Objects and subtyping in the λΠ-calculus modulo

Raphaël Cauderlier, Catherine Dubois

Objects and subtyping in the λΠ-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 λΠ-calculus modulo, an extension of the λΠ-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 λΠ-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 λΠ-calculus modulo[13] (λΠm) 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[13]. Moreover, translation tools from real-world proof assistant like Coq[12, 4] and the HOL family[3] to Dedukti[5], a type-checker for λΠm, 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 λΠm, more precisely the simply-typed ς-calculus[2]. A major feature of object oriented type systems is subtyping, and it will be the focus of this article. The simply-typed ς-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[20] is a logical system featuring class-based object mechanisms which are translated in λΠm. In order to generalize this encoding of objects in λΠm 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[27, 6] 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[28].

However, following the example of the λ-calculus, 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[14] 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 of Abadi and Cardelli[2] 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 fewer constructs and objects and their types are fixed records. Hence they are somewhat simpler but they still are very expressive.
- Featherweight Java[21] 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 λ-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[22] and Isabelle/HOL[15] using extensible records and subject reduction for Object Calculi has been proved in Coq[8] and Isabelle/HOL[19]. For the untyped Object Calculus, confluence has also been formally proved in Isabelle/HOL[18].

Encodings of objects based on rewrite techniques have also been studied; for example, in the ρ-calculus[10], a full encoding of the untyped Object Calculus and λ-calculus of objects[9] and a partial encoding of the simply-typed Object Calculus[10] have been designed. In the Maude specification environment[11], 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 λΠm such that operational semantic, typing derivations, and binding operation are preserved by this translation.

The next section of this article describes λΠm, our target language, Section 3 describes the simply-typed ς-calculus, our source language. Section 4 is the main section of this article; it defines a strongly-normalizing encoding of the simply-typed ς-calculus in λΠm. 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 λΠ-calculus modulo

2.1 The λΠ-calculus

The λΠ-calculus[16], also known as LF and λP, is an extension of the simply typed λ-calculus with dependent types. λΠ terms and types have the following syntax:

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

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 that may depend on x. If x does not appear free in τ_2, the term $\Pi x : \tau_1.\tau_2$ will be abbreviated 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 variable typing declarations is called a ($\lambda\Pi$) context:

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

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

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 $t_0\{t_1/x\}$ 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 β-reduction (the smallest congruence such that $(\lambda x : \tau_1.t_0)t_1 \equiv_\beta t_0\{t_1/x\}$).

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

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

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

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

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

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

Figure 1 inference rules for the $\lambda\Pi$-calculus

The $\lambda\Pi$-calculus is the type-system on which logical frameworks such as Automath[24] and Twelf[26] 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 β-equivalent but also when they are congruent for a given rewrite system.

$$
\begin{array}{l|l}
 & \mathtt{A} : \mathtt{Type}. \\
\mathtt{Nat} : \mathtt{Type}. & \mathtt{List} : \mathtt{Nat} \to \mathtt{Type}. \\
0 : \mathtt{Nat}. & \mathtt{empty} : \mathtt{List}\ 0. \\
\mathtt{S} : \mathtt{Nat} \to \mathtt{Nat}. & \mathtt{cons} : \Pi n : \mathtt{Nat}.\mathtt{A} \to \mathtt{List}\ n \to \mathtt{List}\ (\mathtt{S}\ n). \\
 & \\
\mathtt{plus} : \mathtt{Nat} \to \mathtt{Nat} \to \mathtt{Nat}. & \mathtt{append} : \Pi n_1.\Pi n_2. \\
 & \quad \mathtt{List}\ n_1 \to \mathtt{List}\ n_2 \to \mathtt{List}\ (\mathtt{plus}\ n_1 n_2). \\
 & \\
\mathtt{plus}\ 0\ n \hookrightarrow n. & \mathtt{append}\ 0\ n\ \mathtt{empty}\ l \hookrightarrow l. \\
\mathtt{plus}\ n\ 0 \hookrightarrow n. & \mathtt{append}\ n\ 0\ l\ \mathtt{empty} \hookrightarrow l. \\
\mathtt{plus}\ (\mathtt{S}\ n_1)\ n_2 \hookrightarrow \mathtt{S}\ (\mathtt{plus}\ n_1\ n_2). & \mathtt{append}\ (\mathtt{S}\ n_1)\ n_2\ (\mathtt{cons}\ n_1\ a\ l_1)\ l_2 \hookrightarrow \\
\mathtt{plus}\ n_1\ (\mathtt{S}\ n_2) \hookrightarrow \mathtt{S}\ (\mathtt{plus}\ n_1\ n_2). & \quad \mathtt{cons}\ (\mathtt{plus}\ n_1\ n_2)\ a\ (\mathtt{append}\ n_1\ n_2\ l_1\ l_2)
\end{array}
$$

Figure 2 Example of $\lambda\Pi$m-context: Peano natural numbers and concatenation of dependent lists

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.

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:

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

where Λ stands for $\lambda\Pi$ contexts.

