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

Raphaël Cauderlier, Catherine Dubois

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

Raphaël Cauderlier[1,2] and Catherine Dubois[2,3]

[1] INRIA Paris-Rocquencourt, Paris, France
[2] CNAM, Paris, France
[3] ENSIIE, Évry, France

Abstract

We present a shallow embedding of the Object Calculus of Abadi and Cardelli in the $\lambda\Pi$-calculus modulo, an extension of the $\lambda\Pi$-calculus in which conversion is considered modulo a rewrite system.

This embedding may be used as an example of translation of subtyping, a feature also present in some proof assistants like Coq and PVS.

This embedding is proved correct with respect to the operational semantics and the type system of the Object Calculus.

It has been implemented as a translation tool from the Object Calculus to Dedukti, a type-checker for the $\lambda\Pi$-calculus modulo.

Introduction

Motivation

The $\lambda\Pi$-calculus modulo[9] is a type system with dependent types in which the conversion congruence can be extended by a user-supplied rewrite system.

It can be used as a logical framework to encode all the functional Pure Type Systems[9]. Moreover, translation tools from real-world proof assistant like Coq[8, 4] and the HOL family[3] to Dedukti[17], a type-checker for the $\lambda\Pi$-calculus modulo, permit to recheck proofs done in these complex systems using a small, easy to trust, checker.

A major feature of OO type systems is subtyping, and it will be the focus of this article, but subtyping is not limited to the world of objects; it is also present in the Coq proof assistant which is also translated to Dedukti.

Translating an object calculus with subtyping is hence a way to understand one special case of subtyping to be compared with other like universe cumulativity in Coq or predicate subtyping in PVS.

We also believe that objects may be useful for proof assistants like they already are for programming; we would like to be able to develop proofs using OO concepts and mechanisms such as inheritance, method redefinition and late binding. For this we would need complex objects where methods would be typed with dependent types. This work is a first step in this direction starting from a very simple type-system for objects.

Related work

Many encodings of objects have been developed, studied, and compared in the 90s. In order to express complex but common object mechanisms such as self reference and inheritance, the target language is usually chosen to be very rich like System $F^{\omega}_{<:}$ (a type system featuring polymorphism, existential types, type operators and subtyping).

Because of the complexity of System $F^{\omega}_{<:}$ and the limitations of these encodings, they are of limited practicality to study OOL or to implement OO mechanisms in proof systems; the only implementation to our knowledge is the Yarrow proof assistant[18].

However, following the example of λ-calculi, small calculi taking objects as core notions have been designed and their type systems have been proved safe. For example:

- The λ-calculus of objects[10] is an extension of the simply-typed λ-calculus with object construction, method call and method update. In this system, objects are extended with their types using extensible records.
- The Object Calculi from Abadi and Cardelli[1] are a collection of calculi based on objects. They differ from the λ-calculus of objects in two important ways: they are not based on the λ-calculus so they have less constructs and objects and their types are fixed records. Hence they are somewhat simpler but they still are very expressive.
- Featherweight Java[13] is a core calculus for the popular class-based Java programming language. It is a small class-based OO calculus designed to study extensions of class-based languages such as Java.

These three calculi can easily encode the λ-calculus, allow non-terminating recursion and have some form of subtyping: respectively raw-polymorphism, structural subtyping and class-based subtyping.

These calculi's type systems have been formalized in proof assistants; these formalizations can be seen as deep embeddings of the calculi in type theory; some of these are:

- encodings of Featherweight Java in Coq[14] and Isabelle/HOL[11] using extensible records
- Object calculi in Coq, LF, PVS and LEGO. (`http://www-sop.inria.fr/members/Luigi.Liquori/PAPERS/jar-07.pdf` et les trucs cits page 39)

Encodings of objects based on rewrite techniques have also been studied; for example, in the ρ-calculus[6], a full encoding of the untyped Object Calculus and λ-calculus of objects[5] and a partial encoding of the simply-typed Object Calculus[6] have been designed. In the Maude specification environment[7], objects are also encoded using a rewrite system thanks to the reflection mechanism of Maude.

Contribution

In contrast with these deep encodings, our contribution is a shallow embedding. What we mean by the term "shallow" is that the elements of the source language, the simply-typed ς-calculus: terms, values and types are respectively translated to terms, values and types in the λΠ-calculus modulo such that operational semantic, typing derivations and binding operation are preserved by this translation.

The first section of this article describes the λΠ-calculus modulo, our target language, the second section describes the simply-typed ς-calculus, our source language. section 3 is the main section of this article; it defines a strongly-normalizing encoding of the simply-typed ς-calculus in the λΠ-calculus modulo. This encoding is not fully shallow because it does not preserve the operational semantics. In section 4, we add two rewrite rules to this encoding to reflect the operational semantics; doing so we lose strong-normalization. section 5 describes the implementation of our encoding in Dedukti.

1 The λΠ-calculus modulo

1.1 The λΠ-calculus

The λΠ-calculus[12], also known as LF and λP, is an extension of the simply typed λ-calculus adding dependent types. λΠ terms and types have the following syntax:

$$t, u, v, \ldots, \tau ::= x \mid t\ u \mid \lambda x : \tau.t \mid \Pi x : \tau_1.\tau_2 \mid \mathbf{Type} \mid \mathbf{Kind}$$

There is no syntactic distinction between terms and types but we use latin letters starting at t to denote terms and the greek letter τ to denote types.

We use the letter s to denote a sort, either **Type** or **Kind**.

The term $\Pi x : \tau_1.\tau_2$ where the variable x may appear free in τ_2 is called a dependent product and represents the type of functions taking an argument x of type τ_1 and returning a value of type τ_2.

If x does not appear free in τ_2, the term $\Pi x : \tau_1.\tau_2$ is the type of functions from type τ_1 to type τ_2 and we will abbreviate it as $\tau_1 \to \tau_2$.

If τ_1 is clear from context, the term $\Pi x : \tau_1.\tau_2$ will be abbreviated as $\Pi x.\tau_2$.

A list of term typing declarations is called a (λΠ) context:

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau$$

where $\emptyset$ denotes the empty context.

Some contexts are called well-formed. When the context Γ is well-formed, we write $\Gamma \vdash_d$.

Some terms are called well-typed. When the term t is well-typed of type τ in context Γ, we write $\Gamma \vdash_d t : \tau$.

These two notions are mutually defined in Figure 1 where $\equiv_\beta$ is the congruence induced by β-reduction.

The λΠ-calculus is the type-system on which logical frameworks such as Automath[15] and Twelf[16] are based.

1.2 The λΠ-calculus modulo

The λΠ-calculus modulo is an extension of the λΠ-calculus which weakens the conversion rule; terms are considered convertible not only when they are β-equivalent but also when they are congruent for a given rewrite system.

λΠ modulo terms are λΠ terms but λΠ modulo contexts may also contain rewrite rules which also need to be well-typed.

Rewrite rules are composed of three parts; a rule context which is a λΠ context used to type free variables, a left-hand side and a right-hand side which are both terms. In order to make the rewrite system computable, we need to add the following restrictions on rewrite rules:

- the left-hand side is a term without λ nor Π abstraction (we call it a pattern)
- free variables of the right-hand side also appear free in the left-hand side
- free variables of the left-hand side are declared in the rule context.

So the new syntax for contexts is as follows:

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau \mid \Gamma, (\Lambda t \hookrightarrow u)$$

$$s \in \{\mathbf{Type}, \mathbf{Kind}\}$$

$$\frac{}{\emptyset \vdash_d} \text{(Empty)} \qquad \frac{\Gamma \vdash_d \qquad \Gamma \vdash_d \tau : s \qquad x \notin \Gamma}{\Gamma, x : \tau \vdash_d} \text{(Decl)} \qquad \frac{\Gamma \vdash_d}{\Gamma \vdash_d \mathbf{Type} : \mathbf{Kind}} \text{(Sort)}$$

$$\frac{\Gamma \vdash_d \qquad x : \tau \in \Gamma}{\Gamma \vdash_d x : \tau} \text{(Var)} \qquad \frac{\Gamma \vdash_d \tau_1 : \mathbf{Type} \qquad \Gamma, x : \tau_1 \vdash_d \tau_2 : s}{\Gamma \vdash_d \Pi x : \tau_1 . \tau_2 : s} \text{(Prod)}$$

$$\frac{\Gamma \vdash_d \tau_1 : \mathbf{Type} \qquad \Gamma, x : \tau_1 \vdash_d \tau_2 : s \qquad \Gamma, x : \tau_1 \vdash_d t : \tau_2}{\Gamma \vdash_d \lambda x : \tau_1 . t : \Pi x : \tau_1 . \tau_2} \text{(Abs)}$$

$$\frac{\Gamma \vdash_d t_0 : \Pi x : \tau_1 . \tau_2 \qquad \Gamma \vdash_d t_1 : \tau_1}{\Gamma \vdash_d t_0 t_1 : \tau_2\{t_1/x\}} \text{(App)}$$

$$\frac{\Gamma \vdash_d t : \tau_1 \qquad \Gamma \vdash_d \tau_1 : s \qquad \Gamma \vdash_d \tau_2 : s \qquad \tau_1 \equiv_\beta \tau_2}{\Gamma \vdash_d t : \tau_2} \text{(Conv)}$$

Figure 1: inference rules for the λΠ-calculus

$\texttt{Nat} : \mathbf{Type}.$
$0 : \texttt{Nat}.$
$\texttt{S} : \texttt{Nat} \to \texttt{Nat}.$

$\texttt{plus} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}.$

$\texttt{plus}\ 0\ n \hookrightarrow n.$
$\texttt{plus}\ (\texttt{S}\ n_1)\ n_2 \hookrightarrow \texttt{S}\ (\texttt{plus}\ n_1\ n_2).$

$\texttt{A} : \mathbf{Type}.\texttt{List} : \texttt{Nat} \to \mathbf{Type}.$
$\texttt{empty} : \texttt{List}\ 0.$
$\texttt{cons} : \Pi n : \texttt{Nat}.\texttt{A} \to \texttt{List}\ n \to \texttt{List}\ (\texttt{S}\ n).$

$\texttt{append} : \Pi n_1 . \Pi n_2 .$
$\quad \texttt{List}\ n_1 \to \texttt{List}\ n_2 \to \texttt{List}\ (\texttt{plus}\ n_1 n_2).$

$\texttt{append}\ 0\ n\ \texttt{empty}\ l \hookrightarrow l.$
$\texttt{append}\ (\texttt{S}\ n_1)\ n_2\ (\texttt{cons}\ n_1\ a\ l_1)\ l_2 \hookrightarrow$
$\quad \texttt{cons}\ (\texttt{plus}\ n_1\ n_2)\ a\ (\texttt{append}\ n_1\ n_2\ l_1\ l_2)$

Figure 2: Example of λΠ modulo context: Peano natural numbers and concatenation of dependent lists

where Λ stands for λΠ contexts.

