A Step-indexed Semantics of Imperative Objects

Cătălin Hrițcu and Jan Schwinghammer
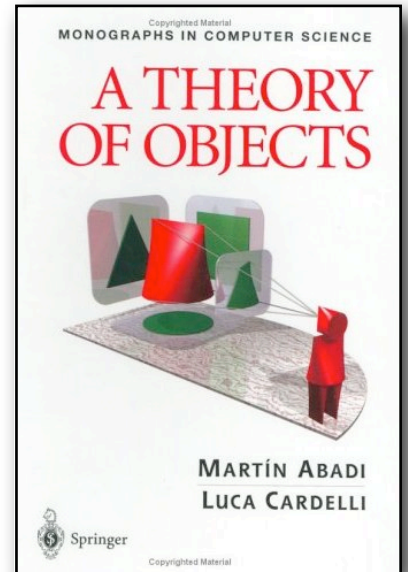
Saarland University, Saarbrücken, Germany

1

Imperative object calculus

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

MONOGRAPHS IN COMPUTER SCIENCE

A THEORY
OF OBJECTS

MARTÍN ABADI
LUCA CARDELLI

Springer

Imperative object calculus

[Abadi and Cardelli, '96]

$$a, b ::= \ x \ | \ [\text{m}_d = \varsigma(x_d)b_d]_{d \in D}$$
$$| \ a.\text{m} \ | \ a.\text{m} := \varsigma(x)b$$
$$| \ \text{clone } a \ | \ \lambda(x)b \ | \ a \ b$$
$$v ::= \{\text{m}_d = l_d\}_{d \in D} \ | \ \lambda(x)b$$

dynamic allocation

heap stores code

recursion involves heap

$$\langle \emptyset, [\text{m} = \varsigma(x)x.\text{m}].\text{m} \rangle \xrightarrow{\text{OBJ}}$$
$$\rightarrow \langle \{l \mapsto \lambda(x)x.\text{m}\}, \{\text{m} = l\}.\text{m} \rangle \xrightarrow{\text{INV}}$$
$$\rightarrow \langle \{l \mapsto \lambda(x)x.\text{m}\}, (\lambda(x)x.\text{m}) \ \{\text{m} = l\} \rangle \xrightarrow{\text{BETA}}$$
$$\rightarrow \langle \{l \mapsto \lambda(x)x.\text{m}\}, \{m = l\}.\text{m} \rangle \xrightarrow{\text{INV}} \dots$$

Imperative object calculus

$$a, b ::= \; x \; | \; [\mathrm{m}_d{=}\varsigma(x_d)b_d]_{d \in D}$$
$$| \; a.\mathrm{m} \; | \; a.\mathrm{m} := \varsigma(x)b$$
$$| \; \mathrm{clone} \; a \; | \; \lambda(x)b \; | \; a \; b$$
$$v ::= \{\mathrm{m}_d{=}l_d\}_{d \in D} \; | \; \lambda(x)b$$
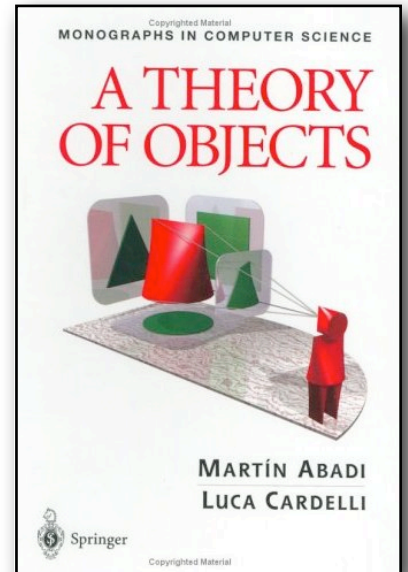
dynamically-allocated,
higher-order store

\+ expressive type system

- Object types and subtyping

- Impredicative second-order types

- Recursive types

2

Hard to find good semantic models

- For domain-theoretic models ...

- Higher-order store

  - Solving recursive domain equations

- + Dynamic allocation - possible-worlds models

  - Recursively defined functor categories over CPOs

- Existing domain-theoretic models
  [Levi, '02] [Reus & Schwinghammer, '06]

  - Despite being complex are not abstract enough

- + Polymorphic values on the heap (impredicative)

  - No domain-theoretic models known, in general!

