Depth Subtyping and Type Inference for Object Calculi

Michele Bugliesi
Dipartimento di Informatica
Universita Ca' Foscari di Venezia
michele@dsi.unive.it
http://www.dsi.unive.it/~michele

Santiago M. Pericás-Geertsen
Department of Computer Science
Boston University
santiago@cs.bu.edu
http://www.cs.bu.edu/~santiago

January 22, 2000
Introduction

- Our interest:
  - Type systems for objects with decidable and feasible type inference.

- Existing systems:
  - Recursive types + subtyping: decidable + feasible [Palsberg 95].
  - Self types: decidable + unfeasible [Jim/Palsberg 97].\(^a\)

⇒ Split Types:

- Type inference: decidable + feasible.
- Strictly more powerful than recursive types, but incomparable to Self types.
- Can be used to type: encoded functions, references, channels.

\(^a\)Type inference for a restricted form of Self types is NP-complete.
Sigma Calculus: Syntax

1. Syntax:

   \[ a, b, c, d ::= q \mid s \mid [\ell_i = \varsigma(s) b_i \{ s \}^{i \in I}] \mid a.\ell \mid a.\ell \leftarrow \varsigma(s)b \]

2. Notation:

   - \([..., \ell = a, ...]\) stands for \([..., \ell = \varsigma(s)a, ...]\) where \(s \not\in \text{FV}(a)\).
   - \(a.\ell := b\) stands for \(a.\ell \leftarrow \varsigma(s)b\) where \(s \not\in \text{FV}(b)\).
   - \(b\{s\}\) stands for a term \(b\) where \(s\) may occur free.
   - \(b\{c\}\) stands for a term \(b\{s\}\) where occurrences of \(s\) were substituted by \(c\).
Recursive Types + Subtyping

- The rules (Sub Object) and (Sub μ) do not coexist naturally :-(

\[ E \not\vdash \mu(X)[x : \text{int}, y : \text{int}, \text{move} : X] \leq \mu(Y)[x : \text{int}, \text{move} : Y] \]

Because of invariant subtyping, this would require \( X \leq Y \) and also \( Y \leq X \) since \( X \) must be the same type as \( Y \).

- Typing rules:

\[
\begin{align*}
(\text{Sub } \mu) & \quad E, X \leq Y \vdash A \leq B \\
E & \vdash \mu(X)A \leq \mu(Y)B \\
(\text{Sub Object}) & \quad J \subseteq I \\
E & \vdash [\ell_i : B_i]_{i \in I} \leq [\ell_j : B_j]_{j \in J}
\end{align*}
\]
Depth Subtyping?

- Is it sound to replace (Sub Object) by a more flexible typing rule?

\[
P_1 \equiv \mu(Y)[x : \text{int}, \text{move} : Y]
\]
\[
P_2 \equiv \mu(X)[x : \text{int}, y : \text{int}, \text{move} : X]
\]

Let \( p_1 \) be an arbitrary term of (proper) type \( P_1 \) and \( p_2 \) of type \( P_2 \).

\[
p_2 \triangleq [x = \varsigma(s)s.\text{move}.y, y = 0, \text{move} = \varsigma(s)s.y := s.y + 1]
\]

Then, the following program generates a \textit{run-time} error:

\[
(p_2.\text{move} := p_1).x \quad \text{(oops!!)}
\]
Some terms are still typable ...

- Define,

\[
p_1 \overset{\vartriangle}{=} [x = 0, move = \varsigma(s)s.x := s.x + 1]
\]
\[
p_2 \overset{\vartriangle}{=} [x = 0, y = 0, move = \varsigma(s)s.y := s.y + 1]
\]
\[
o \overset{\vartriangle}{=} p_2.\text{move} := p_1
\]

Clearly,

\[
\emptyset \vdash p_1 : \mu(X)[x : \text{int}, move : X] \equiv P_1
\]
\[
\emptyset \vdash p_2 : \mu(X)[x : \text{int}, y : \text{int}, move : X] \equiv P_2
\]

The term \( o \) can be assigned the type \( P \equiv [x : \text{int}, y : \text{int}, move : P_1] \) since \( P \leq [x : \text{int}, move : P_1] = P_1 \). Notice that we assigned a type to the term \( o \) even though \( P_2 \not\equiv P_1 \).
Others are not ...

