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 with rewriting. This embedding may be used as an example of translation of subtyping. We prove this embedding correct with respect to the operational semantics and the type system of the Object Calculus. We implemented a translation tool from the Object Calculus to Dedukti, a type-checker for the $\lambda\Pi$-calculus modulo.

1998 ACM Subject Classification F.4.1 Lambda calculus and related systems

Keywords and phrases object, calculus, encoding, dependent type, rewrite system

1 Introduction

Motivation. The $\lambda\Pi$-calculus modulo [16] ($\lambda\Pi m$) 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 [16]. Moreover, translation tools from real-world proof assistants like Coq [15, 4] and the HOL family [3] to Dedukti [6], a type-checker for $\lambda\Pi m$, allow for the verification of proofs done in these complex systems using a small, easy to trust, checker.

In this paper we present an encoding of an object calculus in $\lambda\Pi m$, more precisely the simply-typed $\varsigma$-calculus [2]. A major feature of object oriented type systems is subtyping, and it will be the focus of this article. The simply-typed $\varsigma$-calculus is the simplest object calculus featuring subtyping. We chose it as our source language to understand the special case of structural object subtyping to be compared with other forms of subtyping 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 object oriented concepts and mechanisms such as inheritance, method redefinition and late binding. FoCaLiZe [24] is a logical system featuring class-based object mechanisms which are translated in $\lambda\Pi m$ [9].

In order to generalize this encoding of objects in $\lambda\Pi m$ to more primitive object-based mechanisms, we would need complex objects where methods would be typed with dependent types. This work is a first step in that direction starting from a very simple type-system for objects.

Related work. Many encodings [31, 7] 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 object oriented languages or to implement object oriented mechanisms in proof systems; the only implementation to our knowledge is the Yarrow proof assistant [32]. However, following the example of the $\lambda$-calculus, small calculi taking objects as core notions have been designed and their type systems have been proved safe. For example:

- The $\lambda$-calculus of objects [18] is an extension of the simply-typed $\lambda$-calculus with object construction, method call, and method update. In this system, objects are extended with their types using extensible records.
- The Object Calculi of Abadi and Cardelli [2] are a collection of calculi based on objects. They differ from the $\lambda$-calculus of objects in two important ways: they are not based on the $\lambda$-calculus so they have fewer constructs and objects and their types are fixed records. Hence they are somewhat simpler but they still are very expressive.
- Featherweight Java [25] is a core calculus for the popular class-based Java programming language. It is a small class-based object oriented calculus designed to study extensions of class-based languages such as Java.

These three calculi can easily encode the $\lambda$-calculus, allow possibly non-terminating recursion and have some form of subtyping: respectively row polymorphism, structural subtyping, and class-based subtyping.

The type systems of these three calculi have been formalized in proof assistants; those formalizations can be seen as deep embeddings of the calculi in type theory. For example, Featherweight Java has been formalized in Coq [26] and Isabelle/HOL [19] using extensible records and subject reduction for Object Calculi has been proved in Coq [11] and Isabelle/HOL [23]. For the untyped Object Calculus, confluence has also been formally proved in Isabelle/HOL [22].

Encodings of objects based on rewrite techniques have also been studied; for example, in the $\rho$-calculus [13], a full encoding of the untyped Object Calculus and $\lambda$-calculus of objects [12] and a partial encoding of the simply-typed Object Calculus [13] have been designed. In the Maude specification environment [14], 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 in the sense of [8, 5, 17]; the elements of the source language, the simply-typed $\varsigma$-calculus: terms, values, and types are respectively translated to terms, values, and types in $\lambda\Pi m$ such that operational semantics, typing derivations, and binding operation are preserved by this translation.

The next section of this article describes $\lambda\Pi m$, our target language, Section 3 describes the simply-typed $\varsigma$-calculus, our source language. Section 4 is the main section of this article; it defines a strongly-normalizing encoding of the simply-typed $\varsigma$-calculus in $\lambda\Pi m$. This encoding is not fully shallow because it does not preserve the operational semantics. In Section 5, we add two rewrite rules to this encoding to reflect the operational semantics; doing so we lose strong-normalization.

## 2 The $\lambda\Pi$-calculus modulo

### 2.1 The $\lambda\Pi$-calculus

The $\lambda\Pi$-calculus [20], also known as LF and $\lambda P$, is an extension of the simply typed $\lambda$-calculus with dependent types. $\lambda\Pi$ terms and types have the following syntax:
\[ s \in \{ \text{Type, Kind} \} \]

\[
\begin{array}{c}
\emptyset \vdash_d \quad (\text{Empty}) \\
\Gamma \vdash_d \quad \text{Decl} \\
\Gamma, x : \tau \vdash_d x \not\in \Gamma \\
\Gamma \vdash_d \quad \text{Sort} \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash_d x : \tau \in \Gamma \\
\Gamma \vdash_d x : \tau \\
\Gamma, x : \tau_1 \vdash_d \tau_2 : s \\
\Gamma, x : \tau_1 \vdash_d t : \tau_2 \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash_d \tau_1 : \text{Type} \\
\Gamma, x : \tau_1 \Gamma, x : \tau_2 \vdash \tau_1 \Gamma, x : \tau_2 \vdash s \\
\Gamma \vdash_d \Pi x : \tau_1.\tau_2 \\
\Gamma \vdash_d \Pi x : \tau_1.\tau_2 \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash_d \lambda x : \tau_1. t : \Pi x : \tau_1.\tau_2 \\
\Gamma \vdash_d t_0 : \Pi x : \tau_1.\tau_2 \\
\Gamma \vdash_d t_1 : \tau_1 \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash_d t_1 : \tau_1 \Gamma \vdash_d \tau_1 : s \\
\Gamma \vdash_d \tau_2 : s \\
\tau_1 \equiv_\beta \tau_2 \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash_d \tau_2 \\
\end{array}
\]

**Figure 1** inference rules for the \(\lambda\Pi\)-calculus.

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

There is no syntactic distinction between terms and types but we use latin letters starting at \(t\) to denote terms and the greek letter \(\tau\) 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 \(\tau_2\) is called a dependent product and represents the type of functions taking an argument \(x\) of type \(\tau_1\) and returning a value of type \(\tau_2\) that may depend on \(x\). If \(x\) does not appear free in \(\tau_2\), the term \(\Pi x : \tau_1.\tau_2\) will be abbreviated as \(\tau_1 \to \tau_2\). If \(\tau_1\) is clear from context, the term \(\Pi x : \tau_1.\tau_2\) will be abbreviated as \(\Pi x.\tau_2\).

A list of variable typing declarations is called a (\(\lambda\Pi\)) context:

\[ \Gamma ::= \emptyset \mid \Gamma, x : \tau \]

where \(\emptyset\) denotes the empty context. We implicitly use \(\alpha\)-conversion to avoid variable capture. In particular, contexts contain distinct variables.

Some contexts are called well-formed. When the context \(\Gamma\) is well-formed, we write \(\Gamma \vdash_d\). Some terms are called well-typed. When the term \(t\) is well-typed of type \(\tau\) in context \(\Gamma\), we write \(\Gamma \vdash_d t : \tau\). These two notions are mutually defined in Figure 1 where \(t_0[t_1/x]\) denotes the capture-avoiding substitution of the variable \(x\) by the term \(t_1\) in term \(t_0\) and \(\equiv_\beta\) is the congruence induced by \(\beta\)-reduction (the smallest congruence such that \((\lambda x : \tau_1.t_0) t_1 \equiv_\beta t_0\{t_1/x\}\)).

The \(\lambda\Pi\)-calculus is the type-system on which logical frameworks such as Automath [28] and Twelf [30] are based.

### 2.2 The \(\lambda\Pi\)-calculus modulo

The \(\lambda\Pi\)-calculus modulo (\(\lambda\Pi_m\)) is an extension of the \(\lambda\Pi\)-calculus which extends the conversion rule; terms are considered convertible not only when they are \(\beta\)-equivalent but also when they are congruent for a given rewrite system.

The terms are the same as in the \(\lambda\Pi\)-calculus but contexts may also contain rewrite rules which also need to be well-typed.
Objects and Subtyping in the $\lambda\Pi$-Calculus Modulo

Rewrite rules are composed of three parts: a rule context which is a $\lambda\Pi$ 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 decidable\(^1\), we need to add the following restrictions on rewrite rules:

- the left-hand side is a first-order pattern (a term built only from variables and applications)
- 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:

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

where $\Lambda$ stands for $\lambda\Pi$ contexts. The rule context $\Lambda$ will often be omitted when clear from context.

For any context $\Gamma$, a reduction relation on terms $\rightarrow_{\beta\Gamma}$ is defined by:

- for any terms $t_1, t_2$ and any variable $x$, $(\lambda x.t_1)t_2 \rightarrow_{\beta\Gamma} t_1[t_2/x]$  
- for any rule $(\Lambda l \rightarrow r) \in \Gamma$ and any substitution $\theta$ of the variables of $\Lambda$, $\theta l \rightarrow_{\beta\Gamma} \theta r$.

We denote by $\equiv_{\beta\Gamma}$ the smallest congruence containing $\rightarrow_{\beta\Gamma}$.