Types and heap typings

- In a set-theoretic term model of our calculus
  are types just sets of values?

- 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 semantic types

- Model types as sets of pairs?

$$Type = \mathcal{P}(\mathit{HeapTyping} \times \mathit{CVal})$$

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

  - There are no set-theoretic solutions to this!

Step-indexed models

- Alternative to subject-reduction [Appel & Felty, '00]

  - Simpler machine-checkable proofs of type soundness

- Much simpler than the domain-theoretic models

  - Only based on a small-step operational semantics

- Model of types 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

Types and heap typings

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

- We can solve this by a stratified construction
$$Type_{k+1} = \mathcal{P}(j \in [0, k] \times HeapTyping_j \times CVal)$$
$$HeapTyping_j = Loc \rightharpoonup_{fin} Type_j$$

- k-th approximation: $\lfloor \tau \rfloor_k = \{ \langle j, \Psi, v \rangle \in \tau \mid j < k \}$

  - We have that $\lfloor \tau \rfloor_k \in Type_k$
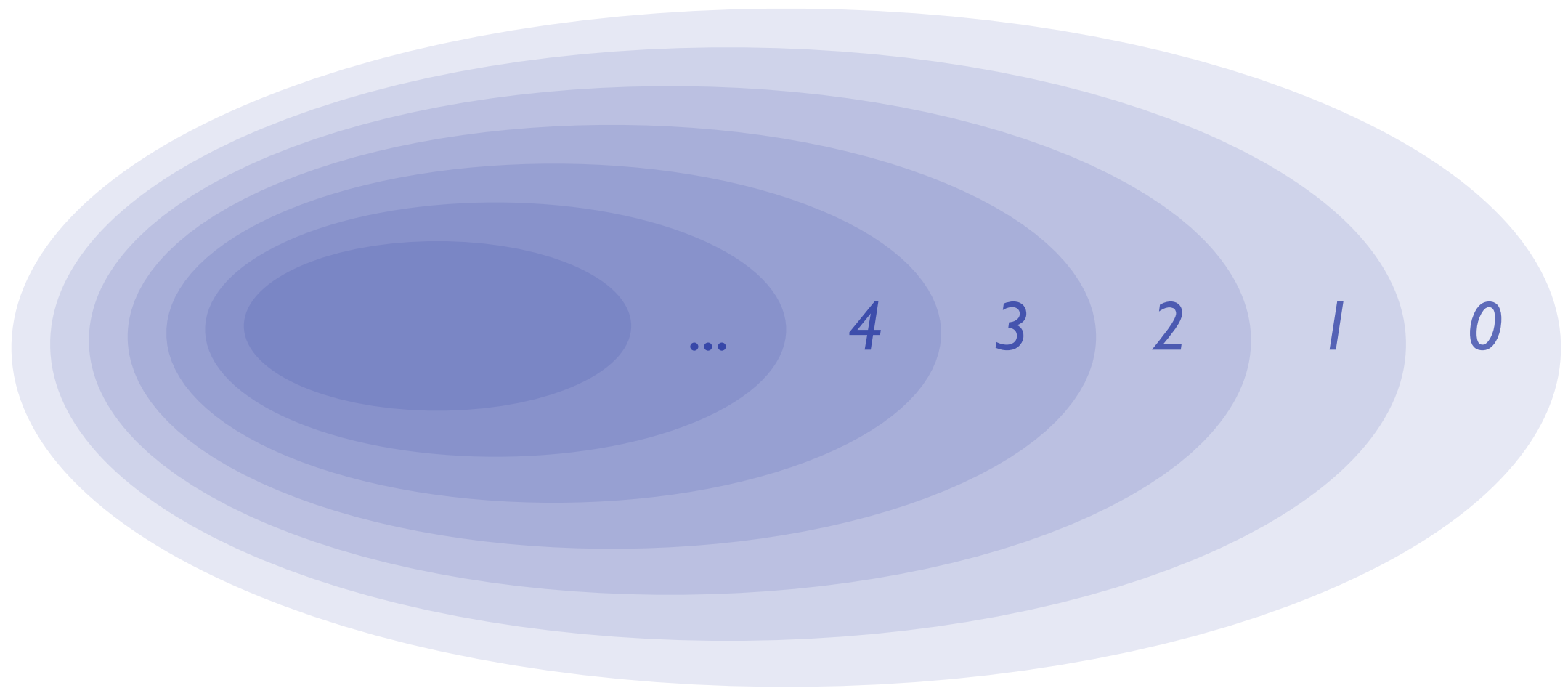
- Stratification invariant:

