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 ...
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

• Step-indexed Semantic Models
• Functional Object Calculus
  • Step-indexed Model
    • Object Types
    • Subtyping
    • Variance Annotations
  • Syntactic Type System
    • Semantic Soundness
• Conclusion and Further Work
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 \((\tau, \alpha, \beta)\)

• \(\langle k, v \rangle \in \tau\) if one cannot distinguish \(v\) from a “real” value of type \(\tau\) in less than \(k\) computation steps

• For example: \(\langle 1, \lambda x. \text{true} \rangle \in \text{Nat} \rightarrow \text{Nat}\)
  \(\langle 2, \lambda x. \text{true} \rangle \not\in \text{Nat} \rightarrow \text{Nat}\)

• Equivalently:
  \(\langle k, v \rangle \in \tau\) if every context of type \(\tau\) 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:

\[ a :_k \tau \iff \forall j < k. (a \rightarrow^j b \land b \not\leftrightarrow) \Rightarrow \langle k - j, b \rangle \in \tau \]

• For example:

\[ \lambda x. \text{true} :_1 \text{Nat} \rightarrow \text{Nat} \]
\[ (\lambda x. x) (\lambda x. \text{true}) :_2 \text{Nat} \rightarrow \text{Nat} \]
\[ (\lambda x. x) ((\lambda x. x) \text{true}) :_2 \text{Nat} \rightarrow \text{Nat} \]

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

- **Base types**
  \[ \text{Bool} \triangleq \{ \langle k, v \rangle \mid k \in \mathbb{N}, v \in \{\text{true}, \text{false}\} \} \]
  \[ \text{Nat} \triangleq \{ \langle k, n \rangle \mid k, n \in \mathbb{N} \} \]

- **Function types**
  \[ \alpha \rightarrow \beta \triangleq \{ \langle k, \lambda x. b \rangle \mid \forall v. \langle k - 1, v \rangle \in \alpha \Rightarrow [x \mapsto v] (b) : k - 1 \beta \} \]
Simple Semantic Types

- **Base types**
  
  $\text{Bool} \triangleq \{ \langle k, v \rangle \mid k \in \mathbb{N}, v \in \{ \text{true}, \text{false} \} \}$

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

- **Function types**
  
  $\alpha \rightarrow \beta \triangleq \{ \langle k, \lambda x. b \rangle \mid \forall j < k. \forall v. \langle j, v \rangle \in \alpha \Rightarrow [x \mapsto v] (b) : j \beta \}$
Semantic Typing Judgement

- **Definition**
  \[ \Sigma \models a : \tau \iff \forall k \geq 0. \forall \sigma : k \Sigma. \sigma(a) : k \tau \]

- **Typing open terms; not approximative**

- **Semantic type environment** \((\Sigma : \text{Var} \rightarrow_{\text{fin}} \text{Type})\)

- **Value environment** \((\sigma : \text{Var} \rightarrow_{\text{fin}} \text{CVal})\)

- **Agreement:** \(\sigma : k \Sigma \iff \forall x \in \text{Dom}(\Sigma). \sigma(x) : k \Sigma(x)\)

- **This definition directly enforces type safety**
Semantic Typing Judgement

- Defined independently of any typing rules
- One can prove everything from definitions
  \[
  \emptyset \models \lambda x. \lambda y. x + y : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}
  \]
  \[
  [x \mapsto \text{Nat}] [y \mapsto \text{Nat}] \models x + y : \text{Nat}
  \]
- 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

\[(\text{VAR})\quad \Sigma \models x : \Sigma(x)\quad (\text{ADD})\quad \frac{\Sigma \models b : \text{Nat}}{\Sigma \models x + y : \text{Nat}}\]

\[(\text{LAM})\quad \frac{\Sigma[x \mapsto \alpha] \models b : \beta}{\Sigma \models \lambda x. b : \alpha \rightarrow \beta}\quad (\text{APP})\quad \frac{\Sigma \models a : \beta \rightarrow \alpha \quad \Sigma \models b : \beta}{\Sigma \models a \ b : \alpha}\]