The rule context Λ will often be omitted when clear from context.

To check if contexts are well-formed, we add a rule for the new case of rewrite rule; a rewrite rule is well-formed in a context Γ if the left-hand side and the right-hand side have the same type in context Γ:

$$\frac{\Gamma \vdash_d \qquad \Gamma, \Lambda \vdash_d t : \tau \qquad \Gamma, \Lambda \vdash_d u : \tau}{\Gamma, (\Lambda t \hookrightarrow u) \vdash_d} \text{(RewriteRule)}$$

The conversion typing rule for the λΠ-calculus modulo is the same as the one for the λΠ-calculus except that the β-equivalence is replaced by the congruence modulo β and the rewrite system induced by the typing context Γ.

$$\frac{\Gamma \vdash_d t : T_1 \qquad \Gamma \vdash_d T_1 : s \qquad \Gamma \vdash_d T_2 : s \qquad T_1 \equiv_{\beta\Gamma} T_2}{\Gamma \vdash_d t : T_2} \text{ (Conv)}$$

Other typing rules are unchanged. In particular, if the typing judgment $\Gamma \vdash_d t : T$ is derivable in the λΠ-calculus, then it is also derivable in the λΠ-calculus modulo with the exact same derivation and an empty rewrite system.

An example of well-formed λΠ modulo context is shown in Figure 2. This example is composed of the definitions of the addition in Peano arithmetic and the concatenation of lists depending on their length. The definition of the addition is needed to convert the types of the left hand side to the type of the right hand side of each rewrite rule defining the concatenation; for instance, let us check that the rule $\texttt{append}\ 0\ n\ \texttt{empty}\ l \hookrightarrow l$ is well-formed in the context $\Gamma = \texttt{Nat} : \mathbf{Type}, 0 : \texttt{Nat}, \ldots, \texttt{append} : \Pi n_1.\Pi n_2.\texttt{List}\ n_1 \to \texttt{List}\ n_2 \to \texttt{List}\ (\texttt{plus}\ n_1\ n_2)$:

- The implicit rule context is $\Lambda = (n : \texttt{Nat}, l : \texttt{List}\ n)$.
- The constants 0, `empty` and `append` have respectively the types `Nat`, $\texttt{List}\ 0$ and $\Pi n_1.\Pi n_2.\texttt{List}\ n_1 \to \texttt{List}\ n_2 \to \texttt{List}\ (\texttt{plus}\ n_1\ n_2)$ in Γ.
- By successive applications of the (App) rule, we can type the left-hand side $\texttt{append}\ 0\ n\ \texttt{empty}\ l$ with type $\texttt{List}\ (\texttt{plus}\ 0\ n)$ in Γ, Λ.
- The rule $\texttt{plus}\ 0\ n \hookrightarrow n$ is in Γ so $\texttt{List}\ (\texttt{plus}\ 0\ n) \equiv_{\beta(\Gamma,\Delta)} \texttt{List}\ n$ and we can retype the left-hand side to type $\texttt{List}\ n$ in context Γ, Λ by an application of the (Conv) rule.
- The left-hand side $\texttt{append}\ 0\ n\ \texttt{empty}\ l$ and the right-hand side l have the same type $\texttt{List}\ n$ in context Γ, Λ so the rule $\Lambda(\texttt{append}\ 0\ n\ \texttt{empty}\ l) \hookrightarrow l$ is well-formed in context Γ.

The interesting properties about a λΠ modulo context and the associated rewrite system are confluence, strong normalization and well-formedness. None of them is decidable but when the rewrite system is both confluent and strongly normalizing, convertibility check can be decided by comparing the normal forms so well-formedness becomes decidable and is indeed implemented in the Dedukti[17] type checker.

2 The simply-typed ς-calculus

In this section, we will describe the source language of our encoding.

The simply-typed ς-calculus is an object calculus defined by Abadi and Cardelli[1, 2]. This calculus is an object-based (classes are not primitive constructs) calculus with functional semantics (values are immutable). Its type system features structural subtyping (as opposed to class subtyping). Contrary to simply-typed λ-calculus, well-typed ς-terms do not always terminate.

2.1 Syntax

The syntax of the simply-typed ς-calculus is divided between types and terms.

Types are (maybe empty) records of types:

$$A, B, \ldots ::= [l_i : A_i]_{i \in 1 \ldots n}$$

The order in which the labels appear does not matter as long as each l_i remains associated to the same A_i.

Terms are records of methods introduced by a self binder ς. Methods can be selected and updated.

$$\begin{array}{rcll} a, b, \ldots & ::= & x & \textit{variable} \\ & | & [l_i = \varsigma(x_i : A)a_i]_{i \in 1 \ldots n} & \textit{object} \\ & | & a.l & \textit{method selection} \\ & | & a.l \Leftarrow \varsigma(x : A)b & \textit{method update} \end{array}$$

When the variable introduced by the ς binder is unused, we may omit the binder and write $l = b$ and $a.l \Leftarrow b$ instead of, respectively, $l = \varsigma(x : A)b$ and $a.l \Leftarrow \varsigma(x : A)b$ where x does not appear free in b.

Typing contexts are lists of typing declarations:

$$\Delta ::= \emptyset \mid \Delta, x : A$$

in which each variable may appear at most once.

When x appears in Δ, we denote by $\Delta(x)$ the associated type.

2.2 Typing

The following rules, where A stands for $[l_i : A_i]_{i \in 1 \ldots n}$, define a type system for the simply-typed ς-calculus:

$$\frac{}{\Delta \vdash_\varsigma x : \Delta(x)}\ (\text{var}) \qquad \frac{\forall i \in 1 \ldots n \qquad \Delta, x_i : A \vdash_\varsigma a_i : A_i}{\Delta \vdash_\varsigma [l_i = \varsigma(x_i : A)a_i]_{i \in 1 \ldots n} : A}\ (\text{obj})$$

$$\frac{\Delta \vdash_\varsigma a : A \qquad i \in 1 \ldots n}{\Delta \vdash_\varsigma a.l_i : A_i}\ (\text{select}) \qquad \frac{\Delta \vdash_\varsigma a : A \qquad \Delta, x : A \vdash_\varsigma b : A_i}{\Delta \vdash_\varsigma a.l_i \Leftarrow \varsigma(x : A)b : A}\ (\text{update})$$

2.2.1 Subtyping

This type system is extended by a subtyping relation $<:$ defined as follows:

$$\frac{}{[l_i : A_i]_{i \in 1 \ldots n+m} <: [l_i : A_i]_{i \in 1 \ldots n}}\ (\text{subtype})$$

Since the order of labels is irrelevant, this definition actually states that A is a subtype of B whenever every label of B is also in A, with the same type.

This subtyping relation can be used to transtype terms with the following subsumption rule:

$$\frac{\Delta \vdash_\varsigma a : A \qquad A <: B}{\Delta \vdash_\varsigma a : B}\ (\text{subsume})$$

2.2.2 Minimum types

Abadi and Cardelli have proven that the simply-typed ς-calculus enjoys minimum typing[1] : for each well-typed term a in a context Δ, we can compute a type $\mathbf{mintype}_\Delta(a)$ such that

- $\Delta \vdash_\varsigma a : \mathbf{mintype}_\Delta(a)$
- for all A such that $\Delta \vdash_\varsigma a : A$, we have $\mathbf{mintype}_\Delta(a) <: A$.

The meta-level function **mintype**[1] is defined as follows:

- $\mathbf{mintype}_\Delta(x) = \Delta(x)$
- $\mathbf{mintype}_\Delta([\,]) = [\,]$
- $\mathbf{mintype}_\Delta([l_i = \varsigma(x_i : A)a_i]_{i\in 1\dots n+1}) = A$
- $\mathbf{mintype}_\Delta(a.l_j) = B_j$ where $\mathbf{mintype}_\Delta(a) = [l_i : B_i]_{i\in 1\dots n}$
- $\mathbf{mintype}_\Delta(a.l \Leftarrow \varsigma(x : A)) = A$

2.3 Operational Semantics

The values of the simply-typed ς-calculus are plain objects. Selection and update are reduced by the following operational semantics rules where A stands for $[l_i : A_i]_{i\in 1\dots n}$ and a stands for $[l_i = \varsigma(x_i : A)a_i]_{i\in 1\dots n}$:

$$\begin{array}{lcl} a.l_j & \rightarrowtail & a_j\{a/x_j\} \\ a.l_j \Leftarrow \varsigma(x : A')u & \rightarrowtail & [l_j = \varsigma(x : A)u, l_i = \varsigma(x_i : A)a_i]_{i\in 1\dots n, i\neq j} \end{array}$$

where $a_j\{a/x\}$ denotes the substitution of the variable x by the term a in term a_j.

