Refinement Types | Algebraic Effects

Danel Ahman
LFCS, University of Edinburgh

HOPE Workshop, 28 September 2013
Overview

- Refinement types & effects

- What do we feel is missing from refinement type systems?
  - A uniform treatment of various computational effects
  - General logical specifications for arbitrary effects

- Our way of bridging this gap
  - Algebraic effects and their logics
  - General effectful ref. types through algebraic effectful reasoning
  - Hopefully leads us to a general theory of effectful refinement types

- Some examples
  - State and pre-/post-conditions
  - Communication and sessions
  - Combination of the two
Overview

- Refinement types & effects

- What do we feel is missing from refinement type systems?
  - A uniform treatment of various computational effects
  - General logical specifications for arbitrary effects

- Our way of bridging this gap
  - Algebraic effects and their logics
  - General effectful ref. types through algebraic effectful reasoning
  - Hopefully leads us to a general theory of effectful refinement types

- Some examples
  - State and pre-/post-conditions
  - Communication and sessions
  - Combination of the two
Overview

- Refinement types & effects

- What do we feel is missing from refinement type systems?
  - A uniform treatment of various computational effects
  - General logical specifications for arbitrary effects

- Our way of bridging this gap
  - Algebraic effects and their logics
  - General effectful ref. types through algebraic effectful reasoning
  - Hopefully leads us to a general theory of effectful refinement types

- Some examples
  - State and pre-/post-conditions
  - Communication and sessions
  - Combination of the two
Overview

- Refinement types & effects

- What do we feel is missing from refinement type systems?
  - A uniform treatment of various computational effects
  - General logical specifications for arbitrary effects

- Our way of bridging this gap
  - Algebraic effects and their logics
  - General effectful ref. types through algebraic effectful reasoning
  - Hopefully leads us to a general theory of effectful refinement types

- Some examples
  - State and pre-/post-conditions
  - Communication and sessions
  - Combination of the two
Effects in refinement type systems

- Most current refinement type systems target specific effects:
  - F7 extended with a refined state monad \[ ((s_0)\varphi_0)x : \sigma((s_1)\varphi_1) \] \cite{Borgströmetal09}
  
  - Monadic F* with a Dijkstra monad \[ M\sigma wp \] \cite{Swamyetal13}
  
  - Session types with linear refinement types \{ x : T | \varphi \} to session types (with \varphi in MLL) \cite{Baltazaretal12}

- Some systems are more abstract in effects they consider:
  - Effective theory of type refinements \cite{Mandelbaumetal03}
    
    - term refinements \( \varphi : \text{bool, its}(t), \varphi_1 \rightarrow \varphi_2, (\varphi_1, \psi_1) \rightarrow (\varphi_2, \psi_2) \)
    
    - world refinements \( \psi : \text{formulas in linear logic} \)
    
    - parametrized by a set of operations (together with a signature of operation refinements and a transition function for operations)
Most current refinement type systems target specific effects:

- F7 extended with a refined state monad  
  by adding a new computation type \( \{(s_0)\varphi_0\}x : \sigma\{(s_1)\varphi_1\} \)

- Monadic F* with a Dijkstra monad  
  by adding a comp. type \( M\sigma wp \)

- Session types with linear refinement types  
  by adding ref. ty. \( \{x : T \mid \varphi\} \) to session types (with \( \varphi \) in MLL)

Some systems are more abstract in effects they consider:

- Effective theory of type refinements  
  term refinements \( \varphi : \text{bool}, \text{its}(t), \varphi_1 \rightarrow \varphi_2, (\varphi_1, \psi_1) \rightarrow (\varphi_2, \psi_2) \)  
  world refinements \( \psi : \text{formulas in linear logic} \)  
  parametrized by a set of operations (together with a signature of operation refinements and a transition function for operations)
Effects in refinement type systems

- Most current refinement type systems target specific effects:
  - F7 extended with a refined state monad \cite{Borgstrometal09} by adding a new computation type \((\varphi_0)x : \sigma\{\varphi_1\}\) 
  - Monadic F* with a Dijkstra monad \cite{Swamyetal13} by adding a comp. type \(M\sigma wp\) 
  - Session types with linear refinement types \cite{Baltazaretal12} by adding ref. ty. \(\{x : T \mid \varphi\}\) to session types (with \(\varphi\) in MLL)

- Some systems are more abstract in effects they consider:
  - Effective theory of type refinements \cite{Mandelbaumetal03}
    - term refinements \(\varphi : \text{bool}, \text{its}(t), \varphi_1 \rightarrow \varphi_2, (\varphi_1, \psi_1) \rightarrow (\varphi_2, \psi_2)\)
    - world refinements \(\psi : \text{formulas in linear logic}\)
    - parametrized by a set of operations (together with a signature of operation refinements and a transition function for operations)
Effects in refinement type systems

Most current refinement type systems target specific effects:

- F7 extended with a refined state monad by adding a new computation type \( \{(s_0)\phi_0\}x : \sigma\{(s_1)\phi_1\} \)

- Monadic F* with a Dijkstra monad by adding a comp. type \( M\sigma wp \)

- Session types with linear refinement types by adding ref. ty. \( \{x : T \mid \phi\} \) to session types (with \( \phi \) in MLL)

Some systems are more abstract in effects they consider:

- Effective theory of type refinements by adding a new computation type \( \{(s_0)\phi_0\}x : \sigma\{(s_1)\phi_1\} \)

- term refinements \( \phi : \text{bool}, \text{its}(t), \phi_1 \rightarrow \phi_2, (\phi_1, \psi_1) \rightarrow (\phi_2, \psi_2) \)

- world refinements \( \psi : \text{formulas in linear logic} \)

- parametrized by a set of operations (together with a signature of operation refinements and a transition function for operations)
Consider a (fragment of a) simple communication language:

\[
\Gamma \vdash \text{return } t : FA \\
\Gamma \vdash \text{send}_t(u) : FA
\]

Session refinements (inspired by session types)

\[
S(A) ::= \text{end}(A) \mid ?(x : \text{nat}).S(A) \mid !(x : \text{nat} \mid \varphi).S(A)
\]

Example programs with their refinements:

\[
\Gamma \vdash \text{receive}(x.\text{receive}(y.t)) : ?(x : \text{nat}).?(y : \text{nat}).S(1)
\]

\[
\Gamma \vdash \text{send}_t(\text{send}_{t+1}(u)) : !(x : \text{nat} \mid \top).!(y : \text{nat} \mid y > x).S(1)
\]
Consider a (fragment of a) simple communication language:

\[
\begin{align*}
\Gamma \vdash t : A & \quad \Gamma, x : A \vdash u : S(B) \\
\Gamma \vdash \text{return } t : \text{end}(A) & \quad \Gamma \vdash t \text{ to } x. u : S(A); S(B) \\
\Gamma, x : \text{nat} \vdash t : S(A) & \quad \Gamma \vdash t : \text{nat} \\
\Gamma \vdash \text{receive}(x.t) : ?(x : \text{nat}).S(A) & \quad \Gamma \vdash \varphi[t/x] \\
& \quad \Gamma \vdash u : S(A) \\
& \quad \Gamma \vdash \text{send}_t(u) : !(x : \text{nat} | \varphi).S(A)
\end{align*}
\]

Session refinements (similar syntax to session types):

- \( S(A) ::= \text{end}(A) \mid ?(x : \text{nat}).S(A) \mid !(x : \text{nat} | \varphi).S(A) \)

Example programs with their refinements:

- \( \Gamma \vdash \text{receive}(x.\text{receive}(y.t)) : ?(x : \text{nat}).?(y : \text{nat}).S(1) \)
- \( \Gamma \vdash \text{send}_t(\text{send}_{t+1}(u)) : !(x : \text{nat} | \top).!(y : \text{nat} | y > x).S(1) \)
Different effects, languages and specifications

- Consider a (fragment of a) simple communication language:

\[
\begin{align*}
\Gamma & \vdash t : A & \quad & \Gamma & \vdash t : S(A) & \quad & \Gamma, x : A \vdash u : S(B) \\
\Gamma & \vdash \text{return } t : end(A) & \quad & \Gamma & \vdash t \text{ to } x. u : S(A); S(B) \\
\Gamma, x : \text{nat} & \vdash t : S(A) & \quad & \Gamma & \vdash t : \text{nat} & \quad & \Gamma & \vdash \varphi[t/x] & \quad & \Gamma & \vdash u : S(A) \\
\Gamma & \vdash \text{receive}(x.t) : ?(x : \text{nat}).S(A) & \quad & \Gamma & \vdash \text{send}_t(u) : !(x : \text{nat} | \varphi).S(A)
\end{align*}
\]

