A Step-indexed Semantics of Imperative Objects

Cătălin Hriţcu and Jan Schwinghammer

Information Security and Cryptography Reading Group
8 January 2008, Saarbrücken, Germany

IN-DEPTH TALK

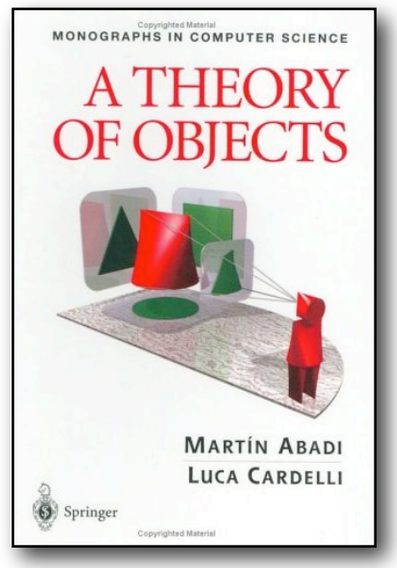
Outline

- Imperative object calculus
  - Why denotational models are hard to build?
- Step-indexed model of types
  - Step-indexing ... in general
  - Ahmed's model of general references and polymorphism
  - Modelling object types and subtyping
  - Type safety
- Future work: from types to specifications
  - Why do we need semantic models in the first place?

Imperative object calculus

$$
\begin{aligned}
a, b ::= \;& x \mid [\mathrm{m}_d{=}\varsigma(x_d)b_d]_{d\in D} \\
& \mid a.\mathrm{m} \mid a.\mathrm{m} := \varsigma(x)b \\
& \mid \mathrm{clone}\ a \mid \lambda(x)b \mid a\ b \\
v ::= & \{\mathrm{m}_d{=}l_d\}_{d\in D} \mid \lambda(x)b
\end{aligned}
$$

[Abadi and Cardelli, '96]



dynamically-allocated, higher-order store

+ expressive type system

$$
\begin{aligned}
\langle \emptyset, [\mathrm{m}{=}\varsigma(x)x & \\
\to \langle \{\!\{ l \mapsto \lambda(x)x.\mathrm{m} \}\!\}, \{\mathrm{m}{=}l\}.\mathrm{m} & \rangle \\
\to \langle \{\!\{ l \mapsto \lambda(x)x.\mathrm{m} \}\!\}, (\lambda(x)x.\mathrm{m})\ \{\mathrm{m}{=}l\} & \rangle \xrightarrow{\textsc{Beta}} \\
\to \langle \{\!\{ l \mapsto \lambda(x)x.\mathrm{m} \}\!\}, \{m{=}l\}.\mathrm{m} & \rangle \xrightarrow{\textsc{Inv}} \ldots
\end{aligned}
$$

Dynamically-allocated, higher-order store

- Feature present in many real languages:
  - General references in ML
  - Pointers to functions in C/C++
- Hard to find good semantic models
  - Reasoning about the behavior of programs
- Denotational semantics has troubles in this setting
  - Especially in the presence of expressive type system

Why are classic denotational models hard to build in this setting?

Problem 1: Semantic domains

- Higher-order store
  - Solving recursive domain equation

$$D_{Val} = (D_{Heaps} \times D_{Val} \rightharpoonup D_{Heaps} \times D_{Val}) + \ldots$$

$$D_{Heaps} = Loc \rightharpoonup_{fin} D_{Val}$$

  - For the imperative object calculus done in: [Kamin & Reddy, 94] [Reus & Streicher, '04]

- + polymorphic values stored (impredicative)
  - No domain-theoretic models known!

Problem 2: Types and heap typings

- If D_{Val} is a set then $Type = \mathcal{P}(D_{Val})$?
- No! Our values depend on the heap, e.g. $\{\mathrm{m}_d{=}l_d\}_{d \in D}$
  - so semantic types depend on heap typings
  - heap typings are maps from locations to types
    - No set-theoretic solution for

$$Type = \mathcal{P}(HeapTyping \times D_{Val})$$

$$HeapTyping = Loc \rightharpoonup_{fin} Type$$

- One trick: consider syntactic heap typings [Levi, '02]
  - powerdomain construction might also work

Problem 3: Dynamic allocation

- Heaps evolve during computation
  - No deallocation, updates type preserving
    - ➡ Heap typings can only "grow"
- Index semantic entities by heap typings
- Domain-theoretic possible-worlds models
  - Recursively defined functor categories over CPOs
  - Complex, no second-order types, not abstract enough
  - [Levi, '02] [Reus & Schwinghammer, '06]

Step-indexed models

Step-indexed models

- Introduced by Appel et al. [Appel & Felty, '00]
- Alternative to subject-reduction
  - Simpler machine-checkable proofs of type soundness
- Model for the lambda calculus with recursive types [Appel & McAllester, '01]
- Later extended to general references and impredicative polymorphism [Ahmed, '04]
  - We further extended it with object types and subtyping
  - Used it to prove the soundness of an expressive, standard type system for the imperative object calculus