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 $\Gamma$ if the left-hand side and the right-hand side have the same type in $\Gamma$, $\Lambda$ ($\Gamma$ augmented with the rule context):

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

The set of rewrite rules in a context $\Gamma$ defines a rewrite system; the conversion rule for $\lambda\Pi m$ is the same as the one for the $\lambda\Pi$-calculus except that the $\beta$-equivalence is replaced by the congruence $\equiv_{\beta\Gamma}$.

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

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

An example of well-formed $\lambda\Pi m$-context\(^2\) 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. Here and in rest of the paper, we omit in such definitions the types of variables introduced by $\Pi$ and $\lambda$ when it is not ambiguous. 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 $\text{append } 0 n \text{ empty } l \rightarrow l$ is well-formed in the context $\Gamma := \text{Nat} : \text{Type}, 0 : \text{Nat}, \ldots, \text{append} : \Pi n_1.\Pi n_2.\text{List } n_1 \rightarrow \text{List } n_2 \rightarrow \text{List } (\text{plus } n_1 n_2)$:

\(^1\) That is, to decide whether a given term matches a rewrite rule.

\(^2\) Examples and other contexts in $\lambda\Pi m$ are preceded in this article by a vertical bar in order to distinguish them from examples in the $\varsigma$-calculus.
we call such a proof scheme a rewrite system is both confluent and strongly normalizing, convertibility check can be

To state a theorem, we declare a symbol whose type is the theorem statement and to prove the theorem we add one or more rewrite rules defining this symbol as a (total and terminating) function. A \(\lambda I\) proof of the addition commutativity is given in Figure 3. This proof is composed of two rewrite rules that mimic a proof by induction on the first argument of \(\text{plus}\). In the following, we call such a proof scheme a \(\lambda I\) induction proof.

The interesting properties about a \(\lambda I\)-context and its 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

\[\begin{align*}
\text{Nat} & : \text{Type}. \\
0 & : \text{Nat}. \\
S & : \text{Nat} \rightarrow \text{Nat}. \\
\text{plus} & : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}. \\
\text{append} & : \Pi n_1.\Pi n_2. \text{List} n_1 \rightarrow \text{List} n_2 \rightarrow \text{List} (\text{plus} n_1 n_2).
\end{align*}\]

\[\begin{align*}
\text{equal} & : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Type}. \\
\text{refl} & : \Pi n. \text{equal} n n.
\end{align*}\]

\[\begin{align*}
\text{equal}_{\text{S}} & : \Pi n_1.\Pi n_2. \text{equal} n_1 n_2 \rightarrow \text{equal} (\text{S} n_1) (\text{S} n_2). \\
\text{equal}_{\text{S}} & n n (\text{refl} n) \leftrightarrow \text{refl} (\text{S} n).
\end{align*}\]

\[\begin{align*}
\text{plus}_{\text{comm}} & : \Pi n_1.\Pi n_2. \text{equal} (\text{plus} n_1 n_2) (\text{plus} n_2 n_1). \\
\text{plus}_{\text{comm}} & 0 n_2 \leftrightarrow \text{refl} n_2. \\
\text{plus}_{\text{comm}} & (\text{S} n_1) n_2 \leftrightarrow \text{equal}_{\text{S}} (\text{plus} n_1 n_2) (\text{plus} n_2 n_1) (\text{plus}_{\text{comm}} n_1 n_2).
\end{align*}\]
decided by comparing normal forms so well-formedness becomes decidable and is indeed implemented in the Dedukti [6] type checker.

However, the correctness of Dedukti relies on confluence only; strong normalization is only used to ensure termination.

3 The simply-typed $\varsigma$-calculus

In this section, we describe the source language of our encoding, that is the simply-typed $\varsigma$-calculus defined by Abadi and Cardelli [2, 1] (also called $Obj_{1<}$). 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 $\lambda$-calculus, well-typed $\varsigma$-terms do not always terminate.

3.1 Syntax

The syntax of the simply-typed $\varsigma$-calculus is divided between types and terms.

Types are (possibly empty) records of types:

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

Labels are distinct and their order 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 $\varsigma$. Methods can be selected and updated.

$$a, b, \ldots ::= x \quad \text{variable}$$

$$\mid \left[ l_i = \varsigma(x_i : A)a_i \right]_{i \in 1 \ldots n} \quad \text{object}$$

$$\mid a.l \quad \text{method selection}$$

$$\mid a.l \leftarrow \varsigma(x : A)b \quad \text{method update}$$

Again, labels in objects are distinct and their order does not matter. When the variable introduced by the $\varsigma$ 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 $\Delta$, we denote by $\Delta(x)$ the associated type.

3.2 Typing

The following rules, where $A$ stands for $\left[ l_i : A_i \right]_{i \in 1 \ldots n}$, define a type system for the simply-typed $\varsigma$-calculus³:

³ Abadi and Cardelli also consider a ground type that they call $K$ or $Top$ to ease comparison with the simply-typed $\lambda$-calculus. It can be replaced by the empty object type $\left[ \right]$ so we omit it here to simplify the calculus.
### 3.2.1 Subtyping

This type system is extended by a subtyping relation $<: \text{defined as follows:}$

\[
\Delta \vdash \varsigma A_i \quad \forall i \in 1 \ldots n + m \quad (\text{subtype})
\]

\[
\Delta \vdash \varsigma A <: B \quad \Delta \vdash \varsigma B <: C \quad (\text{trans})
\]

Since the order of labels is irrelevant, the (subtype) rule 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 change the type of terms with the following subsumption rule:

\[
\Delta \vdash \varsigma a : A <: B \quad \Delta \vdash \varsigma A <: B \quad (\text{subsume})
\]

### 3.2.2 Minimum types

Abadi and Cardelli have proved that the simply-typed $\varsigma$-calculus enjoys minimum typing [2]: for each well-typed term $a$ in a context $\Delta$, we can compute a type $\text{mintype}_\Delta (a)$ such that:

1. $\Delta \vdash \varsigma a : \text{mintype}_\Delta (a)$
2. For all $A$ such that $\Delta \vdash \varsigma a : A$, we have $\Delta \vdash \varsigma \text{mintype}_\Delta (a) <: A$.

The meta-level function $\text{mintype}^4$ is defined as follows:

\[
\text{mintype}_{\Delta} (x) := \Delta (x)
\]

\[
\text{mintype}_{\Delta} ([]) := [
\]

\[
\text{mintype}_{\Delta} ([l_i = \varsigma (x_i : A) a_i]_{i \in 1 \ldots n+1}) := A
\]

\[
\text{mintype}_{\Delta} (a, j) := B_j \text{ when } \text{mintype}_{\Delta} (a) \text{ is } [l_i : B_i]_{i \in 1 \ldots n}
\]

### 3.3 Operational Semantics

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

---

4 Bold face is here used to distinguish the meta-level.
54 Objects and Subtyping in the $\lambda\Pi$-Calculus Modulo

\[
\begin{align*}
  a.l_j & \Rightarrow a_j(a/x) \\
  a.l_j \leftarrow \varsigma(x : A')u & \Rightarrow [l_j = \varsigma(x : A)u, l_i = \varsigma(x_i : A)a_i]_{i \in \ldots, i \neq j}
\end{align*}
\]

where $a_j(a/x)$ denotes the substitution of the variable $x$ by the term $a$ in term $a_j$.

The type $A'$ used in the binder for updating the object $a$ does not need to be equal to $A$ but may be any supertype of it.

Subject reduction has been proved by Abadi and Cardelli [1]. However, reduction does not preserve minimum typing since $\text{mintype}_\Delta(a.l_j \leftarrow \varsigma(x : A')u)$ is (by definition) $A'$ but this term reduces to a value of type $A$.

### 3.4 Example

The expressivity of the $\varsigma$-calculus can be illustrated by the following example from Abadi and Cardelli [2] assuming that we have a type $\text{Num}$ for numbers and that the simply-typed $\lambda$-calculus has been encoded:

\[
\begin{align*}
  \text{RomCell} & := [\text{get} : \text{Num}] \\
  \text{PromCell} & := [\text{get} : \text{Num}, \text{set} : \text{Num} \rightarrow \text{RomCell}] \\
  \text{PrivateCell} & := [\text{get} : \text{Num}, \text{contents} : \text{Num}, \text{set} : \text{Num} \rightarrow \text{RomCell}] \\
  \text{myCell} : \text{PromCell} & := [\text{get} = \varsigma(x : \text{PrivateCell}).x.\text{contents}, \\
  & \quad \text{contents} = \varsigma(x : \text{PrivateCell})0, \\
  & \quad \text{set} = \varsigma(x : \text{PrivateCell})\lambda(n : \text{Num}).x.\text{contents} \leftarrow n]
\end{align*}
\]

$\text{RomCell}$ is the type of read-only memory cells; the only action that we can perform on a $\text{RomCell}$ is to read it ($\text{get}$ method).

A $\text{PromCell}$ is a memory cell which can be written once ($\text{set}$ method), we can either read it now or write it and get a $\text{RomCell}$.