2.4 Example

The power of the ς-calculus can be illustrated by the following example from Abadi and Cardelli[1] assuming we have a type Num for numbers and encoded the simply-typed λ-calculus:

$$\begin{array}{ll} \mathit{RomCell} & := [\ \mathit{get} : \mathit{Num}\ ] \\ \mathit{PromCell} & := [\ \mathit{get} : \mathit{Num}, \mathit{set} : \mathit{Num} \to \mathit{RomCell}\ ] \\ \mathit{PrivateCell} & := [\ \mathit{get} : \mathit{Num}, \mathit{contents} : \mathit{Num}, \mathit{set} : \mathit{Num} \to \mathit{RomCell}\ ] \\ \mathit{myCell} : \mathit{PromCell} & := [\ \mathit{get} = \varsigma(x : \mathit{PrivateCell})x.\mathit{contents}, \\ & \quad \mathit{contents} = \varsigma(x : \mathit{PrivateCell})0, \\ & \quad \mathit{set} = \varsigma(x : \mathit{PrivateCell})\lambda(n : \mathit{Num})x.\mathit{contents} \Leftarrow n\ ] \end{array}$$

RomCell is the type of read-only memory cell; the only action that we can perform on a *RomCell* is to read it.

[1] bold face is here used to distinguish the meta-level

A *PromCell* is a memory cell which can be written once, we can either read it now or write it and get a *RomCell*.

PrivateCell is a type used for implementation; it extends *PromCell* with a *contents* field which should not be seen from the outside.

The object *myCell* is a *PromCell* implemented as *PrivateCell* thanks to subsumption.

3 Encoding of the simply-typed ς-calculus in the λΠ-calculus modulo

This section describes an encoding of the simply-typed ς-calculus; a λΠ-modulo context together with a translation function from ς types, terms and contexts.

The rewrite system associated with this λΠ-modulo context will be confluent and strongly-normalizing, making type-checking of encoded terms decidable.

We want this encoding to be shallow in the sense discussed in the introduction : we want to preserve typing, reduction and the binding operation. The encoding described in this section will preserve typing and binding but preserving reduction of a non terminating system can, of course, not been achieved using a strongly-normalizing rewrite system.

In next section, we will add a few rewrite rules to this encoding in order to preserve reduction at the price of losing normalization.

This encoding is implemented as a translation tool producing Dedukti terms.

3.1 Encoding of types

We assume given an infinite λΠ-type `label` with a decidable equality.

Unit, product and Σ-types can be encoded in the λΠ-calculus modulo in a straightforward way so we will consider that they are given with the usual notations (respectively `unit`, $A \times B$ and $\Sigma x : A.B$).

3.1.1 Domains

Domains are lists of labels:

$$\begin{aligned}&\texttt{domain} : \textbf{Type}.\\ &\texttt{nil} : \texttt{domain}.\\ &\texttt{cons} : \texttt{label} \to \texttt{domain} \to \texttt{domain}.\end{aligned}$$

we will use the notation $[l_1; \ldots; l_n]$ for $(\texttt{cons}\ l_1\ (\ldots(\texttt{cons}\ l_n\ \texttt{nil})\ldots))$.

We avoid assuming that our domains are duplicate-free and we consider relevant proofs of membership of labels. The computational content of such a membership proof is a position in the list where the label appears. We simply call membership proofs *positions*:

$$\begin{aligned}&\bullet \in \bullet : \texttt{label} \to \texttt{domain} \to \textbf{Type}.\\ &\texttt{at-head} : \Pi l.\Pi d.l \in \texttt{cons}\ l\ d.\\ &\texttt{in-tail} : \Pi l_1.\Pi l_2.\Pi d.l_1 \in d \to l_1 \in \texttt{cons}\ l_2\ d.\end{aligned}$$

Most functions in the encoding are defined by induction on positions.

3.1.2 Object types

Types are encoded as sorted association lists. Sorting is done at translation time so we don't need an ordering on labels in the target language.

Formally, we declare the following type and terms:

$$\begin{aligned}&\texttt{type} : \textbf{Type}.\\&\texttt{typenil} : \texttt{type}.\\&\texttt{typecons} : \texttt{label} \to \texttt{type} \to \texttt{type} \to \texttt{type}.\end{aligned}$$

A translation function $[\![\bullet]\!]$ from ς-types to λΠ-terms of type `type`is given by

$$[\![[l_i : A_i]_{i\in 1\ldots n, l_1<\ldots<l_n}]\!] := \texttt{typecons}\ l_1\ [\![A_1]\!]\ (\ldots(\texttt{typecons}\ l_n\ [\![A_n]\!]\ \texttt{typenil})\ldots).$$

For example, the types *RomCell*, *PromCell* and *PrivateCell* defined in subsection 2.4 get translated as follows:

$$\begin{aligned}[\![\mathit{RomCell}]\!] &= \texttt{typecons}\ \mathit{get}\ [\![\mathit{Num}]\!]\ \texttt{typenil}\\ [\![\mathit{PromCell}]\!] &= \texttt{typecons}\ \mathit{get}\ [\![\mathit{Num}]\!]\ (\\ &\qquad \texttt{typecons}\ \mathit{set}\ [\![\mathit{Num} \to \mathit{RomCell}]\!]\\ &\qquad\quad \texttt{typenil})\\ [\![\mathit{PrivateCell}]\!] &= \texttt{typecons}\ \mathit{contents}\ [\![\mathit{Num}]\!]\ (\\ &\qquad \texttt{typecons}\ \mathit{get}\ [\![\mathit{Num}]\!]\ (\\ &\qquad\quad \texttt{typecons}\ \mathit{set}\ [\![\mathit{Num} \to \mathit{RomCell}]\!]\\ &\qquad\qquad \texttt{typenil}))\end{aligned}$$

3.1.3 Design choices

We could have added a proof that the consed label minors the tail of the list as an extra argument to the `typecons` constructor but this increases a lot the size of the translated types so we prefer to live with the existence of λΠ-terms of type `type`not coming from the encoding.

It is also possible to quotient the association lists by a rule exchanging the order of entries:

$$\texttt{typecons}\ l_1\ A_1\ (\texttt{typecons}\ l_2\ A_2\ B) \quad \hookrightarrow \quad \texttt{typecons}\ l_2\ A_2\ (\texttt{typecons}\ l_1\ A_1\ B).$$

In order to preserve normalization, we have to guard this rule by a condition like $l_2 < l_1$. The resulting rewrite system becomes hard to keep confluent so we also avoid this.

3.1.4 assoc and dom

Since types are translated as association lists, we define the usual functions for looking for an association and listing the domain:

$$\begin{aligned}&\texttt{dom} : \texttt{type} \to \texttt{domain}.\\&\texttt{dom}\ \texttt{typenil} &&\hookrightarrow\ \texttt{nil}.\\&\texttt{dom}\ (\texttt{typecons}\ l\ A\ B) &&\hookrightarrow\ \texttt{cons}\ l\ (\texttt{dom}\ B).\end{aligned}$$

$$\begin{aligned}&\texttt{assoc} : \Pi A : \texttt{type}.\Pi l : \texttt{label}.l \in \texttt{dom}\ A \to \texttt{type}.\\&\texttt{assoc}\ (\texttt{typecons}\ l\ A\ B)\ l\ (\texttt{at-head}\ l\ (\texttt{dom}\ B)) &&\hookrightarrow\ A.\\&\texttt{assoc}\ (\texttt{typecons}\ l_2\ A\ B)\ l_1\ (\texttt{in-tail}\ l_1\ l_2\ (\texttt{dom}\ B)\ p) &&\hookrightarrow\ \texttt{assoc}\ B\ l_1\ p.\end{aligned}$$

We will abbreviate $\texttt{assoc}\ A\ l\ p$ as $A._p l$.

3.1.5 Type equality and subtyping relations

The subtyping relation is defined by:

$$\begin{array}{lll} \bullet \leq \bullet : \texttt{type} \to \texttt{type} \to \textbf{Type}. & & \\ A \leq \texttt{typenil} & \hookrightarrow & \texttt{unit}. \\ A \leq \texttt{typecons}\ l\ B\ C & \hookrightarrow & \Sigma p : l \in \texttt{dom}\ A.(A._p l = B) \times (A \leq C). \end{array}$$

We use the notation $d_1 \subset d_2$ as an abbreviation for $\Pi l.l \in d_1 \to l \in d_2$.

3.1.6 Properties of the subtyping relation

Lemma 1 (subtype-weakening)**.** *The $\leq$ relation enjoys weakening; in the λΠ-calculus modulo, we can define a function subtype-weakening of type $\Pi A.\Pi B.\Pi l.\Pi C.A \leq B \to (\texttt{typecons}\ l\ A\ C) \leq B$.*

Proof. Direct by induction on B. □

Lemma 2 (subtype-refl)**.** *The $\leq$ relation is reflexive; in the λΠ-calculus modulo, we can define a function subtype-refl of type $\Pi A.A \leq A$.*

Proof. By induction on A using the previous lemma. □

Lemma 3 (subtype-dom)**.** *The dom function is compatible with $\leq$; in the λΠ-calculus modulo, we can define a function subtype-dom of type $\Pi A.\Pi B.A \leq B \to \texttt{dom}\ B \subset \texttt{dom}\ A$.*

Proof. By induction on B.