Step-indexed models

- Based on a small-step operational semantics
- Simpler than the domain-theoretic models
- No need to search for semantic domains (problem 1)
  - We just take $D_{Val} = CVal$
- Can also model impredicative second-order types

Problem 2: Types and heap typings

- Circular definition $Type = \mathcal{P}(HeapTyping \times CVal)$

$$HeapTyping = Loc \rightharpoonup_{fin} Type$$

- We solve this by a stratified construction

$$Type_{k+1} \approx \mathcal{P}(\mathbb{N} \times (\bigcup_{j \leq k} HeapTyping_j) \times CVal)$$

$$HeapTyping_k = Loc \rightharpoonup_{fin} Type_k$$

- *k*-th approximation: $\lfloor \tau \rfloor_k = \{\langle j, \Psi, v \rangle \in \tau \mid j < k\}$
  - We have that $\lfloor \tau \rfloor_k \in Type_k$
- Stratification invariant:
  - $\lfloor \alpha \rfloor_{k+1}$ is only defined in terms of $\lfloor \Psi \rfloor_k$ and $\lfloor \tau \rfloor_k$

Semantic approximation

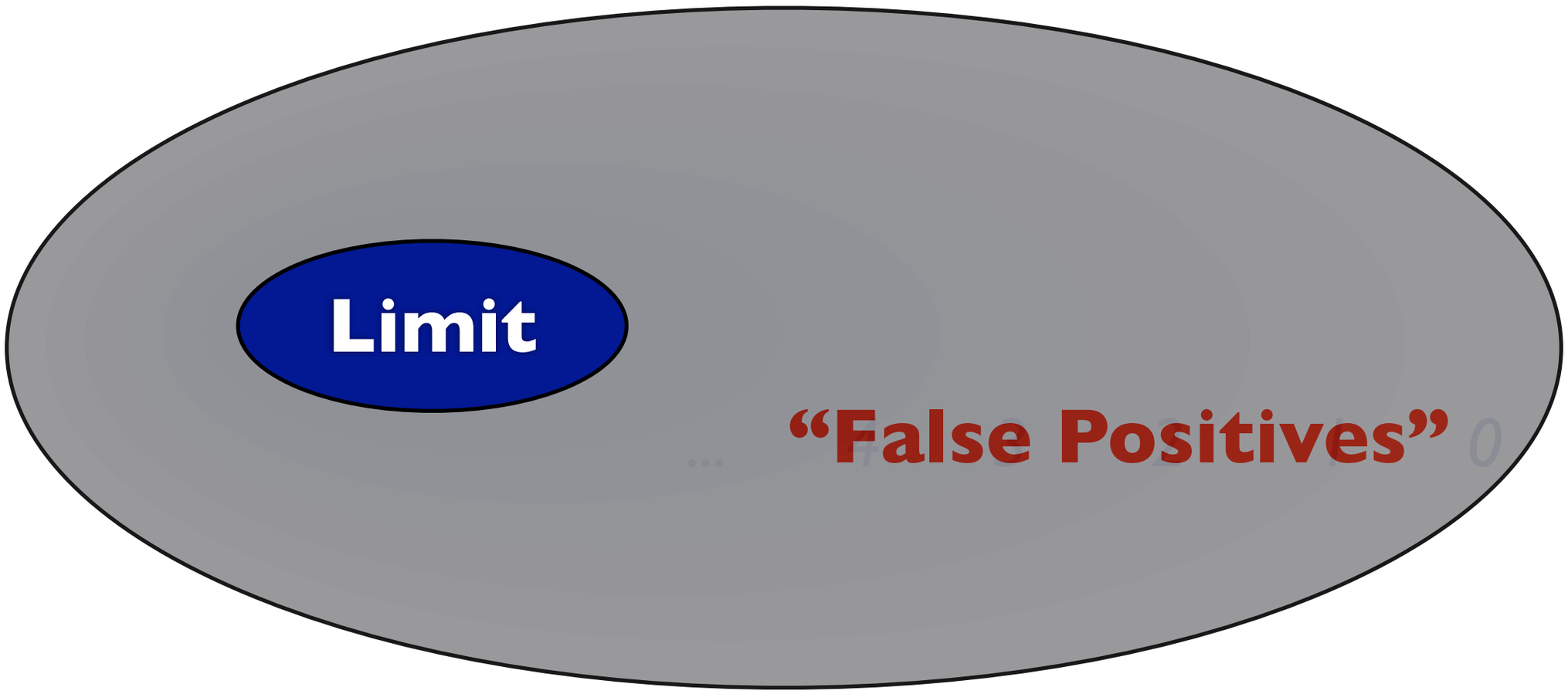
- Semantic types are sets of triples
- $\langle k, \Psi, v \rangle \in \tau$ if v safely executes for at least k steps in every context of type τ, and for every $h :_k \Psi$
- Example: $\langle 1, \emptyset, \lambda x.\, \mathrm{true} \rangle \in Nat \to Nat$