- Define,

\[ p_0 \triangleq [\text{move} = \varsigma(s)s] \]
\[ p_2' \triangleq [x = \varsigma(s)s.\text{move}.y, y = 0, \text{move} = \varsigma(s)s.y := s.y + 1] \]
\[ o \triangleq [\ell = p_2'], \ell := p_0 \]

Clearly,

\[ \emptyset \vdash p_0 : \mu(X)[\text{move} : X] \equiv P_0 \]
\[ \emptyset \vdash p_2' : \mu(X)[x : \text{int}, y : \text{int}, \text{move} : X] \equiv P_2 \]

The most informative type for \( o \) is \([\ell : []]\). Consequently, \( o.\ell.\text{move} \) is not typable.
Split Types

- Split types are object types of the form $\mu(X)[\ell_i : (B_{i}^{u}, B_{i}^{s})_{i \in I}]$ where $B_{i}^{u} \leq B_{i}^{s}$, for every $i \in I$.

- Intuitively, the component $B_{i}^{u}$ – or update component – is used to type an update for $\ell_i$, whereas the component $B_{i}^{s}$ – or select component – is used to type a selection for $\ell_i$.

- The restriction $B_{i}^{u} \leq B_{i}^{s}$ means that every method does not ``advertise'' (select component) more structure than what it actually ``has'' (update component). This restriction is required for soundness (i.e. for subject reduction).
Subtyping over Split Types

- Subtyping is no longer *invariant*, but *covariant* in the select components and *contravariant* in the update components.

(Sub Object)

\[
\vdash C^u_j \leq B^u_j \quad \vdash B^s_j \leq C^s_j \quad (J \subseteq I)
\]

\[
\vdash \left[ \ell_i : (B^u_{i}, B^s_{i})^{i \in I} \right] \leq \left[ \ell_j : (C^u_{j}, C^s_{j})^{j \in J} \right]
\]

- Every pair of object types \( A \) and \( A' \) has a *least upper bound* \( A \sqcup A' \) and a *greatest lower bound* \( A \sqcap A' \). The type \( A \sqcup A' \) is *often* more informative than any common supertype found with recursive types.
Split Types: System $\text{Ob}^{\downarrow\uparrow}$

- Let $A \equiv [\ell_i : (B^u_i, B^s_i)^{i \in I}]$ in,

\[
\begin{align*}
(\text{Val Select}) & \quad E \vdash a : A \quad \vdash A \leq [\ell_j : (\bot, D)] \quad \ldots \quad E \vdash a.\ell_j : D \\
(\text{Val Update}) & \quad E \vdash a : A \quad \vdash A \leq [\ell_j : (D, \top)] \quad E, s : A \vdash b : D \quad \ldots \quad E \vdash a.\ell_j \leftarrow \varsigma(s) b : A \\
(\text{Val Object}) & \quad E, s : [\ell_i : (B^u_i, B^s_i)^{i \in I}] \vdash b_i : B^u_i \quad \vdash B^u_i \leq B^s_i \quad \ldots \quad E \vdash [\ell_i = \varsigma(s) b_i^s]^{i \in I} : A
\end{align*}
\]
More terms are typable in Ob$^\uparrow\uparrow$

- Define,

\[ p_0 \triangleq \text{move} = \zeta(s)s \]
\[ p_2' \triangleq [x = \zeta(s)s.\text{move}.y, y = 0, \text{move} = \zeta(s)s.y := s.y + 1] \]
\[ o \triangleq [\ell = p_2'].\ell := p_0 \]

We can prove,

\[ \emptyset \vdash p_0 : \mu(X)[\text{move} : (X, X)] \equiv S_0 \]
\[ \emptyset \vdash p_2' : \mu(X)[x : (\text{int, int}), y : (\text{int, int}), \text{move} : (X, X)] \equiv S_2 \]

The most informative type for \( o \) in Ob$^\uparrow\uparrow$ is,

\[ [\ell : \mu(X)[\text{move} : (S_0 \cap S_2, X)]] \]

Consequently, \( o.\ell.\text{move} \) is \textit{now} typable.
Encoding the Lambda Calculus

- Encoding from [Abadi/Cardelli 96]:

1. \([ q ] = q,\)
2. \([ x ] = x,\)
3. \([ \lambda(x)b\{x\}] = [arg = \varsigma(s)s.a.rg,\]
   \hspace{1cm}val = \varsigma(s)[b\{x\}]\{x := s.a.rg\}]\),
4. \([ a(b) ] = ([ a ].a.rg := [ b ]).val.\)
Preservation of Typings

- Encoding of types:

1. $\llbracket Q \rrbracket = Q$,

2. $\llbracket A \rightarrow B \rrbracket = \llbracket \text{arg}:(\llbracket A \rrbracket, \top), \text{val}:(\bot, \llbracket B \rrbracket) \rrbracket$.