- Session refinements (similar syntax to session types):

  - \[ S(A) ::= end(A) \mid ?(x : \text{nat}).S(A) \mid !(x : \text{nat} | \varphi).S(A) \]

- Example programs with their refinements:

  - \[ \Gamma \vdash \text{receive}(x.\text{receive}(y.t)) : ?(x : \text{nat}).?(y : \text{nat}).S(1) \]
  - \[ \Gamma \vdash \text{send}_t(\text{send}_{t+1}(u)) : !(x : \text{nat} \mid \top).!(y : \text{nat} \mid y > x).S(1) \]
Consider a (fragment of a) simple state language:

\[
\begin{align*}
\Gamma \vdash t : A & \quad \Gamma, x : A \vdash u : FB \\
\Gamma \vdash \text{return } t : FA & \quad \Gamma \vdash t \to x. u : FB \\
\Gamma, x : \text{nat} \vdash t : FA & \quad \Gamma \vdash t : \text{nat} \\
\Gamma \vdash \text{lookup}(x.t) : FA & \quad \Gamma \vdash \text{update}_{t}(u) : FA
\end{align*}
\]

Pre- & post-condition specifications:

\[\forall \vec{x}. \{(x_0).\varphi_P\} x. A \{(x_1).\varphi_Q\}\]

Example program with its refinement:

\[\Gamma \vdash \text{lookup}(x.\text{update}_{x+1}(\text{return } \ast)) : \{(x_0).\text{odd}(x_0)\} x : 1 \{(x_1).\text{even}(x_1)\}\]
Different effects, languages and specifications

- Consider a (fragment of a) simple state language:

\[
\Gamma, x : \text{nat} \vdash t : \forall \vec{x}, x_0.\{(x_1).\varphi_Q\} y : A\{(x_2).\varphi_R\}
\]
\[
\Gamma \vdash \forall \vec{x}.\{(x_0).\top\} x : \text{nat}\{(x_1).\varphi\} x_0 \equiv \forall \vec{x}.\{(x_0).\varphi_P\} x : \text{nat}\{(x_1).\varphi_Q\}
\]
\[
\Gamma \vdash \text{lookup}(x.t) : \forall \vec{x}.\{(x_0).\varphi_P\} y : A\{(x_2).\varphi_R\}
\]
\[
\Gamma \vdash t : \text{nat} \quad \Gamma \vdash u : \forall \vec{x}, x_0.\{(x_1).\varphi_Q\} x : A\{(x_2).\varphi_R\}
\]
\[
\Gamma \vdash \forall \vec{x}.\{(x_0).\top\} _{-} : 1\{(x_1).\varphi\} x_0 \equiv \forall \vec{x}.\{(x_0).\varphi_P\} _{-} : 1\{(x_1).\varphi_Q\}
\]
\[
\Gamma \vdash \text{update}_t(u) : \forall \vec{x}.\{(x_0).\varphi_P\} x : A\{(x_2).\varphi_R\}
\]

- Pre- & post-condition specifications:

\[
\forall \vec{x}.\{(x_0).\varphi_P\} x : A\{(x_1).\varphi_Q\}
\]

- Example program with its refinement:

\[
\Gamma \vdash \text{lookup}(x.\text{update}_{x+1}(\text{return } \star)) : \{(x_0).\text{odd}(x_0)\} x : 1\{(x_1).\text{even}(x_1)\}
\]
Different effects, languages and specifications

- Consider a (fragment of a) simple state language:

\[ \Gamma, x : \text{nat} \vdash t : \forall \vec{x}, x_0.\{(x_1).\varphi_Q\}y : A\{(x_2).\varphi_R\} \]
\[ \Gamma \vdash \forall \vec{x}.\{(x_0).\top\}x : \text{nat}\{(x_1).x_1 = x_0 \land x_1 = y\} \sqsubseteq \forall \vec{x}.\{(x_0).\varphi_P\}x : \text{nat}\{(x_1).\varphi_Q\} \]

\[ \Gamma \vdash \text{lookup}(x.t) : \forall \vec{x}.\{(x_0).\varphi_P\}y : A\{(x_2).\varphi_R\} \]

\[ \Gamma \vdash t : \text{nat} \quad \Gamma \vdash u : \forall \vec{x}, x_0.\{(x_1).\varphi_Q\}x : A\{(x_2).\varphi_R\} \]
\[ \Gamma \vdash \forall \vec{x}.\{(x_0).\top\}_- : 1\{(x_1).x_1 = t\} \sqsubseteq \forall \vec{x}.\{(x_0).\varphi_P\}_- : 1\{(x_1).\varphi_Q\} \]

\[ \Gamma \vdash \text{update}_t(u) : \forall \vec{x}.\{(x_0).\varphi_P\}x : A\{(x_2).\varphi_R\} \]

- Pre- & post-condition specifications:

\[ \forall \vec{x}.\{(x_0).\varphi_P\}x : A\{(x_1).\varphi_Q\} \]

- Example program with its refinement:

\[ \Gamma \vdash \text{lookup}(x.\text{update}_{x+1}(\text{return } \star)) : \{(x_0).\text{odd}(x_0)\}x : 1\{(x_1).\text{even}(x_1)\} \]
Also want a combination of these languages and specifications

For example, combining state and communication:

\[ \forall \vec{x}.\{(x_0).\varphi_P\}(S(A) \Rightarrow x : A)\{(x_1).\varphi_Q\} \]

Example program with a composite refinement:

\[ \langle \rangle \vdash \text{receive}(x.\text{lookup}(y.\text{if } y > x \text{ then update}_{y-x}(\text{return } \star) \text{ else return } \star)) : \]

\[ \{(x_0).\top\}(?(x : \text{nat}).\text{end}(1) \Rightarrow y : 1)\{(x_1). (x > x_0) \quad \implies \quad x_1 = x_0 - x\} \]

Other effects and their specs.?

Non-standard combinations of specs.?
Different effects, languages and specifications

- Also want a combination of these languages and specifications
- For example, combining state and communication:

$$\forall \vec{x}.\{(x_0).\varphi_P\}(S(A) \Rightarrow x : A)\{(x_1).\varphi_Q\}$$

- Example program with a composite refinement:

$$\langle \rangle \vdash \text{receive}(x.\text{lookup}(y.\text{if } y > x \text{ then update}_{y-x}(\text{return } \star) \text{ else return } \star)) :$$

$$\{(x_0).\top\}(\exists(x : \text{nat}).\text{end}(1) \Rightarrow y : 1)\{(x_1).(x > x_0) \implies x_1 = x_0 - x\}$$

- Other effects and their specs.
- Non-standard combinations of specs.
Different effects, languages and specifications

- Also want a combination of these languages and specifications
- For example, combining state and communication:

\[ \forall \vec{x}.\{(x_0).\phi_P\}(S(A) \Rightarrow x : A)\{(x_1).\phi_Q\} \]

- Example program with a composite refinement:

\[
\langle \rangle \vdash \text{receive}(x.\text{lookup}(y.\text{if } y > x \text{ then } \text{update}_{y-x}(\text{return } \star) \text{ else } \text{return } \star)): \\
\{(x_0).\top\}(\exists(x : \text{nat}).\text{end}(1) \Rightarrow y : 1)\{(x_1).(x > x_0) \implies x_1 = x_0 - x\}
\]

- Other effects and their specs.
- Non-standard combinations of specs.
Different effects, languages and specifications

- Also want a combination of these languages and specifications

- For example, combining state and communication:

  \[ \forall \vec{x}.\{(x_0).\varphi_P\}(S(A) \Rightarrow x : A)\{(x_1).\varphi_Q\} \]

- Example program with a composite refinement:

  \[ \langle \rangle \vdash \text{receive} (x.\text{lookup}(y.\text{if } y > x \text{ then update}_{y-x}(\text{return } \star) \text{ else return } \star)) : \]

  \[ \{(x_0).\top\}(?(x : \text{nat}).\text{end}(1) \Rightarrow y : 1)\{(x_1).(x > x_0) \implies x_1 = x_0 - x\} \]

- Other effects and their specs.?