  $\langle 2, \emptyset, \lambda x.\, \mathrm{true} \rangle \notin Nat \to Nat$

- For values depending the store
  - $\langle k, \Psi, l \rangle \in \mathrm{ref}\ \tau \ \Rightarrow\ \forall h :_k \Psi.\, \langle k-1, \Psi, h(l) \rangle \in \tau$
  - Reading and writing to the store takes one step
  - Actual definition $\langle k, \Psi, l \rangle \in \mathrm{ref}\ \tau \ \Leftrightarrow\ \lfloor \Psi(l) \rfloor_k = \lfloor \tau \rfloor_k$

Semantic types

- Sequences of increasingly accurate approximations



- In the end we are only interested in the limit
- Approximation crucial for well-founded construction
  - And also extremely useful when dealing with recursion

State extension

- Dynamic allocation
  - Heap typings can only "grow" during computation
- The precision of our approximation can also decrease
- State extension

$$(k, \Psi) \sqsubseteq (j, \Psi') \quad \Leftrightarrow \quad j \leq k \ \wedge \ dom(\Psi) \subseteq dom(\Psi') \\ \wedge \ \forall l \in dom(\Psi).\ \lfloor \Psi' \rfloor_j (l) = \lfloor \Psi \rfloor_j (l)$$

- Possible-worlds model
- Semantic types should be closed under state extension

$$\langle k, \Psi, v \rangle \in \alpha \ \wedge \ (k, \Psi) \sqsubseteq (j, \Psi') \ \Rightarrow \ \langle j, \Psi', v \rangle \in \alpha$$

The type of a closed term

- Defined as

$$a :_{k,\Psi} \tau \;\Leftrightarrow \forall j < k.\, (h :_k \Psi \;\wedge\; \langle h, a\rangle \rightarrow^j \langle h', b\rangle \not\rightarrow)$$
$$\Rightarrow \exists \Psi'.\, (k, \Psi) \sqsubseteq (k-j, \Psi') \;\wedge\; h' :_{k-j} \Psi'$$
$$\wedge\; \langle k-j, \Psi', b\rangle \in \tau$$

- For example:

$$\lambda x.\, \mathrm{true} :_{1,\emptyset} Nat \to Nat$$
$$(\lambda x.\, x)\ (\lambda x.\, \mathrm{true}) :_{2,\emptyset} Nat \to Nat$$
$$(\lambda x.\, x)\ ((\lambda x.\, x)\ \mathrm{true}) :_{2,\emptyset} Nat \to Nat$$

(none of these holds if we increase the index by 1)

Simple semantic types

- Base types

$$Bool \triangleq \{\langle k, \Psi, v\rangle \mid k \in \mathbb{N}, \Psi \in HeapTyping_k, v \in \{\text{true}, \text{false}\}\}$$

$$Nat \triangleq \{\langle k, \Psi, \underline{n}\rangle \mid k, n \in \mathbb{N}, \Psi \in HeapTyping_k\}$$

- Procedure types

$$\alpha \to \beta \triangleq \{\langle k, \Psi, \lambda(x)b\rangle \mid \forall j < k, \Psi' \sqsupseteq \Psi, v. \langle j, \Psi', v\rangle \in \alpha \Rightarrow \{\!\{x \mapsto v\}\!\}(b) :_{j,\Psi'} \beta\}$$

Semantic typing judgement

- Typing open terms; not approximative

$$\Sigma \models a : \alpha \;\Leftrightarrow\; \forall k \geq 0.\, \forall \Psi.\, \forall \sigma :_{k,\Psi} \Sigma.\, \sigma(a) :_{k,\Psi} \alpha$$

- This definition directly enforces type safety
- But we still need to prove the typing rules sound
  - We first prove the validity of semantic typing lemmas
  - Then use these lemmas to prove the syntactic typing rules
- Example: subtyping recursive types (the amber rule)

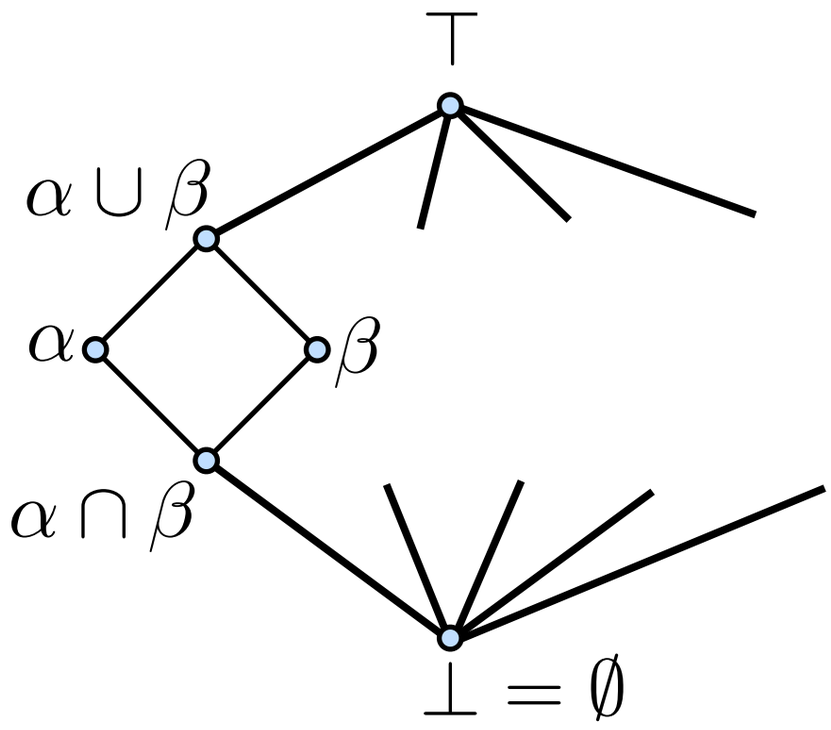
$$(\textsc{Semantic})\ \frac{\forall \alpha, \beta \in \mathit{Type}.\, \alpha \subseteq \beta \Rightarrow F(\alpha) \subseteq G(\beta)}{\mu F \subseteq \mu G}$$