- base case is trivial
- if $B = \texttt{typecons}\ l'\ B_1\ B_2$, we have some position $p' : l' \in \texttt{dom}\ A$ and $A \leq B_2$. For any l and any position $p : l \in \texttt{cons}\ l\ (\texttt{dom}\ B_2)$, either p is at head in which case $l = l'$ and p' proves the goal, or p is in tail and we conclude by the induction hypothesis.

□

Lemma 4 (subtype-assoc)**.** *The assoc function is compatible with $\leq$; in the λΠ-calculus modulo, we can define a function subtype-assoc of type $\Pi A.\Pi B.\Pi st : A \leq B.\Pi l.\Pi p : l \in \texttt{dom}\ B.B._p l = A._{\texttt{subtype-dom}\ A\ B\ st\ l\ p} l$.*

Proof. By induction on B.

- base case is trivial
- if $B = \texttt{typecons}\ l'\ B_1\ B_2$, we have some position $p' : l' \in \texttt{dom}\ A$ such that $A._{p'} l' = B_1$ and $A \leq B_2$. For any l and any position $p : l' \in \texttt{cons}\ l\ (\texttt{dom}\ B_2)$,
  - either p is at head in which case $l = l'$ and $B._p l = B_1$. $A._{\texttt{subtype-dom}\ A\ B\ st\ l\ p} l' = A._{p'} l' = B_1$
  - or p is in tail in which case we conclude again by the induction hypothesis.

□

Lemma 5 (subtype-trans)**.** *The subtyping relation is transitive; in the λΠ-calculus modulo, we can define a function subtype-trans of type* $\Pi A.\Pi B.\Pi C.A \leq B \to B \leq C \to A \leq C$.

Proof. By induction on C, using subtype-dom and subtype-assoc. □

3.2 Encoding of terms

As we did for types, we will define translation functions from terms and contexts of the simply-typed ς-calculus to terms and contexts of the λΠ-calculus modulo.

These functions will preserve typing in the sense that we will define, in the λΠ-calculus modulo, a function `Expr` such that whenever the judgment $\Delta \vdash_\varsigma a : A$ is valid in the simply-typed ς-calculus, the judgment $[\![\Delta]\!] \vdash_d [\![a]\!]_{\Delta,A} : \texttt{Expr}\ [\![A]\!]$ is valid in the λΠ-calculus modulo.

3.2.1 Objects

`Expr` $[\![A]\!]$ is the λΠ-type of well-typed objects of type A and `Meth` $[\![A]\!]$ $[\![B]\!]$ is the λΠ-type of methods of A returning a B.

We can declare `Expr` and `Meth`:

$$\texttt{Expr} : \texttt{type} \to \textbf{Type}.$$
$$\texttt{Meth} : \texttt{type} \to \texttt{type} \to \textbf{Type}.$$

Unfortunately, we cannot define `Expr` directly by some nil and cons constructors, as we did for types, because a sublist of a well-typed object is not well-typed.

We call a sublist of a well-typed object of type A, defined on some set of labels d, a *preobject* of type (A, d).

Formally, we define a λΠ-type `Preobj` A d by the following declarations:

$$\texttt{Preobj} : \texttt{type} \to \texttt{domain} \to \textbf{Type}.$$
$$\texttt{prenil} : \Pi A.\texttt{Preobj}\ A\ \texttt{nil}.$$
$$\texttt{precons} : \Pi A.\Pi d.\Pi l.\Pi p : l \in \texttt{dom}\ A.\texttt{Meth}\ A\ A._p l \to \texttt{Preobj}\ A\ d \to \texttt{Preobj}\ A\ (\texttt{cons}\ l\ d).$$

With preobjects at hand, we can define objects of type A:

$$\texttt{Obj}\ A := \texttt{Preobj}\ A\ (\texttt{dom}\ A).$$

and expressions of type B are objects of type A, subtype of B:

$$\texttt{Expr}\ B := \Sigma A : \texttt{type}.(\texttt{Obj}\ A) \times (A \leq B).$$

We would like to define `Meth` A B as `Expr` $A \to$ `Expr` B to close this set of definitions but this is not a positive inductive definition so we would lose normalization.

We solve this problem by adding the following axioms:

$$\begin{aligned}
&\texttt{Meth} && : \texttt{type} \to \texttt{type} \to \textbf{Type}.\\
&\texttt{Eval-meth} && : \Pi A.\Pi B.\texttt{Meth}\ A\ B \to \texttt{Expr}\ A \to \texttt{Expr}\ B.\\
&\texttt{Make-meth} && : \Pi A.\Pi B.(\texttt{Expr}\ A \to \texttt{Expr}\ B) \to \texttt{Meth}\ A\ B.\\
&\texttt{reduce-meth} && : \Pi A.\Pi B.\Pi f.\Pi a.\texttt{Eval-meth}\ A\ B\ (\texttt{Make-meth}\ A\ B\ f)\ a = f\ a.
\end{aligned}$$

The key point here is that the encoding does not preserve reduction but it (axiomatically)

preserves equality of objects so we are a few rewrite rules away from a reduction-preserving encoding.

The translation of a looping ς-term like $[l = \varsigma(x : [l : []])x.l].l$ will be a term whose normalization will freeze at an occurrence of the pattern $\texttt{Eval-meth}\ A\ B\ (\texttt{Make-meth}\ A\ B\ f)\ a$ which will not be matched by any rewrite rule.

3.2.2 Coercions

Implicit subtyping can not be expressed in the λΠ-calculus modulo because each λΠ-term has at most one type modulo β and rewriting. Hence we cannot simply rewrite any type A to any of its subtype or supertype; rewriting is oriented but conversion is symmetric.

Since we cannot use implicit subtyping, we have to define some explicit coercion operation to be used instead of the subsumption typing rule.

These coercions are actually very easy to define thanks to our definition of `Expr` and the lemma subtype-trans:

$$\begin{aligned}&\texttt{coerce} : \Pi B.C : \texttt{type}.B \leq C \to \texttt{Expr}\ B \to \texttt{Expr}\ C.\\ &\texttt{coerce}\ B\ C\ st_{BC}\ (A, a, st_{AB}) \hookrightarrow (A, a, \texttt{subtype-trans}\ st_{AB}\ st_{BC}).\end{aligned}$$

We will use the notation $a \uparrow_A^B$ for the term $\texttt{coerce}\ A\ B\ st\ a$ of type $\texttt{Expr}\ B$, omitting the subtyping proof.

3.2.3 Operational semantics

The select and update functions explore the object until they find the corresponding method and either return it or rebuild another object.

Their definitions follow the definitions of `Expr` and `Obj`; they work recursively on the `Preobj` structure by the way of auxiliary functions called preselect and preupdate. These functions operate on a preobject of type (A, d) and are defined by induction on a position $p : l \in d$ which can be converted to a position of type $l \in \texttt{dom}\ A$ thanks to the following lemma:

Lemma 6 (preobj-subset)**.** *Preobjects are defined on subsets of the domain: in the λΠ-calculus modulo, we can define a function preobj-subset of type* $\texttt{Preobj}\ A\ d \to d \subset \texttt{dom}\ A$.

Proof. Direct by induction. □

The definition of update is not hard:

$\texttt{preupdate} : \Pi A.\Pi d.\Pi l.\Pi p : l \in d.\Pi po : \texttt{Preobj}\ A\ d.\texttt{Meth}\ A\ A._{\texttt{preobj-subset}\ po\ l\ p} l \to \texttt{Preobj}\ A\ d.$
$\texttt{obj-update}\Pi A.\Pi l.\Pi p : l \in \texttt{dom}\ A.\texttt{Obj}\ A \to \texttt{Meth}\ A\ A._p l \to \texttt{Obj}\ A.$
$\texttt{update} : \Pi A.\Pi l.\Pi p : l \in \texttt{dom}\ A.\texttt{Expr}\ A \to \texttt{Meth}\ A\ A._p l \to \texttt{Expr}\ A.$

$\texttt{preupdate}\ A\ (\texttt{cons}\ l\ d)\ l\ (\texttt{at-head}\ l\ d)\ (\texttt{precons}\ A\ d\ l\ p'\ m'\ po)\ m$
$\quad\hookrightarrow \texttt{precons}\ A\ d\ l\ p'\ m\ po.$
$\texttt{preupdate}\ A\ (\texttt{cons}\ l'\ d)\ l\ (\texttt{in-tail}\ l\ l'\ d\ p)\ (\texttt{precons}\ A\ d\ l'\ p'\ m'\ po)\ m$
$\quad\hookrightarrow \texttt{precons}\ A\ d\ l'\ p'\ m'\ (\texttt{preupdate}\ A\ d\ l\ p\ po\ m).$

$\texttt{obj-update}\ A\ l\ p\ a\ m \quad\hookrightarrow\quad \texttt{preupdate}\ A\ (\texttt{dom}\ A)\ l\ p\ a\ m.$

`obj-update` can be used to update a method of an object at type A ; if we want to update an expression at type B where $A \leq B$, we only have at hand a method of type `Meth` B $A.l$ (for some l) where `obj-update` needs a `Meth` A $A.l$. This can be solved by a substitution of the $self$ variable by $self \uparrow_A^B$ in the method body which, in HOAS, is easy to write as (`Make-meth` A $A.l$ ((λ($self : A$) (`Eval-meth` B $A.l$ m ($self \uparrow_A^B$))))).

$$\begin{aligned}
&\texttt{update}\ B\ l\ p\ (A, a, st)\ m\\
&\quad\hookrightarrow (A,\\
&\qquad \texttt{obj-update}\ A\ l\ (\texttt{subtype-dom}\ A\ B\ st\ p\ )\ a\\
&\qquad\quad (\texttt{Make-meth}\ A\ A.l\ (\lambda(self : A)\ (\texttt{Eval-meth}\ B\ A.l\ m\ (\texttt{coerce}\ A\ B\ st\ self))))),\\
&\qquad st).
\end{aligned}$$

Selection is a bit more subtle because it has to perform the substitution by the whole object which is lost by recursive destruction of the object. Hence preselect doesn't return an object but the method associated with the label and the substitution is done by select.

