $(\forall\zeta'.\zeta \equiv \text{send}_i,0(\zeta') \Rightarrow \bot \land \zeta \equiv \text{send}_i,1(\zeta') \Rightarrow \bot)(\zeta)$

- Unfortunately no true concurrency in algebraic effects yet
  - Plotkin’s CONCUR’12 work on CCS-style parallel interleaving
Conclusions

- Showed our preliminary work towards combining the following:
  - algebraic effects
  - refinement types
  - program specifications

- Still a lot of work to be done
  - Generalizing the semantics of this simply-typed refinement type system to a context-dependent calculus
  - Accommodate local effects and instances of effects
  - Extend effect theories with built in behavioral refinements
  - Extend effect theories with cost measures and label algebras
  - Combinations of specifications induced by combinations of effect theories
  - Find more practical applications