$$(\textsc{Syntactic})\ \frac{\Gamma \vdash \mu X.\underline{A} \quad \Gamma \vdash \mu Y.\underline{B} \quad \Gamma, Y \leqslant \mathit{Top}, X \leqslant Y \vdash \underline{A} \leqslant \underline{B}}{\Gamma \vdash \mu X.\underline{A} \leqslant \mu Y.\underline{B}}$$

Object types and subtyping

Subtyping

- Since types are sets, subtyping is set inclusion
- Subtyping forms a complete lattice on types



Definition of object types

- Methods stored in the heap as procedures

$$[\mathrm{m}_d : \tau_d]_{d \in D} \approx \{\mathrm{m}_d : \mathrm{ref}\ (? \to \tau_d)\}_{d \in D}$$

- Self-application semantics of method invocation

$$\langle h, \{\mathrm{m}_d{=}l_d\}_{d \in D}\, .\mathrm{m}_e \rangle \ \to\ \langle h, h(l_e)\ \{\mathrm{m}_d{=}l_d\}_{d \in D} \rangle$$

suggests that object types need to be recursive

$$[\mathrm{m}_d : \tau_d]_{d \in D} \approx \mu(\alpha).\{\mathrm{m}_d : \mathrm{ref}\ (\alpha \to \tau_d)\}_{d \in D}$$

Definition of object types (first attempt)

$$\langle k, \Psi, \{\mathrm{m}_d{=}l_d\}_{d\in D}\rangle \in \alpha = [\mathrm{m}_d : \tau_d]_{d\in D} \;\Leftrightarrow$$
$$\forall d \in D.\, \langle k, \Psi, l\rangle \in \mathrm{ref}\ (\alpha \to \tau_d)$$

- Where the reference type is defined in [Ahmed '04]

$$\mathrm{ref}\ \tau = \{\langle k, \Psi, l\rangle \mid \lfloor \Psi(l) \rfloor_k = \lfloor \tau \rfloor_k\}$$

- This definition is well-founded (inductive on k)
  - $\lfloor \alpha \rfloor_{k+1}$ is defined in terms of $\lfloor \alpha \rfloor_k$
- The set it defines satisfies the stratification invariant
  - $\lfloor \alpha \rfloor_{k+1}$ is only defined in terms of $\lfloor \Psi \rfloor_k$ and $\lfloor \tau_d \rfloor_k$
- The set it defines is indeed a type
  - i.e. it is closed under state extension

$$\langle k, \Psi, v\rangle \in \alpha \ \wedge\ (k, \Psi) \sqsubseteq (j, \Psi') \ \Rightarrow\ \langle j, \Psi', v\rangle \in \alpha$$

Definition of object types (first attempt)

- This definition validates all typing lemmas for objects
  - Let $\alpha = [m_d : \tau_d]_{d \in D}$

$$(\textsc{Obj})\ \frac{\forall d \in D.\ \Sigma[x_d \mapsto \alpha] \models b_d : \tau_d}{\Sigma \models [m_d{=}\varsigma(x_d)b_d]_{d \in D} : \alpha} \qquad (\textsc{Clone})\ \frac{\Sigma \models a : \alpha}{\Sigma \models \text{clone } a : \alpha}$$

