International Max Planck Research School
for Computer Science

Step-indexed Semantic Model of Types for the Functional Object Calculus

Catalin Hritcu
Information Security and Cryptography Group

Advisor: *Jan Schwinghammer*
Supervisor: *Prof. Gert Smolka*
Programming Systems Lab

Master's Thesis - Final Talk - Saarland University - June 2007

Master's Thesis

- Proved the soundness of a type system with:
  - object types,
  - subtyping,
  - recursive types,
  - and bounded quantified types ...
- ... with respect to a semantic model.

Soundness of Type Systems

- Most common technique is purely syntactic
  - Subject-reduction [Wright & Felleisen, '94]
    - This is not the only way
- Can be proved wrt. a semantic model
  - Denotational semantics
    - Popular in the '70s and '80s
    - Models usually very involved
  - In my thesis we constructed a much simpler model using "step-indexing"

Outline

Thesis

- Step-indexed Semantic Models — Chapter 1
- Functional Object Calculus — Chapter 2
  - Step-indexed Model — Chapter 3
    - Object Types
    - Subtyping
    - Variance Annotations
  - Syntactic Type System — Chapter 4
    - Semantic Soundness
- Conclusion and Further Work — Chapter 5

IN-DEPTH TALK

Step-indexed Semantic Models

Step-indexed Semantic Models

- Introduced by Appel et al. [Appel & Felty, '00]
- Alternative to subject-reduction
  - More elementary and more modular proofs
  - Easier to check automatically
- Lambda calculus with recursive types [Appel & McAllester, '01]
  - + parametric polymorphism [Ahmed, '04]
  - We extended it with object types and subtyping, and used it for the functional object calculus