$\texttt{preselect} : \Pi A.\Pi d.\Pi l.\Pi p : l \in d.\texttt{Preobj}\ A\ d \to \texttt{Meth}\ A\ (A._p l)$.
$\texttt{obj-select} : \Pi A.\Pi l.\Pi p : l \in \texttt{dom}\ A.\texttt{Obj}\ A \to \texttt{Meth}\ A\ (A._p l)$.
$\texttt{select} : \Pi A.\Pi l.\Pi p : l \in \texttt{dom}\ A.\texttt{Expr}\ A \to \texttt{Expr}\ A._p l$.

$$\begin{aligned}
&\texttt{preselect}\ A\ (\texttt{cons}\ l\ d)\ l\ (\texttt{at-head}\ l\ d)\ (\texttt{precons}\ A\ d\ l\ p'\ m\ po) &&\hookrightarrow\ m.\\
&\texttt{preselect}\ A\ (\texttt{cons}\ l'\ d)\ l\ (\texttt{in-tail}\ l\ l'\ d\ p)\ (\texttt{precons}\ A\ d\ l'\ p'\ m'\ po) &&\hookrightarrow\ \texttt{preselect}\ A\ d\ l\ p\ po.
\end{aligned}$$

$$\texttt{obj-select}\ A\ l\ p\ a \ \hookrightarrow\ \texttt{preselect}\ A\ (\texttt{dom}\ A)\ l\ p\ a.$$

$$\begin{aligned}
&\texttt{select}\ B\ l\ p\ (A, a, st) \ \hookrightarrow\ \texttt{Eval-meth}\ A\ A._p\ l\\
&\qquad\qquad (\texttt{obj-select}\ A\ l\ p\ a)\ (A, a, st).
\end{aligned}$$

3.2.4 Translation function for expressions

We now have all we need to define a translation function from simply-typed ς terms to the λΠ-calculus modulo.

In fact we will give two such functions; given a typing judgment $\Delta \vdash_\varsigma a : A$, we define $[\![a]\!]_{\Delta,A}$ as $[\![a]\!]_\Delta \uparrow^{[\![A]\!]}_{[\![\mathbf{mintype}_\Delta(a)]\!]}$ where the proof of $[\![\mathbf{mintype}_\Delta(a)]\!] \leq [\![A]\!]$ is computed by a meta-level function **decide-subtype**.

and the $[\![\bullet]\!]_\Delta$ function is defined by:

$$\begin{aligned}
&[\![[l_i = \varsigma(x : A)a_i]_{i\in 1\ldots n, l_1<\ldots<l_n}]\!]_\Delta\\
&\quad := ([\![A]\!],\\
&\qquad \texttt{precons}\ [\![A]\!]\ [l_2; \ldots; l_n]\ l_1\ p_1\ [\![\varsigma(x : A)a_1]\!]_{\Delta,A._{p_1} l_1}\ (\\
&\qquad \ldots(\texttt{precons}\ [\![A]\!]\ [\,]\ l_n\ p_n\ [\![\varsigma(x : A)a_n]\!]_{\Delta,A._{p_n} l_n}\ \texttt{prenil}\ [\![A]\!])),\\
&\qquad \texttt{subtype-refl}\ [\![A]\!])
\end{aligned}$$

$$\begin{aligned}
&[\![a.l]\!]_\Delta := \texttt{select}\ [\![\mathbf{mintype}_\Delta(a)]\!]\ l\ p\ [\![a]\!]_\Delta\\
&[\![a.l \Leftarrow \varsigma(x : A)b]\!]_\Delta := \texttt{update}\ [\![A]\!]\ l\ p\ [\![a]\!]_{\Delta,A}\ [\![\varsigma(x : A)b]\!]_{\Delta,A._p l}
\end{aligned}$$

$$\begin{aligned}
&[\![\varsigma(x : A)b]\!]_{\Delta,B}\\
&\quad := \texttt{Make-meth}\ [\![A]\!]\ [\![B]\!]\ (\lambda x : \texttt{Expr}\ [\![A]\!].[\![b]\!]_{(\Delta,x:A),B})
\end{aligned}$$

The positions p_i and p in this encoding can be computed for any well-typed ς-term : p_i is

the ith position (p_1 is $\texttt{at-head}\ l_1\ [l_2;\ldots;l_n]$, p_2 is $\texttt{in-tail}\ l_2\ l_1\ (\texttt{at-head}\ l_2\ [l_3;\ldots;l_n])$, p_n is $\texttt{in-tail}\ l_n\ l_1\ (\ldots(\texttt{in-tail}\ l_n\ l_{n-1}\ (\texttt{at-head}\ l_n\ [l_n])\ )\ldots)$, and p is the p_i such that l is l_i).

The translation of the binding operation of our source language (the ς binder) is done by a binding operation in the target language (the λ binder).

We can now compute the translation of our example term *myCell*:

$$
\begin{aligned}
&[\![myCell]\!]_{\Delta,PromCell}\\
&\quad = [\![myCell]\!]_\Delta \uparrow^{[\![PromCell]\!]}_{[\![\mathbf{mintype}_\Delta(myCell)]\!]}\\
&\quad = [\![myCell]\!]_\Delta \uparrow^{[\![PromCell]\!]}_{[\![PrivateCell]\!]}\\
&\quad = ([\![PrivateCell]\!],\\
&\qquad \texttt{precons}\ [\![PrivateCell]\!]\ [get;set]\ contents\ p_1\\
&\qquad\quad [\![\varsigma(x:PrivateCell)0]\!]_{\Delta,Num}(\\
&\qquad\quad \texttt{precons}\ [\![PrivateCell]\!]\ [set]\ get\ p_2\\
&\qquad\qquad [\![\varsigma(x:PrivateCell)x.contents]\!]_{\Delta,Num}(\\
&\qquad\qquad \texttt{precons}\ [\![PrivateCell]\!]\ [\,]\ set\ p_3\\
&\qquad\qquad\quad [\![\varsigma(x:PrivateCell)\lambda(n:Num)x.contents \Leftarrow n]\!]_{\Delta,Num\to RomCell}(\\
&\qquad\qquad\quad \texttt{prenil}\ [\![PrivateCell]\!]))),\\
&\qquad \textbf{decide-subtype}\ [\![PrivateCell]\!]\ [\![PromCell]\!])
\end{aligned}
$$

$$
\begin{aligned}
&[\![\varsigma(x:PrivateCell)0]\!]_{\Delta,Num}\\
&\quad = \texttt{Make-meth}\ [\![PrivateCell]\!]\ [\![Num]\!]\\
&\qquad (\lambda x:\texttt{Expr}\ [\![PrivateCell]\!].[\![0]\!]_{(\Delta,x:PrivateCell),Num})
\end{aligned}
$$

$$
\begin{aligned}
&[\![\varsigma(x:PrivateCell)x.contents]\!]_{\Delta,Num}\\
&\quad = \texttt{Make-meth}\ [\![PrivateCell]\!]\ [\![Num]\!]\\
&\qquad (\lambda x:\texttt{Expr}\ [\![PrivateCell]\!].\texttt{select}\ [\![Num]\!]\ contents\ p_1\ x)
\end{aligned}
$$

$$
\begin{aligned}
&[\![\varsigma(x:PrivateCell)\lambda(n:Num)x.contents \Leftarrow n]\!]_{\Delta,Num\to RomCell}\\
&\quad = \texttt{Make-meth}\ [\![PrivateCell]\!]\ [\![Num\to RomCell]\!]\\
&\qquad (\lambda x:\texttt{Expr}\ [\![PrivateCell]\!].[\![\lambda(n:Num)x.contents \Leftarrow n]\!]_{\Delta,Num\to RomCell})
\end{aligned}
$$

3.3 Properties of the encoding

Let Γ_0 be the λΠ-modulo context composed of the declarations and rewrite rules of this section, we investigate properties of the rewrite system $\mathbf{R}$ associated with Γ_0 and of translated ς-terms in contexts of the form Γ_0, Λ where Λ is a λΠ-context (a λΠ modulo context without rewrite rule) so the rewrite system associated with Γ_0, Λ is $\mathbf{R}$.

3.3.1 Normalization and confluence

$\mathbf{R}$ is strongly normalizing because recursive calls are performed on strict subterms and variables of left-hand sides are never applied in the right-hand side. It is also confluent because it is left-linear and normalizing.

Confluence of $\mathbf{R}$ is needed to ensure correctness of the type-checking algorithm for the λΠ-calculus modulo implemented in Dedukti and normalization is needed to ensure termination of this algorithm. These important properties of the rewrite system are currently not checked by Dedukti which is a mere type checker.

In order to be extra-confident in these properties, we implemented the definitions of Δ

in the Calculus of Inductive Constructions, which is known to be strongly normalizing and confluent[8], and type-checked this implementation with Coq.

Our code is available at `https://www.rocq.inria.fr/deducteam/Sigmaid/`. However this translation to Coq is not trustworthy because it uses axioms (`Make-meth`, `Eval-meth` and `reduce-meth`) which are not provable in Coq.

3.3.2 Correctness of the subtyping relation

In this subsection we prove that our translation of types preserves subtyping: given two ς-types A and B, we have $A <: B$ if and only if $[\![A]\!] \leq [\![B]\!]$.