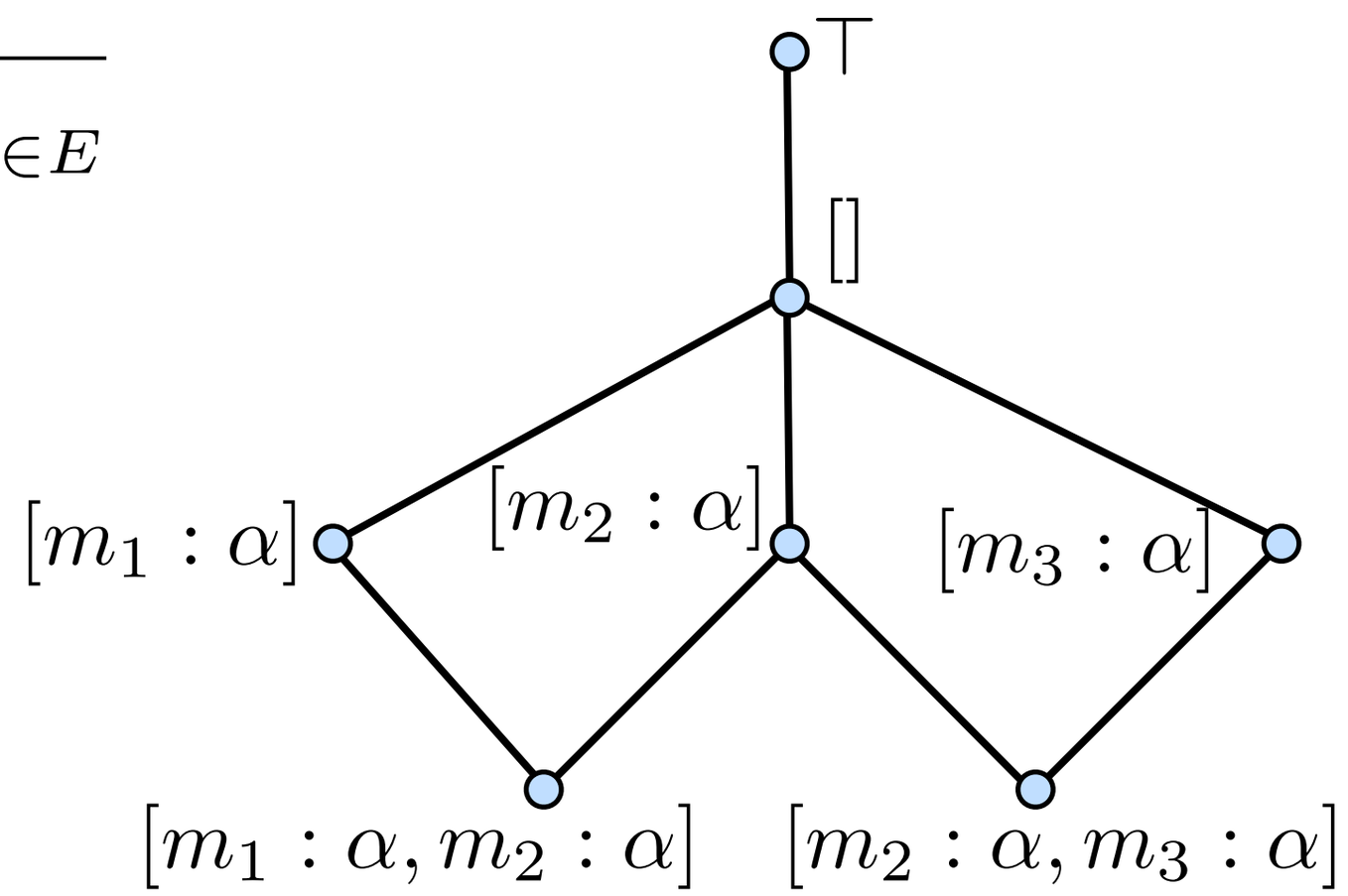
$$(\textsc{Inv})\ \frac{\Sigma \models a : \alpha \quad e \in D}{\Sigma \models a.m_e : \tau_e} \qquad (\textsc{Upd})\ \frac{\Sigma \models a : \alpha \quad e \in D \quad \Sigma[x \mapsto \alpha] \models b : \tau_e}{\Sigma \models a.m_e := \varsigma(x)b : \alpha}$$

- But none of the subtyping lemmas!

Subtyping in width

- Object types with more methods are subtypes of object types with less methods
- Assuming the same type for the common methods

$$\frac{E \subseteq D}{[m_d : \tau_d]_{d \in D} \subseteq [m_e : \tau_e]_{e \in E}}$$



Subtyping in width

- Fix definition to accommodate subtyping in width

$$\langle k, \Psi, \{\mathrm{m}_d{=}l_d\}_{d\in D}\rangle \in \alpha = [\mathrm{m}_d : \tau_d]_{d\in D} \;\Leftrightarrow$$
$$\forall d \in D.\, \langle k, \Psi, l\rangle \in \mathrm{ref}\ (\alpha \to \tau_d)$$

- But why does it fail in the first place?
- One reason: an object type contains only those objects which have exactly the methods specified by it, and nothing more
- This is easy to fix:

$$\langle k, \Psi, \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle \in \alpha = [\mathrm{m}_d : \tau_d]_{d\in D} \;\Leftrightarrow$$
$$D \subseteq E \;\wedge\; \forall d \in D.\, \langle k, \Psi, l\rangle \in \mathrm{ref}\ (\alpha \to \tau_d)$$

Subtyping in width

- Unfortunately this is not the only reason

$$[\mathrm{m}_d : \tau_d]_{d \in D} \approx \mu(\alpha).\{\mathrm{m}_d : \mathrm{ref}\ (\alpha \to \tau_d)\}_{d \in D}$$

- Second reason:
  - positions of recursion variable are invariant
  - even without reference position contravariant
  - they should be covariant!