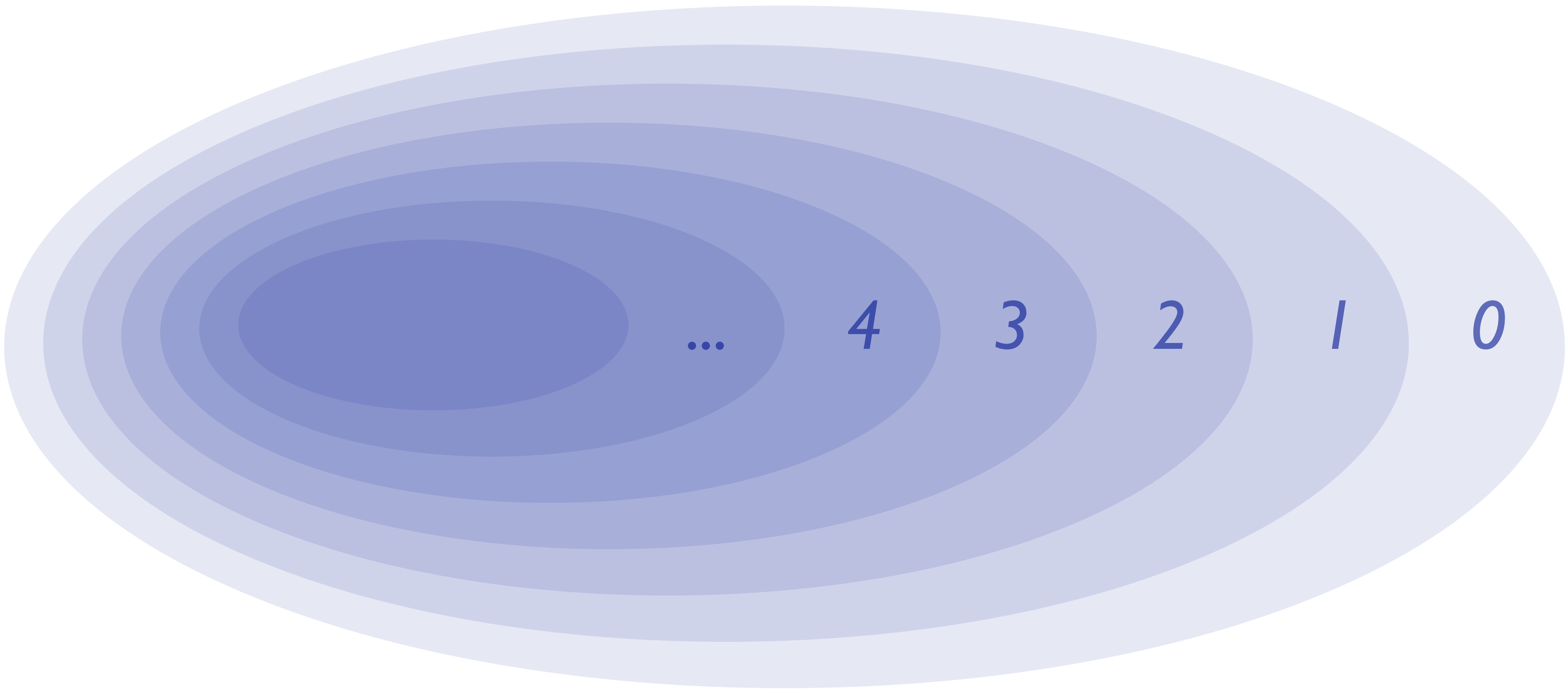
Semantic Types

- Semantic types are sets of indexed values (τ, α, β)
- $\langle k, v \rangle \in \tau$ if one cannot distinguish v from a "real" value of type τ in less than k computation steps
- For example: $\langle 1, \lambda x.\, \mathrm{true} \rangle \in Nat \to Nat$

  $\langle 2, \lambda x.\, \mathrm{true} \rangle \notin Nat \to Nat$
- Equivalently:
  $\langle k, v \rangle \in \tau$ if every context of type τ safely executes for at least k steps when applied to v

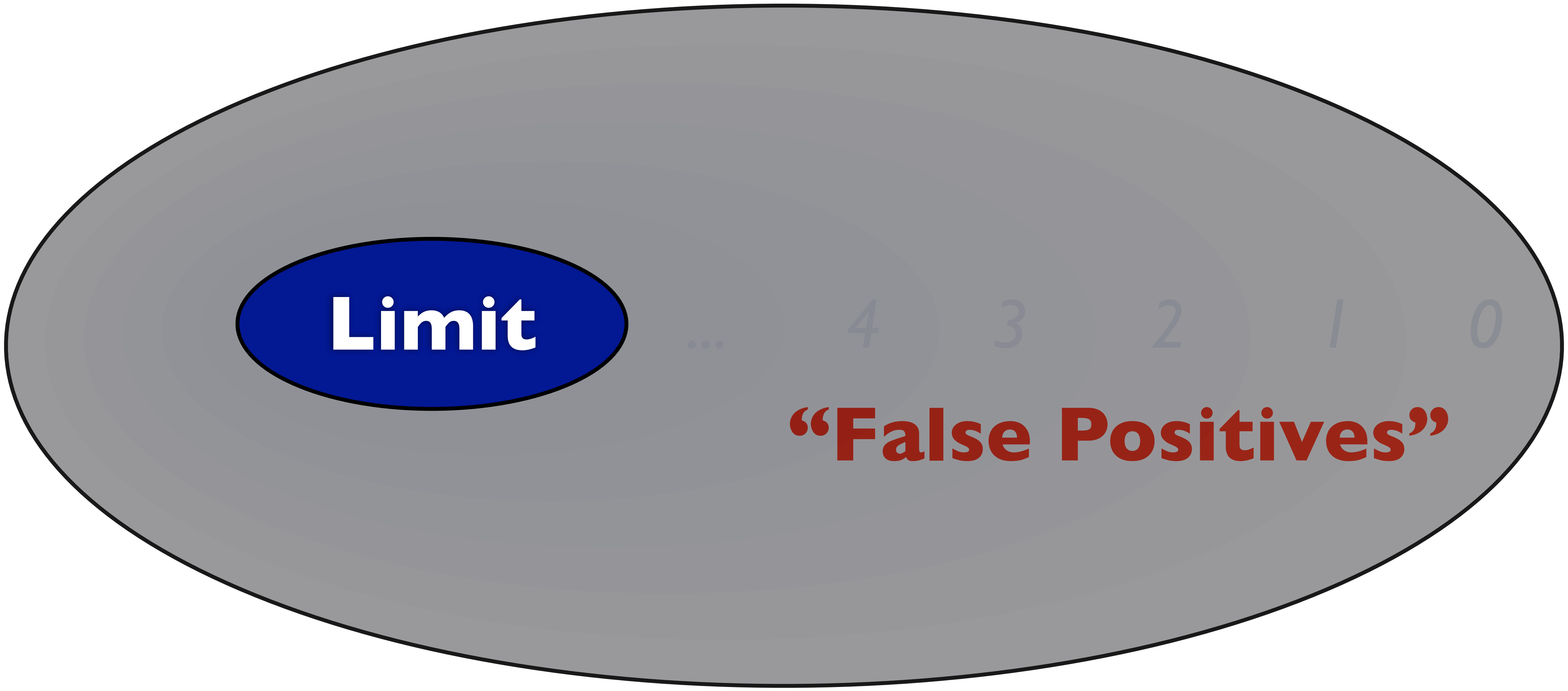
Semantic Types

- Sequences of increasingly accurate approximations



Semantic Types

- Sequences of increasingly accurate approximations



- In the end we are only interested in the limit
- However, approximating is crucial for recursive types