- Non-standard combinations of specs.?
Our proposed approach

A computational language with algebraic effects

- ref. types for general effectful specs.
- using algebraic effectful reasoning

State language ⊆ Communication language ⊆ Language X
Refinement types

- The style of ref. types we work with (no effects for time being):
  - \( \lambda \)-calculus with types \( A ::= \alpha \mid 1 \mid A_1 \times A_2 \mid A_1 \to A_2 \)

- Refinement types \( \sigma ::= \alpha \mid 1 \mid \Sigma_{x: \sigma_1} \sigma_2 \mid \Pi_{x: \sigma_1} \sigma_2 \mid \{ x : \sigma \mid \varphi \} \)

- Well-formed refinement types \( \Gamma \vdash \sigma : \text{Ref}(A) \), e.g.:
  \[
  \begin{array}{c}
  \vdash \Gamma \text{ wf} \\
  \hline
  \Gamma \vdash \alpha : \text{Ref}(\alpha)
  \end{array}
  \]
  \[
  \begin{array}{c}
  \Gamma \vdash \sigma_1 : \text{Ref}(A_1) \\
  \Gamma, x : \sigma_1 \vdash \sigma_2 : \text{Ref}(A_2)
  \end{array}
  \]
  \[
  \begin{array}{c}
  \Gamma \vdash \Pi_{x: \sigma_1} \sigma_2 : \text{Ref}(A_1 \to A_2)
  \end{array}
  \]
  \[
  \begin{array}{c}
  \Gamma \vdash \sigma : \text{Ref}(A) \\
  \Gamma, x : A \vdash \varphi : \text{prop}
  \end{array}
  \]
  \[
  \begin{array}{c}
  \Gamma \vdash \{ x : \sigma \mid \varphi \} : \text{Ref}(A)
  \end{array}
  \]

- Well-typed refined terms \( \Gamma \vdash t : \sigma \), e.g.:
  \[
  \begin{array}{c}
  \Gamma \vdash t : \sigma \\
  |\Gamma| | \Gamma^\circ \vdash \varphi[|t|/x]
  \end{array}
  \]
  \[
  \begin{array}{c}
  \Gamma \vdash t : \{ x : \sigma \mid \varphi \}
  \end{array}
  \]

Denney '98
The style of ref. types we work with (no effects for time being):

- \( \lambda \)-calculus with types \( A ::= \alpha \mid 1 \mid A_1 \times A_2 \mid A_1 \to A_2 \)

- Refinement types \( \sigma ::= \alpha \mid 1 \mid \Sigma_{x : \sigma_1} \sigma_2 \mid \Pi_{x : \sigma_1} \sigma_2 \mid \{ x : \sigma \mid \varphi \} \)

Well-formed refinement types \( \Gamma \vdash \sigma : \text{Ref}(A) \), e.g.:

\[
\begin{align*}
\vdash \Gamma \text{ wf} & \quad \Gamma \vdash \sigma_1 : \text{Ref}(A_1) & \quad \Gamma, x : \sigma_1 \vdash \sigma_2 : \text{Ref}(A_2) \\
\Gamma \vdash \alpha : \text{Ref}(\alpha) & \quad \Gamma \vdash \Pi_{x : \sigma_1} \sigma_2 : \text{Ref}(A_1 \to A_2) \\
\Gamma \vdash \sigma : \text{Ref}(A) & \quad \Gamma, x : A \vdash \varphi : \text{prop} \quad \Gamma \vdash \{ x : \sigma \mid \varphi \} : \text{Ref}(A)
\end{align*}
\]

Well-typed refined terms \( \Gamma \vdash t : \sigma \), e.g.:

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

Denney '98
Refinement types

- The style of ref. types we work with (no effects for time being):
  - \(\lambda\)-calculus with types \(A ::= \alpha \mid 1 \mid A_1 \times A_2 \mid A_1 \rightarrow A_2\)
  - Refinement types \(\sigma ::= \alpha \mid 1 \mid \Sigma_{x:\sigma_1}\sigma_2 \mid \Pi_{x:\sigma_1}\sigma_2 \mid \{x : \sigma \mid \varphi\}\)

- Well-formed refinement types: \(\Gamma \vdash \sigma : \text{Ref}(A)\), e.g.:

  \[
  \begin{align*}
  \Gamma \vdash \alpha : \text{Ref}(\alpha) & \quad \Gamma \vdash \sigma_1 : \text{Ref}(A_1) \quad \Gamma, x : \sigma_1 \vdash \sigma_2 : \text{Ref}(A_2) \\
  \Gamma \vdash \Pi_{x:\sigma_1}\sigma_2 : \text{Ref}(A_1 \rightarrow A_2) & \quad \Gamma \vdash \sigma : \text{Ref}(A) \quad \Gamma, x : A \vdash \varphi : \text{prop} \\
  \Gamma \vdash \{x : \sigma \mid \varphi\} : \text{Ref}(A)
  \end{align*}
  \]

- Well-typed refined terms: \(\Gamma \vdash t : \sigma\), e.g.:

  \[
  \begin{align*}
  \Gamma \vdash t : \sigma & \quad |\Gamma| \cup \Gamma^\circ \vdash \varphi[|t|/x] \\
  \Gamma \vdash t : \{x : \sigma \mid \varphi\}
  \end{align*}
  \]
Refinement types

- The style of ref. types we work with (no effects for time being):
  - $\lambda$-calculus with types $A ::= \alpha \mid 1 \mid A_1 \times A_2 \mid A_1 \to A_2$
  - Refinement types $\sigma ::= \alpha \mid 1 \mid \Sigma x:\sigma_1 \sigma_2 \mid \Pi x:\sigma_1 \sigma_2 \mid \{x : \sigma \mid \varphi\}$

- Well-formed refinement types $\Gamma \vdash \sigma : \text{Ref}(A)$, e.g.:

\[
\begin{align*}
\vdash \Gamma \text{ wf} & \quad \Gamma \vdash \sigma_1 : \text{Ref}(A_1) & \quad \Gamma, x : \sigma_1 \vdash \sigma_2 : \text{Ref}(A_2) \\
\hline
\Gamma \vdash \alpha : \text{Ref}(\alpha) & \quad \Gamma \vdash \Pi x : \sigma_1 \sigma_2 : \text{Ref}(A_1 \to A_2)
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \sigma : \text{Ref}(A) & \quad \Gamma, x : A \vdash \varphi : \text{prop} \\
\hline
\Gamma \vdash \{x : \sigma \mid \varphi\} : \text{Ref}(A)
\end{align*}
\]

- Well-typed refined terms $\Gamma \vdash t : \sigma$, e.g.:

\[
\begin{align*}
\vdash \Gamma \vdash t : \sigma & \quad |\Gamma| \mid \Gamma^\circ \vdash \varphi[|t|/x] \\
\hline
\Gamma \vdash t : \{x : \sigma \mid \varphi\}
\end{align*}
\]

Denney '98
Algebraic effects

- Let’s look at effects algebraically (for example: state)
- Types (sets) of values (countable) and locations (fin.): Val, Loc
- Operation symbols:
  - lookup : Loc \( \rightarrow \) Val
  - update : Loc, Val \( \rightarrow \) 1
- Enforce equations on derived terms:
  - \( \text{update}_{l,v}(\text{lookup}_l(x.t)) = \text{update}_{l,v}(t[v/x]) \)
  - \( \text{update}_{l,v}(\text{update}_{l,v'}(t)) = \text{update}_{l,v'}(t) \)
  - \( t = \text{lookup}_l(x.\text{update}_{l,x}(t)) \)
  - \( \text{update}_{l,v}(\text{update}_{l',v'}(t)) = \text{update}_{l',v'}(\text{update}_{l,v}(t)) \quad (l \neq l') \)
- ...  
- Your usual monad through free algebra construction:
  - \( T = UF = (\text{Val}^\text{Loc} \times -)^{\text{Val}^\text{Loc}} \)
Algebraic effects