$\text{PrivateCell}$ is a type used for implementation; it extends $\text{PromCell}$ with a $\text{contents}$ field which should not be seen from the outside.

The object $\text{myCell}$ implemented as an object of type $\text{PrivateCell}$ can be given the type $\text{PromCell}$ thanks to subsumption.

### 4 Encoding of the simply-typed $\varsigma$-calculus in the $\lambda\Pi$-calculus modulo

This section describes an encoding of the simply-typed $\varsigma$-calculus given by a $\lambda\Pi$-context and a translation of $\varsigma$-types, terms, and contexts. We want it to be shallow in the sense discussed in the introduction. However, the encoding described in the current section will only preserve typing and binding, since preserving reduction of a non terminating system cannot, of course, be achieved using a strongly-normalizing rewrite system. The associated rewrite system will be confluent and strongly normalizing, making type-checking of encoded terms decidable. In the next section, we will add a few rewrite rules in order to preserve reduction at the price of losing normalization.

This encoding is implemented as a translation tool [10] producing Dedukti terms from $\varsigma$-terms and types.

#### 4.1 Encoding of types

We assume given an infinite $\lambda\Pi$-type label with a decidable equality.

Unit, product, $\Sigma$-types, and Leibnitz equality can all be encoded in $\lambda\Pi$ (they are special cases of inductive types, which are translated to $\lambda\Pi$ by Coqine [4]) so we consider that they are available with the usual notations (respectively $\text{unit}$, $A \times B$, $\Sigma x : A.B$, and $=A$).

To avoid confusion with Leibnitz equality, we write $\equiv$ for the equality at meta-level.
4.1.1 Domains

Domains are lists of labels:

\[
\begin{align*}
\text{domain} & : \text{Type}. \\
\text{nil} & : \text{domain}.
\end{align*}
\]

\[
\text{cons} : \text{label} \to \text{domain} \to \text{domain}.
\]

We use the notation \([l_1; \ldots; l_n]\) for \((\text{cons} \ l_1 \ (... (\text{cons} \ l_n \ \text{nil}) ...).\)

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

\[
\begin{align*}
\epsilon & : \text{label} \to \text{domain} \to \text{Type}. \\
\text{at-head} & : \Pi l. \Pi d. l \in \text{cons} \ l \ d. \\
\text{in-tail} & : \Pi l_1. \Pi l_2. \Pi d. l_1 \in d \to l_1 \in \text{cons} \ l_2 \ d.
\end{align*}
\]

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

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

4.1.2 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:

\[
\begin{align*}
\text{type} & : \text{Type}. \\
\text{typenil} & : \text{type}. \\
\text{typecons} & : \text{label} \to \text{type} \to \text{type} \to \text{type}.
\end{align*}
\]

The \(\lambda\Pi\)-term \(\text{type}\) should not be confused with the \(\lambda\Pi\)-term \(\text{Type}\); the former is the \(\lambda\Pi\) equivalent of \(\varsigma\)-types and the latter is sort of all the \(\lambda\Pi\)-types.

A translation function \(\[,\]\) from \(\varsigma\)-types to \(\lambda\Pi\)-terms of type \(\text{type}\) is given by

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

For example, the types \(\text{RomCell}, \text{PromCell}\), and \(\text{PrivateCell}\) defined in Section 3.4 are translated as follows:

\[
\begin{align*}
[[\text{RomCell}]] & \equiv \text{typecons} \ \text{get} \ [[\text{Num}]] \ \text{typenil} \\
[[\text{PromCell}]] & \equiv \text{typecons} \ \text{get} \ [[\text{Num}]] ( \\
& \hspace{1cm} \text{typecons} \ \text{set} \ [[\text{Num} \to \text{RomCell}]] \ \\
& \hspace{2cm} \text{typenil}) \\
[[\text{PrivateCell}]] & \equiv \text{typecons} \ \text{contents} \ [[\text{Num}]] ( \\
& \hspace{1cm} \text{typecons} \ \text{get} \ [[\text{Num}]] ( \\
& \hspace{2cm} \text{typecons} \ \text{set} \ [[\text{Num} \to \text{RomCell}]] \ \\
& \hspace{3cm} \text{typenil})))
\end{align*}
\]

4.1.3 Design choices

This encoding of \(\varsigma\)-types as association lists is a bit under-specified: the type \(\text{type}\) does not impose unicity of label nor sorting. We know two ways to impose these two restrictions:
We can add an extra argument to the `typecons` constructor, witnessing that the added label minors the elements in the tail of the list:

\[
\begin{align*}
type &: \text{Type}. \\
\text{minors} &: \text{label} \to \text{type} \to \text{Type}. \\
\text{typenil} &: \text{type}. \\
\text{typecons} &: \Pi l. (\Pi A. \Pi B. \text{minors} l B) \to \text{type}. \\
\text{minors-nil} &: \Pi l. (\text{typenil} \to \text{minors} l \text{typenil}). \\
\text{minors-cons} &: \Pi l. (\Pi l'. \Pi A. \Pi B. (\text{minors} l' B \to l < l' \to \text{minors} l (\text{typecons} l' A B)).
\end{align*}
\]

But this increases a lot the size of the translated types.

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

\[
\begin{align*}
\text{typecons} l_1 A_1 (\text{typecons} l_2 A_2 B) & \leftrightarrow \text{typecons} l_2 A_2 (\text{typecons} l_1 A_1 B). \\
\text{typecons} l A_1 (\text{typecons} l A_2 B) & \leftrightarrow \text{typecons} l A_1 B.
\end{align*}
\]

In order to preserve normalization, we have to guard the first rule by a condition like \(l_2 < l_1\). Unfortunately, the resulting rewrite system becomes hard to keep confluent with definitions of functions on \text{type}. Moreover this requires an ordering on labels and the use of conditional rewriting which is not yet implemented in Dedukti.

The benefit from excluding unsorted association lists does not seem worth the drawbacks of these solutions hence we prefer to live with the existence of \(\lambda\Pi\)-terms of type \text{type} not coming from the encoding.

### 4.1.4 Domain and association

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

\[
\begin{align*}
\text{dom} &: \text{type} \to \text{domain}. \\
\text{dom} \text{typenil} & \leftrightarrow \text{nil}. \\
\text{dom} (\text{typecons} l A B) & \leftrightarrow \text{cons} l (\text{dom} B). \\
\text{assoc} &: \Pi A. \Pi l. (l \in \text{dom} A \to \text{type}). \\
\text{assoc} (\text{typecons} l A B) l (\text{at-head} l (\text{dom} B)) & \leftrightarrow A. \\
\text{assoc} (\text{typecons} l_2 A B) l_1 (\text{in-tail} l_1 l_2 (\text{dom} B) p) & \leftrightarrow \text{assoc} B l_1 p.
\end{align*}
\]

We abbreviate \text{assoc} \(A l p\) as \(A.p l\) or \(A.l\) making the position \(p\) implicit.

### 4.1.5 Subtyping relations

The subtyping relation is defined by:

\[
\begin{align*}
\bullet \leq \bullet &: \text{type} \to \text{type} \to \text{Type}. \\
A \leq \text{typenil} & \leftrightarrow \text{unit}. \\
A \leq \text{typecons} l B C & \leftrightarrow \Sigma p : l \in \text{dom} A. (A.p l \equiv \text{type} B) \times (A \leq C).
\end{align*}
\]

where \(\equiv_{\text{type}}\) is the Leibnitz equality defined on \text{type}.
4.1.6 Properties of the subtyping relation

This subsection lists a few useful properties of the \( \leq \) relation. These properties are provable directly in \( \lambda \Pi m \), as opposed to the correctness of the translation of subtyping which will be addressed in Section 4.3.2. These proofs can be found at [10].

- **Lemma 1 (subtype-weakening).** The \( \leq \) relation enjoys weakening; it means that in \( \lambda \Pi m \), we can define a total function \( \text{subtype-weakening} \) of type

\[
\Pi A.\Pi B.\Pi I.\Pi C.A \leq B \rightarrow (\text{typecons } l A C) \leq B.
\]

**Proof.** Straightforward by induction on \( B \) (as explained previously, the function \( \text{subtype-weakening} \) is defined by two rewrite rules, one for \( B \equiv \text{typenil} \) and another for \( B \equiv \text{typecons } \ldots \)).

- **Lemma 2 (subtype-refl).** The \( \leq \) relation is reflexive; in \( \lambda \Pi m \), we can define a total function \( \text{subtype-refl} \) of type \( \Pi A.A \leq A \).

**Proof.** By induction on \( A \) using the previous lemma.

- **Lemma 3 (subtype-dom).** The \( \text{dom} \) function is compatible with \( \leq \); in \( \lambda \Pi m \), we can define a total function \( \text{subtype-dom} \) of type \( \Pi A.\Pi B.\Pi A \leq B \rightarrow \text{dom } B \subseteq \text{dom } A \).

**Proof.** By induction on \( B \).

- Base case is trivial (there is no rewrite rule for this case because it is an empty case)
- If \( B \) is \( \text{typecons } l' B_1 B_2 \), we have some position \( p' : l' \in \text{dom } A \) and \( A \leq B_2 \). For any \( l \) and any position \( p : l \in \text{cons } l \) \( (\text{dom } B_2) \), either \( p \) is at head in which case \( l \equiv l' \) and \( p' \) proves the goal, or \( p \) is in tail and we conclude using the induction hypothesis.