The Type of a Closed Term

- Defined as:

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

- For example:

$$\lambda x.\, \mathrm{true} :_1 Nat \rightarrow Nat$$

$$(\lambda x.\, x)\ (\lambda x.\, \mathrm{true}) :_2 Nat \rightarrow Nat$$

$$(\lambda x.\, x)\ ((\lambda x.\, x)\ \mathrm{true}) :_2 Nat \rightarrow Nat$$

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

Simple Semantic Types

- Base types

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

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

- Function types

$$\alpha \to \beta \triangleq \{\langle k, \lambda x.\, b\rangle \mid \forall j{<}k.\ \forall v.\ \langle j, v\rangle \in \alpha \Rightarrow [x \mapsto v]\,(b) :_j \beta\}$$

Semantic Typing Judgement

- Definition

$$\Sigma \models a : \tau \quad :\Leftrightarrow \quad \forall k \geq 0.\ \forall \sigma :_k \Sigma.\ \sigma(a) :_k \tau$$

- Typing open terms; not approximative
- Semantic type environment $(\Sigma : \mathsf{Var} \rightharpoonup_{fin} \mathsf{Type})$
- Value environment $(\sigma : \mathsf{Var} \rightharpoonup_{fin} \mathsf{CVal})$
- Agreement: $\sigma :_k \Sigma \quad :\Leftrightarrow \quad \forall x \in Dom(\Sigma).\ \sigma(x) :_k \Sigma(x)$
- This definition directly enforces type safety

Semantic Typing Judgement

- Defined independently of any typing rules
- One can prove everything from definitions

$$\emptyset \models \lambda x.\, \lambda y.\, x + y : Nat \to Nat \to Nat$$

$$[x \mapsto Nat]\,[y \mapsto Nat] \models x + y : Nat$$

- Lots of duplication between the proofs
- Solution: semantic typing rules
  - Prove general typing lemmas first
  - Then build type derivations in the usual way

Semantic Typing Rules

$$(\textsc{Var})\ \Sigma \models x : \Sigma(x) \qquad (\textsc{Add})\ \frac{\Sigma \models a : Nat \quad \Sigma \models b : Nat}{\Sigma \models a + b : Nat}$$

$$(\textsc{Lam})\ \frac{\Sigma[x \mapsto \alpha] \models b : \beta}{\Sigma \models \lambda x.\, b : \alpha \to \beta} \qquad (\textsc{App})\ \frac{\Sigma \models a : \beta \to \alpha \quad \Sigma \models b : \beta}{\Sigma \models a\, b : \alpha}$$

- Example of a semantic type derivation:

$$\textstyle (\textsc{Lam})\ \dfrac{(\textsc{Lam})\ \dfrac{(\textsc{Add})\ \dfrac{(\textsc{Var})\ \dfrac{}{[x \mapsto Nat]\,[y \mapsto Nat] \models x : Nat} \qquad \dfrac{}{[x \mapsto Nat]\,[y \mapsto Nat] \models y : Nat}\ (\textsc{Var})}{[x \mapsto Nat]\,[y \mapsto Nat] \models x + y : Nat}}{[x \mapsto Nat] \models \lambda y.\, x + y : Nat \to Nat}}{\emptyset \models \lambda x.\, \lambda y.\, x + y : Nat \to Nat \to Nat}$$

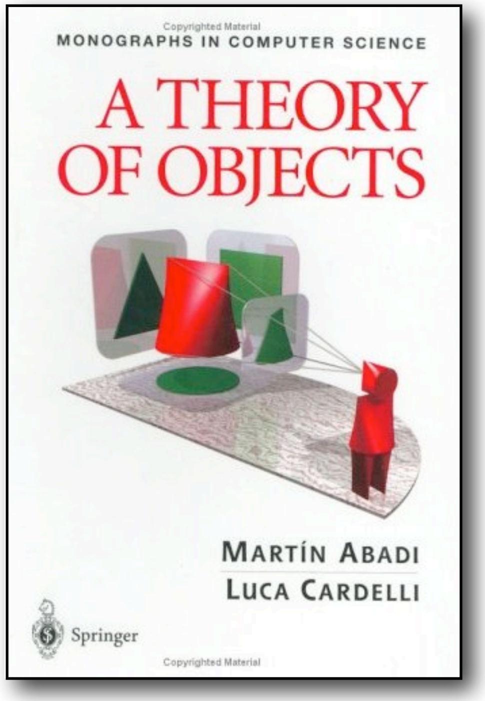
Semantic Typing Rules

- Derive true judgements from true judgements
- Have to be proved sound wrt. semantic model
  - Each rule proved independently: modularity
- Afterwards they can be used to build derivations [Appel & Felty, '00], [Appel & McAllester, '01] ...
  - For more complex type systems
    - Models more complex (type variables)
    - Undecidable type checking
- We only use them for proving the soundness of a syntactic type system (decidable type checking)