  - $\lfloor \alpha \rfloor_{k+1}$ 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 executes for at least k steps without getting stuck in every context of type τ, for every $h :_k \Psi$

- Example: $\langle 1, \emptyset, (\lambda x. true) \rangle \in Nat \rightarrow Nat$

  $$\langle 2, \emptyset, (\lambda x. true) \rangle \notin Nat \rightarrow Nat,$$
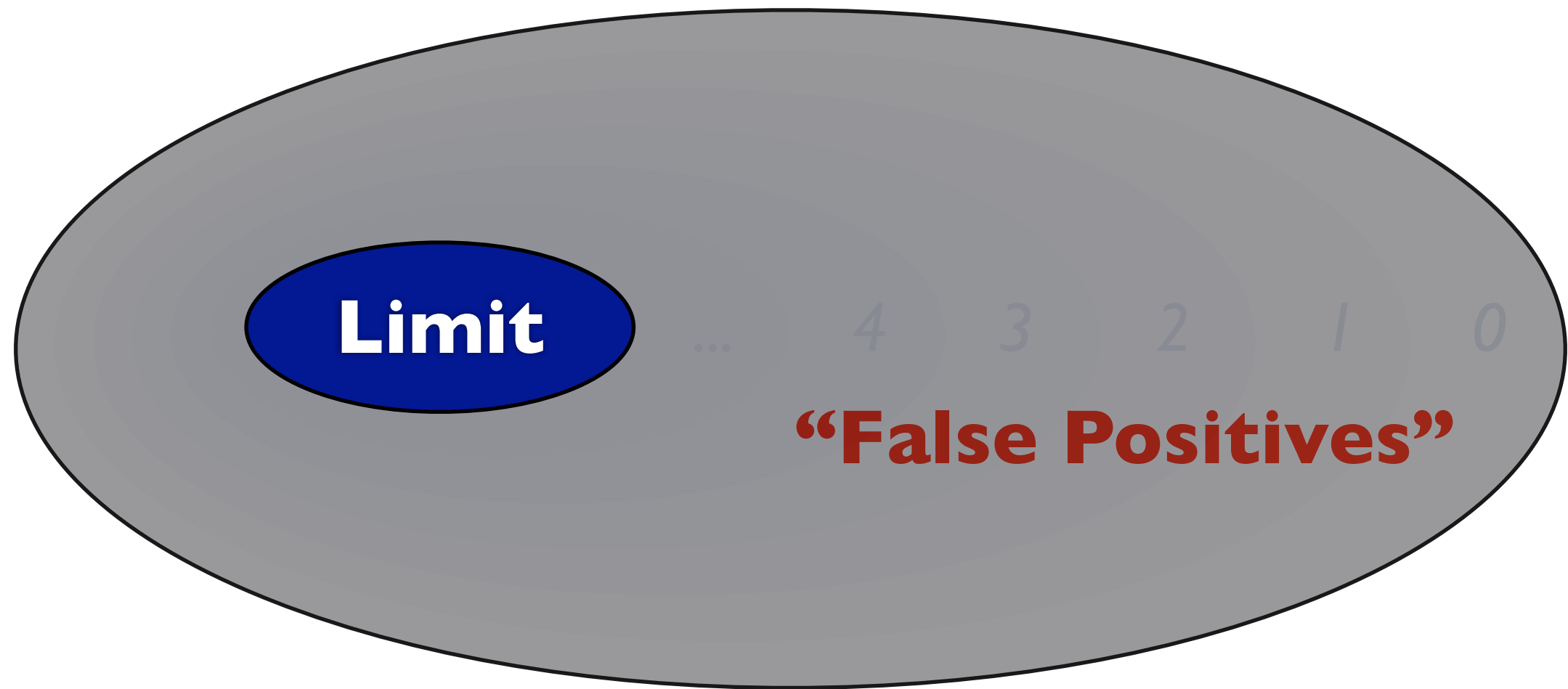
  $$C[\cdot] = ([\cdot] \; 42) + 2$$