Notice that, if $\vdash A \rightarrow B \leq A' \rightarrow B'$ is derivable in $\mathbf{F}_\leq$ (simply typed + subtyping) then $\vdash \llbracket A \rightarrow B \rrbracket \leq \llbracket A' \rightarrow B' \rrbracket$ is derivable in $\mathbf{Ob}^\uparrow\uparrow$.

- Theorem (Preservation of Typing).

Let $a$ be an arbitrary $\lambda$-term and $A$ an arbitrary type. If $E \vdash a : A$ is derivable in $\mathbf{F}_\leq$ then $\llbracket E \rrbracket \vdash \llbracket a \rrbracket : \llbracket A \rrbracket$ is derivable in $\mathbf{Ob}^\uparrow\uparrow$. 
**Type Inference Algorithm**

**Init.** Form the initial pair \((\{\Gamma \triangleright a : \alpha\}, \emptyset)\), where \(\alpha\) is a fresh type variable and \(\Gamma\) an environment mapping the free variables of \(a\) to fresh type variables.

**Iterate.** Let \((J, C)\) be the current pair. If \(J\) is empty, then stop. Otherwise, select a judgement from \(J\) and rewrite it using the appropriate rule.

**Theorem (Soundness and Completeness)**
The type inference algorithm is sound and complete with respect to type derivations in \(\mathbb{Ob}^{\uparrow\uparrow}\).
Split Types vs. Recursive Types

- Recursive types: a recursive type can be encoded as a Split type where update and select components are identical (invariant subtyping).

- Recursive types + variance annotations: the object type $[\ell_i \nu_i : B_i^{i \in I}]$ where $\nu_i \in \{^+, ^-, \circ\}$ can be encoded as $[\ell_i : (B_i^u, B_i^s)^{i \in I}]$ where:
  
  - $B_i^u = B_i$ and $B_i^s = \top$ when $\nu_i = ^-$,
  - $B_i^u = \bot$ and $B_i^s = B_i$ when $\nu_i = ^+$,
  - $B_i^u = B_i^s = B_i$ when $\nu_i = \circ$. 

Santiago M. Pericás-Geertsen
Split Types vs. Self Types

- Update rule for Self types:

\[(A \equiv \varsigma(X)[..., \ell : B\{X\}, ...])\]

\[
\frac{E \vdash a : A \quad E, Y \leq A, s : Y \vdash b : B\{Y\}}{E \vdash a.\ell \leftarrow \varsigma(s)b : A}
\]

- The typing power of the two systems is \textit{incomparable}:
  
  - Not typable with Self types:

\[
[move = \varsigma(s)s].move := [move = \varsigma(s)s]
\]

  - Not typable with Split types:

\[
p_0 \triangleq [move = \varsigma(s)s]
\]

\[
p_2' \triangleq [x = \varsigma(s)s.move.y, y = 0, move = \varsigma(s)s.y := s.y + 1]
\]

\[
o \triangleq ([\ell = p_2'].\ell := p_0).\ell.move \leftarrow \varsigma(s)s
\]
Split Types vs. Self Types

- Given,

\[ S_1 \equiv \mu(X)[x : (\text{int, int}), move : (X, X)] \]
\[ S_2 \equiv \mu(X)[x : (\text{int, int}), y : (\text{int, int}), move : (X, X)] \]

We have,

\[ \vdash S_1 \leq \mu(X)[x : (\text{int, int}), move : (S_1 \cap S_2, X)] \]
\[ \vdash S_2 \leq \mu(X)[x : (\text{int, int}), move : (S_1 \cap S_2, X)] \]

However,

\[ \not\vdash S_2 \leq S_1 \]
Conclusions

Split Types:

+ Provide a flexible (in depth) form of subtyping.
+ Generalize recursive types and variance annotations.
+ Support a typed encoding for functions.

? Useful for other applications: reference types and channel types.

+/- Are not comparable with Self Types.

Type Inference Algorithm:

+ Runs in polynomial time.\textsuperscript{a}

- Constraint sets are often hard to read.

\textsuperscript{a}No formal proof is included in the FOOL version of the paper.