Lemma 7. *if $l \in \texttt{dom}\ [\![[l_i : A_i]_{i\in 1\ldots n}]\!]$ then $l = l_j$ for some $j \in 1\ldots n$.*

Proof. Trivial by induction on the position of type $l \in \texttt{dom}\ [\![[l_i : A_i]_{i\in 1\ldots n}]\!]$. □

Lemma 8. *if $j \in 1\ldots n$, then $l_j \in \texttt{dom}\ [\![[l_i : A_i]_{i\in 1\ldots n}]\!]$.*

Proof. Without loss of generality, we assume that $l_1 > \ldots > l_n$. $\texttt{dom}\ [\![[l_i : A_i]_{i\in 1\ldots n}]\!] = [l_n; \ldots; l_1]$. We prove that $l_j \in [l_n; \ldots; l_1]$ by induction on n:

- case $n = 0$: the hypothesis $j \in 1\ldots n$ is a contradiction.
- case $n = p+1$: if $j = p+1$ then `at-head` $j\ [l_p; \ldots; l_1]$ proves $l_j \in [l_{p+1}; \ldots l_1]$ else $j \in 1\ldots p$ so by induction hypothesis, $l_j \in [l_p; \ldots l_1]$ thus $l_j \in [l_{p+1}; \ldots; l_1]$ by `in-tail`.

□

Lemma 9. *$j \in 1\ldots n \to [\![[l_i : A_i]_{i\in 1\ldots n}]\!].{}_{pos} l_j = [\![A_j]\!]$ where pos is the proof of the previous lemma.*

Proof. This is trivial by following the same steps as the previous lemma. □

Theorem 1. *For every type A and B, if $A <: B$ then $[\![A]\!] \leq [\![B]\!]$.*

Proof. Since $A <: B$, A is some $[l_i : A_i]_{i\in 1\ldots n+m}$ whith $B = [l_i : A_i]_{i\in 1\ldots n}$. Without loss of generality, we may assume $l_n < l_{n-1} < \ldots < l_2 < l_1$. We proceed by induction on n:

- case $n = 0$: $[\![B]\!] = \texttt{typenil}$ hence $[\![A]\!] \leq [\![B]\!]$.
- case $n = p + 1$: $[\![B]\!] = \texttt{typecons}\ l_{p+1}\ [\![A_{p+1}]\!]\ [\![[l_i : A_i]_{i\in 1\ldots p}]\!]$.

$$\begin{aligned}
&[\![A]\!] \leq [\![B]\!] \\
&\quad = [\![A]\!] \leq \texttt{typecons}\ l_{p+1}\ [\![A_{p+1}]\!]\ [\![[l_i : A_i]_{i\in 1\ldots p}]\!] \\
&\quad = \Sigma pos : l_{p+1} \in \texttt{dom}\ [\![A]\!]. \\
&\qquad ([\![A]\!].{}_{pos} l_{p+1} = A_{p+1}) \times ([\![A]\!] \leq [\![[l_i : A_i]_{i\in 1\ldots p}]\!])
\end{aligned}$$

pos and the equality proof are given by the lemmata. The proof of $[\![A]\!] \leq [\![[l_i : A_i]_{i\in 1\ldots p}]\!]$ is given by the induction hypothesis.

□

Theorem 2. *The translation function on types is injective: if $[\![A]\!] = [\![B]\!]$ then $A = B$.*

Proof. A type and its encoding have the same size so A and B have the same size. The proof is by induction on this common size; both case are trivial. □

Theorem 3. *For every type A and B, if $[\![A]\!] \leq [\![B]\!]$ then $A <: B$.*

Proof. By induction on the size n of $B = [l_i : B_i]_{l_1 > \ldots > l_n}$.

- case $n = 0$: $B = [\,]$ hence $A <: B$.
- case $n = p + 1$: $[\![B]\!] = \texttt{typecons}\ l_{p+1}\ [\![B_{p+1}]\!]\ [\![[l_i : B_i]_{l_1 > \ldots > l_p}]\!]$. Our hypothesis simplifies to:

$$\begin{aligned}&[\![A]\!] \leq [\![B]\!]\\ &= [\![A]\!] \leq \texttt{typecons}\ l_{p+1}\ [\![B_{p+1}]\!]\ [\![[l_i : B_i]_{l_1 > \ldots > l_p}]\!]\\ &= \Sigma pos : l_{p+1} \in \texttt{dom}\ [\![A]\!].\\ &\quad ([\![A]\!].{}_{pos} l_{p+1} = [\![B_{p+1}]\!]) \times \left([\![A]\!] \leq [\![[l_i : B_i]_{l_1 > \ldots > l_p}]\!]\right)\end{aligned}$$

By induction hypothesis, A is of the form $[l_i : B_i; l'_j : A_j]_{i \in 1 \ldots p, j \in 1 \ldots m+1}$. From the lemmata and the injectivity theorem, we get $l_{p+1} = l'_j$ and $A_j = B_{p+1}$ for some $j \in 1 \ldots m+1$. By renaming the l's, we can choose $j = m + 1$ and we get $A = [l_i : B_i; l'_j : A_j]_{i \in 1 \ldots p, j \in 1 \ldots m}$ so $A <: B$. □

3.3.3 Type preservation

We want to prove the following type preservation theorem:

Theorem 4. *If, in the simply typed ς-calculus, the judgment $\Delta \vdash_\varsigma a : A$ is valid, then the encoded judgment $[\![\Delta]\!] \vdash_d [\![a]\!]_{\Delta,A} : \texttt{Expr}\ [\![A]\!]$, is valid in the λΠ-calculus modulo.*

For this, we first need to define $[\![\Delta]\!]$:

$$\begin{aligned}[\![\emptyset]\!] &:= \Gamma_0\\ [\![\Delta, x : A]\!] &:= [\![\Delta]\!], x : \texttt{Expr}\ [\![A]\!]\end{aligned}$$

Since the translation function $[\![\bullet]\!]_{\Delta,A}$ is recursively defined together with $[\![\bullet]\!]_\Delta$ and the translation function for methods, we need lemmata to relate these three functions:

Lemma 10. *If, in the simply typed ς-calculus, the judgment $\Delta \vdash_\varsigma a : A$ is valid, and, in the λΠ-calculus modulo, the judgment $[\![\Delta]\!] \vdash_d [\![a]\!]_\Delta : \texttt{Expr}\ [\![\mathbf{mintype}_\Delta\,(a)]\!]$ is valid, then so is the judgment $[\![\Delta]\!] \vdash_d [\![a]\!]_{\Delta,A} : \texttt{Expr}\ [\![A]\!]$.*

Proof. From $\Delta \vdash_\varsigma a : A$ we get, by minimality, $\mathbf{mintype}_\Delta\,(a) <: A$ hence $[\![\mathbf{mintype}_\Delta\,(a)]\!] \leq [\![A]\!]$ by Theorem 1. So $[\![a]\!]_{\Delta,A} = [\![a]\!]_\Delta \uparrow^{[\![A]\!]}_{[\![\mathbf{mintype}_\Delta(a)]\!]}$ has type $\texttt{Expr}\ [\![A]\!]$. □

Lemma 11. *If, in the λΠ-calculus modulo, the judgment $[\![\Delta]\!], x : \texttt{Expr}\ [\![A]\!] \vdash_d [\![b]\!]_{(\Delta, x:A), B} : \texttt{Expr}\ [\![B]\!]$ is valid, then so is the judgment $[\![\Delta]\!] \vdash_d [\![\varsigma(x : A)b]\!]_{\Delta,B} : \texttt{Meth}\ [\![A]\!]\ [\![B]\!]$.*

Proof. x doesn't occur free in $[\![B]\!]$ because it is a closed term.

Hence we can type the λ-abstraction with an arrow type: $[\![\Delta]\!] \vdash_d \lambda x : \texttt{Expr}\ [\![A]\!].[\![b]\!]_{(\Delta, x:A), B} : \texttt{Expr}\ [\![A]\!] \to \texttt{Expr}\ [\![B]\!]$.

So $[\![\varsigma(x : A)b]\!]_{\Delta,B} = \texttt{Make-meth}\ [\![A]\!]\ [\![B]\!]\ (\lambda x : \texttt{Expr}\ [\![A]\!].[\![b]\!]_{(\Delta, x:A), B})$ has type $\texttt{Meth}\ [\![A]\!]\ [\![B]\!]$. □

Theorem 5. *If, in the simply typed ς-calculus, the judgment $\Delta \vdash_\varsigma a : A$ is valid, then the judgment $[\![\Delta]\!] \vdash_d [\![a]\!]_\Delta : \texttt{Expr}\ [\![\mathbf{mintype}_\Delta\,(a)]\!]$ is valid in the λΠ-calculus modulo.*

Proof. By minimality, $\Delta \vdash_\varsigma a : \mathbf{mintype}_\Delta\,(a)$. We proceed by induction on this typing derivation; we have one case for each typing rule in the simply-typed ς-calculus:

- case (var): a is a variable x appearing in Δ and $\mathbf{mintype}_\Delta(a) = \mathbf{mintype}_\Delta(x) = \Delta(x)$. By definition of $[\![\Delta]\!]$, $x \in [\![\Delta]\!]$ and $[\![\Delta]\!](x) = \texttt{Expr}\ [\![\Delta(x)]\!] = \texttt{Expr}\ [\![\mathbf{mintype}_\Delta(a)]\!]$.