Semantic types

- Sequences of increasingly accurate approximations

... 4 3 2 1 0

Semantic types

- Sequences of increasingly accurate approximations

- In the end we are only interested in the limit

- Approximation crucial for well-founded construction

  + Extremely useful when giving recursive definitions of types

State extension

- Heaps evolve during computation

  - Dynamic allocation, no deallocation, weak updates

    ➡ Heap typings can only "grow"

- The precision of our approximation decreases with each reduction step

- State extension relation: $(k, \Psi) \sqsubseteq (j, \Psi')$

- Closure under state extension (Kripke monotonicity)

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

- Semantic types must be closed under state extension

- Possible-worlds model

The type of arbitrary terms

- For a closed term a, $a :_{k,\Psi} \tau$ iff

$$\langle h, a \rangle \rightarrow^j \langle h', b \rangle \nrightarrow, \text{ for any } j < k, \ h :_k \Psi, \ b, \text{ and } h'$$

$$\Rightarrow \langle k - j, \Psi', b \rangle \in \tau, \text{ for some } \Psi' \text{ such that}$$

$$(k, \Psi) \sqsubseteq (k - j, \Psi') \text{ and } h' :_{k-j} \Psi'$$

- Semantic typing judgement

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

  - Typing open terms; not approximative

- This definition directly enforces type safety

  - Still need to prove the soundness of the typing rules

Simple semantic types

- Base types

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

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

- Procedure types

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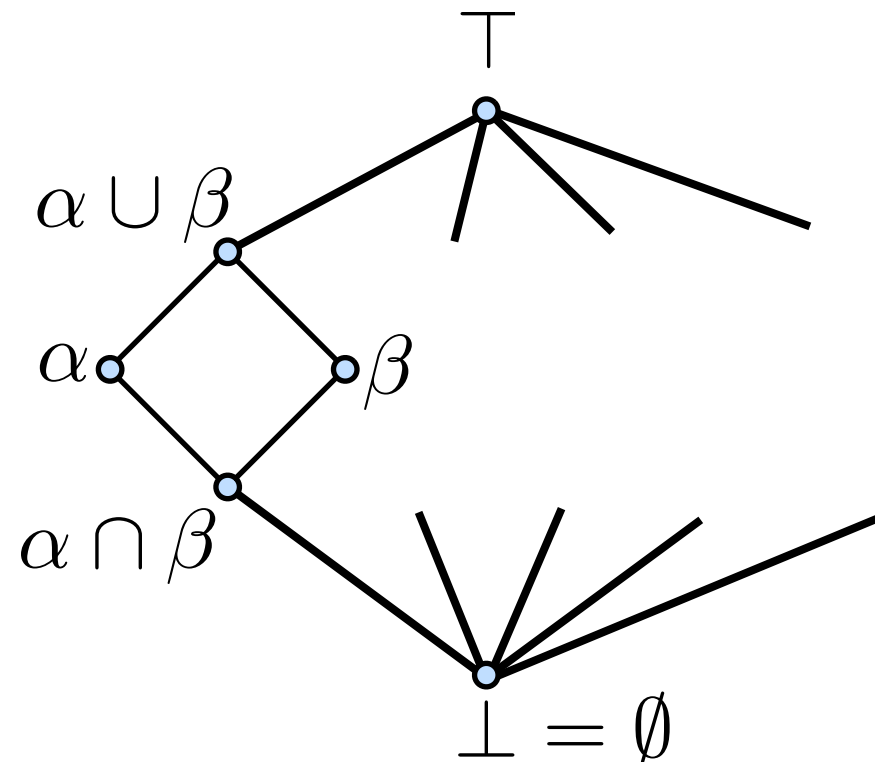
- Reference types

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

Object types and subtyping

Subtyping

- Since types are sets, subtyping is set inclusion

- Subtyping forms a lattice on types

- Simple, but not orthogonal to the other features

  - e.g. non-trivial interaction with object types

Definition of object types

- Methods stored in the heap as procedures and self-application semantics of method invocation suggest