- Let’s look at effects algebraically (for example: state)
- Types (sets) of values (countable) and locations (fin.): Val, Loc
- Operation symbols:
  - lookup : Loc → Val
  - update : Loc, Val → 1
- Enforce equations on derived terms:
  - \( \text{update}_{l,v}(\text{lookup}_l(x.t)) = \text{update}_{l,v}(t[v/x]) \)
  - \( \text{update}_{l,v}(\text{update}_{l,v'}(t)) = \text{update}_{l,v'}(t) \)
  - \( t = \text{lookup}_l(x.\text{update}_{l,x}(t)) \)
  - \( \text{update}_{l,v}(\text{update}_{l',v'}(t)) = \text{update}_{l',v'}(\text{update}_{l,v}(t)) \quad (l \neq l') \)
  - ...
- Your usual monad through free algebra construction:
  - \( T = UF = (\text{Val}^{\text{Loc}} \times -)^{\text{Val}^{\text{Loc}}} \)
Algebraic effects

- Let’s look at effects algebraically (for example: state)
- Types (sets) of values (countable) and locations (fin.): Val, Loc
- Operation symbols:
  - lookup : Loc → Val
  - update : Loc, Val → 1

- Enforce equations on derived terms:
  - $\text{update}_{l,v}(\text{lookup}_{l}(x.t)) = \text{update}_{l,v}(t[v/x])$
  - $\text{update}_{l,v}(\text{update}_{l,v'}(t)) = \text{update}_{l,v'}(t)$
  - $t = \text{lookup}_{l}(x.\text{update}_{l,x}(t))$
  - $\text{update}_{l,v}(\text{update}_{l',v'}(t)) = \text{update}_{l',v'}(\text{update}_{l,v}(t))$ (l ≠ l')
  - ...

- Your usual monad through free algebra construction:
  - $T = UF = (\text{Val}^{\text{Loc}} \times -)^{\text{Val}^{\text{Loc}}}$
Algebraic effects

- Let’s look at effects algebraically (for example: state)
- Types (sets) of values (countable) and locations (finite): Val, Loc
- Operation symbols:
  - lookup : Loc $\rightarrow$ Val
  - update : Loc, Val $\rightarrow$ 1
- Enforce equations on derived terms:
  - $\text{update}_{l,v}(\text{lookup}_l(x.t)) = \text{update}_{l,v}(t[v/x])$
  - $\text{update}_{l,v}(\text{update}_{l,v'}(t)) = \text{update}_{l,v'}(t)$
  - $t = \text{lookup}_l(x.\text{update}_l,x(t))$
  - $\text{update}_{l,v}(\text{update}_{l',v'}(t)) = \text{update}_{l',v'}(\text{update}_{l,v}(t))$ \hspace{1cm} (l $\neq$ $l'$)
- ...

Your usual monad through free algebra construction:

- $T = UF = (\text{Val}^{\text{Loc}} \times -)^{\text{Val}^{\text{Loc}}}$

Plotkin & Power ’02
Algebraic effects

- Let’s look at effects algebraically (for example: state)
- Types (sets) of values (countable) and locations (fin.): $\text{Val}, \text{Loc}$
- Operation symbols:
  - $\text{lookup} : \text{Loc} \rightarrow \text{Val}$
  - $\text{update} : \text{Loc}, \text{Val} \rightarrow 1$
- Enforce equations on derived terms:
  - $\text{update}_{l,v}(\text{lookup}_{l}(x.t)) = \text{update}_{l,v}(t[v/x])$
  - $\text{update}_{l,v}(\text{update}_{l,v'}(t)) = \text{update}_{l,v'}(t)$
  - $t = \text{lookup}_{l}(x.\text{update}_{l,x}(t))$
  - $\text{update}_{l,v}(\text{update}_{l',v'}(t)) = \text{update}_{l',v'}(\text{update}_{l,v}(t)) \quad (l \neq l')$
  - $\ldots$

- Your usual monad through free algebra construction:
  - $T = UF = (\text{Val}^{\text{Loc}} \times -)^{\text{Val}^{\text{Loc}}}$

Plotkin & Power ’02
The programming language

- We use a variant of the Effect Calculus (closely related to Call-by-Push-Value)
  (Egger et. al. ’09, ’12, Levy ’01, ’04)

- Value and computation types:
  \[ A ::= \alpha | 1 | A_1 \times A_2 | A_1 \to A_2 | FA \]
  \[ A ::= A_1 \times A_2 | A_1 \to A_2 | FA \]

- Terms \( t \):
  \[ t ::= x | \star | \langle t_1, t_2 \rangle | \text{proj}_i t | \lambda x.t | t_1(t_2) | \text{return } t | t_1 \to x.t_2 | \text{op}_{t_1}(x.t_2) \]

- Well-typed terms \( \Gamma \vdash t : A \), e.g.:

\[
\begin{align*}
\Gamma \vdash t : A & \quad \Gamma \vdash t_1 : FA_1 & \quad \Gamma, x : A_1 \vdash t_2 : A_2 \\
\Gamma \vdash \text{return } t : FA & \quad \Gamma \vdash t_1 \to x.t_2 : A_2 \\
\Gamma \vdash t_1 : \beta & \quad \Gamma, x : \alpha \vdash t_2 : A & \quad (\text{op} : \beta \to \alpha) \\
\Gamma \vdash \text{op}_{t_1}(x.t_2) : A
\end{align*}
\]
The programming language

- We use a variant of the Effect Calculus (closely related to Call-by-Push-Value)
  - Egger et. al. ’09, ’12
  - Levy ’01,’04

- Value and computation types:
  - \( A ::= \alpha | 1 | A_1 \times A_2 | A_1 \rightarrow A_2 | FA \)
  - \( \overline{A} ::= A_1 \times A_2 | A_1 \rightarrow A_2 | FA \)

- Terms \( t \):
  - \( t ::= x | \ast | \langle t_1, t_2 \rangle | \text{proj}_i t | \lambda x.t | t_1(t_2) | \text{return} t | t_1 \text{ to } x.t_2 | \text{op}_{t_1}(x.t_2) \)

- Well-typed terms \( \Gamma \vdash t : A \), e.g.:
  - \[
    \frac{\Gamma \vdash t : A}{\Gamma \vdash \text{return } t : FA}
  \]
  - \[
    \frac{\Gamma \vdash t_1 : FA_1 \quad \Gamma, x : A_1 \vdash t_2 : A_2}{\Gamma \vdash \text{to } x.t_2 : A_2}
  \]
  - \[
    \frac{\Gamma \vdash t_1 : \beta \quad \Gamma, x : \alpha \vdash t_2 : A}{\Gamma \vdash \text{op}_{t_1}(x.t_2) : A} \quad (\text{op : } \beta \rightarrow \alpha)
  \]
The programming language

- We use a variant of the Effect Calculus (closely related to Call-by-Push-Value).
  - Egger et. al. ’09, ’12
  - Levy ’01,’04

- Value and computation types:
  - \( A ::= \alpha \mid 1 \mid A_1 \times A_2 \mid A_1 \rightarrow A_2 \mid FA \)
  - \( A ::= A_1 \times A_2 \mid A_1 \rightarrow A_2 \mid FA \)

- Terms \( t \):
  - \( t ::= x \mid \star \mid \langle t_1, t_2 \rangle \mid \text{proj}_i t \mid \lambda x.t \mid t_1(t_2) \mid \text{return } t \mid t_1 \text{ to } x. t_2 \mid \text{op}_{t_1}(x.t_2) \)

- Well-typed terms \( \Gamma \vdash t : A \), e.g.:
  - \( \Gamma \vdash t : A \)
  - \( \Gamma \vdash t_1 : FA_1 \quad \Gamma, x : A_1 \vdash t_2 : A_2 \)
  - \( \Gamma \vdash \text{return } t : FA \)
  - \( \Gamma \vdash t_1 \text{ to } x. t_2 : A_2 \)
  - \( \Gamma \vdash t_1 : \beta \quad \Gamma, x : \alpha \vdash t_2 : A \)
  - \( \Gamma \vdash \text{op}_{t_1}(x.t_2) : A \quad (\text{op} : \beta \rightarrow \alpha) \)
Algebraic effectful reasoning