- **Lemma 4 (subtype-assoc).** The \( \text{assoc} \) function is compatible with \( \leq \); in \( \lambda \Pi m \), we can define a total function \( \text{subtype-assoc} \) of type \( \Pi A.\Pi B.\Pi st : A \leq B.\Pi Ip : l \in \text{dom } B.B.\Pi l =_{\text{type}} A.\Pi \text{subtype-dom } A B st l p l' \).

**Proof.** By induction on \( B \).

- Base case is trivial
- If \( B \) is \( \text{typecons } l' B_1 B_2 \), we have some position \( p' : l' \in \text{dom } A \) such that \( A.p' l' =_{\text{type}} B_1 \) and \( A \leq B_2 \). For any \( l \) and any position \( p : l' \in \text{cons } l \) \( (\text{dom } B_2) \), either \( p \) is at head in which case \( l \equiv l' \) and \( B_2.p l \equiv B_1.l \). \( A.\Pi \text{subtype-dom } A B st l p l' \equiv A.p' l' \equiv B_1 \)
  - or \( p \) is in tail in which case we conclude again using the induction hypothesis.

- **Lemma 5 (subtype-trans).** The subtyping relation is transitive; in \( \lambda \Pi m \), we can define a total function \( \text{subtype-trans} \) of type \( \Pi A.\Pi B.\Pi C.A \leq B \rightarrow B \leq C \rightarrow A \leq C \).

**Proof.** By induction on \( C \), using \( \text{subtype-dom} \) and \( \text{subtype-assoc} \).

4.2 Encoding of terms

As we did for types, we define translation functions from terms and contexts of the simply-typed \( \varsigma \)-calculus to terms and contexts of \( \lambda \Pi m \).

These functions preserve typing in the sense that we can define, in \( \lambda \Pi m \), a function \( \text{Expr} \) such that whenever the judgment \( \Delta \vdash_\varsigma a : A \) is valid in the simply-typed \( \varsigma \)-calculus, the judgment \( \llbracket \Delta \rrbracket \vdash [a]_{\Delta.A} : \text{Expr} \llbracket [A] \rrbracket \) is valid in \( \lambda \Pi m \).

We define a \( \lambda \Pi m \)-context reflecting the syntax and the semantics of the \( \varsigma \)-calculus. We start with concrete objects, we then define coercions reflecting the use of the subsumption rule. From these declarations, we define the \( \Pi m \) version of selection and update, and finally we give the translation function for terms.
4.2.1 Objects

Expr \( A \) represents the \( \lambda \Pi \)m-type of well-typed objects of type \( A \) and Meth \( A B \) represents the \( \lambda \Pi \)m-type of methods of \( A \) returning an object of type \( B \).

We can declare Expr and Meth:

\[
\begin{align*}
\text{Expr} : & \text{type} \rightarrow \text{Type}. \\
\text{Meth} : & \text{type} \rightarrow \text{type} \rightarrow \text{Type}.
\end{align*}
\]

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 \( \lambda \Pi \)m-type Preobj \( A d \) by the following declarations:

\[
\begin{align*}
\text{Preobj} : & \text{type} \rightarrow \text{domain} \rightarrow \text{Type}. \\
prenil : & \Pi A . \text{Preobj} A \text{nil}. \\
precons : & \Pi A. \Pi d. \Pi l. \Pi p : l \in \text{dom} A. \\
& \text{Meth} A A. p l \rightarrow \text{Preobj} A d \rightarrow \text{Preobj} (\text{cons} l d).
\end{align*}
\]

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

\[
\text{Obj} A : \leftrightarrow \text{Preobj} A (\text{dom} A).
\]

and expressions of type \( B \) are objects of a type \( A \), subtype of \( B \):

\[
\text{Expr} B : \leftrightarrow \Sigma A : \text{type}. (\text{Obj} A) \times (A \leq B).
\]

Since the subtyping relation is reflexive, we can inject objects into expressions:

\[
\begin{align*}
\text{expr-of-obj} : & \Pi A. \text{Obj} A \rightarrow \text{Expr} A. \\
\text{expr-of-obj a} : & (A, a, \text{subtype-refl} A).
\end{align*}
\]

We would like to define Meth \( A B \) as Expr \( A \rightarrow \text{Expr} B \) to end this set of definitions but then the negative occurrence of Expr would be a source of non-termination.

We solve this problem by adding axioms stating that Meth \( A B \) is equivalent to Expr \( A \rightarrow \text{Expr} B \):

\[
\begin{align*}
\text{Eval-meth} : & \Pi A. \Pi B. \text{Meth} A B \rightarrow \text{Expr} A \rightarrow \text{Expr} B. \\
\text{Make-meth} : & \Pi A. \Pi B. (\text{Expr} A \rightarrow \text{Expr} B) \rightarrow \text{Meth} A B.
\end{align*}
\]

The key point here is that Eval-meth and Make-meth will freeze reduction. For example the translation of a looping \( \varsigma \)-term like \( [l = \varsigma(x : [l : []])x.l]l \) will be a term whose normalization will freeze at an occurrence of the pattern Eval-meth \( A B \) (Make-meth \( A B f \)) \( a \) which will not be matched by any rewrite rule.

To get a reduction-preserving encoding, we just have to add some rewrite rules; either the rule Eval-meth \( A B \) (Make-meth \( A B f \)) \( a \leftrightarrow f \) \( a \) or the following one Meth \( A B \leftrightarrow \text{Expr} A \rightarrow \text{Expr} B \) (and Eval-meth and Make-meth both reduce to identity).

4.2.2 Coercions

Implicit subtyping cannot be expressed in \( \lambda \Pi \)m because each \( \lambda \Pi \)-term has at most one type modulo \( \beta \) and rewriting. Hence we cannot simply rewrite any type \( A \) to any of its subtypes or supertypes; 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 $\text{Expr}$ and Lemma 5; if $a$ is an object of type $A$ subtype of $B$ seen as an expression of type $B$, seeing $a$ as an expression of type $C$ supertype of $B$ only requires a proof of $A \leq C$ which may be obtained by transitivity of $\leq$:

\[
\begin{align*}
\text{coerce} : \Pi B : \text{type}. \Pi C : \text{type}. B \leq C \rightarrow \text{Expr} B \rightarrow \text{Expr} C. \\
\text{coerce} B C \text{ st}_{BC} (A, a, \text{st}_{AB}) & \mapsto (A, a, \text{subtype-trans} \text{ st}_{AB} \text{ st}_{BC}).
\end{align*}
\]

We use the notation $a \uparrow^B_A$ for the term $\text{coerce} A B \text{ st } a$ of type $\text{Expr} B$, leaving the subtyping proof implicit.

### 4.2.3 Operational semantics

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

Their definitions follow the definitions of $\text{Expr}$ and $\text{Obj}$; they work recursively on the $\text{Preobj}$ structure using auxiliary functions called $\text{preselect}$ and $\text{preupdate}$. 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 $l \in \text{dom } A$ thanks to the following lemma:

▶ **Lemma 6 (preobj-subset).** Preobjects are defined on subsets of the domain: in $\lambda \Pi m, w$, we can define a total function $\text{preobj-subset}$ of type $\Pi A. \Pi l. \Pi d. \text{Preobj } A \rightarrow l \in \text{dom } A$.

**Proof.** Straightforward by induction on $d$. ▶

The definition of update is straightforward:

\[
\begin{align*}
\text{preupdate} : \Pi A. \Pi l. \Pi p : l \in \text{dom } A. \Pi d. \Pi l. \Pi p : \text{Preobj } A \rightarrow \text{Meth } A A \rightarrow \text{Expr } A.
\end{align*}
\]

\[
\begin{align*}
\text{update} A l p & \rightarrow \text{precons } A d l \in \text{dom } A. \text{Expr } A \rightarrow \text{Meth } A A \rightarrow \text{Expr } A.
\end{align*}
\]

The obj-update function can be used to update a method of an object of type $A$; if we want to update an expression of type $B$ where $A \leq B$, we only have at hand a method of type $\text{Meth } B A l$ (for some $l$) where obj-update needs a $\text{Meth } A A l$. This can be solved by a substitution of the self variable by its coercion $\text{self} \uparrow^B_A$ in the method body, which is easy to write as $(\lambda l. \text{self} : \text{Expr } A) (\text{Eval-meth } B A l m (\text{self} \uparrow^B_A))$. Hence we can define update as follows:

\[
\begin{align*}
\text{update} B l p & \rightarrow (A, a, \text{st}) \rightarrow (A, a, \text{st}).
\end{align*}
\]
We now have all we need to define a translation function from simply-typed \( \lambda \)-terms of different types because \( \Lambda \Pi \) lacks subtyping and subsumption. Hence we have to parameterize our translation function by the targeted type \( A \) in the \( \lambda \)-calculus. Fortunately, it is enough to define the translation function for the minimum type of \( a \), written \( \llbracket a \rrbracket_\Delta \). We can then define the general translation function for type \( A \) as \( \llbracket a \rrbracket_{\Delta, A} := \llbracket a \rrbracket_\Delta \uparrow^{[\text{mintype}_\Delta(a)]} \) where the proof of \( [\text{mintype}_\Delta(a)] \leq [A] \) is computed by a meta-level\(^5\) function decide-subtype (omitted here).