- case (obj): $a = [l_i = \varsigma(x_i : A)\ a_i]_{l_1<\ldots<l_n}$ with $\mathbf{mintype}_\Delta(a) = A = [l_i : A_i]_{l_1<\ldots<l_n}$.

  $$\begin{aligned}[\![a]\!]_\Delta &= [\![[l_i = \varsigma(x_i : A)\ a_i]_{l_1<\ldots<l_n}]\!]_\Delta \\ &= ([\![A]\!], \\ &\qquad \texttt{precons}\ [\![A]\!]\ [l_2;\ldots;l_n]\ l_1\ p_1\ [\![\varsigma(x : A)a_1]\!]_{\Delta,A.p_1 l_1}\ (\\ &\qquad \ldots(\texttt{precons}\ [\![A]\!]\ [\,]\ l_n\ p_n\ [\![\varsigma(x : A)a_n]\!]_{\Delta,A.p_n l_n}\ \texttt{prenil}\ [\![A]\!])), \\ &\qquad \texttt{subtype-refl}\ [\![A]\!])\end{aligned}$$

  The term $\texttt{subtype-refl}\ [\![A]\!]$ has type $[\![A]\!] \leq [\![A]\!]$ so we just need to check that $\texttt{precons}\ [\![A]\!]\ [l_2;\ldots;l_n]\ l_1\ p_1\ [\![\varsigma(x : A)a_1]\!]_{\Delta,A.p_1 l_1}\ (\ldots(\texttt{precons}\ [\![A]\!]\ [\,]\ l_n\ p_n\ [\![\varsigma(x : A)a_n]\!]_{\Delta,A.p_n l_n}\ \texttt{prenil}\ [\![A]\!]))$ has type $\texttt{Obj}\ [\![A]\!]$.

  - To compute $\texttt{Obj}\ [\![A]\!]$, we first compute $\texttt{dom}\ [\![A]\!]$:

    $$\begin{aligned}\texttt{dom}\ [\![A]\!] &= \texttt{dom}\ [\![[l_i : A_i]_{l_1<\ldots<l_n}]\!] \\ &= \texttt{dom}\ (\texttt{typecons}\ l_1\ [\![A_1]\!]\ (\ldots(\texttt{typecons}\ l_n\ [\![A_n]\!]\ \texttt{typenil})\ldots)) \\ &= [l_1;\ldots;l_n]\end{aligned}$$

    hence $\texttt{Obj}\ [\![A]\!] = \texttt{Preobj}\ [\![A]\!]\ (\texttt{dom}\ [\![A]\!]) = \texttt{Preobj}\ [\![A]\!]\ [l_1;\ldots;l_n]$.

  - We show by induction that each built preobject is well-typed with the expected type. For all $i \in 1\ldots n$,

    $$\begin{aligned}[\![\Delta]\!] \vdash_d\ & \texttt{precons}\ [\![A]\!]\ [l_{i+1};\ldots;l_n]\ l_i\ p_i\ [\![\varsigma(x : A)a_i]\!]_{\Delta,A.p_i l_i}\ (\\ & \ldots(\texttt{precons}\ [\![A]\!]\ [\,]\ l_n\ p_n\ [\![\varsigma(x : A)a_n]\!]_{\Delta,A.p_n l_n}\ \texttt{prenil}\ [\![A]\!])) \\ & \quad : \texttt{Preobj}\ [\![A]\!]\ [l_i;\ldots;l_n]\end{aligned}$$

    This is trivial by decreasing recursion on i.

  Finally $[\![\Delta]\!] \vdash_d [\![[l_i = \varsigma(x_i : A)\ a_i]_{l_1<\ldots<l_n}]\!]_\Delta : \texttt{Expr}\ [\![A]\!]$.

- case (select): a is of the form $a = a'.l_j$ with $j \in 1\ldots n$ and $\Delta \vdash_\varsigma a' : A'$ where $A' = [l_i : A_i]_{i\in 1\ldots n}$. Without loss of generality, we can assume that A' is the minimal type of a':

  $\mathbf{mintype}_\Delta(a') = [l_i : A_i]_{i\in 1\ldots n}$ so $\mathbf{mintype}_\Delta(a) = A_j$.

  Lemma 8 gives us a position $p : l_j \in \texttt{dom}\ [\![\mathbf{mintype}_\Delta(a')]\!]$ so by **??**, $[\![\mathbf{mintype}_\Delta(a')]\!]._p l_j = [\![A_j]\!] = [\![\mathbf{mintype}_\Delta(a)]\!]$.

  Moreover, $\Delta \vdash_\varsigma [\![a']\!]_\Delta : \texttt{Expr}\ [\![\mathbf{mintype}_\Delta(a')]\!]$ by induction hypothesis so $[\![a]\!]_\Delta = \texttt{select}\ [\![\mathbf{mintype}_\Delta(a')]\!]\ l_j\ p\ [\![a']\!]_\Delta$ has type $\texttt{Expr}\ [\![\mathbf{mintype}_\Delta(a)]\!]$.

- case (update): a is of the form $a = a'.l_j \Leftarrow \varsigma(x : A)b$ with $j \in 1\ldots n$, $\Delta \vdash_\varsigma a' : A$ and $\Delta, x : A \vdash_\varsigma b : A_j$ where $A = [l_i : A_i]_{i\in 1\ldots n}$.

  By induction hypothesis and Lemma 10, $[\![\Delta]\!] \vdash_d [\![a']\!]_{\Delta,A} : \texttt{Expr}\ A$. By Lemma 11, $[\![\Delta]\!] \vdash_d [\![\varsigma(x : A)b]\!]_{\Delta,A_j} : \texttt{Meth}\ [\![A]\!]\ [\![A_j]\!]$.

  Like in the previous case, Lemma 8 gives us a position $p : l_j \in \texttt{dom}\ [\![A]\!]$ and by **??**, $[\![A]\!]._p l_j = [\![A_j]\!]$.

  Hence $[\![a]\!]_\Delta = \texttt{update}\ [\![A]\!]\ l_j\ p\ [\![a']\!]_{\Delta,A}\ [\![\varsigma(x : A)b]\!]_{\Delta,A_j}$ has type $[\![A]\!] = [\![\mathbf{mintype}_\Delta(a)]\!]$.

- case (subsume): The only possible instantiation of the subsumption rule which derives a minimum typing is the trivial case

$$\frac{\Delta \vdash_\varsigma a : \mathbf{mintype}_\Delta(a) \qquad \mathbf{mintype}_\Delta(a) <: \mathbf{mintype}_\Delta(a)}{\Delta \vdash_\varsigma a : \mathbf{mintype}_\Delta(a)}\ (\text{subsume})$$

In this case, our goal is exactly the induction hypothesis $[\![\Delta]\!] \vdash_d [\![a]\!]_\Delta : \mathtt{Expr}\ [\![\mathbf{mintype}_\Delta(a)]\!]$.

□

3.3.4 Semantics preservation

Lemma 12. *The translation function is stable by substitution:* $[\![a]\!]_{(\Delta_1,x:B),A}\{[\![b]\!]_{(\Delta_1,\Delta_2),B}/x\} = [\![a\{b/x\}]\!]_{(\Delta_1,\Delta_2),A}$.

Proof. This comes from the fact that binding operation is preserved by the encoding. This can be proved by induction on a. □

Lemma 13. *Unicity of subtype proofs: if* st_1 *and* st_2 *both have type* $[\![A]\!] \leq [\![B]\!]$ *then* $st_1 = st_2$.

This lemma justifies our use of implicit subtype proofs in the notation $\bullet \uparrow^{[\![\bullet]\!]}_{[\![\bullet]\!]}$.

Proof. Unicity of subtype proofs comes from the fact that $[\![A]\!]$ is duplicate-free. We don't use, however, the fact that $[\![B]\!]$ is duplicate-free and prove this theorem for any $\overline{B}$ of type `type`: if st_1 and st_2 both have type $[\![A]\!] \leq \overline{B}$ then $st_1 = st_2$.

We proceed by induction on $\overline{B}$.

- base case: $\overline{B} = \mathtt{typenil}$

  $[\![A]\!] \leq \overline{B} = [\![A]\!] \leq \mathtt{typenil} = \mathtt{unit}$. The type `unit` has only one inhabitant so $st_1 = st_2$.

- inductive case: $\overline{B} = \mathtt{typecons}\ l\ \overline{B_1}\ \overline{B_2}$

  A is some $[l_i : A_i]_{l_1<\ldots<l_n}$.

  By definition of $\leq$,

  $$[\![A]\!] \leq \mathtt{typecons}\ l\ \overline{B_1}\ \overline{B_2} = \Sigma p : l \in [l_1;\ldots;l_n].([\![A]\!].{}_p l = \overline{B_1}) \times ([\![A]\!] \leq \overline{B_2})$$

  But there is only one $p : l \in [l_1;\ldots;l_n]$ because the l_is are different. Let us call it p_0.

  $[\![A]\!] \leq \mathtt{typecons}\ l\ \overline{B_1}\ \overline{B_2}$ is isomorphic to $([\![A]\!].{}_{p_0} = \overline{B_1}) \times ([\![A]\!] \leq \overline{B_2})$.

  The left type $[\![A]\!].{}_{p_0} = \overline{B_1}$ has at most one inhabitant because equality on `type` is decidable; the right type $[\![A]\!] \leq \overline{B_2}$ has only one element by induction hypothesis.

□

Theorem 6. *If* $\Delta \vdash_\varsigma a : A$ *and* $a \rightarrowtail a'$ *then* $[\![a]\!]_{\Delta,A} = [\![a']\!]_{\Delta,A}$ *is inhabited in context* $[\![\Delta]\!]$.