- This algebraic treatment of effects induces an effectful multi-sorted logic on EC:
  - Value types: \( A ::= \alpha | 1 | A_1 \times A_2 | A_1 \rightarrow A_2 | FA \)
  - Computation types: \( A ::= A_1 \times A_2 | A_1 \rightarrow A_2 | FA \)
  - Terms: \( t ::= x | \star | \langle t_1, t_2 \rangle | \text{proj}_i t | \lambda x.t | t_1(t_2) | \text{return } t | t_1 \text{ to } x.t_2 | \text{op}_{t_1}(x.t_2) \)
  - Formulas: \( \varphi ::= t_1 = t_2 | R(t) | \pi(t) | \neg \varphi | \varphi_1 \lor \varphi_2 | \exists x.\varphi \)
  - Predicates: \( \pi ::= X | (\bar{x}).\varphi | \mu X.\pi | \nu X.\pi \)
- Allows algebraic effectful reasoning:
  - Reasoning in terms of equivalence classes of computation trees
  - Based on the logic of algebraic effects for CBPV

Plotkin & Pretnar '08
Algebraic effectful reasoning

- This algebraic treatment of effects induces an effectful multi-sorted logic on EC:
  - Value types: $A ::= \alpha \mid 1 \mid A_1 \times A_2 \mid A_1 \rightarrow A_2 \mid FA$
  - Computation types: $A ::= A_1 \times A_2 \mid A_1 \rightarrow A_2 \mid FA$
  - Terms: $t ::= x \mid \star \mid \langle t_1, t_2 \rangle \mid proj_i t \mid \lambda x.t \mid t_1(t_2) \mid return \ t \mid t_1 \text{to} \ x.t_2 \mid \text{op}_{t_1}(x.t_2)$
  - Formulas: $\varphi ::= t_1 = t_2 \mid R(t) \mid \pi(t) \mid \neg \varphi \mid \varphi_1 \lor \varphi_2 \mid \exists x.\varphi$
  - Predicates: $\pi ::= X \mid (\vec{x}).\varphi \mid \mu X.\pi \mid \nu X.\pi$

- Allows algebraic effectful reasoning:
  - Reasoning in terms of equivalence classes of computation trees

- Based on the logic of algebraic effects for CBPV

Plotkin & Pretnar '08
Algebraic effectful reasoning

- This algebraic treatment of effects induces an effectful multi-sorted logic on EC:
  - **Value types:** \( A ::= \alpha \mid 1 \mid A_1 \times A_2 \mid A_1 \to A_2 \mid FA \)
  - **Computation types:** \( A ::= A_1 \times A_2 \mid A_1 \to A_2 \mid FA \)
  - **Terms:** \( t ::= \)
    - \( x \mid \star \mid \langle t_1, t_2 \rangle \mid \text{proj}_i t \mid \lambda x.t \mid t_1(t_2) \mid \text{return} \ t \mid t_1 \, \text{to} \, x \cdot t_2 \mid \text{op}_{t_1}(x \cdot t_2) \)
  - **Formulas:** \( \varphi ::= t_1 = t_2 \mid R(t) \mid \pi(t) \mid \neg \varphi \mid \varphi_1 \lor \varphi_2 \mid \exists x.\varphi \)
  - **Predicates:** \( \pi ::= X \mid (\vec{x}).\varphi \mid \mu X.\pi \mid \nu X.\pi \)

- Allows algebraic effectful reasoning:
  - Reasoning in terms of equivalence classes of computation trees
- Based on the logic of algebraic effects for CBPV

Plotkin & Pretnar '08
Algebraic effectful reasoning

- This algebraic treatment of effects induces an effectful multi-sorted logic on EC:
  - Value types: $A ::= \alpha \mid 1 \mid A_1 \times A_2 \mid A_1 \rightarrow A_2 \mid FA$
  - Computation types: $A ::= A_1 \times A_2 \mid A_1 \rightarrow A_2 \mid FA$
  - Terms: $t ::= x \mid \ast \mid \langle t_1, t_2 \rangle \mid \text{proj}_i t \mid \lambda x.t \mid t_1(t_2) \mid \text{return } t \mid t_1 \text{ to } x.t_2 \mid \text{op}_{t_1}(x.t_2)$
  - Formulas: $\varphi ::= t_1 = t_2 \mid R(t) \mid \pi(t) \mid \neg \varphi \mid \varphi_1 \lor \varphi_2 \mid \exists x.\varphi$
  - Predicates: $\pi ::= X \mid (\tilde{x}).\varphi \mid \mu X.\pi \mid \nu X.\pi$
- Allows algebraic effectful reasoning:
  - Reasoning in terms of equivalence classes of computation trees
- Based on the logic of algebraic effects for CBPV

Plotkin & Pretnar '08
Refinement types for effectful computations

- The story is similar to the \(\lambda\)-calc. ref. types \(\Gamma \vdash \sigma : \text{Ref}(A)\).

- We start with EC and its value & computation types:
  - \(A ::= \alpha \mid 1 \mid A_1 \times A_2 \mid A_1 \to A_2 \mid FA\)
  - \(A ::= A_1 \times A_2 \mid A_1 \to A_2 \mid FA\)

- We define the refinement types as:
  - \(\sigma ::= \alpha \mid 1 \mid \Sigma_{x:\sigma_1} \sigma_2 \mid \Pi_{x:\sigma_1} \sigma_2 \mid F\sigma \mid \{x : \sigma \mid \varphi\}\)
  - \(\tau ::= \tau_1 \times \tau_2 \mid \Pi_{x:\sigma} \tau \mid F\sigma\)

- Notice: no refinements on computation types
  - \(\varphi\)'s do not induce subalgebras in general
    - would break the adj. model principle (comp. types as algebras)

- Well-formed ref. types similar to \(\lambda\)-calc wf. ref. types, e.g.:
  - \(\Gamma \vdash \sigma : \text{Ref}(A)\)
  - \(\Gamma, x : A \vdash \varphi : \text{prop}\)
  - \(\Gamma \vdash \sigma : \text{Ref}(A)\)
  - \(\Gamma \vdash \{x : \sigma \mid \varphi\} : \text{Ref}(A)\)
  - \(\Gamma \vdash F\sigma : \text{Ref}(FA)\)
Refinement types for effectful computations

The story is similar to the $\lambda$-calc. ref. types $\Gamma \vdash \sigma : \text{Ref}(A)$

We start with EC and its value & computation types:

- $A ::= \alpha \mid 1 \mid A_1 \times A_2 \mid A_1 \to A_2 \mid FA$
- $\tau ::= \tau_1 \times \tau_2 \mid \Sigma_{x:\tau_1} \tau_2 \mid \Pi_{x:\tau_1} \tau_2 \mid F\sigma \mid \{x : \sigma \mid \varphi\}$

We define the refinement types as:

- $\sigma ::= \alpha \mid 1 \mid \Sigma_{x:\sigma_1} \sigma_2 \mid \Pi_{x:\sigma_1} \sigma_2 \mid F\sigma \mid \{x : \sigma \mid \varphi\}$

Notice: no refinements on computation types

- $\varphi$’s do not induce subalgebras in general
- would break the adj. model principle (comp. types as algebras)

Well-formed ref. types similar to $\lambda$-calc wf. ref. types, e.g.:

<table>
<thead>
<tr>
<th>$\Gamma \vdash \sigma : \text{Ref}(A)$</th>
<th>$\Gamma, x : A \vdash \varphi : \text{prop}$</th>
<th>$\Gamma \vdash \sigma : \text{Ref}(A)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash {x : \sigma \mid \varphi} : \text{Ref}(A)$</td>
<td>$\Gamma \vdash F\sigma : \text{Ref}(FA)$</td>
<td></td>
</tr>
</tbody>
</table>
Refinement types for effectful computations

- The story is similar to the $\lambda$-calc. ref. types $\Gamma \vdash \sigma : \text{Ref}(A)$

- We start with EC and its value & computation types:
  - $A ::= \alpha | 1 | A_1 \times A_2 | A_1 \to A_2 | FA$
  - $A ::= A_1 \times A_2 | A_1 \to A_2 | FA$

- We define the refinement types as:
  - $\sigma ::= \alpha | 1 | \Sigma x:\sigma_1 \sigma_2 | \Pi x:\sigma_1 \sigma_2 | F\sigma | \{ x : \sigma | \varphi \}$
  - $\tau ::= \tau_1 \times \tau_2 | \Pi x:\tau \tau | F\sigma$