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

- This validates all typing rules for objects

  - Let $\alpha = [m_d : \tau_d]_{d \in D}$

$$(\mathrm{OBJ})\ \frac{\forall d \in D.\ \Sigma[x_d \mapsto \alpha] \models b_d : \tau_d}{\Sigma \models [m_d = \varsigma(x_d)b_d]_{d \in D} : \alpha} \qquad (\mathrm{CLONE})\ \frac{\Sigma \models a : \alpha}{\Sigma \models \mathrm{clone}\ a : \alpha}$$
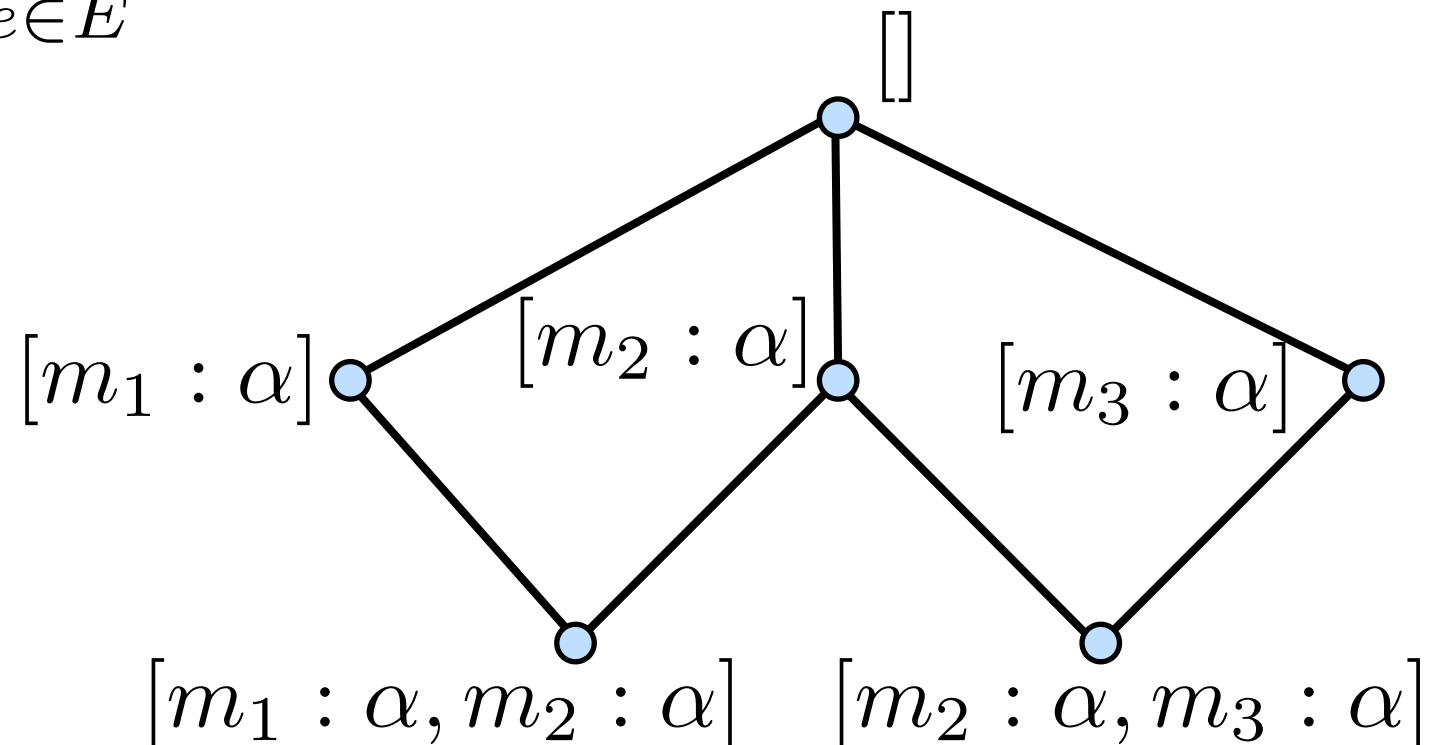
$$(\mathrm{INV})\ \frac{\Sigma \models a : \alpha \quad e \in D}{\Sigma \models a.m_e : \tau_e} \qquad (\mathrm{UPD})\ \frac{\Sigma \models a : \alpha \quad e \in D \quad \Sigma[x \mapsto \alpha] \models b : \tau_e}{\Sigma \models a.m_e := \varsigma(x)b : \alpha}$$

- But none of the subtyping rules!