- Example of a semantic type derivation:

\[(\text{VAR})\quad [x \mapsto \text{Nat}] \models y \mapsto \text{Nat} \models x : \text{Nat} \quad [x \mapsto \text{Nat}] \models y \mapsto \text{Nat} \models y : \text{Nat}\]

\[(\text{ADD})\quad \frac{[x \mapsto \text{Nat}] \models y \mapsto \text{Nat} \models x + y : \text{Nat}}{[x \mapsto \text{Nat}] \models \lambda y. x + y : \text{Nat} \rightarrow \text{Nat}}\]

\[(\text{LAM})\quad \frac{[x \mapsto \text{Nat}] \models \lambda y. x + y : \text{Nat} \rightarrow \text{Nat}}{\emptyset \models \lambda x. \lambda y. x + y : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}}\]
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

• ζ-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

\[
a, b ::= x \quad \text{(variable)}
\mid [m_d = \varsigma(x_d)b_d]_{d \in D} \quad \text{(object creation)}
\mid a.m \quad \text{(method invocation)}
\mid a.m := \varsigma(x)b \quad \text{(method update)}
\]
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
  \[ v.m_e ::= \varsigma(x)b \rightarrow [m_e = \varsigma(x)b, m_d = \varsigma(x_d)b_d]_{d \in D\setminus\{e\}} \]
- Evaluation contexts
  \[ C[\bullet] ::= \bullet \mid C.m \mid C.m ::= \varsigma(x)b \]
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 \(\tau_d\) for all \(d\).

• Method types (basically same as function types)

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

• The simplest definition of object types

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

• This definition is well-founded (indexing crucial)
Object Types

- This simple definition validates the rules for object creation, method invocation and update

- Let \( \alpha \equiv [m_d : \tau_d]_{d \in D} \)

\[
\begin{align*}
\text{(OBJ)} & & \forall d \in D. \; \Sigma[x_d \mapsto \alpha] \models b_d : \tau_d \\
& & \Sigma \models [m_d = \varsigma(x_d) b_d]_{d \in D} : \alpha
\end{align*}
\]

\[
\begin{align*}
\text{(INV)} & & \Sigma \models a : \alpha \quad e \in D \\
& & \Sigma \models a.m_e : \tau_e
\end{align*}
\]

\[
\begin{align*}
\text{(UPD)} & & \Sigma \models a : \alpha \quad e \in D \\
& & \Sigma[x \mapsto \alpha] \models b : \tau_e \\
& & \Sigma \models a.m_e := \varsigma(x) b : \alpha
\end{align*}
\]

- 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

\[ \top \triangleq \{ \langle k, v \rangle \mid k \in \mathbb{N}, v \in \text{CVal} \} \]

\[ \alpha \cup \beta \]

\[ \alpha \cap \beta \]

\[ \bot = \emptyset \]
Subtyping Object Types

- Subtyping in width

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

- Object types with more methods are subtypes of object types with less

- Method types are exactly matched

\[ [m_1 : \alpha] \]
\[ [m_2 : \alpha] \]
\[ [m_3 : \alpha] \]
\[ [m_1 : \alpha, m_2 : \alpha] \]
\[ [m_2 : \alpha, m_3 : \alpha] \]
Subtyping in Width
Subtyping in Width

• Fix definition to accommodate subtyping in width

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

• Fix definition to accommodate subtyping in width

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

• But why does it fail in the first place?
Subtyping in Width

• Fix definition to accommodate subtyping in width

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

• 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
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
Subtyping in Width

- Unfortunately this is not the only reason

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

- Unfortunately this is not the only reason

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

- Second reason: highlighted position is contravariant
Subtyping in Width

• Unfortunately this is not the only reason

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

• Second reason: highlighted position is contravariant

• Attempt to circumvent this

  • “unroll” the definition of method types

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

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

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

• This gives us subtyping in width

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

• This gives us subtyping in width

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

• But it no longer validates the update rule
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. ( [x_d \mapsto [m_e = \varsigma(x_e)b_e]_{e \in E}] (b_d) : j \tau_d) \}