The Functional Object Calculus

Functional Object Calculus

- ς-calculus [Abadi & Cardelli, '96]
- Very expressive, yet extremely simple object-oriented programming language
- Only one primitive: objects
- Objects are collections of methods that can be invoked and updated
- Syntax



$$
\begin{array}{lcll}
a, b & ::= & x & \text{(variable)} \\
 & | & [m_d = \varsigma(x_d) b_d]_{d \in D} & \text{(object creation)} \\
 & | & a.m & \text{(method invocation)} \\
 & | & a.m := \varsigma(x) b & \text{(method update)}
\end{array}
$$

Operational Semantics

- Small-step operational semantics
- Let $v ::= [m_d{=}\varsigma(x_d)b_d]_{d \in D}$
  - Method invocation

$$v.m_e \rightarrow [x_e \mapsto v]\,(b_e)$$

  - Method update

$$v.m_e \coloneqq \varsigma(x)b \rightarrow [m_e{=}\varsigma(x)b, m_d{=}\varsigma(x_d)b_d]_{d \in D \setminus \{e\}}$$

  - Evaluation contexts

$$C[\bullet] ::= \bullet \mid C.m \mid C.m \coloneqq \varsigma(x)b$$

Step-indexed Model for the Functional Object Calculus

Step-indexed Model for the Functional Object Calculus

- Extends model by Appel and McAllester
  - Object types
  - Subtyping
  - Bounded quantified types [Ahmed, ‘04]

Object Types

- An object $[m_d{=}\varsigma(x_d)b_d]_{d\in D}$ has type $[m_d : \tau_d]_{d\in D}$ if m_d has type τ_d for all d
- Method types (basically same as function types)

$$\alpha \rightsquigarrow \tau \triangleq \{\langle k, \varsigma(x)b\rangle \mid \forall j{<}k.\ \forall v.\ \langle j, v\rangle \in \alpha \Rightarrow [x \mapsto v]\,(b) :_j \tau\}$$

- The simplest definition of object types

$$[m_d : \tau_d]_{d\in D} \triangleq \{\langle k, [m_d{=}\varsigma(x_d)b_d]_{d\in D}\rangle \mid \forall d{\in}D,\\ \langle k, \varsigma(x_d)b_d\rangle{\in} [m_d : \tau_d]_{d\in D} \rightsquigarrow \tau_d\}$$

- This definition is well-founded (indexing crucial)

Object Types

- This simple definition validates the rules for object creation, method invocation and update
- Let $\alpha \equiv [m_d : \tau_d]_{d \in D}$

