Objects and subtyping in the $\lambda\Pi$-calculus modulo

Ali Assaf, Raphaël Cauderlier, Catherine Dubois

TYPES 2014, May 12
Motivations

- The $\lambda\Pi$-calculus modulo has been designed to encode other calculi
  - Functional Pure Type Systems
  - Proof assistants: Coq, HOL, FoCaLize
  - Theorem provers: Zenon, iProver

- We use $\lambda\Pi$-calculus modulo rewriting to study OOL semantics
  - How can we translate object mechanisms in the $\lambda\Pi$-calculus modulo?

- Object calculi have type systems with (object) subtyping
  - The $\lambda\Pi$-calculus modulo lacks subtyping

- Subtyping is a common feature of type systems, also present in Coq (universes)
Related work

- In System $\text{F}_\omega$ (polymorphism, type operators and subtyping)
  - Implemented in Yarrow (1997): a proof assistant with object subtyping

- Object calculi (a.k.a $\zeta$-calculi) from Abadi and Cardelli, *A Theory of Objects*, Springer Verlag, 1996
  - Deep encodings in Coq, focus on proving properties on the type system
    - by Gillard and Despeyroux (1999): reasoning on binders encoded via DeBruijn indices
    - and Liquori (2007): proof of the subject-reduction theorem

- In Isabelle/HOL: deep formalisation of class-based languages (parts of Java and Scala) with extensible records: Klein and Nipkow (2005), Foster and Vytiniotis (2006)
This work

- Encoding of an object calculus: the simply-typed $\varsigma$-calculus
- Shallow embedding
  - semantically equal terms, types or proofs should not be distinguishable after the encoding
  - expected efficiency
  - readability

- In the $\lambda\Pi$-calculus modulo
Outline

1. The $\lambda\Pi$-calculus modulo and Dedukti

2. The simply-typed $\varsigma$-calculus

3. Explicit subtyping in the $\lambda\Pi$-calculus modulo
The $\lambda\Pi$-calculus modulo

- The $\lambda\Pi$-calculus is a typed $\lambda$ calculus with dependent types
- The $\lambda\Pi$-calculus modulo, introduced by Cousineau and Dowek in 2007, extends the $\lambda\Pi$-calculus with a rewrite system $R$.

$$\frac{\Gamma \vdash t : A \quad A \equiv_{\beta R} B}{\Gamma \vdash t : B} \quad \text{(Conv)}$$
Dedukti

- Type-checker for the $\lambda\Pi$-calculus modulo
  It is a free software, available at
  https://www.rocq.inria.fr/deducteam/Dedukti/
- Dependent types
- Rewriting on terms and types
- Partial functions and proofs
- Non-linear pattern-matching

- Functional semantics (imperative semantics also studied)
- Model of both class-based and object-based languages
- No termination guaranteed by typing
- Structural subtyping
Syntax and semantics

- **Types**

  \[ A ::= [ l_i : A_i ]_{i=1..n} \quad \text{labels are unordered} \]

- **Terms**

  \[ t, u ::= [ l_i = \varsigma(x : A) \ t_i ]_{i=1..n} \]
  \[ t.l \]
  \[ t.l \leftarrow \varsigma(x : A) \ u \]

  \( (t.l \leftarrow u) \) abbreviates \( (t.l \leftarrow \varsigma(x : A) \ u) \) where \( x \not\in \text{FV}(u) \).

  \( (l = u) \) abbreviates \( (l = \varsigma(x : A) \ u) \) where \( x \not\in \text{FV}(u) \).

- **Operational semantics**

  \[ A ::= [ l_i : A_i ]_{i=1..n} \]
  \[ t ::= [ l_i = \varsigma(x : A) \ t_i ]_{i=1..n} \]

  \[ t.l_j \quad \rightarrow \quad t_j \ [t/x] \]
  \[ t.l_j \leftarrow \varsigma(x : A) \ u \quad \rightarrow \quad [ l_j = \varsigma(x : A) \ u, l_i = \varsigma(x : A) \ t_i ]_{i=1..n, i \neq j} \]
Typing and subtyping

\[ A := [ l_i : A_i ]_{i=1..n} \]