The \( \llbracket \cdot \rrbracket_{\Delta} \) function, the \( \llbracket \cdot \rrbracket_{\Delta, A} \) function and the translation function for methods are mutually defined by:

\[
\begin{align*}
\llbracket a \rrbracket_{\Delta, A} &:= \llbracket a \rrbracket_\Delta \uparrow^{[A]}_{[\text{mintype}_\Delta(a)]} \\
\llbracket l_1 = s(x : A)a_1, \ldots, l_n \rrbracket_{\Delta} &:= \text{expr-of-obj} ( \\
&\quad \text{precons} \ [A] \ [l_2; \ldots; l_n] l_1 p_1 [s(x : A)a_1]_{\Delta, A, p_1, l_1} ( \\
&\quad \quad \ldots (\text{precons} \ [A] [A] [l_n p_n [s(x : A)a_n]_{\Delta, A, p_n, l_n} \text{ prenil} [A])))) \\
\llbracket a.l \rrbracket_{\Delta} &:= \text{select} \ [\text{mintype}_{\Delta}(a)] \ l \ p \ [a]_{\Delta} \\
\llbracket a.l \rrbracket_{\Delta} &:= \llbracket\cdot\rrbracket_{\Delta} \ l \ p \ [a]_{\Delta, A} [s(x : A)b]_{\Delta, A, p,l} \\
\llbracket s(x : A)b \rrbracket_{\Delta, B} &:= \text{Make-meth} \ [A] \ [B] (\lambda x : \text{Expr} \ [A].b)_{(\Delta, x : A), B})
\end{align*}
\]

\(^5\) The function decide-subtype is easy to define at the meta-level but could also be defined in \( \Lambda \Pi \).
The positions \( p_i \) and \( p \) in this encoding can be computed for any well-typed \( \varsigma \)-term: \( p_i \) is the \( i \)th position (\( p_1 \) is at-head \( l_1 \) [\( l_2; \ldots; l_n \)], \( p_2 \) is in-tail \( l_2 \) \( l_1 \) (at-head \( l_2 \) [\( l_3; \ldots; l_n \)], \( p_n \) is in-tail \( l_n \) \( l_1 \) (\( \ldots \)(in-tail \( l_n \) \( l_{n-1} \) (at-head \( l_n \) [\( ]\) \( \ldots \))), and \( p \) is the \( p_i \) such that \( l \) is \( l_i \).

The translation of the binding operation of our source language (the \( \varsigma \) binder) is done by a binding operation in the target language (the \( \lambda \) binder). This technique is generally known as Higher-Order Abstract Syntax (HOAS) [29].

We can now compute the translation of our example term \( \text{myCell} \). We translate a term \( a \) by an object of type \( \text{mintype}_\Delta (a) \) seen as an expression of the required type. In this case, \( \text{mintype}_\Delta (\text{myCell}) \) is PrivateCell and the required type is PromCell.

\[
\begin{align*}
\text{myCell} &\triangleq \text{myCell} \Delta \text{PromCell} \\
&\equiv \text{myCell} \Delta [\text{PrivateCell}] \\
&\equiv \text{myCell} \Delta [\text{PromCell}] \\
&\equiv ([\text{PrivateCell}], \\
&\text{precons} \ [\text{PrivateCell}] [\text{get; set}] \text{contents} \ p_1 \\
&\equiv \text{precons} \ [\text{PrivateCell}] [\text{set}] \text{get} \ p_2 \\
&\equiv \text{precons} \ [\text{PrivateCell}] [\text{set}] \text{get} \ p_3 \\
&\equiv \text{decide-subtype} \ [\text{PrivateCell}] [\text{PromCell}]
\end{align*}
\]

As expected, the translation of the looping \( \varsigma \)-term \( l = \varsigma (x : [l : [ ]]) \cdot x . l \) normalizes to an instance of the pattern \( \text{Eval-meth } A \ B F (\text{Make-meth } A \ B \ F) \ a \):

\[
\begin{align*}
\text{eval} &\equiv \text{typecons} \ l \text{typenil typenil} \\
&\equiv \text{select} \ [[l : [ ]][x : [l : [ ]]]] l p_1 x \\
&\equiv \text{Make-meth} \ [[l : [ ]][\text{typenil}] [\lambda x : \text{Expr} \ [[l : [ ]][\text{typenil}][x : [l : [ ]]]]) \\
&\equiv \text{subtype-refl} \ [[l : [ ]][\text{typenil}]
\end{align*}
\]
\[
\[ l = s(x : [l : []])x.\ell.]_0 \equiv \text{select } \[ l = s(x : [l : []])x.\ell.]_\Delta \[ l = s((x : [l : []])x.\ell)]_0
\]
\[ \rightarrow \text{Eval-meth } \[ l = s(x : [l : []])x.\ell.]_0 [ ]\ ]\ ] l \]
\[ \text{(obj-select } \[ l = s(x : [l : []])x.\ell.]_0 [ ]\ ]\ ] l \]
\[ \text{(precons ...)}\]
\[ \rightarrow \text{Eval-meth } \[ l = s(x : [l : []])x.\ell.]_0 [ ]\ ]\ ] l \]
\[ \text{(precons ...)}\]
\[ \rightarrow \text{Eval-meth } \[ l = s(x : [l : []])x.\ell.]_0 [ ]\ ]\ ] l \]
\[ \text{(preselect } \[ l = s(x : [l : []])x.\ell.]_0 [ ]\ ]\ ] l \]
\[ \text{(Make-method } \[ l = s(x : [l : []])x.\ell.]_0 [ ]\ ]\ ] l \]
\[ \text{(lambda } \lambda x : \text{Expr } \[ l = s(x : [l : []])x.\ell.]_0 [x.\ell].x.[l : []])\]
\[ \rightarrow \text{Eval-meth } \[ l = s(x : [l : []])x.\ell.]_0 [ ]\ ]\ ] l \]

4.3 Properties of the encoding

Let \( \Gamma_0 \) be the \( \Pi \) context composed of the declarations and rewrite rules presented in the previous section. We investigate properties of the rewrite system \( R_0 \) associated with \( \Gamma_0 \) and of translated \( \varsigma \) terms in contexts of the form \( \Gamma_0, \Lambda \) where \( \Lambda \) is a \( \Pi \) context (a \( \Pi \) context without rewrite rule) so the rewrite system associated with \( \Gamma_0, \Lambda \) is \( R_0 \).

The proofs in this section are done at the metalevel and are pen-and-paper proofs.

4.3.1 Normalization and confluence

The rewrite system \( R_0 \) 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 [27].

In order to be extra-confident in these properties, we implemented the definitions of \( \Gamma_0 \) in the Calculus of Inductive Constructions, which is known to be strongly normalizing and confluent [15], and type-checked this implementation with Coq.

Our code is available at \( \text{http://sigmaid.gforge.inria.fr} \). However this translation to Coq uses axioms \( \text{(Meth, Make-method, and Eval-method)} \) which are \( \text{a priori} \) not provable in Coq.

4.3.2 Preservation of the subtyping relation by the translation

In this subsection we prove that our translation of types preserves subtyping: given two \( \varsigma \) types \( A \) and \( B \), we have \( A <: B \) if and only if \( [A] \leq [B] \). The proof requires some intermediate results we detail below.

\textbf{Lemma 7.} If \( l \in \text{dom } [l_i : A_i]_{i \in 1 \ldots n} \) then \( l \equiv l_j \) for some \( j \in 1 \ldots n \).

\textbf{Proof.} Trivial by induction on the position of type \( l \in \text{dom } [l_i : A_i]_{i \in 1 \ldots n} \).

\textbf{Lemma 8.} If \( j \in 1 \ldots n \), then \( l_j \in \text{dom } [l_i : A_i]_{i \in 1 \ldots n} \).