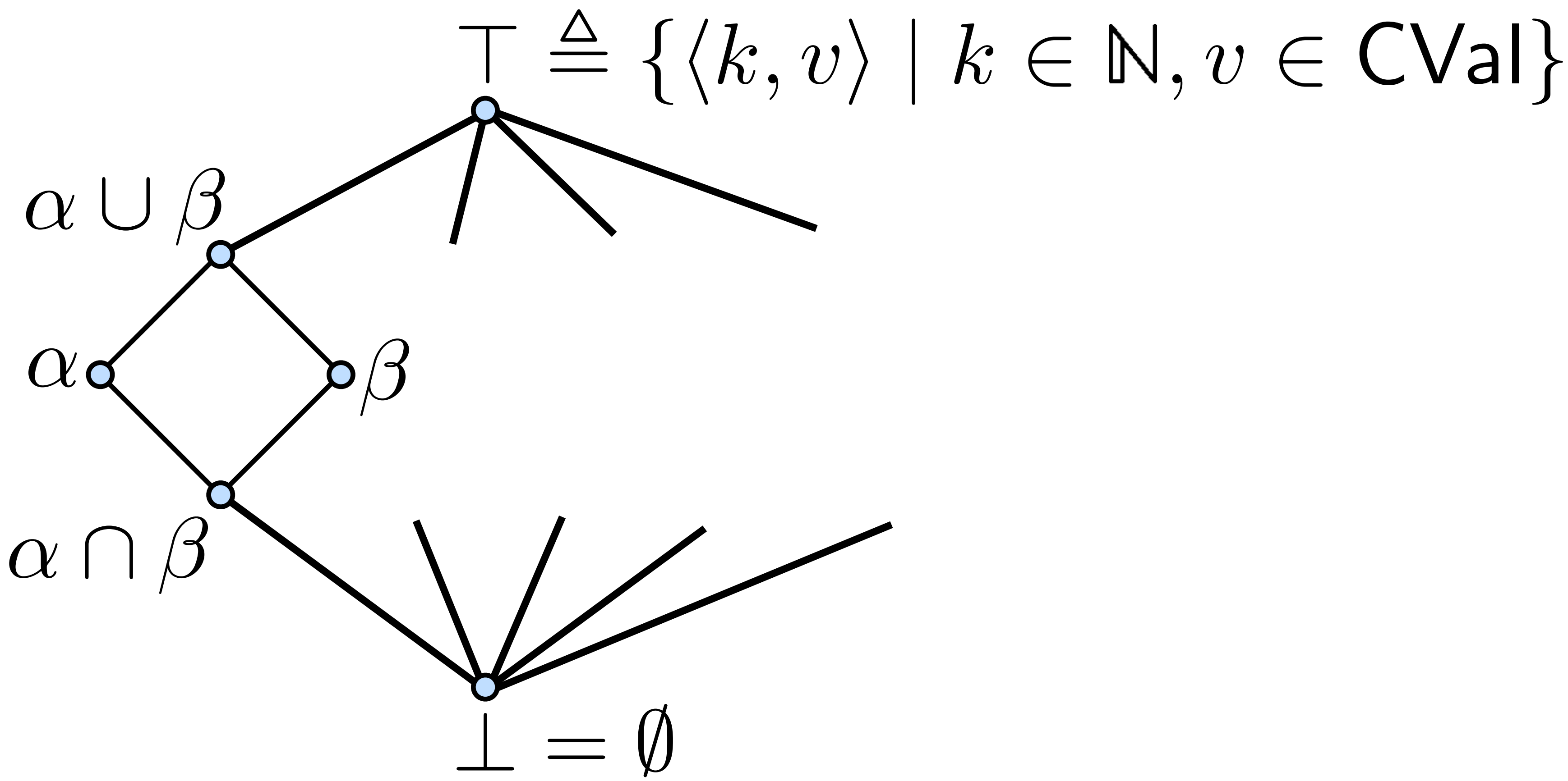
$$(\textsc{Obj})\ \frac{\forall d \in D.\ \Sigma[x_d \mapsto \alpha] \models b_d : \tau_d}{\Sigma \models [m_d{=}\varsigma(x_d)b_d]_{d \in D} : \alpha} \qquad (\textsc{Inv})\ \frac{\Sigma \models a : \alpha \quad e \in D}{\Sigma \models a.m_e : \tau_e}$$

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

- But not the one for subtyping (we will fix this!)