- Notice: no refinements on computation types
  - $\varphi$'s do not induce subalgebras in general
  - would break the adj. model principle (comp. types as algebras)

- Well-formed ref. types similar to $\lambda$-calc wf. ref. types, e.g.:

\[
\begin{align*}
\Gamma \vdash \sigma : \text{Ref}(A) & \quad \Gamma, x : A \vdash \varphi : \text{prop} & \quad \Gamma \vdash \sigma : \text{Ref}(A) \\
\hline
\Gamma \vdash \{ x : \sigma | \varphi \} : \text{Ref}(A) & \quad \Gamma \vdash F\sigma : \text{Ref}(FA)
\end{align*}
\]
Refinement types for effectful computations

- The story is similar to the $\lambda$-calc. ref. types $\Gamma \vdash \sigma : \text{Ref}(A)$
- We start with EC and its value & computation types:
  - $A ::= \alpha \mid 1 \mid A_1 \times A_2 \mid A_1 \to A_2 \mid FA$
  - $A ::= A_1 \times A_2 \mid A_1 \to A_2 \mid FA$
- We define the refinement types as:
  - $\sigma ::= \alpha \mid 1 \mid \Sigma_{x:\sigma_1} \sigma_2 \mid \Pi_{x:\sigma_1} \sigma_2 \mid F\sigma \mid \{x : \sigma \mid \varphi\}$
  - $\tau ::= \tau_1 \times \tau_2 \mid \Pi_{x:\tau} \tau \mid F\sigma$
- Notice: no refinements on computation types
  - $\varphi$’s do not induce subalgebras in general
  - would break the adj. model principle (comp. types as algebras)
- Well-formed ref. types similar to $\lambda$-calc wf. ref. types, e.g.:
  - $\Gamma \vdash \sigma : \text{Ref}(A)$
  - $\Gamma, x : A \vdash \varphi : \text{prop}$
  - $\Gamma \vdash \sigma : \text{Ref}(A)$
  - $\Gamma \vdash \{x : \sigma \mid \varphi\} : \text{Ref}(A)$
  - $\Gamma \vdash F\sigma : \text{Ref}(FA)$
Refinement types for effectful computations

- The story is similar to the λ-calc. ref. types $\Gamma \vdash \sigma : \text{Ref}(A)$

- We start with EC and its value & computation types:
  - $A ::= \alpha \mid 1 \mid A_1 \times A_2 \mid A_1 \to A_2 \mid FA$
  - $A ::= A_1 \times A_2 \mid A_1 \to A_2 \mid FA$

- We define the refinement types as:
  - $\sigma ::= \alpha \mid 1 \mid \Sigma_{x:\sigma_1}\sigma_2 \mid \Pi_{x:\sigma_1}\sigma_2 \mid F\sigma \mid \{x : \sigma \mid \varphi\}$
  - $\tau ::= \tau_1 \times \tau_2 \mid \Pi_{x:\sigma}\tau \mid F\sigma$

- Notice: no refinements on computation types
  - $\varphi$’s do not induce subalgebras in general
  - would break the adj. model principle (comp. types as algebras)

- Well-formed ref. types similar to λ-calc wf. ref. types, e.g.:
  $\Gamma \vdash \sigma : \text{Ref}(A)$
  $\Gamma, x : A \vdash \varphi : \text{prop}$
  $\Gamma \vdash \sigma : \text{Ref}(A)$
  \[\Gamma \vdash \{x : \sigma \mid \varphi\} : \text{Ref}(A)\]
  $\Gamma \vdash F\sigma : \text{Ref}(FA)$
Refinement types for effectful computations ctd.

- Well-typed terms follow the adj. model considerations:
  \[
  \begin{align*}
  \Gamma &\vdash t : \sigma & |\Gamma| &\vdash \varphi[|t|/x] \\
  \hline \\
  \Gamma &\vdash t : \{x : \sigma | \varphi\} & |\Gamma| &\vdash \varphi[|t|/x] \\
  \hline \\
  \Gamma &\vdash t : \sigma \\
  \hline \\
  \Gamma &\vdash \text{return } t : F\sigma \\
  \hline \\
  \Gamma &\vdash t_1 : F\sigma_1 & \Gamma, x : \sigma_1 &\vdash t_2 : \tau \\
  \hline \\
  \Gamma &\vdash t_1 \text{ to } x \cdot t_2 : \tau \\
  \hline \\
  \Gamma &\vdash t_1 : \beta & \Gamma, x : \alpha &\vdash t_2 : \tau \\
  \hline \\
  \Gamma &\vdash \text{op}_{t_1}(x \cdot t_2) : \tau
  \end{align*}
  \]

- Also, more modular verification rules are derivable, e.g.:
  \[
  \begin{align*}
  \Gamma &\vdash t_1 : \sigma_1 & |\sigma_1| = \beta & \Gamma, x : \alpha &\vdash t_2 : \sigma_2 & |\sigma| = |\sigma_2| = A \\
  \Gamma &\vdash \{x : A | \exists x', x''.x = \text{op}_{x'}(x \cdot x''(x)) \land \sigma_1[x'/x] \land \forall x'''.\sigma_2[x''(x''')/x]\} \sqsubseteq \sigma \\
  \Gamma &\vdash \text{op}_{t_1}(x \cdot t_2) : \sigma
  \end{align*}
  \]
Refinement types for effectful computations ctd.

Well-typed terms follow the adj. model considerations:

\[
\begin{align*}
\Gamma \vdash t : \sigma & \quad |\Gamma| \mid \Gamma^\circ \vdash \varphi[|t|/x] \\
\hline
\Gamma \vdash t : \{x : \sigma \mid \varphi\} & \quad \Gamma \vdash \varphi[|t|/x] \\
\Gamma \vdash t : \sigma & \quad |\Gamma| \mid \Gamma^\circ \vdash \varphi[|t|/x] \\
\hline
\Gamma \vdash \text{return } t : F\sigma & \quad \Gamma, x : \sigma_1 \vdash t_2 : \tau \\
\Gamma \vdash t_1 : F\sigma_1 & \\
\hline
\Gamma \vdash t_1 \text{ to } x. t_2 : \tau & \quad \Gamma \vdash \text{op}_{t_1}(x. t_2) : \tau \\
\Gamma \vdash t_1 : \beta & \quad \Gamma, x : \alpha \vdash t_2 : \tau \\
\hline
\Gamma \vdash \text{op}_{t_1}(x. t_2) : \tau
\end{align*}
\]

Also, more modular verification rules are derivable, e.g.:

\[
\begin{align*}
\Gamma \vdash t_1 : \sigma_1 & \quad |\sigma_1| = \beta & \Gamma, x : \alpha \vdash t_2 : \sigma_2 & \quad |\sigma| = |\sigma_2| = A \\
\Gamma \vdash \{x : A \mid \exists x', x''. x = \text{op}_{x'}(x. x''(x)) \land \sigma_1^\circ[x'/x] \land \forall x'''. \sigma_2^\circ[x''(x''')/x]\} \sqsubseteq \sigma \\
\hline
\Gamma \vdash \text{op}_{t_1}(x. t_2) : \sigma
\end{align*}
\]
Refinement types for effectful computations ctd.