• But it no longer validates the update rule

• We add an extra condition that fixes this last bug

• Let \( \alpha \equiv [m_d : \tau_d]_{d \in D} \)

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

• Comes in two flavours

\[ \begin{align*}
\alpha \quad [m : \alpha] & \quad \Rightarrow \quad \beta \quad [m : \beta] \\
\beta \quad [m : \beta] & \quad \Rightarrow \quad \alpha \quad [m : \alpha]
\end{align*} \]

covariant (read-only)  \hspace{2cm}  contravariant (write-only)
Subtyping in Depth

• Comes in two flavours

\[ \beta \quad [m : \beta] \quad \Rightarrow \quad \alpha \quad [m : \alpha] \]

covariant (read-only)

\[ \beta \quad [m : \alpha] \quad \Rightarrow \quad \beta \quad [m : \beta] \]

contravariant (write-only)

• Our usual methods can be both invoked and updated

• They need to be invariant (no subtyping in depth)
Subtyping in Depth

- Comes in two flavours

\[
\begin{align*}
\alpha & \quad [m : \alpha] \\
\beta & \quad [m : \beta] \\
\Rightarrow & \quad \Rightarrow \\
\beta & \quad [m : \alpha] \\
\alpha & \quad [m : \beta]
\end{align*}
\]

covariant (read-only) \quad 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
Variance Annotations

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

$$
\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
\land \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 \} \]
$$
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 \( \alpha \equiv [m_d : \nu_d \tau_d]_{d \in D} \)

\[
\alpha \triangleq \{ \langle k, \lfloor m_e=\varphi(x_e)b_e \rfloor_{e \in E} \rangle \mid D \subseteq E, \forall d \in D. \forall j<k. \left( (\nu_d \in \{+, 0\} \Rightarrow [x_d \mapsto [m_d : \nu_d \tau_d]_{d \in D}] (b_d) : j \tau_d \right) \right.
\]

\[
\left. \land (\nu_d \in \{-, 0\} \Rightarrow \forall \varphi(x)b. \langle j, \varphi(x)b \rangle \in \alpha \rightsquigarrow \tau_d \right) \Rightarrow \langle j, \lfloor m_d=\varphi(x)b, m_e=\varphi(x_e)b_e \rfloor_{e \in E \setminus \{d\}} \rangle \in \alpha \} \} \}
Subtyping in Width and Depth
Subtyping in Width and Depth

- This gives us subtyping in width and depth

\[ E \subseteq D \quad \forall e \in E. (\nu_e \in \{+, 0\} \Rightarrow \alpha_e \subseteq \beta_e) \]
\[ \land (\nu_e \in \{-, 0\} \Rightarrow \beta_e \subseteq \alpha_e) \]

\[ [m_d : \nu_d \alpha_d]_{d \in D} \subseteq [m_e : \nu_e \beta_e]_{e \in E} \]
Subtyping in Width and Depth

• This gives us subtyping in width and depth

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

• And extra flexibility

\[ [m : _{\pm} \alpha] \]
\[ [m : _{-} \alpha] \]
\[ [m : _{0} \alpha] \]

• 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)

\[ \forall \alpha, \beta \in \text{Type}. \alpha \subseteq \beta \Rightarrow F(\alpha) \subseteq G(\beta) \]

\[ \mu F \subseteq \mu G \]

\[ \Gamma \vdash \mu X.A \quad \Gamma \vdash \mu Y.B \quad \Gamma, Y \leq \text{Top}, X \leq Y \vdash A \leq B \]

\[ \Gamma \vdash \mu X.A \leq \mu Y.B \]
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 $\Gamma \vdash A \leq B$ and $\eta \models \Gamma$, then $\llbracket A \rrbracket_\eta \subseteq \llbracket B \rrbracket_\eta$
- Semantic soundness
  If $\Gamma \vdash a : A$ and $\eta \models \Gamma$, then $\llbracket \Gamma \rrbracket_\eta \models E(a) : \llbracket A \rrbracket_\eta$.
- 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 \( \lambda \)-calculus with dependent products and sums
References


(and many more)
Thank You