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

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

- Subtyping in width fails because:

  - positions of recursion variable are invariant

  - even without reference positions contravariant

  - they should be covariant! (see below)

$$\dfrac{\alpha \subseteq \beta \Rightarrow \dfrac{E \subseteq D \quad \forall d \in D.\ \mathrm{ref}\ (\alpha \to \tau_d) \subseteq \mathrm{ref}\ (\beta \to \tau_d)}{\{\mathrm{m}_d : \mathrm{ref}\ (\alpha \to \tau_d)\}_{d \in D} \subseteq \{\mathrm{m}_e : \mathrm{ref}\ (\beta \to \tau_e)\}_{e \in E}}}{\mu(\alpha).\{\mathrm{m}_d : \mathrm{ref}\ (\alpha \to \tau_d)\}_{d \in D} \subseteq \mu(\beta).\{\mathrm{m}_e : \mathrm{ref}\ (\beta \to \tau_e)\}_{e \in E}}$$

Subtyping in width

- We force covariance for recursion variable using a bounded existential

$$\frac{\alpha \subseteq \beta \quad \forall \tau \subseteq \alpha.\ F(\tau) \subseteq G(\tau)}{\exists \alpha' \subseteq \alpha.F(\alpha') \subseteq \exists \beta' \subseteq \beta.G(\beta')}$$

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

- 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]

Subtyping in depth

- Our methods can be both invoked and updated

  - They need to be invariant (o)

- 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 (-)

- Moreover, we would like that $[\mathrm{m} :_+ \alpha]$ $[\mathrm{m} :_- \alpha]$

$$[\mathrm{m} :_\circ \alpha]$$
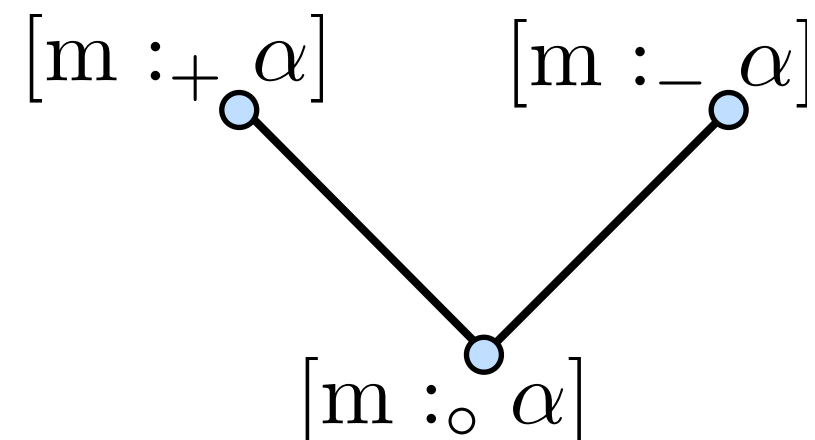
Extending reference types

- However, 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
    $$\mathrm{ref}_+ \tau = \{\langle k, \Psi, l\rangle \mid \lfloor\Psi\rfloor_k (l) \subseteq \lfloor\tau\rfloor_k\}$$
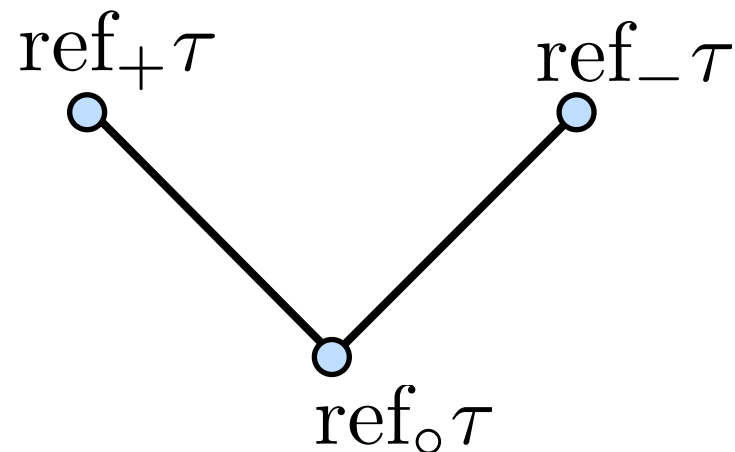
    - This is **not** read-only!