\textbf{Proof.} Without loss of generality, we assume that \( l_1 > \ldots > l_n \). \( \text{dom } [l_i : A_i]_{i \in 1 \ldots n} \) is \( [l_n; \ldots; l_1] \). We prove that \( l_j \in [l_n; \ldots; l_1] \) by induction on \( n \):

\text{case } n \equiv 0: \text{ the hypothesis } j \in 1 \ldots n \text{ is a contradiction.}
\[ \text{case } n \equiv p+1: \text{ if } j \equiv p+1 \text{ then } \text{at-head } j \langle l_p; \ldots; l_1 \rangle \text{ proves } l_j \in \langle l_{p+1}; \ldots; l_1 \rangle \text{ else } j \in 1 \ldots p \]

so by induction hypothesis, \( l_j \in \langle l_{p+1}; \ldots; l_1 \rangle \) thus \( l_j \in \langle l_{p+1}; \ldots; l_1 \rangle \) by \text{in-tail}.

\[ \text{Lemma 9. If } j \in 1 \ldots n, \text{ then } \langle l_i : A_i \rangle_{i=1 \ldots n} \text{-pos}\ l_j \equiv \langle A_j \rangle \] where \text{pos} is the proof of the previous lemma.

\[ \text{Proof. This is trivial by following the same steps as in the previous lemma.} \]

\[ \text{Theorem 10. For every type } A \text{ and } B, \text{ if } \Delta \vdash A : B \text{ then } \llbracket A \rrbracket \leq \llbracket B \rrbracket. \]

\[ \text{Proof. We proceed by induction on the derivation of } \Delta \vdash A : B, \text{ there are three cases:} \]

\[ \text{case (subtype)} \]

\[ A \text{ is some } \llbracket l_i : A_i \rrbracket_{i=1 \ldots n+m} \text{ with } B \equiv \llbracket l_i : A_i \rrbracket_{i=1 \ldots n}. \text{ Without loss of generality, we may assume } l_n < l_{n-1} < \ldots < l_2 < l_1. \text{ We proceed by induction on } n: \]

\[ \text{case } n \equiv 0: \llbracket B \rrbracket \equiv \text{typeni1 hence } \llbracket A \rrbracket \leq \llbracket B \rrbracket. \]

\[ \text{case } n \equiv p+1: \llbracket B \rrbracket \equiv \text{typecons } l_{p+1} \llbracket A_{p+1} \rrbracket \llbracket l_i : A_i \rrbracket_{i=1 \ldots p}. \]

\[ \llbracket A \rrbracket \leq \llbracket B \rrbracket \
\equiv \llbracket A \rrbracket \leq \text{typecons } l_{p+1} \llbracket A_{p+1} \rrbracket \llbracket l_i : A_i \rrbracket_{i=1 \ldots p} \]
\[ \equiv \Sigma \text{pos} : l_{p+1} \in \text{dom } \llbracket A \rrbracket. \]
\[ (\llbracket A \rrbracket \_\text{pos}\ l_{p+1} = \text{type } \llbracket A_{p+1} \rrbracket) \times (\llbracket A \rrbracket \leq \llbracket l_i : A_i \rrbracket_{i=1 \ldots p}) \]

\[ \text{pos and the equality proof are given by Lemma 8 and Lemma 9. The proof of } \llbracket A \rrbracket \leq \llbracket l_i : A_i \rrbracket_{i=1 \ldots p} \text{ is given by the induction hypothesis.} \]

\[ \text{case (red)} \]

\[ \text{This is trivial by Lemma 2.} \]

\[ \text{case (trans)} \]

\[ \text{This is trivial by Lemma 5.} \]

\[ \text{Theorem 11. The translation function on types is injective: if } \llbracket A \rrbracket = \text{type } \llbracket B \rrbracket \text{ then } A \equiv B. \]

\[ \text{Proof. A type and its encoding have the same size hence } A \text{ and } B \text{ have the same size. The proof is by induction on this common size; both cases are trivial.} \]

\[ \text{Theorem 12. For every type } A \text{ and } B, \text{ well-formed in context } \Delta, \text{ if } \llbracket A \rrbracket \leq \llbracket B \rrbracket \text{ then } \Delta \vdash A : B. \]

\[ \text{Proof. By induction on the size } n \text{ of } B := \llbracket l_i : B_i \rrbracket_{i=1 \ldots n}. \]

\[ \text{case } n \equiv 0: B \equiv [ ] \text{ hence } \Delta \vdash A : B. \]

\[ \text{case } n \equiv p+1: \llbracket B \rrbracket \equiv \text{typecons } l_{p+1} \llbracket B_{p+1} \rrbracket \llbracket l_i : B_i \rrbracket_{i=1 \ldots p}. \] Our hypothesis simplifies to:

\[ \llbracket A \rrbracket \leq \llbracket B \rrbracket \
\equiv \llbracket A \rrbracket \leq \text{typecons } l_{p+1} \llbracket B_{p+1} \rrbracket \llbracket l_i : B_i \rrbracket_{i=1 \ldots p} \]
\[ \equiv \Sigma \text{pos} : l_{p+1} \in \text{dom } \llbracket A \rrbracket. \]
\[ (\llbracket A \rrbracket \_\text{pos}\ l_{p+1} = \text{type } \llbracket B_{p+1} \rrbracket) \times (\llbracket A \rrbracket \leq \llbracket l_i : B_i \rrbracket_{i=1 \ldots p}) \]

By induction hypothesis, \( A \) is of the form \( \llbracket l_i : B_i; l'_j : A_j \rrbracket_{i=1 \ldots p; j=1 \ldots m+1} \). From the lemmata and the injectivity theorem, we get \( l_{p+1} \equiv l'_j \) and \( A_j \equiv B_{p+1} \) for some \( j \in 1 \ldots m + 1 \). By renaming the \( l \)'s, we can choose \( j \equiv m + 1 \) and we get \( A \equiv \llbracket l_i : B_i; l'_j : A_j \rrbracket_{i=1 \ldots p; j=1 \ldots m} \) so \( \Delta \vdash A : B \) by rule (subtype).
4.3.3 Type preservation

We want to prove the following type preservation theorem:

**Theorem 13.** If, in the simply typed $\varsigma$-calculus, the judgment $\Delta \vdash_\varsigma a : A$ is valid, and, in $\lambda \Pi m$, the judgment $[\Delta] \vdash_d [a]_{\Delta,A} : \text{Expr} \ [\text{mintype}_\Delta (a)]$ is valid, then so is the judgment $[\Delta] \vdash_d [a]_{\Delta,A} : \text{Expr} \ [A]$.

Proof. From $\Delta \vdash_\varsigma a : A$ we get, by minimality, $\Delta \vdash_{\varsigma \varsigma} \text{mintype}_\Delta (a) \ll : A$ hence $[\text{mintype}_\Delta (a)] : \ll \ [A]$ by Theorem 10. Therefore $[a]_{\Delta,A} \equiv [a]_{\Delta} \uparrow [\text{mintype}_\Delta (a)]$ has type $\text{Expr} \ [A]$.

**Lemma 14.** If, in the simply typed $\varsigma$-calculus, the judgment $\Delta \vdash_\varsigma a : A$ is valid, and, in $\lambda \Pi m$, the judgment $[\Delta] \vdash_d [a]_{\Delta} : \text{Expr} \ [\text{mintype}_\Delta (a)]$ is valid, then so is the judgment $[\Delta] \vdash_d [a]_{\Delta,A} : \text{Expr} \ [A]$.

Proof. From $\Delta \vdash_\varsigma a : A$ we get, by minimality, $\Delta \vdash_{\varsigma \varsigma} \text{mintype}_\Delta (a) \ll : A$ hence $[\text{mintype}_\Delta (a)] : \ll \ [A]$ by Theorem 10. Therefore $[a]_{\Delta,A} \equiv [a]_{\Delta} \uparrow [\text{mintype}_\Delta (a)]$ has type $\text{Expr} \ [A]$.

**Theorem 16.** If, in the simply typed $\varsigma$-calculus, the judgment $\Delta \vdash_\varsigma a : A$ is valid, then the judgment $[\Delta] \vdash_d [a]_{\Delta} : \text{Expr} \ [\text{mintype}_\Delta (a)]$ is valid in $\lambda \Pi m$.

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

- case (var): $a$ is a variable $x$ appearing in $\Delta$ and $\text{mintype}_\Delta (a) \equiv \text{mintype}_\Delta (x) \equiv \Delta (x)$.

  By definition of $[\Delta]$, $x \in [\Delta]$ and $[\Delta] (x) \equiv \text{Expr} \ [\Delta (x)] \equiv \text{Expr} \ [\text{mintype}_\Delta (a)]$.

- case (obj): $a$ is $[l_1 = \varsigma (x : A) a_i | i_1 < \ldots < i_n]$ with $\text{mintype}_\Delta (a) \equiv A \equiv [l_1 : A_i | i_1 < \ldots < i_n] \equiv \text{Expr} \ [\text{mintype}_\Delta (a)]$.

  $[a]_{\Delta} \equiv [l_1 = \varsigma (x : A) a_i | i_1 < \ldots < i_n]_{\Delta}$

  $\equiv \text{expr-of-obj} (\ldots)$

  $\text{precons} \ [A] \ [l_1 \ldots l_n] \ l_1 \ p_1 \ [k : (x : A) a_1]_{\Delta A p_1 l_1} (\ldots)$

  $\text{precons} \ [A] \ [l_1 \ldots l_n] \ l_1 \ p_1 \ [k : (x : A) a_1]_{\Delta A p_1 l_1} (\ldots)$

  The term $\text{expr-of-obj}$ has type $\text{Obj} \ [A] \rightarrow \text{Expr} \ [A]$ so we just need to check that $\text{precons} \ [A] \ [l_1 \ldots l_n] \ l_1 \ p_1 \ [k : (x : A) a_1]_{\Delta A p_1 l_1} (\ldots)$

  $\equiv \text{Obj} \ [A] \equiv \text{Preobj} \ [A] \ (\text{dom} \ [A]) \equiv \text{Preobj} \ [A] \ [l_1 \ldots l_n]$. 