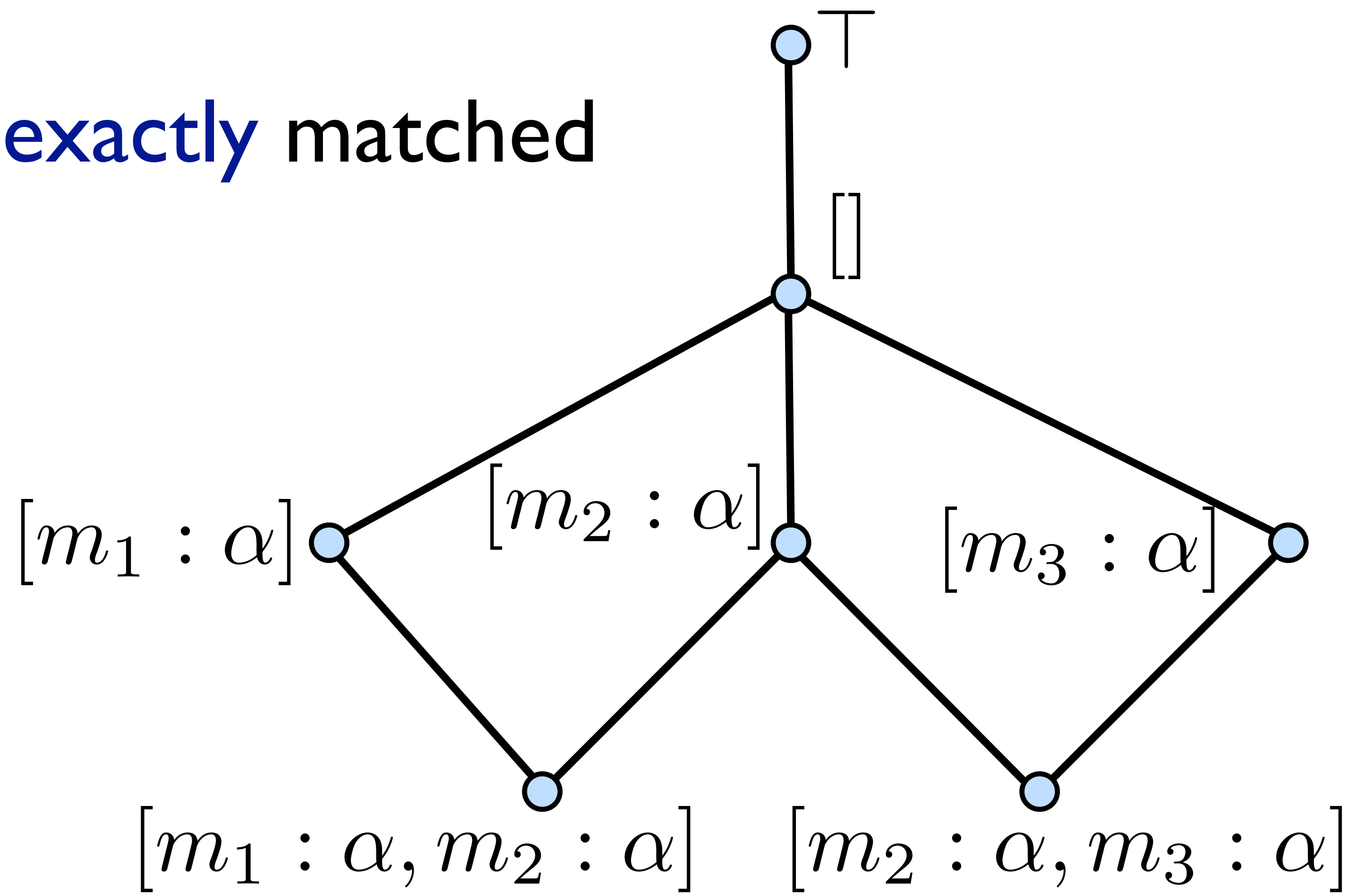
Subtyping

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



Subtyping Object Types

- Subtyping in width $$\frac{E \subseteq D}{[m_d : \tau_d]_{d \in D} \subseteq [m_e : \tau_e]_{e \in E}}$$
  - Object types with more methods are subtypes of object types with less
  - Method types are exactly matched



Subtyping in Width

- Fix definition to accommodate subtyping in width

$$[m_d : \tau_d]_{d \in D} \triangleq \{\langle k, [m_d {=} \varsigma(x_d) b_d]_{d \in D} \rangle \mid \forall d {\in} D,$$
$$\langle k, \varsigma(x_d) b_d \rangle {\in} [m_d : \tau_d]_{d \in D} \rightsquigarrow \tau_d\}$$

- 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 not more
- Easy fix:

$$[m_d : \tau_d]_{d \in D} \triangleq \{\langle k, [m_e {=} \varsigma(x_e) b_e]_{e \in E} \rangle \mid D \subseteq E, \forall d {\in} D,$$
$$\langle k, \varsigma(x_d) b_d \rangle \in [m_d : \tau_d]_{d \in D} \rightsquigarrow \tau_d\}$$

Subtyping in Width

- Unfortunately this is not the only reason

$$[m_d : \tau_d]_{d\in D} \triangleq \{\langle k, [m_e{=}\varsigma(x_e)b_e]_{e\in E}\rangle \mid D \subseteq E, \forall d{\in}D, \langle k, \varsigma(x_d)b_d\rangle \in [m_d : \tau_d]_{d\in D} \rightsquigarrow \tau_d\}$$

- Second reason: highlighted position is contravariant
- Attempt to circumvent this
  - "unroll" the definition of method types

$$\alpha \rightsquigarrow \tau \triangleq \{\langle k, \varsigma(x)b\rangle \mid \forall j{<}k.\ \forall v.\ \langle j, v\rangle \in \alpha \Rightarrow [x \mapsto v]\,(b) :_j \tau\}$$

  - only require methods to work with the current object as the self argument

$$[m_d : \tau_d]_{d\in D} \triangleq \{\langle k, [m_e{=}\varsigma(x_e)b_e]_{e\in E}\rangle \mid D \subseteq E, \forall d{\in}D, \forall j{<}k.\ \left(\left[x_d \mapsto [m_e{=}\varsigma(x_e)b_e]_{e\in E}\right](b_d) :_j \tau_d\right)\}$$