  - 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

$$\mathrm{ref}_+\,\tau \qquad \mathrm{ref}_-\,\tau$$

$$\mathrm{ref}_\circ\,\tau$$

- Not really new [Reynolds, '88] [Pierce & Sangiorgi, '96]

Definition of object types

$$\langle k, \Psi, \{\mathrm{m}_e = l_e\}_{e \in E} \rangle \in \alpha = [\mathrm{m}_d : \tau_d]_{d \in D} \;\Leftrightarrow\; D \subseteq E$$

$$\wedge\; \exists \alpha' \subseteq \lfloor \alpha \rfloor_k.\; (\forall d \in D.\; \langle k, \Psi, l_d \rangle \in \mathrm{ref}_{\nu_d}(\alpha' \to \tau_d))$$

$$\Uparrow$$

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

Definition of object types

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

- But, because α' is kept abstract

  - invocation and cloning rules are no longer validated

- Fixing invocation

  - We need to permit self-application

  - We explicitly enforce that α' contains $\{\mathrm{m}_e = l_e\}_{e \in E}$

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

- Fixing clone

  - We enforce that α' contains all clones of $\{\mathrm{m}_e = l_e\}_{e \in E}$
    i.e. all objects that satisfy the same typing assumptions

Definition of object types

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

- This definition is well-founded (inductive on k)

  - $\lfloor\alpha\rfloor_{k+1}$ is defined in terms of $\lfloor\alpha\rfloor_k$

- Validates all typing and subtyping rules for objects

  - Most interesting proof is for object creation
    (nested induction on naturals)

- Main contribution of the paper

Conclusion

- We extended the step-indexed model of Ahmed et. al. with object types and subtyping, and used it for the imperative object calculus

- Our interpretation of object types uses

  - Recursive types and bounded existentials

  - Readable and writable reference types

- Resulting model

  - is much simpler than a domain-theoretic ones

  - interprets a richer type discipline - impredicative 2nd order types, subtyping in depth wrt. variance annotations

  - However, it only deals with types and type safety

Beyond types

- Purely syntactic argument would have sufficed for proving the safety of our type system (subject-reduction)

  - So why do we need models?

- For more expressive deduction systems, e.g. program logics

  - Meaning of assertions no longer obvious

    - They should describe the code in the (higher-order) heap

  - Subject-reduction limited to whole programs of base type

  - Proving soundness using semantic model (derivability implies validity in the model) gives much stronger guarantees

- **Future work:** Prove the soundness of a program logic for the imperative object calculus using step-indexed model

Backup slides

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!

Semantic typing judgement

- Typing open terms; not approximative

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

- 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)

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

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

Semantic soundness

- We relate the syntactic type expressions to their corresponding semantic types

- We prove that the two are in close correspondence

- Theorem: Soundness of subtyping
  $$\text{If } \Gamma \vdash A \leqslant B \text{ and } \eta \models \Gamma, \text{ then } [\![A]\!]_\eta \subseteq [\![B]\!]_\eta$$

- Theorem: Semantic soundness
  $$\text{If } \Gamma \vdash a : A \text{ and } \eta \models \Gamma, \text{ then } [\![\Gamma]\!]_\eta \models a : [\![A]\!]_\eta$$

- Corollary (Type safety)
  Well-typed terms are safe to evaluate.

More than types (related work)

- Step-indexed PER model for lambda calculus with recursive and impredicative quantified types [Ahmed, '06]

  - Captures exactly observational equivalence, no state

- Soundness of compositional program logic for a very simple stack-based abstract machine [Benton, '05]

- Floyd-Hoare-style framework based on relational parametricity for machine code programs [Benton, '06]

More extensions and future work

- Generalizing reference types ... and object types

$$\mathrm{ref_\circ}\,\tau = \mathrm{ref_+}\,\tau \cap \mathrm{ref_-}\,\tau$$

$$\mathrm{ref}(\alpha, \beta) = \mathrm{ref_-}\,\alpha \cap \mathrm{ref_+}\,\beta$$

- Accommodating self types (easy)

- More realistic languages