\[ \forall i=1..n \quad \Gamma, x : A \vdash t_i : A_i \quad (\text{obj}) \]
\[ \Gamma \vdash [ l_i = \varsigma(x : A) t_i ]_{i=1..n} : A \quad (\text{obj}) \]
\[ \Gamma \vdash t : A \quad (\text{select}) \]
\[ \Gamma \vdash t.l_i : A_i \]

\[ \Gamma \vdash t : A \quad \Gamma, x : A \vdash u : A_i \quad (\text{update}) \]
\[ \Gamma \vdash t.l_i \leftrightarrow \varsigma(x : A) u : A \]

\[ [ l_i : A_i ]_{i=1..n+m} <: [ l_i : A_i ]_{i=1..n} \]

\[ \Gamma \vdash t : A \quad A <: B \quad (\text{subsume}) \]
\[ \Gamma \vdash t : B \]
Example: Encoding of booleans

\[
\begin{align*}
\text{Bool}_A & := [\text{if} : A, \text{then} : A, \text{else} : A] \\
\text{true}_A & := [\text{if} = \varsigma(\text{self} : A) \text{self.then}, \\
& \quad \text{then} = \varsigma(\text{self} : A) \text{self.then}, \\
& \quad \text{else} = \varsigma(\text{self} : A) \text{self.else}] \\
\text{false}_A & := [\text{if} = \varsigma(\text{self} : A) \text{self.else}, \\
& \quad \text{then} = \varsigma(\text{self} : A) \text{self.then}, \\
& \quad \text{else} = \varsigma(\text{self} : A) \text{self.else}] \\
\text{if} & := \text{if}_A \ b \ \text{then} \ t \ \text{else} \ e := ((b.\text{then} \leftarrow t).\text{else} \leftarrow e).\text{if}
\end{align*}
\]

"then" and "else" methods are updated before "if" is selected
Subtyping example

RomCell :=  [ get : nat ]  
PromCell :=  [ get : nat, set : nat → RomCell ]  

PromCell <: RomCell  

myCell : PromCell := [ get = 0,  
set = λ self : PromCell. λ n : nat. self.get ← n ]  

myCell.set(42).get ↦* 42
Translation scheme from simply-typed $\zeta$-calculus to $\lambda\Pi$-calculus modulo

- Types and objects are translated as association lists
- The operational semantics is translated to rewrite rules
- Subtyping is explicit
Explicit subtyping

- In the $\lambda\Pi$-calculus modulo, each term has at most one type modulo the rewrite system $+ \beta$ conversion.
- Convertibility is a symmetric relation.
- We cannot rewrite $A$ to $B$ whenever $A <: B$ because that would make both types equal.
- Hence we ask the user to provide explicit coercions (subtyping annotations).
Translation of types

- Types are translated by normalized association lists
- Equality and subtyping relations on types are decidable:

\[
\begin{align*}
A = A & \quad \iff \quad \text{true} \\
[] = (_, _) :: _ & \quad \iff \quad \text{false} \\
(_, _) :: _ = [] & \quad \iff \quad \text{false} \\
(l_1, A_1) :: B_1 = (l_2, A_2) :: B_2 & \quad \iff \quad l_1 = l_2 \land A_1 = A_2 \land B_1 = B_2 \\
A <: [] & \quad \iff \quad \text{true} \\
A <: (l, B_1) :: B_2 & \quad \iff \quad B_1 = \text{assoc} \ A \ l \land A <: B_2
\end{align*}
\]
Translation of objects

Objects are also translated by association lists with labels in the same order than in the corresponding type

- an object of type \( A \) is something of the form
  \[
  [l = \varsigma(x : A) (t : \text{assoc } A \ l)]_{l \in \text{dom}(A)}
  \]
- sublists are not well-typed objects
- to construct objects, we need to consider (ill-typed) objects defined on subsets of \( \text{dom}(A) \)
- to coerce objects, we need to consider (ill-typed) objects with methods typed by (assoc \( B \)).

\[ \Rightarrow \text{ A pre-object of type } (A, f, D) \text{ is something of the form} \]
\[
[l = \varsigma(x : A) (t : f \ l)]_{l \in D}
\]
Semantics

- **preselect**: $\forall A, f, D, \text{PreObj}(A, f, D) \to \forall l, A \to f(l)$.
  
  preselect $((l_1, m) :: o) \ l_2$
  
  $\rightarrow$

  if $(l_1 = l_2)$ then $m$ else preselect $(o, l_2)$

- **select**: $\forall A, A \to \forall l, \text{assoc} A \ l$.