Subtyping in Width

- This gives us subtyping in width

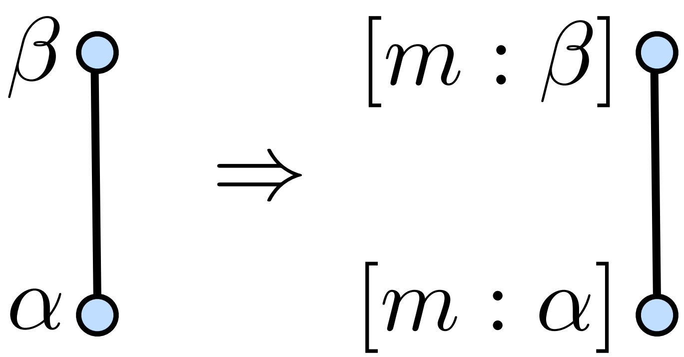
$$[m_d : \tau_d]_{d \in D} \triangleq \{\langle k, [m_e{=}\varsigma(x_e)b_e]_{e \in E}\rangle \mid D \subseteq E, \forall d{\in}D,$$
$$\forall j{<}k.\ (\left[x_d \mapsto [m_e{=}\varsigma(x_e)b_e]_{e \in E}\right](b_d) :_j \tau_d\}$$

- But it no longer validates the update rule
- We add an extra condition that fixes this last bug
- Let $\alpha \equiv [m_d : \tau_d]_{d \in D}$

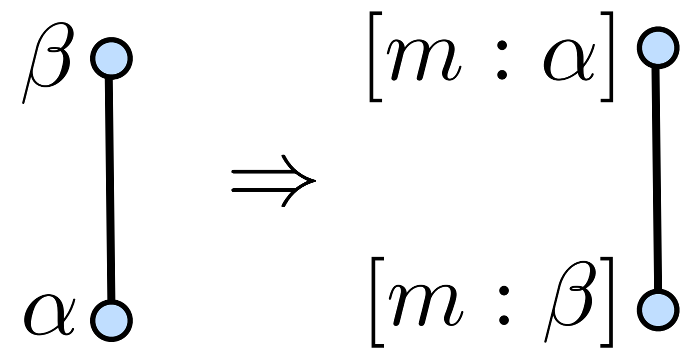
$$\alpha \triangleq \{\langle k, [m_e{=}\varsigma(x_e)b_e]_{e \in E}\rangle \mid D \subseteq E, \forall d{\in}D.$$
$$\forall j{<}k.\ (\left[x_d \mapsto [m_e{=}\varsigma(x_e)b_e]_{e \in E}\right](b_d) :_j \tau_d$$
$$\wedge\ \forall \varsigma(x)b.\ \langle j, \varsigma(x)b\rangle \in \alpha \rightsquigarrow \tau_d$$
$$\Rightarrow \langle j, [m_d{=}\varsigma(x)b, m_e{=}\varsigma(x_e)b_e]_{e \in E \setminus \{d\}}\rangle \in \alpha)\}$$

Subtyping in Depth

- Comes in two flavours



covariant (read-only)



contravariant (write-only)

- Our usual methods can be both invoked and updated
    - They need to be invariant (no subtyping in depth)
- Still, if we restrict invocations and updates
    - Covariant subtyping for read-only methods
    - Contravariant subtyping for write-only methods

Variance Annotations

- Extend object types by annotating each method
    - Covariant (+), contravariant (-) or invariant (0)
    - Restrict reads and writes accordingly
- Adapt the definition of object types (easy)
    - Let $\alpha \equiv [m_d : \tau_d]_{d \in D}$

$$
\begin{aligned}
\alpha \triangleq \{\langle k, [m_e{=}\varsigma(x_e)b_e]_{e\in E}\rangle \mid D \subseteq E, \forall d{\in}D.&\\
\forall j{<}k.\ \big(\big[x_d \mapsto [m_e{=}\varsigma(x_e)b_e]_{e\in E}\big]\,(b_d) :_j \tau_d &\quad \textbf{invocation}\\
\wedge\ \forall\varsigma(x)b.\ \langle j, \varsigma(x)b\rangle \in \alpha \rightsquigarrow \tau_d &\quad \textbf{update}\\
\Rightarrow \langle j, [m_d{=}\varsigma(x)b, m_e{=}\varsigma(x_e)b_e]_{e\in E\setminus\{d\}}\rangle \in \alpha\big)\}&
\end{aligned}
$$