$$\alpha \subseteq \beta \Rightarrow \frac{\dfrac{E \subseteq D \quad \forall d \in D.\ \mathrm{ref}\ (\alpha \to \tau_d) \subseteq \mathrm{ref}\ (\beta \to \tau_d)}{F(\alpha) \subseteq G(\beta)}}{\mu F \subseteq \mu G}$$

$$F(\alpha) = \{\mathrm{m}_d : \mathrm{ref}\ (\alpha \to \tau_d)\}_{d \in D} \qquad G(\beta) = \{\mathrm{m}_e : \mathrm{ref}\ (\beta \to \tau_e)\}_{e \in E}$$

Subtyping in width

- We force covariance for recursion variable using a bounded existential

$$[\mathrm{m}_d : \tau_d]_{d \in D} \approx \mu(\alpha).\exists \alpha' \subseteq \alpha.\{\mathrm{m}_d : \mathrm{ref}\ (\alpha' \to \tau_d)\}_{d \in D}$$

- α' can be viewed as the "true" type of the object

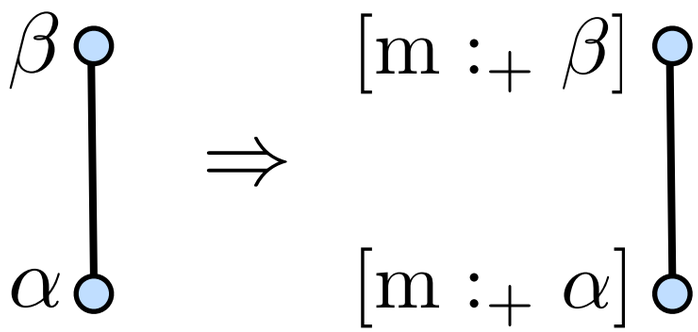
- Similar to some encodings of the functional obj. calculus [Abadi & Cardelli, '96] and [Abadi, Cardelli & Viswanathan, '96]

$$\alpha \subseteq \beta \Rightarrow \frac{\alpha \subseteq \beta \quad \forall \tau \subseteq \alpha.\ \dfrac{D \subseteq E \quad \forall d \in D.\ \mathrm{ref}\ (\tau \to \tau_d) \subseteq \mathrm{ref}\ (\tau \to \tau_d)}{F(\tau) \subseteq G(\tau)}}{\dfrac{}{\exists \alpha' \subseteq \alpha.F(\alpha') \subseteq \exists \beta' \subseteq \beta.G(\beta')}}{\dfrac{}{\mu(\alpha).\exists \alpha' \subseteq \alpha.F(\alpha') \subseteq \mu(\beta).\exists \beta' \subseteq \beta.G(\beta')}}$$

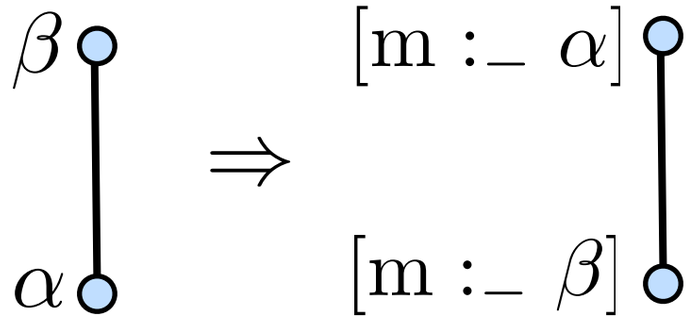
$$F(\alpha) = \{\mathrm{m}_d : \mathrm{ref}\ (\alpha \to \tau_d)\}_{d \in D} \qquad G(\beta) = \{\mathrm{m}_e : \mathrm{ref}\ (\beta \to \tau_e)\}_{e \in E}$$

Subtyping in depth

- Our methods can be both invoked and updated
  - They need to be invariant (no subtyping in depth)
- Still, if we mark methods with their desired variance and restrict invocations and updates accordingly
  - Covariant subtyping for invoke-only methods
  - Contravariant subtyping for update-only methods



covariant (invoke-only)



contravariant (update-only)

- This is what we would like to have :)

Extending reference types

- The usual reference types are invariant

$$\mathrm{ref}_\circ \tau = \{\langle k, \Psi, l\rangle \mid \lfloor \Psi \rfloor_k (l) = \lfloor \tau \rfloor_k\}$$

- The type of the location is precisely known
  - So both reading and writing are safe at type τ
- If we only give a bound on $\Psi(l)$ then only one of these operations is safe at a meaningful type
  - Readable reference type

$$\mathrm{ref}_+ \tau = \{\langle k, \Psi, l\rangle \mid \lfloor \Psi \rfloor_k (l) \subseteq \lfloor \tau \rfloor_k\}$$

    - This is not read-only (it is writable at type $\bot$)!
  - Writable reference types