Well-typed terms follow the adj. model considerations:

\[
\begin{align*}
\Gamma \vdash t : \sigma & \quad |\Gamma| \mid \Gamma^\circ \vdash \varphi[|t|/x] \\
\overline{\Gamma \vdash t : \{x : \sigma \mid \varphi\}} \\
\Gamma \vdash t : \varphi[|t|/x] & \quad |\Gamma| \mid \Gamma^\circ \vdash \varphi[|t|/x] \\
\overline{\Gamma \vdash \text{return } t : F\sigma} \\
\Gamma \vdash t_1 : F\sigma_1 & \quad \Gamma, x : \sigma_1 \vdash t_2 : \tau \\
\overline{\Gamma \vdash t_1 \text{ to } x.t_2 : \tau} \\
\Gamma \vdash t_1 : \beta & \quad \Gamma, x : \alpha \vdash t_2 : \tau \\
\overline{\Gamma \vdash \text{op}_{t_1}(x.t_2) : \tau}
\end{align*}
\]

Also, more modular verification rules are derivable, e.g.:

\[
\begin{align*}
\Gamma \vdash t_1 : \sigma_1 & \quad |\sigma_1| = \beta \quad \Gamma, x : \alpha \vdash t_2 : \sigma_2 \\
|\sigma| = |\sigma_2| = A \\
\Gamma \vdash \{x : A \mid \exists x', x''.x = \text{op}_{x'}(x.x''(x)) \land \sigma_1^\circ[x'/x] \land \forall x'''.\sigma_2^\circ[x''(x''')/x]\} \sqsubseteq \sigma \\
\Gamma \vdash \text{op}_{t_1}(x.t_2) : \sigma
\end{align*}
\]
Examples: communication

- Recall the small state language:
  - induced by the 1-location state theory
  - receive: \(1 \rightarrow \text{nat} \), send: \(\text{nat} \rightarrow 1\)

- Recall the a grammar of session refinements:
  - \(S(A) ::= \text{end}(A) | !(x : \text{nat} | \varphi).S(A) | \)
    
    \(? (y : \text{nat}).S(A) | S_1(B); S_2(A)\)

- They are defined as operations on predicates, e.g.:
  - \(\text{end}(A) \overset{\text{def}}{=} (x : FA).\exists x'.x = \text{return } x'\)
  - \(!(x : \text{nat} | \varphi).S(A) \overset{\text{def}}{=} (x : FA).\exists x', x''.x = \text{send}_{x'}(x'') \land \varphi[x'/x] \land (S(A)[x'/x])(x'')\)
  - \(? (x : \text{nat}.S(A)) \overset{\text{def}}{=} (x : FA).\exists x'.x = \text{receive}(x.x'(x)) \land \forall x''.(S(A)[x''/x])(x'(x''))\)
  - \(S(A); S(B) \overset{\text{def}}{=} ...\)
Examples: communication

- Recall the small state language:
  - induced by the 1-location state theory
  - receive : 1 → nat  ,  send : nat → 1

- Recall the a grammar of session refinements:
  - \( S(A) ::= end(A) | !(x : \text{nat} \mid \varphi).S(A) | \\
  \quad ?(y : \text{nat}).S(A) | S_1(B); S_2(A) \)

- They are defined as operations on predicates, e.g.:
  - \( end(A) \overset{\text{def}}{=} (x : FA).\exists x'.x = \text{return } x' \)
  - \( !(x : \text{nat} \mid \varphi).S(A) \overset{\text{def}}{=} (x : FA).\exists x', x''.x = \text{send}_{x'}(x'') \land \\
  \quad \varphi[x'/x] \land (S(A)[x'/x])(x'') \)
  - \( ?(x : \text{nat}.S(A)) \overset{\text{def}}{=} (x : FA).\exists x'.x = \text{receive}(x.x'(x)) \land \\
  \quad \forall x''.(S(A)[x''/x])(x'(x'')) \)
  - \( S(A); S(B) \overset{\text{def}}{=} ... \)
Examples: communication

- Recall the small state language:
  - induced by the 1-location state theory
  - receive: $1 \rightarrow \text{nat}$, send: $\text{nat} \rightarrow 1$

- Recall the a grammar of session refinements:
  - $S(A) ::= \text{end}(A) \mid !(x : \text{nat} \mid \varphi).S(A) \mid$
    $\ ?(y : \text{nat}).S(A) \mid S_1(B); S_2(A)$

- They are defined as operations on predicates, e.g.:
  - $\text{end}(A) \stackrel{\text{def}}{=} (x : FA).\exists x'.x = \text{return } x'$
  - $!(x : \text{nat} \mid \varphi).S(A) \stackrel{\text{def}}{=} (x : FA).\exists x', x''.x = \text{send}_{x'}(x'') \land$
    $\varphi[x'/x] \land (S(A)[x'/x])(x'')$
  - $?(x : \text{nat}.S(A)) \stackrel{\text{def}}{=} (x : FA).\exists x'.x = \text{receive}(x.x'(x)) \land$
    $\forall x''.(S(A)[x''/x])(x'(x''))$
  - $S(A); S(B) \stackrel{\text{def}}{=} ...$
Examples: communication

- Recall the small state language:
  - induced by the 1-location state theory
  - receive: \( 1 \rightarrow \text{nat} \), send: \( \text{nat} \rightarrow 1 \)

- Recall the grammar of session refinements:
  - \( S(A) ::= \text{end}(A) \mid !(x : \text{nat} \mid \varphi).S(A) \mid \)
    \( ?(y : \text{nat}).S(A) \mid S_1(B) ; S_2(A) \)

- They are defined as operations on predicates, e.g.:

  - \( \text{end}(A) \overset{\text{def}}{=} (x : FA).\exists x'.x = \text{return } x' \)

  - \( !(x : \text{nat} \mid \varphi).S(A) \overset{\text{def}}{=} (x : FA).\exists x', x''.x = \text{send}_x(x'') \land \)
    \( \varphi[x'/x] \land (S(A)[x'/x])(x'') \)

  - \( ?(x : \text{nat}.S(A)) \overset{\text{def}}{=} (x : FA).\exists x'.x = \text{receive}(x.x'(x)) \land \)
    \( \forall x''.(S(A)[x''/x])(x'(x'')) \)

  - \( S'(A); S(B) \overset{\text{def}}{=} \ldots \)
Examples: communication

- Recall the small state language:
  - induced by the 1-location state theory
  - receive : 1 → nat , send : nat → 1

- Recall the a grammar of session refinements:
  - \( S(A) ::= end(A) \mid !(x : \text{nat} \mid \varphi).S(A) \mid ?(y : \text{nat}).S(A) \mid S_1(B); S_2(A) \)

- They are defined as operations on predicates, e.g.:
  - \( \text{end}(A) \overset{\text{def}}{=} (x : FA).\exists x'.x = \text{return } x' \)
  - \( !(x : \text{nat} \mid \varphi).S(A) \overset{\text{def}}{=} (x : FA).\exists x', x''.x = \text{send}_{x'}(x'') \land \varphi[x'/x] \land (S(A)[x'/x])(x'') \)
  - \( ?(x : \text{nat}.S(A)) \overset{\text{def}}{=} (x : FA).\exists x'.x = \text{receive}(x.x'(x)) \land \forall x''.(S(A)[x''/x])(x'(x'')) \)
  - \( S'(A); S(B) \overset{\text{def}}{=} \ldots \)
Examples: communication

- Recall the small state language:
  - induced by the 1-location state theory
  - receive : 1 → nat , send : nat → 1

- Recall the a grammar of session refinements:
  - $S(A) ::= end(A) | !(x : nat | \varphi).S(A) | \quad \text{??(y : nat).S(A) | } S_1(B); S_2(A)

- They are defined as operations on predicates, e.g.:
  - $end(A) \triangleq (x : FA).\exists x'.x = \text{return } x'$
  - $!(x : \text{nat} | \varphi).S(A) \triangleq (x : FA).\exists x', x''.x = \text{send}_{x'}(x'') \land \varphi[x'/x] \land (S(A)[x'/x])(x'')$
  - $?(x : \text{nat}.S(A)) \triangleq (x : FA).\exists x'.x = \text{receive}(x.x'(x)) \land \forall x''.(S(A)[x''/x])(x'(x''))$
  - $S(A); S(B) \triangleq \ldots$
Examples: communication

- Recall the small state language:
  - induced by the 1-location state theory
  - receive: \(1 \to \text{nat}\), send: \(\text{nat} \to 1\)

- Recall the a grammar of session refinements:
  - \(S(A) ::= \text{end}(A) \mid !(x : \text{nat} \mid \varphi).S(A) \mid\)
  - \(?(y : \text{nat}).S(A) \mid S_1(B); S_2(A)\)

- They are defined as operations on predicates, e.g.:
  - \(\text{end}(A) \overset{\text{def}}{=} (x : FA).\exists x'. x = \text{return } x'\)
  - \( !(x : \text{nat} \mid \varphi).S(A) \overset{\text{def}}{=} (x : FA).\exists x', x''. x = \text{send}_{x'}(x'') \land \varphi[x'/x] \land (S(A)[x'/x])(x'')\)
  - \(?(x : \text{nat}.S(A)) \overset{\text{def}}{=} (x : FA).\exists x'. x = \text{receive}(x.x'(x)) \land \forall x''.(S(A)[x''/x])(x'(x''))\)
  - \(S(A); S(B) \overset{\text{def}}{=} ...\)
Examples: state