Variance Annotations

- Extend object types by annotating each method
  - Covariant (+), contravariant (-) or invariant (0)
  - Restrict reads and writes accordingly
- Adapt the definition of object types (easy)
  - Let $\alpha \equiv [m_d :_{\nu_d} \tau_d]_{d \in D}$

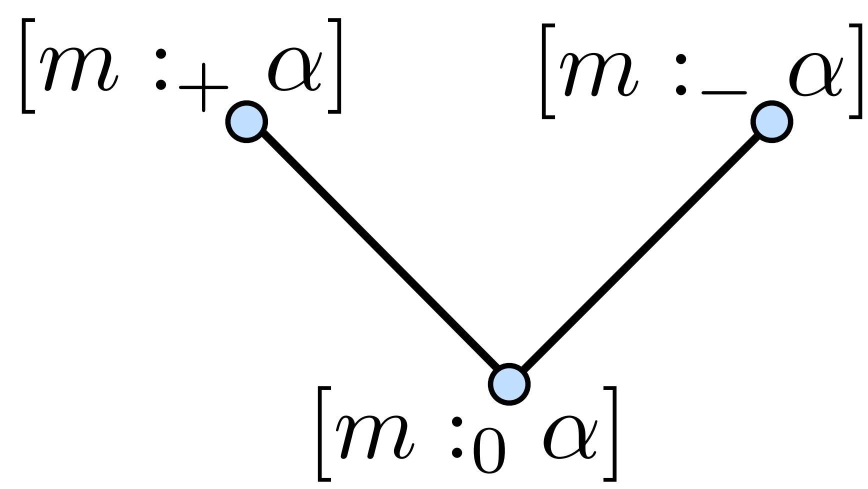
$$
\begin{aligned}
\alpha \triangleq \{\langle k, [m_e{=}\varsigma(x_e)b_e]_{e\in E}\rangle \mid\ & D \subseteq E, \forall d{\in}D. \\
& \forall j{<}k.\ ((\nu_d \in \{+,0\} \Rightarrow \left[x_d \mapsto [m_d :_{\nu_d} \tau_d]_{d\in D}\right](b_d) :_j \tau_d) \\
& \quad \wedge\ (\nu_d \in \{-,0\} \Rightarrow \forall \varsigma(x)b.\ \langle j, \varsigma(x)b\rangle \in \alpha \rightsquigarrow \tau_d \\
& \qquad \Rightarrow \langle j, [m_d{=}\varsigma(x)b, m_e{=}\varsigma(x_e)b_e]_{e\in E\setminus\{d\}}\rangle \in \alpha))\}
\end{aligned}
$$

Subtyping in Width and Depth

- This gives us subtyping in width and depth

$$\frac{E \subseteq D \quad \forall e \in E.\ (\nu_e \in \{+, 0\} \Rightarrow \alpha_e \subseteq \beta_e) \\ \wedge (\nu_e \in \{-, 0\} \Rightarrow \beta_e \subseteq \alpha_e)}{[m_d :_{\nu_d} \alpha_d]_{d \in D} \subseteq [m_e :_{\nu_e} \beta_e]_{e \in E}}$$

- And extra flexibility



  - This allows us to treat external accesses differently from accesses through self

Syntactic Type System

Syntactic Type System

- "Semantic type system" is sound but undecidable
- We introduce a syntactic type system (standard)
  - Prove its soundness wrt. the semantic model
- For example (Amber rule)

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

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

Semantic Soundness

- We relate the syntactic type expressions to their corresponding semantic types
- We prove that the two are in close correspondence
- Soundness of subtyping

  If $\Gamma \vdash A \leqslant B$ and $\eta \models \Gamma$, then $[\![A]\!]_\eta \subseteq [\![B]\!]_\eta$
- Semantic soundness

  If $\Gamma \vdash a : A$ and $\eta \models \Gamma$, then $[\![\Gamma]\!]_\eta \models E(a) : [\![A]\!]_\eta$.
- Corollary (Type Safety)

  Well-typed terms evaluate safely once erased.

Conclusion and Further Work

Conclusion

- Constructed step-indexed semantic model of types for the functional object calculus
- Used it to prove the soundness of an expressive syntactic type system
- Contributions to the step-indexing method
    - Object types
    - Subtyping
    - Bounded quantified types
    - Relating model to a syntactic type system

Further Work

- Step-indexed model of types for the imperative object calculus
  - The original goal of my thesis
  - Basically done
  - We will try to publish it separately
- Program logic for the imperative object calculus
  - Our original long-term goal
  - One small step done
    - Step-indexed model for λ-calculus with dependent products and sums

References

Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5): 657-683, September 2001.

Amal J. Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.

(and many more)

Thank You