Proof. The simply-typed ς-calculus is known to be Church-Rosser[1] so $\Delta \vdash_\varsigma a' : A$. From the type-preservation theorem, $[\![a]\!]_{\Delta,A}$ and $[\![a']\!]_{\Delta,A}$ have type $\mathtt{Expr}\ [\![A]\!]$ in context $[\![\Delta]\!]$.

We proceed by induction on the operational semantics definition; there are two cases:

- case (select): $a \rightarrowtail a'$ is an instance of $a''.l_j \rightarrowtail a_j\{a''/x_j\}$ with $a'' = [l_i = \varsigma(x_i : A'')a_i]_{i=1\ldots n}$ and $A'' = [l_i : A_i]_{i=1\ldots n}$. So $a = a''.l_j$ and $a' = a_j\{a''/x_j\}$.

  We look at the minimum types of a and a':

  - $\mathbf{mintype}_\Delta(a'') = A'' = [l_i : A_i]_{i=1\ldots n}$ so $\mathbf{mintype}_\Delta(a) = \mathbf{mintype}_\Delta(a''.l_j) = A_j$

– We call A' the minimum type of a', by minimality we know that $A' <: A_j$.

$[\![a'']\!]_\Delta = ([\![A'']\!], \alpha, \texttt{subtype-refl}\ [\![A'']\!])$ where α is a term of the form $(\dots(\texttt{precons}\ [\![A'']\!]\ [l_{j+1};\dots;l_n]\ l_j\ [\![\varsigma(x_j : A'')a_j]\!]_{\Delta,A_j}\dots\ )\ )$.

$$\begin{array}{lll}
[\![a]\!]_\Delta & = & \texttt{select}\ [\![A'']\!]\ l_j\ p\ [\![a'']\!]_\Delta \\
& \hookrightarrow & \texttt{Eval-meth}\ [\![A'']\!]\ [\![A_j]\!]\ (\texttt{obj-select}\ [\![A'']\!]\ l_j\ p\ \alpha)\ [\![a'']\!]_\Delta \\
& \hookrightarrow & \texttt{Eval-meth}\ [\![A'']\!]\ [\![A_j]\!]\ (\texttt{preselect}\ [\![A'']\!]\ [l_1;\dots;l_n]\ l_j\ p\ \alpha)\ [\![a'']\!]_\Delta \\
& \hookrightarrow^* & \texttt{Eval-meth}\ [\![A'']\!]\ [\![A_j]\!]\ [\![\varsigma(x_j : A'')a_j]\!]_{\Delta,A_j}\ [\![a'']\!]_\Delta \\
& =_{\texttt{reduce-meth}} & (\lambda x_j : \texttt{Expr}\ [\![A'']\!].[\![a_j]\!]_{(\Delta,x_j:A''),A_j})[\![a'']\!]_\Delta \\
& \longrightarrow_\beta & [\![a_j]\!]_{(\Delta,x_j:A''),A_j}\{[\![a'']\!]_\Delta/x_j\} \\
& =_{Lemma\ 12} & [\![a_j\{a''/x_j\}]\!]_{\Delta,A_j} \\
& = & [\![a']\!]_{\Delta,A_j}
\end{array}$$

the last equality is from the substitution lemma.

Finally,

$$\begin{array}{lll}
[\![a]\!]_{\Delta,A} & = [\![a]\!] \uparrow^{[\![A]\!]}_{[\![\mathbf{mintype}_\Delta(a)]\!]} & \\
& = [\![a']\!]_{\Delta,A_j} \uparrow^{[\![A]\!]}_{[\![A_j]\!]} & \\
& = \left([\![a']\!]_\Delta \uparrow^{[\![A_j]\!]}_{[\![\mathbf{mintype}_\Delta(a')]\!]}\right) \uparrow^{[\![A]\!]}_{[\![A_j]\!]} & \\
& = [\![a']\!]_\Delta \uparrow^{[\![A]\!]}_{[\![\mathbf{mintype}_\Delta(a')]\!]} & \text{(by unicity of subtyping proofs)} \\
& = [\![a']\!]_{\Delta,A} &
\end{array}$$

- case (update): this case is very similar to the previous one, only simpler because we don't need to use `reduce-meth` or the substitution lemma.

□

4 Shallow, non-terminating encoding

In this section, we trade strong-normalization for a shallow encoding.

In order to get a shallow encoding, we want to replace the declaration

$$\texttt{reduce-meth} : \Pi A.\Pi B.\Pi f.\Pi a.\texttt{Eval-meth}\ A\ B\ (\texttt{Make-meth}\ A\ B\ f)\ a = f\ a.$$

by the following rewrite rule

$$\texttt{Eval-meth}\ A\ B\ (\texttt{Make-meth}\ A\ B\ f)\ a \hookrightarrow f\ a.$$

We keep everithing else identical so we call Γ_1 the new context. $\mathbf{R}$' is the new rewrite system ($\mathbf{R}' = \mathbf{R} \cup \{\texttt{Eval-meth}\ A\ B\ (\texttt{Make-meth}\ A\ B\ f)\ a \hookrightarrow f\ a\}$).

$\mathbf{R}$' is a confluent system because the added rewrite rule is orthogonal to the other ones. However, it is not expected to be (strongly or even weakly) normalizing. Hence Dedukti will type-check encoded object programs only if they are well-typed but may not answer on non-terminating terms (actually it will terminate because conversion check, which triggers reduction, only occurs in types).

Proofs of the theorems of the previous section are unchanged and we can restate the semantics preservation as

Theorem 7. *If* $\Delta \vdash_\varsigma a : A$ *and* $a \rightarrowtail a'$ *then* $[\![a]\!]_{\Delta,A} \hookrightarrow^+ [\![a']\!]_{\Delta,A}$.

from which it is clear that **R**' is not normalizing since the simply-typed ς-calculus is not normalizing.

5 Implementation

The encoding described in the previous sections has been implemented as a compiler named sigmaid (SIGMA-calculus In Dedukti) from the simply-typed ς-calculus to Dedukti. It is available at `https://www.rocq.inria.fr/deducteam/Sigmaid/`.

This implementation has been tested on the original examples from Abadi and Cardelli:

- encoding of the simply-typed λ-calculus
- encoding of booleans
- memory cells

Dedukti successfully type-check all these examples in a fraction of a second.

Conclusion

We defined an embedding of the simply-typed ς-calculus to the λΠ-calculus modulo and implemented it in Dedukti.

Despite non-termination of the ς-calculus, we managed to translate it in a very shallow fashion by the mean of two very close encoddings: a normalizing one and a semantics-preserving one.

This embedding is a starting point for other shallow embeddings of typed object oriented calculi with subtyping.

Beside common extensions for object type systems (polymorphism, variance annotations, type operators), we are especially interested in extending this work to object type systems with dependent types in order to study dependently-typed objects combining computational methods and logical methods which depend upon them and prove their specifications.

We would also like to encode class-based calculi like Featherweight Java[13] in the λΠ-calculus modulo in order to compare the encoded versions of structural subtyping and class-based subtyping.

Comparison with other forms of subtyping like universe cumulativity and predicate subtyping will also be under investigation in order to ease interoperability between object oriented systems, Coq and PVS.

Acknowledgment

We would like to thank our colleague Ali Assaf for the fruitful discussions which led to this implementation of subtyping in the λΠ-calculus modulo.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer New York, 1996.

[2] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *In Proc. TACS94, Theoretical Aspects of Computing Sofware*, pages 296–320, 1994.

[3] Ali Assaf and Guillaume Burel. Holide. `https://www.rocq.inria.fr/deducteam/Holide/index.html`.

[4] M. Boespflug and G. Burel. Coqine : Translating the calculus of inductive constructions into the λπ-calculus modulo. In *Proceedings of the Second International Workshop on Proof Exchange for Theorem Proving*, 2012.

[5] Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Matching Power. In *Proceedings of RTA'2001*, Lecture Notes in Computer Science. Springer-Verlag, May 2001.

[6] Horatiu Cirstea, Luigi Liquori, and Benjamin Wack. Rewriting calculus with fixpoints: Untyped and first-order systems. volume 3085. Springer, 2003.

[7] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. Manual distributed as documentation of the Maude system, Computer Science Laboratory, SRI International. `http://maude.csl.sri.com/manual`, January 1999.

[8] The Coq Development Team. *The Coq Reference Manual, version 8.4*, August 2012. Available electronically at `http://coq.inria.fr/doc`.

[9] D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In S. Ronchi Della Rocca, editor, *TLCA*, volume 4583 of *LNCS*, pages 102–117. Springer, 2007.

[10] Kathleen Fisher, Furio Honsell, and John C. Mitchell. A lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1:3–37, 1994.

[11] J. Nathan Foster and Dimitrios Vytiniotis. A theory of featherweight java in isabelle/hol. *Archive of Formal Proofs*, March 2006. `http://afp.sf.net/entries/FeatherweightJava.shtml`, Formal proof development.

[12] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993.

[13] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java - a minimal core calculus for java and gj. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999.

[14] Julian Mackay, Hannes Mehnert, Alex Potanin, Lindsay Groves, and Nicholas Cameron. Encoding featherweight java with assignment and immutability using the coq proof assistant. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 11–19, New York, NY, USA, 2012. ACM.

[15] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer. *Selected papers on Automath*. Elsevier, Amsterdam, 1994.

[16] Frank Pfenning and Carsten Schurmann. System description: Twelf — a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206. Springer-Verlag LNAI, 1999.

[17] R. Saillard. Dedukti: a universal proof checker. In *Foundation of Mathematics for Computer-Aided Formalization Workshop*, Padova, 2013.

[18] Jan Zwanenburg. *Object-Oriented Concepts and Proof Rules: Formalization in Type Theory and Implementation in Yarrow*. PhD thesis, Eindhoven University of Technology, 1999.