To compute $\text{Obj} \ [A]$, we first compute $\text{dom} \ [A]$: 

$\text{dom} \ [A] \equiv \text{dom} \ [l_1 : A_i | i_1 < \ldots < i_n]$

$\equiv \text{dom} \ (\text{typecons} \ l_1 \ [A_i] \ (\ldots) \ (\text{typecons} \ l_n \ [A_n] \ \text{typenil}) \ldots)$

$\equiv [l_1 \ldots l_n]$ 

hence $\text{Obj} \ [A] \equiv \text{Preobj} \ [A] \ (\text{dom} \ [A]) \equiv \text{Preobj} \ [A] \ [l_1 \ldots l_n]$. 


4.3.4 Semantics preservation and consistency

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

$$\Gamma \vdash_d \text{precons} [A] [l_{i+1}; \ldots ; l_n] \quad \vdash_d l_i \quad \Gamma \vdash [x: A]a_i \quad \ldots \quad \vdash_d \text{precons} [A] [l_1 \ldots l_n] \quad \Gamma \vdash \text{prenil} [A])$$

$$\vdash_d \text{Preobj} [A] [l_1; \ldots ; l_n]$$

This is trivial by decreasing recursion on $i$.

Finally $\Gamma \vdash_d [[l_i = \varsigma(x_i : A) a_i]_{i \in 1 \ldots n}] : \text{Expr} [A]$.

- case (select): $a$ is of the form $a', l_j$ with $j \in 1 \ldots n$ and $\Gamma \vdash \varsigma : A'$ 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 $a'$:

  $\text{mintype}_\Gamma (a') \equiv [l_i : A_i]_{i \in 1 \ldots n}$ so $\text{mintype}_\Gamma (a) \equiv A_j$.

  Lemma 8 gives us a position $p : l_j \in \text{dom} [\text{mintype}_\Gamma (a')]$ hence by Lemma 9, $\text{mintype}_\Gamma (a') \equiv A_j$.

  Moreover, $\Gamma \vdash \varsigma : [\text{Expr} [\text{mintype}_\Gamma (a')]]$ by induction hypothesis thus $[a]_\Gamma \equiv \text{select} [\text{mintype}_\Gamma (a')]$.

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

  By induction hypothesis and Lemma 14, $\Gamma \vdash_d [a']_{\Delta, A} : \text{Expr} A$. By Lemma 15, $\Gamma \vdash_d [\varsigma(x : A)b]_{\Delta, A} : \text{Meth} [A] [A_j]$.

  Like in the previous case, Lemma 8 gives us a position $p : l_j \in \text{dom} [\text{Expr} [A]]$ and by Lemma 9, $\text{Expr} [\text{mintype}_\Gamma (a')] \equiv [a]_\Gamma$.

  Hence $[a]_\Gamma \equiv \text{update} [A] l_j p [a']_{\Delta, A} [\varsigma(x : A)b]_{\Delta, A}$ has type $[A]_\Gamma \equiv [\text{mintype}_\Gamma (a')]$.

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

$$\Gamma \vdash \varsigma : \text{mintype}_\Gamma (a) \quad \Gamma \vdash \text{mintype}_\Gamma (a) \equiv [a]_{\Delta, A}$$

$$\Gamma \vdash \varsigma : \text{mintype}_\Gamma (a)$$

In this case, our goal is exactly the induction hypothesis

$$\Gamma \vdash_d [a]_{\Delta} : \text{Expr} [\text{mintype}_\Gamma (a)]$$

From this and Lemma 14, we have proved Theorem 13.

4.3.4 Semantics preservation and consistency

Semantics preservation is not ensured because our rewrite system is strongly normalizing and the simply-typed $\varsigma$-calculus is not.

However, we may want the following weaker result:

**Statement 1.** If $\Gamma \vdash \varsigma : A$ and $a \mapsto a'$ then $[a]_{\Delta, A} \equiv \text{Expr} [A] [a']_{\Delta, A}$ is inhabited in context $[\Delta]$.

In the case where $a$ is a selection $a \equiv a'. l_j$, $[a]_{\Delta, A}$ reduces to an instance of the pattern $\text{Eval-meth} B C (\text{Make-meth} B C f) b$ such that $[a']_{\Delta, A} \equiv f b$.

Hence we would need

$$\text{reduce-meth} : \Pi B . I C . \Pi f . \Pi b . \text{Eval-meth} B C (\text{Make-meth} B C f) b = \text{Expr} B f b$$

---

6. This comes from the proof of minimality in [1] (Propositions 4.1.1-1 to 4.1.1-4); a minimal typing judgment can be derived by allowing subsumption only before the (update) and (obj) rules.
Objects and Subtyping in the λΠ-Calculus Modulo

as an additional axiom. Unfortunately, it would be inconsistent with our encoding so Statement 1 is hopeless. The following inconsistency result has been proved in Coq [10]:

▶ Theorem 17. For any label l, the type
\[
(ΠB.ΠC.Πf.Eval-meth B C (Make-meth B C f) b =Expr C f b) → ([ ] =type [l : [ ]])
\]
is inhabited.

Proof. From an expression, we can extract the type of the underlying object:

\[
| \text{underlying-type : } ΠB.\text{Expr } B → \text{type}. \\
| \text{underlying-type } B (A, a, st) ↦ A.
\]

Let A₀ be the type [l : [ ]] and a₀ an inhabitant of Expr A₀ (for instance a₀ : Expr A₀ := [l = [ ]]). t₀ := a₀ ↑ᵦ₀ is an inhabitant of Expr [ ] which we can distinguish from the empty expression [ ] because they have different underlying types.

We define a function swap : Expr [ ] → Expr [ ] returning an expression different from its argument:

\[
| \text{swap-aux : type } → \text{Expr } [ ] . \\
| \text{swap-aux } \text{typenil} ↦ t₀ . \\
| \text{swap-aux } (\text{typecons } l' B C) ↦ [ ] . \\
| \text{swap : Expr } [ ] → \text{Expr } [ ] . \\
| \text{swap } b ↦ \text{swap-aux } (\text{underlying-type } b).
\]

We remark that Expr A₀ is isomorphic to Expr A₀ → Expr [ ]:

We can define a function elim-A₀ : Expr A₀ → Expr A₀ → Expr [ ] by

\[
\text{elim-A₀ } [l = \varsigma(x)f(x)] := f
\]

and a function intro-A₀ : (Expr A₀ → Expr [ ] ) → Expr A₀ by

\[
\text{intro-A₀ } f := [l = \varsigma(x)f(x)]
\]

let E₀ : Expr A₀ → Expr [ ] be the function defined by E₀ a := swap(elim-A₀ a). Then b₀ : Expr [ ] := E₀ (intro-A₀ E₀) is such that we can prove, using the reduce-meth axiom, b₀ =Expr [ ] swap b₀, hence underlying-type (swap b₀) = underlying-type (swap (swap b₀)) but swap b₀ is either [ ] or t₀ and we get ([ ] =type [l : [ ]]) in both cases. This last step is actually an adaption of the proof of Cantor’s theorem. ◀

Consistency is hard to define in λΠm because we have not even defined anything looking like the false proposition. Consistency is to be defined relatively to a given logic. However, we probably never want ([ ] =type [l : [ ]]) to be inhabited.

5 Shallow, non-terminating encoding

In this section, we trade strong-normalization for a shallow encoding.
5.1 Modified rewrite system

In order to get a shallow encoding, we have to add the following rewrite rules:

\[
\begin{align*}
\text{Meth} & \ A \ B \ \rightarrow \ \text{Expr} \ A \ \rightarrow \ \text{Expr} \ B. \\
\text{Eval-meth} & \ A \ B \ m \ \rightarrow \ m. \\
\text{Make-meth} & \ A \ B \ f \ \rightarrow \ f.
\end{align*}
\]

From this, the \text{reduce-meth} axiom can trivially be proved so we need to change our encoding a bit to forbid the proof of Theorem 17. We do this by disabling the extraction of underlying type and the distinction between objects and expressions. Instead of defining \text{Expr} \ A \ B \ as \ \Sigma \ A : \text{type} \ (\text{Obj} \ A) \times (A \leq B), \ we \ rewrite \ \text{Expr} \ A \ to \ \text{Obj} \ A \ and \ change \ the \ definitions \ of \ the \ functions \ that \ destructed \ expressions: \ \text{update}, \ \text{select}, \ and \ \text{coerce}:

\[
\begin{align*}
\text{Expr} \ A & \ \rightarrow \ \text{Obj} \ A. \\
\text{expr-of-obj} \ a & \ \rightarrow \ a. \\
\text{update} & \ \mathbf{\bullet} \ \mathbf{\bullet} \ (\text{precons} \ \mathbf{\bullet} \ \mathbf{\bullet} \ \mathbf{\bullet} \ \mathbf{\bullet}) \ \rightarrow \ \text{obj-update} \ \mathbf{\bullet} \ \mathbf{\bullet} \ (\text{precons} \ \mathbf{\bullet} \ \mathbf{\bullet} \ \mathbf{\bullet} \ \mathbf{\bullet}) \ \mathbf{\bullet}. \\
\text{update} \ B \ l \ p \ (\text{coerce} \ A \ B \ st \ a) \ m & \ \rightarrow \ \text{coerce} \ A \ B \ st \ (\text{update} \ A \ l \ (\text{subtype-dom} \ A \ B \ st \ l \ p) \ a \ (\lambda (\text{self}).m (\text{self} \uparrow B A)))). \\
\text{select} & \ \mathbf{\bullet} \ \mathbf{\bullet} \ (\text{precons} \ \mathbf{\bullet} \ \mathbf{\bullet} \ \mathbf{\bullet} \ \mathbf{\bullet}) \ \rightarrow \ \text{obj-select} \ \mathbf{\bullet} \ \mathbf{\bullet} \ (\text{precons} \ \mathbf{\bullet} \ \mathbf{\bullet} \ \mathbf{\bullet} \ \mathbf{\bullet} \ \mathbf{\bullet}). \\
\text{select} \ B \ l \ p \ (\text{coerce} \ A \ B \ st \ a) & \ \rightarrow \ \text{select} \ A \ l \ (\text{subtype-dom} \ A \ B \ st \ l \ p) \ a. \\
\text{coerce} \ B \ C \ st_{BC} \ (\text{coerce} \ A \ B \ st_{AB} \ a) & \ \rightarrow \ \text{coerce} \ A \ C \ (\text{subtype-trans} \ st_{AB} \ st_{BC}) \ a
\end{align*}
\]