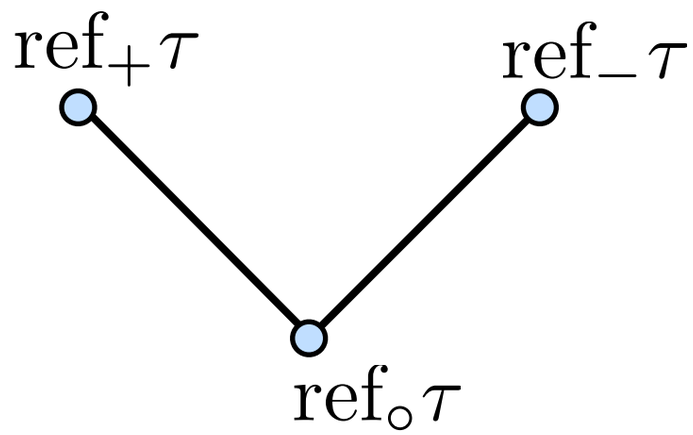
$$\mathrm{ref}_- \tau = \{\langle k, \Psi, l\rangle \mid \lfloor \tau \rfloor_k \subseteq \lfloor \Psi \rfloor_k (l)\}$$

Extending reference types

- Readable reference type is covariant $\dfrac{\alpha \subseteq \beta}{\mathrm{ref}_+\alpha \subseteq \mathrm{ref}_+\beta}$
- Writable reference type is contravariant $\dfrac{\beta \subseteq \alpha}{\mathrm{ref}_-\alpha \subseteq \mathrm{ref}_-\beta}$
- The usual reference types can actually be defined as

$$\mathrm{ref}_\circ\tau = \mathrm{ref}_+\tau \cap \mathrm{ref}_-\tau$$

- So clearly



- [Reynolds, '88] [Pierce & Sangiorgi, '96]

Subtyping in width, depth and more

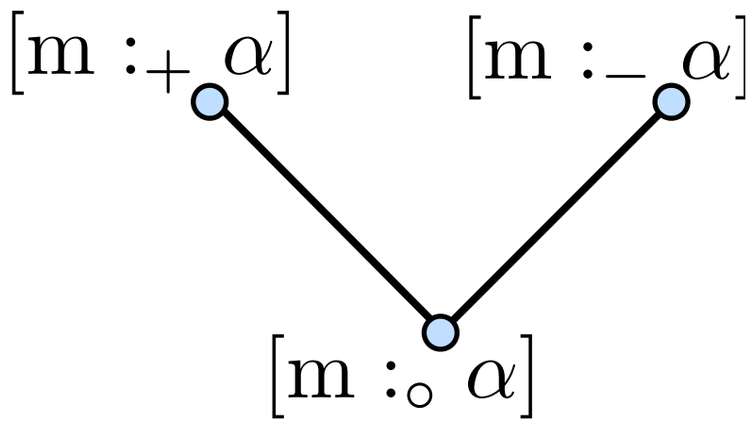
- So we could model objects as

$$[\mathrm{m}_d :_{\nu_d} \tau_d]_{d \in D} \approx \mu(\alpha).\exists \alpha' \subseteq \alpha.\{\mathrm{m}_d : \mathrm{ref}_{\nu_d}(\alpha' \to \tau_d)\}_{d \in D}$$

- This gives us subtyping in width and depth

$$\frac{E \subseteq D \quad \forall e \in E.\, (\nu_e \in \{+, \circ\} \Rightarrow \alpha_e \subseteq \beta_e) \wedge (\nu_e \in \{-, \circ\} \Rightarrow \beta_e \subseteq \alpha_e)}{[\mathrm{m}_d :_{\nu_d} \alpha_d]_{d \in D} \subseteq [\mathrm{m}_e :_{\nu_e} \beta_e]_{e \in E}}$$

- And extra flexibility



Definition of object types?

$$\langle k, \Psi, \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle \in \alpha = [\mathrm{m}_d : \tau_d]_{d\in D} \;\Leftrightarrow\; D \subseteq E$$
$$\wedge\; \exists \alpha' \subseteq \lfloor \alpha \rfloor_k .\; (\forall d \in D.\; \langle k, \Psi, l_d\rangle \in \mathrm{ref}_{\nu_d}(\alpha' \to \tau_d))$$

- But, because α' is kept abstract
  - invocation and cloning are no longer validated

- Invocation $$\frac{\Sigma \models a : \alpha \quad e \in D \quad \nu_e \in \{+, \circ\}}{\Sigma \models a.\mathrm{m}_e : \tau_e}$$
  - we know that $\forall k \geq 0.\; \forall \Psi.\; \forall \sigma :_{k,\Psi} \Sigma.\; \sigma(a) :_{k,\Psi} \alpha$
  - so if $h :_k \Psi \;\wedge\; i < k \;\wedge\; \langle h, \sigma(a)\rangle \to^i \langle h^*, a^*\rangle \not\to$
  - then $\exists \Psi^*.\; (k, \Psi) \sqsupseteq (k - i, \Psi^*) \;\wedge\; h^* :_{k-i} \Psi^*$
  - and $\langle k - i, \Psi^*, a^*\rangle \in \alpha$. So for some $\alpha' \subseteq \lfloor \alpha \rfloor_k$
  - we would need to show that $\langle k - i, \Psi^*, a^*\rangle \in \alpha'$