- Recall the small state language:
  - induced by the 1-location state theory
  - lookup : 1 → nat ,  update : nat → 1

- Formulas \( \varphi_P \) and \( \varphi_Q \) on states (on natural numbers)

- The pre- & post-condition spec.:

  \[
  \forall \vec{x}. \{(x_0).\varphi_P\} y : A\{(x_1).\varphi_Q\} \overset{\text{def}}{=} (x : FA). (\forall \vec{x}'. \forall x_s. \pi_P[\vec{x'}/\vec{x}, x_s/x_0] \implies \pi_{aux}(\vec{x}', x_s, x_s, x))
  \]

  where (for total correctness)

  \[
  \pi_{aux} \overset{\text{def}}{=} \mu X. ((\vec{x}, x_0, x_1, x).
  (\exists y.x = \text{return } y \land \varphi_Q)
  \lor (\exists x'.x = \text{lookup}(x.x'(x)) \land X(\vec{x}, x_0, x_1, x'(x_1)))
  \lor (\exists x', x''.x = \text{update}_{x'}(x'') \land X(\vec{x}, x_0, x', x'')))
  \]
Examples: state

- Recall the small state language:
  - induced by the 1-location state theory
  - lookup : 1 → nat ,  update : nat → 1

- Formulas \( \varphi_P \) and \( \varphi_Q \) on states (on natural numbers)

- The pre- & post-condition spec.:

\[
\forall \vec{x}. \{(x_0).\varphi_P\} y : A\{(x_1).\varphi_Q\} \overset{\text{def}}{=} \\
(x : FA). (\forall x'. \forall x_s. \pi_P[x'/\vec{x}, x_s/x_0] \implies \pi_{aux}(x', x_s, x_s, x))
\]

where (for total correctness)

\[
\pi_{aux} \overset{\text{def}}{=} \mu X.((\vec{x}, x_0, x_1, x).

\left( \exists y. x = \text{return } y \land \varphi_Q \right)
\lor \left( \exists x'. x = \text{lookup}(x.x'(x)) \land X(\vec{x}, x_0, x_1, x'(x_1)) \right)
\lor \left( \exists x', x''. x = \text{update}_{x'}(x'') \land X(\vec{x}, x_0, x', x'') \right)
\]
Examples: state

- Recall the small state language:
  - induced by the 1-location state theory
  - lookup : 1 → nat ,  update : nat → 1

- Formulas $\varphi_P$ and $\varphi_Q$ on states (on natural numbers)

- The pre- & post-condition spec.:

  $\forall \vec{x}. \{(x_0). \varphi_P \} y : A\{(x_1). \varphi_Q \} \overset{\text{def}}{=} (x : FA). (\forall \vec{x}'. \forall x_s. \pi_P[\vec{x}'/\vec{x}, x_s/x_0] \implies \pi_{aux}(\vec{x}', x_s, x_s, x))$

  where (for total correctness)

  $\pi_{aux} \overset{\text{def}}{=} \mu X. ((\vec{x}, x_0, x_1, x)$.

  $\left( \exists y. x = \text{return} y \land \varphi_Q \right)$

  $\lor \left( \exists x'. x = \text{lookup}(x. x'(x)) \land X(\vec{x}, x_0, x_1, x'(x_1)) \right)$

  $\lor \left( \exists x', x''. x = \text{update}_{x'}(x'') \land X(\vec{x}, x_0, x', x'') \right)$
Recall the combined spec. on state & communication:

\[ \forall \vec{x}.\{(x_0).\varphi_P\}(S(A) \Rightarrow x : A)\{(x_1).\varphi_Q\} \]

How well can we represent it in our ref. ty. system?

Combining underlying state & comm. calculi is easy:
- induced by the tensor of effect theories
- semantics induced similarly (i.e., \( T_\otimes = (T_{IO}(Val^{Loc} \times -))^{Val^{Loc}} \))

Combining refinement specs.:
- not so straightforward, no obvious good combinators
- similarity between ref. specs. and monads

\[ \forall \left( \exists x'.x = \text{receive}(x.x'(x)) \right) \land \\
\exists Y.\left( S(x) \iff (\exists y : \text{nat}.Y) \right) \land X(\vec{x}, x_0, x_1, x, Y) \]
Examples: state \( \otimes \) communication

- Recall the combined spec. on state & communication:
  \[
  \forall \vec{x}.\{(x_0).\varphi_P\}(S(A) \Rightarrow x : A)\{(x_1).\varphi_Q\}
  \]

- How well can we represent it in our ref. ty. system?

- Combining underlying state & comm. calculi is easy:
  - induced by the tensor of effect theories
  - semantics induced similarly (i.e., \( T_\otimes = (T_{IO}(Val^Loc \times -))^{Val^Loc} \))

- Combining refinement specs.:
  - not so straightforward, no obvious good combinators
  - similarity between ref. specs. and monads

\[ \exists \vec{x}.'x = \text{receive}(x.x'(x)) \land \exists Y.\big(S(x) \iff (\exists y : \text{nat}.Y)\big) \land X(\vec{x}, x_0, x_1, x, Y) \]
Examples: state $\otimes$ communication

- Recall the combined spec. on state & communication:
  \[ \forall \vec{x}. \{(x_0).\varphi_P\}(S(A) \Rightarrow x : A)\{(x_1).\varphi_Q\} \]

- How well can we represent it in our ref. ty. system?

- Combining underlying state & comm. calculi is easy:
  - induced by the tensor of effect theories
  - semantics induced similarly (i.e., \( T \otimes = (T_{IO}(Val^{Loc} \times -))^{Val^{Loc}} \))

- Combining refinement specs.:
  - not so straightforward, no obvious good combinators
  - similarity between ref. specs. and monads

- \[ \forall \exists x'. x = \text{receive}(x.x'(x)) \land \exists Y. \left( S(x) \iff (?(y : \text{nat}).Y) \right) \land X(\vec{x}, x_0, x_1, x, Y) \]
Examples: state \( \otimes \) communication

- Recall the combined spec. on state & communication:
  \[
  \forall \vec{x}. \{(x_0).\varphi_P\}(S(A) \Rightarrow x : A)\{(x_1).\varphi_Q\}
  \]

- How well can we represent it in our ref. ty. system?

- Combining underlying state & comm. calculi is easy:
  - induced by the tensor of effect theories
  - semantics induced similarly (i.e., \( T_\otimes = (T_{IO}(\text{Val}^{\text{Loc}} \times -))^{\text{Val}^{\text{Loc}}} \))

- Combining refinement specs.:
  - not so straightforward, no obvious good combinators
  - similarity between ref. specs. and monads

- \( \ldots \lor \left( \exists x'. x = \text{receive}(x.x'(x)) \land \right. \left. \exists Y. \left( S(x) \iff (?y : \text{nat}.Y) \right) \land X(\vec{x}, x_0, x_1, x, Y) \right) \)
Examples: state $\otimes$ communication

- Recall the combined spec. on state & communication:
  \[ \forall \vec{x}. \{(x_0).\varphi_P\} (S(A) \supset x : A) \{(x_1).\varphi_Q\} \]

- How well can we represent it in our ref. ty. system?

- Combining underlying state & comm. calculi is easy:
  - induced by the tensor of effect theories
  - semantics induced similarly (i.e., \( T_\otimes = (T_{IO}(\text{Val}^{\text{Loc}} \times -))^{\text{Val}^{\text{Loc}}} \))

- Combining refinement specs.:
  - not so straightforward, no obvious good combinators
  - similarity between ref. specs. and monads

- \( \lor \left( \exists x'. x = \text{receive}(x.x'(x)) \right) \land \\
  \exists Y. \left( S(x) \iff (?y : \text{nat}.Y) \right) \land X(\vec{x}, x_0, x_1, x, Y) \)
To sum it up

A computational language with algebraic effects

+ 

- ref. types for general effectful specs.
- using algebraic effectful reasoning

State language ⊆ Communication language ⊆ Language X

For the future:

- allow ref. types in logic?
- combinations of specs. more painlessly
- other algebraic machinery (locality, handlers)
- extension of simple ty. sys. with dependent refs. fibrationally