  select $a \ l \rightarrow$ preselect $a \ l \ a$

- **preupdate**: $\forall A, f, D, \text{PreObj}(A, f, D) \to \forall l, (A \to f(l)) \to \text{PreObj}(A, f, D)$.

  preupdate $((l_1, m_1) :: o) \ l_2 \ m_2$
  
  $\rightarrow$

  if $(l_1 = l_2)$
  
  then $((l_2, m_2) :: o)$
  
  else $((l_1, m_1) :: (\text{preupdate} A \ f \ D \ o \ l_2 \ m_2))$

- **update**: $\forall A, A \to \forall l, (A \to \text{assoc} A \ l) \to A$.

  update $a \ l \ m \rightarrow$ preupdate $a \ l \ m$
Coercion

- \textbf{coerce}: \( \forall A, B, A <: B \rightarrow A \rightarrow B \)
  - Partial function
    - cases where \( A \not<: B \) don't have to be defined, they will not reduce
  - Decidibility of <:
    - proof of \( A <: B \) is trivial for concrete \( A \) and \( B \)

- Some lemmata about equality, subtyping and pre-objects needed
  \[
  \forall A, f, g, D, \\
  (\forall l \in D, f(l) = g(l)) \rightarrow \text{PreObj}(A, f, D) \rightarrow \text{PreObj}(A, g, D).
  \]
Implementation

- Code and examples available at
  https://www.rocq.inria.fr/deducteam/Sigmaid/sigmaid.tar.gz
- Auxiliary definitions (mostly the definition of labels as strings)
  430 lines, 151 rewrite rules
- Core calculus
  523 lines, 104 rewrite rules
- Time
type-checked by Dedukti v2.2c in 70ms
Tests

Examples from Abadi and Cardelli

myPromCell : PromCell := [get = 42,
    set = \(\varepsilon\)(self : PromCell) \(\lambda\)(x : Nat)
    coerce PromCell RomCell (self.get \rightleftharpoons x)]

<table>
<thead>
<tr>
<th>Expression</th>
<th>(\triangleright)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (_) true (_) then (_) else (_)</td>
<td>(\triangleright)* t</td>
<td>✓</td>
</tr>
<tr>
<td>if (_) false (_) then (_) else (_)</td>
<td>(\triangleright)* e</td>
<td>✓</td>
</tr>
<tr>
<td>((\lambda)(x : A \mapsto b(x))) a</td>
<td>(\triangleright)* b(a)</td>
<td>✓</td>
</tr>
<tr>
<td>(coerce ColorPoint Point [ x = 42, y = 24, c = red ]).x</td>
<td>(\triangleright)* 42</td>
<td>✓</td>
</tr>
<tr>
<td>[get = 42 ].get</td>
<td>(\triangleright)* 42</td>
<td>✓</td>
</tr>
<tr>
<td>myPromCell.get</td>
<td>(\triangleright)* 42</td>
<td>✓</td>
</tr>
<tr>
<td>myPromCell.set(24).get</td>
<td>(\triangleright)* 24</td>
<td>✓</td>
</tr>
<tr>
<td>myCell.set(24).get</td>
<td>(\triangleright)* 24</td>
<td>✓</td>
</tr>
</tbody>
</table>
Conclusion and perspectives

- Shallow embedding of a typed object calculus with subtyping
- Formalized in Dedukti in a few hundred lines
- Validated on examples from Abadi and Cardelli
Conclusion and perspectives

- Study the efficiency
- Check the confluence
- Extend the object calculus with dependent types
  - Specifications and proofs as methods
  - Dependencies between methods
  - Loss of decidable type equality
  - Abstract method / redefinition
- Other object formalizations (featherweight java)
Thank you!
Ordering of labels

- Indistinguishable types in the source language are not always convertible in the target language.

- This could be solved by maintaining the list ordered with this extra rewrite-rule:

  \[(l_1, A_1) :: (l_2, A_2) :: B \quad ? \quad l_1 > l_2\]

  \[\rightarrow\]

  \[(l_2, A_2) :: (l_1, A_1) :: B\]

- But this breaks confluence with the rule:
  \[A = A \rightarrow true\]

- There are other approaches:
  - Add a proof of \(l_1 < l_2\) as argument of cons and define insert without logical argument.
  - Define a guarded version of equal.