Fixing the invocation case

$$\langle k, \Psi, \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle \in \alpha = [\mathrm{m}_d : \tau_d]_{d\in D} \;\Leftrightarrow\; D \subseteq E$$
$$\wedge\; \exists \alpha' \subseteq \lfloor \alpha \rfloor_k \;\wedge\; (\forall d \in D.\, \langle k, \Psi, l_d\rangle \in \mathrm{ref}_{\nu_d}(\alpha' \to \tau_d))$$
$$\wedge\; (\forall j < k.\, \forall \Psi^*.\, (k,\Psi) \sqsubseteq (j,\Psi^*) \Rightarrow \langle j, \Psi^*, \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle \in \alpha')$$

- We explicitly enforce that α' contains $\{\mathrm{m}_e{=}l_e\}_{e\in E}$
  - Not surprising, α' is the "true" type of $\{\mathrm{m}_e{=}l_e\}_{e\in E}$

- Invocation
  - we know that $\forall k \geq 0.\, \forall \Psi.\, \forall \sigma :_{k,\Psi} \Sigma.\, \sigma(a) :_{k,\Psi} \alpha$
  - so if $h :_k \Psi \;\wedge\; i < k \;\wedge\; \langle h, \sigma(a)\rangle \to^i \langle h^*, a^*\rangle \not\to$
  - then $\exists \Psi^*.\, (k,\Psi) \sqsubseteq (k-i, \Psi^*) \;\wedge\; h^* :_{k-i} \Psi^*$
  - and $\langle k-i, \Psi^*, a^*\rangle \in \alpha$. So for some $\alpha' \subseteq \lfloor \alpha \rfloor_k$
  - we would need to show that $\langle k-i, \Psi^*, a^*\rangle \in \alpha'$

Fixing the clone case

$$\langle k, \Psi, \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle \in \alpha = [\mathrm{m}_d : \tau_d]_{d\in D} \;\Leftrightarrow\; D \subseteq E$$
$$\wedge\; \exists \alpha' \subseteq \lfloor \alpha \rfloor_k \;\wedge\; (\forall d \in D.\, \langle k, \Psi, l_d\rangle \in \mathrm{ref}_{\nu_d}(\alpha' \to \tau_d))$$
$$\wedge\; (\forall j < k.\, \forall \Psi^*.\, (k,\Psi) \sqsubseteq (j,\Psi^*) \Rightarrow \langle j, \Psi^*, \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle \in \alpha')$$

- We enforce that α' contains not only $\{\mathrm{m}_e{=}l_e\}_{e\in E}$
  - But also all clones of $\{\mathrm{m}_e{=}l_e\}_{e\in E}$

$$\langle k, \Psi, \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle \in \alpha = [\mathrm{m}_d : \tau_d]_{d\in D} \;\Leftrightarrow\; D \subseteq E$$
$$\wedge\; \exists \alpha' \subseteq \lfloor \alpha \rfloor_k \;\wedge\; (\forall d \in D.\, \langle k, \Psi, l_d\rangle \in \mathrm{ref}_{\nu_d}(\alpha' \to \tau_d))$$
$$\wedge\; (\forall j < k.\, \forall \Psi^*.\, \forall \{\mathrm{m}_e{=}l'_e\}_{e\in E} \cdot (k,\Psi) \sqsubseteq (j,\Psi^*)$$
$$\wedge\, (\forall e \in E.\; \lfloor \Psi^* \rfloor_j (l'_e) = \lfloor \Psi \rfloor_j (l_e))$$
$$\Rightarrow \langle j, \Psi^*, \{\mathrm{m}_e{=}l'_e\}_{e\in E}\rangle \in \alpha')$$

Definition of object types

$$\langle k, \Psi, \{\mathrm{m}_e{=}l_e\}_{e\in E}\rangle \in \alpha = [\mathrm{m}_d : \tau_d]_{d\in D} \;\Leftrightarrow\; D \subseteq E$$
$$\wedge\; \exists \alpha' \subseteq \lfloor \alpha \rfloor_k \;\wedge\; (\forall d \in D.\, \langle k, \Psi, l_d\rangle \in \mathrm{ref}_{\nu_d}(\alpha' \to \tau_d))$$
$$\wedge\; (\forall j < k.\, \forall \Psi^*.\, \forall \{\mathrm{m}_e{=}l'_e\}_{e\in E}\,.\, (k,\Psi) \sqsubseteq (j,\Psi^*)$$
$$\wedge\; (\forall e \in E.\; \lfloor \Psi^* \rfloor_j (l'_e) = \lfloor \Psi \rfloor_j (l_e))$$
$$\Rightarrow \langle j, \Psi^*, \{\mathrm{m}_e{=}l'_e\}_{e\in E}\rangle \in \alpha')$$

- Validates all semantic typing and subtyping lemmas
- We use these lemmas to prove soundness of syntactic type system
  - Other than object types:
    - bounded second-order types
    - recursive types

Why do we need models?

- For languages with higher-order store
- Purely syntactic arguments (subject-reduction)
  - Suffice for proving safety of "traditional" type systems
  - But, for languages with higher-order store and more powerful deduction systems (e.g. program logics)
    - Meaning of assertions no longer obvious
      - Assertions should describe code too
- Proving soundness wrt. semantic model:
  - "Derivability implies validity in the model"

Future work

- Prove the soundness of a program logic for the imperative object calculus using step-indexing

Thank you