The \text{coerce} function is not total anymore because it does not reduce on values but only when applied to another coercion. It is a constructor of \text{Expr} with some computational behaviour; we call such constructors \textit{smart} constructors. The bullets in the previous rules defining \text{update} and \text{select} represent the most general pattern that make these rules well-typed. The idea here is simply that \text{update} and \text{select} are defined by pattern matching on the object, which is either a value or a coercion. We don’t need rules for the \text{prenil} case because there is no label to select or update in that case.

We call \( \Gamma \) this new \( \lambda \Pi \)-context and \( R_1 \) the new rewrite system. We believe that \( R_1 \) is confluent because the non-orthogonal part reflects the simply-typed \( \varsigma \)-calculus known to be confluent [2], but have not formally checked it. However, \( R_1 \) 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\(^7\).

5.2 Semantics preservation

Proofs of the theorems of Section 4 are unchanged because they did not rely on the definitions of \text{update}, \text{select}, and \text{coerce}. The new encoding has the additional property of semantics preservation:

\(^7\) Actually it will terminate because
- conversion check, which triggers reduction, only occurs in types;
- non-termination only occurs at the object level;
- there is no dependent type involving objects coming from our encoding.
Theorem 18. If \( \Delta \vdash \varsigma \ a : A \) and \( a \mapsto a' \) then \( \llbracket a \rrbracket_{\Delta,A} \mapsto^+ \llbracket a' \rrbracket_{\Delta,A} \).

To prove this theorem, we first need two lemmata: stability of the encoding by substitution and unicity of subtyping proofs.

Lemma 19. The translation function is stable by substitution:
\[
\llbracket a \rrbracket_{(\Delta_1,x:B),A} \llbracket b \rrbracket_{(\Delta_2),B/x} = \llbracket a \{ b/x \} \rrbracket_{(\Delta_1,\Delta_2),A}.
\]

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

Lemma 20. Unicity of subtype proofs: if \( s \) and \( s' \) both have type \( [A] \leq [B] \) then \( s \equiv_{[A] \leq [B]} s' \).

This lemma justifies our use of implicit subtype proofs in the notation \( \bullet \uparrow \bullet \).

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 \( \beta^8 \) of type type: if \( s \) and \( s' \) both have type \( [A] \leq \beta \) then \( s \equiv_{[A] \leq \beta} s' \).

We proceed by induction on \( \beta \).

- base case: \( \beta \equiv \text{typenil} \)

\[ [A] \leq \beta \equiv [A] \leq \text{typenil} \equiv \text{unit}. \]

The type unit has only one inhabitant so \( s \equiv_{[A] \leq \beta} s' \).

- inductive case: \( \beta \equiv \text{typecons} l \beta_1 \beta_2 \)

\( A \) is some \( [i : A]_{i_1 < \ldots < i_n} \).

By definition of \( \leq \),
\[ [A] \leq \text{typecons} l \beta_1 \beta_2 \equiv \Sigma p : l \in [i_1; \ldots ; i_n].([A]_p l =_{\text{type}} \beta_1) \times ([A] \leq \beta_2) \]

But there is only one \( p : l \in [i_1; \ldots ; i_n] \) because the \( l_i \)s are different. Let us call it \( p_0 \).

\[ [A] \leq \text{typecons} l \beta_1 \beta_2 \equiv \text{isomorphic to } ([A]_{p_0} l =_{\text{type}} \beta_1) \times ([A] \leq \beta_2). \]

The left type \( [A]_{p_0} l =_{\text{type}} \beta_1 \) has at most one inhabitant thanks to Hedberg Theorem [21].

Because equality on type is decidable; the right type \( [A] \leq \beta_2 \) has only one element by induction hypothesis so \( s \equiv_{[A] \leq \beta} s' \).

We can now prove Theorem 18:

Proof. The simply-typed \( \varsigma \)-calculus enjoys subject-reduction [2] so \( \Delta \vdash \varsigma \ a' : A \). From the type-preservation theorem, \( [a]_{\Delta,A} \) and \( [a']_{\Delta,A} \) have type \( \text{Expr} [A] \) in context \( \Delta \).

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

- case (select): \( a \mapsto a' \) is an instance of \( a''_j l_j \mapsto a_j \{ a''/x_j \} \)

with \( a'' := [i : \varsigma (x_j : A') a_j]_{i = 1 \ldots n} \) and \( A'' := [i : A]_{i = 1 \ldots n} \).

So \( a \equiv a'' l_j \) and \( a' \equiv a_j \{ a''/x_j \} \).

We look at the minimum types of \( a \) and \( a' \):

- \( \text{mintype}_\Delta (a'') \equiv A'' \equiv [i : A]_{i = 1 \ldots n} \) so \( \text{mintype}_\Delta (a) \equiv \text{mintype}_\Delta (a'' l_j) \equiv A_j \)

- We call \( A' \) the minimum type of \( a' \), by minimality we know that \( \Delta \vdash \varsigma \ A' \subset A_j. \)

---

8. We use the greek letter \( \beta \) here to distinguish the \( \varsigma \)-term \( B \) and the \( \lambda \Pi \)-term \( \beta \) which abstracts \( [B] \).
\[ [a''\Delta] \text{ is of the form } (\ldots (\text{precons} [A''\Delta] [l_{j+1}; \ldots; l_n] l_j \ll (x_j : A''\Delta) a_j \Delta, A_j \ldots ) \ldots ), \text{ we abbreviate it as } \alpha. \]

\[
\begin{align*}
[a]_\Delta & \equiv \text{select} [A''\Delta] l_j p \alpha \\
\leftrightarrow & \text{obj-select} [A''\Delta] l_j p \alpha \alpha \\
\leftrightarrow^* & \ll (x_j : A''\Delta) a_j \Delta, A_j \alpha \\
\equiv & (\lambda x_j : \text{Expr} [A''\Delta], a_j \ll [A''\Delta; \Delta_{x_j}, A''\Delta], A_j \alpha) \alpha \\
\rightarrow^\beta & \ll [a_j\ll [\Delta_{x_j}; A''\Delta], A_j \ll [A''\Delta]/x] \\
\end{align*}
\]

Hence, by Lemma 19, we get exactly \([a]_\Delta \leftrightarrow^+ [a'']_\Delta, A_j).\]

Finally,

\[
\begin{align*}
[a]_\Delta, A & \equiv [a] \ll [A]_{\text{minotype}_\Delta(a)} \\
\leftrightarrow & [a'']_\Delta, A_j \ll [A] \\
\equiv & \ll [a'\ll [A_{\text{minotype}_\Delta(a')}], A_j \ll [A] \\
\leftrightarrow & [a'\ll [A]_{\text{minotype}_\Delta(a')} \\
\equiv & [a'\ll [A]_\Delta, A \\
\end{align*}
\]

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

### 6 Conclusion

We defined an embedding of the simply-typed \(\varsigma\)-calculus to \(\lambda\Pi\)m and implemented it in Dedukti as a compiler named sigmaid (SIGMA-calculus In Dedukti) [10]. This implementation has been tested on the following original examples from Abadi and Cardelli:

- encoding of the simply-typed \(\lambda\)-calculus,
- encoding of booleans,
- memory cells.

Despite non-termination of the \(\varsigma\)-calculus, we managed to translate it in a very shallow fashion by means of two encodings: 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. These logical methods would be proofs taking benefits of the mechanisms of object oriented programming.
- We would also like to encode class-based calculi like Featherweight Java [25] in \(\lambda\Pi\)m in order to compare the encoded versions of structural subtyping and class-based subtyping.

Acknowledgment. We would like to thank our colleague Ali Assaf for the fruitful discussions which led to this implementation of subtyping in \(\lambda\Pi\)m. We also thank Alan Schmitt and anonymous referees for their comments that helped us to improve this article.
References


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.

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


