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 $F^{\omega}_{<:}$ (polymorphism, type operators and subtyping)
  - Several deep encodings: Cardelli (1984), Pierce, Turner and Hofmann (1993-1995), Bruce (1993), Abadi, Cardelli and Viswanathan (1996)
  - Implemented in Yarrow (1997): a proof assistant with object subtyping
- Object calculi (a.k.a ς-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 DeBrujn 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 ς-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 ς-calculus
3. Explicit subtyping in the $\lambda\Pi$-calculus modulo

The λΠ-calculus modulo

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

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

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

The simply-typed ς-calculus: Abadi and Cardelli, *A Theory of Objects*, 1996

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

Syntax and semantics

- Types

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

- Terms

$$\begin{aligned} t, u ::= \quad & [\ l_i = \varsigma(x : A)\ t_i\ ]_{i=1..n} \\ & t.l \\ & t.l \Leftarrow \varsigma(x : A)\ u \end{aligned}$$

$(t.l \Leftarrow u)$ abbreviates $(t.l \Leftarrow \varsigma(x : A)\ u)$ where $x \notin FV(u)$.
$(l = u)$ abbreviates $(l = \varsigma(x : A)\ u)$ where $x \notin FV(u)$.

- Operational semantics

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

$$\begin{aligned} t.l_j & \rightarrowtail t_j\ [t/x] \\ t.l_j \Leftarrow \varsigma(x : A)\ u & \rightarrowtail [\ l_j = \varsigma(x : A)\ u, l_i = \varsigma(x : A)\ t_i\ ]_{i=1..n,\ i \neq j} \end{aligned}$$

Typing and subtyping

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

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

$$\frac{\Gamma \vdash t : A \qquad \Gamma, x : A \vdash u : A_i}{\Gamma \vdash t.l_i \Leftarrow \varsigma(x : A)\, u : A}\ (\text{update})$$

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

$$\frac{\Gamma \vdash t : A \qquad A <: B}{\Gamma \vdash t : B}\ (\text{subsume})$$

Example: Encoding of booleans

$$
\begin{aligned}
\mathrm{Bool}_A &:= [\mathrm{if} : A, \mathrm{then} : A, \mathrm{else} : A] \\
\mathrm{true}_A &:= [\mathrm{if} = \varsigma(\mathrm{self} : A)\ \mathrm{self.then}, \\
&\quad\ \ \mathrm{then} = \varsigma(\mathrm{self} : A)\ \mathrm{self.then}, \\
&\quad\ \ \mathrm{else} = \varsigma(\mathrm{self} : A)\ \mathrm{self.else}] \\
\mathrm{false}_A &:= [\mathrm{if} = \varsigma(\mathrm{self} : A)\ \mathrm{self.else}, \\
&\quad\ \ \mathrm{then} = \varsigma(\mathrm{self} : A)\ \mathrm{self.then}, \\
&\quad\ \ \mathrm{else} = \varsigma(\mathrm{self} : A)\ \mathrm{self.else}]
\end{aligned}
$$

$$
\mathrm{if}_A\ b\ \mathrm{then}\ t\ \mathrm{else}\ e := ((b.\mathrm{then} \Leftarrow t).\mathrm{else} \Leftarrow e).\mathrm{if}
$$

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

Subtyping example

$$\begin{aligned} \text{RomCell} &:= [\ \text{get} : \text{nat}\ ] \\ \text{PromCell} &:= [\ \text{get} : \text{nat}, \text{set} : \text{nat} \to \text{RomCell}\ ] \end{aligned}$$

$$\text{PromCell} <: \text{RomCell}$$

$$\begin{aligned} \text{myCell} : \text{PromCell} := [\ &\text{get} = 0, \\ &\text{set} = \varsigma(\text{self} : \text{PromCell})\ \lambda(\text{n} : \text{nat})\ \text{self.get} \Leftarrow \text{n}\ ] \end{aligned}$$

$$\text{myCell.set(42).get} \rightarrowtail^{*} 42$$

Translation scheme from simply-typed ς-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 + β 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:

$$
\begin{array}{l|l}
\mathrm{A} = \mathrm{A} \hookrightarrow \mathrm{true} & \\
[\,] = (\_\,,\_\,) :: \_ \hookrightarrow \mathrm{false} & \mathrm{A} <: [\,] \hookrightarrow \mathrm{true} \\
(\_\,,\_\,) :: \_ = [\,] \hookrightarrow \mathrm{false} & \mathrm{A} <: (\mathrm{l}, \mathrm{B}_1) :: \mathrm{B}_2 \\
(\mathrm{l}_1, \mathrm{A}_1) :: \mathrm{B}_1 = (\mathrm{l}_2, \mathrm{A}_2) :: \mathrm{B}_2 & \quad \hookrightarrow \mathrm{B}_1 = \mathrm{assoc}\ \mathrm{A}\ \mathrm{l} \wedge \mathrm{A} <: \mathrm{B}_2 \\
\quad \hookrightarrow \mathrm{l}_1 = \mathrm{l}_2 \wedge \mathrm{A}_1 = \mathrm{A}_2 \wedge \mathrm{B}_1 = \mathrm{B}_2 &
\end{array}
$$

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

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

Semantics

- preselect : ∀ A, f, D, PreObj(A, f, D) → ∀ l, A → f(l).
  preselect ((l1, m) :: o) l2
  ↪
  if (l1 = l2) then m else preselect (o, l2)
- select : ∀ A, A → ∀ l, assoc A l.
  select a l ↪ preselect a l a
- preupdate : ∀ A, f, D, PreObj(A, f, D) → ∀ l, (A → f(l)) → PreObj(A, f, D).
  preupdate ((l1, m1) :: o) l2 m2
  ↪
  if (l1 = l2)
  then ((l2, m2) :: o)
  else ((l1, m1) :: (preupdate A f D o l2 m2))
- update : ∀ A, A → ∀ l, (A → assoc A l) → A.
  update a l m ↪ preupdate a l m

Coercion

- coerce: $\forall$ A, B, A <: B $\to$ A $\to$ 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

$$\forall \text{ A, f, g, D,}$$
$$(\forall \text{ l} \in \text{D, f(l) = g(l))} \to \text{PreObj(A, f, D)} \to \text{PreObj(A, g, D).}$$

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

$$\begin{aligned}\text{myPromCell} : \text{PromCell} := [&\text{get} = 42,\\ &\text{set} = \varsigma(\text{self} : \text{PromCell})\ \lambda(x : \text{Nat})\\ &\quad \text{coerce PromCell RomCell (self.get} \Leftarrow x)]\end{aligned}$$

$\text{if}_A\ \text{true}_A$ then t else e	$\hookrightarrow^*$ t	✓
$\text{if}_A\ \text{false}_A$ then t else e	$\hookrightarrow^*$ e	✓
$(\lambda\ (x : A \mapsto b(x)))\ a$	$\hookrightarrow^*$ b(a)	✓
(coerce ColorPoint Point [x = 42, y = 24, c = red]).x	$\hookrightarrow^*$ 42	✓
[get = 42].get	$\hookrightarrow^*$ 42	✓
myPromCell.get	$\hookrightarrow^*$ 42	✓
myPromCell.set(24).get	$\hookrightarrow^*$ 24	✓
myCell.set(24).get	$\hookrightarrow^*$ 24	✓

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)

Questions

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

$$(l_1, A_1) :: (l_2, A_2) :: B \quad ?\ l_1 > l_2$$
$$\hookrightarrow$$
$$(l_2, A_2) :: (l_1, A_1) :: B$$

- But this breaks confluence with the rule
  $A = A \hookrightarrow \text{true}$
- 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