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

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

- for any terms t_1, t_2 and any variable x, $(\lambda x.t_1)t_2 \longrightarrow_{\beta\Gamma} t_1\{t_2/x\}$
- for any rule $(\Lambda l \hookrightarrow r) \in \Gamma$ and any substitution θ of the variables of Λ, $\theta l \longrightarrow_{\beta\Gamma} \theta r$.

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

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 Γ if the left-hand side and the right-hand side have the same type in Γ, Λ (Γ augmented with the rule context):

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

[1] that is, to decide whether a given term matches a rewrite rule

The set of rewrite rules in a context Γ 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 β-equivalence is replaced by the congruence $\equiv_{\beta\Gamma}$.

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

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 Π and λ 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 $\texttt{append}\ 0\ n\ \texttt{empty}\ l \hookrightarrow l$ is well-formed in the context $\Gamma := \texttt{Nat} : \texttt{Type}, 0 : \texttt{Nat}, \ldots, \texttt{append} : \Pi n_1.\Pi n_2.\texttt{List}\ n_1 \to \texttt{List}\ n_2 \to \texttt{List}\ (\texttt{plus}\ n_1\ n_2)$:

- 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 $\texttt{List}\ (\texttt{plus}\ 0\ n) \equiv_{\beta(\Gamma,\Delta)} \texttt{List}\ n$, therefore we can also type the left-hand side with the type $\texttt{List}\ n$ in context Γ, Λ using 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 hence the rule $\Lambda(\texttt{append}\ 0\ n\ \texttt{empty}\ l) \hookrightarrow l$ is well-formed in context Γ.

Thanks to dependent types, we can state and prove theorems in $\lambda\Pi$m. 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 $\lambda\Pi$m 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 `plus`. In the following, we call such a proof scheme a $\lambda\Pi$m induction proof.

The interesting properties about a $\lambda\Pi$m-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 decided by comparing normal forms so well-formedness becomes decidable and is indeed implemented in the Dedukti[5] type checker.

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

[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 ς-calculus.

```
equal : Nat → Nat → Type.
refl : Πn : Nat.equal n n.

equal_S : Πn1.Πn2.equal n1 n2 → equal (S n1) (S n2).
equal_S n n (refl n) ↪ refl (S n).

plus_commute : Πn1.Πn2.equal (plus n1 n2) (plus n2 n1).
plus_commute 0 n2 ↪ refl n2.
plus_commute (S n1) n2 ↪ equal_S (plus n1 n2) (plus n2 n1) (plus_commute n1 n2).
```

Figure 3 A proof of the commutativity of addition in λΠm

3 The simply-typed ς-calculus

In this section, we describe the source language of our encoding, that is the simply-typed ς-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 λ-calculus, well-typed ς-terms do not always terminate.

3.1 Syntax

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

Types are (possibly empty) records of types:

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

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 ς. Methods can be selected and updated.

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

Again, labels in objects are distinct and their order does not matter. 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.

3.2 Typing

The following rules, where A stands for $[l_i : A_i]_{i\in 1\dots n}$, define a type system for the simply-typed ς-calculus[3]:

$$\frac{\Delta \vdash_\varsigma A_i \qquad \forall i \in 1\dots n \qquad l_i \text{ distinct}}{\Delta \vdash_\varsigma A}\ (\text{type}) \qquad \frac{}{\emptyset \vdash_\varsigma}\ (\text{empty})$$

$$\frac{\Delta \vdash_\varsigma A \qquad x \notin \Delta}{\Delta, x : A \vdash_\varsigma}\ (\text{decl}) \qquad \frac{\Delta \vdash_\varsigma \qquad x \in \Delta}{\Delta \vdash_\varsigma x : \Delta(x)}\ (\text{var})$$

$$\frac{\Delta, x_i : A \vdash_\varsigma a_i : A_i \qquad \forall i \in 1\dots n}{\Delta \vdash_\varsigma [l_i = \varsigma(x_i : A)a_i]_{i\in 1\dots n} : A}\ (\text{obj}) \qquad \frac{\Delta \vdash_\varsigma a : A \qquad j \in 1\dots n}{\Delta \vdash_\varsigma a.l_j : A_j}\ (\text{select})$$

$$\frac{\Delta \vdash_\varsigma a : A \qquad \Delta, x : A \vdash_\varsigma b : A_j \qquad j \in 1\dots n}{\Delta \vdash_\varsigma a.l_j \Leftarrow \varsigma(x : A)b : A}\ (\text{update})$$

3.2.1 Subtyping

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

$$\frac{\Delta \vdash_\varsigma A_i \qquad \forall i \in 1\dots n+m}{\Delta \vdash_\varsigma [l_i : A_i]_{i\in 1\dots n+m} <: [l_i : A_i]_{i\in 1\dots n}}\ (\text{subtype}) \qquad \frac{\Delta \vdash_\varsigma A}{\Delta \vdash_\varsigma A <: A}\ (\text{refl})$$

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

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:

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

3.2.2 Minimum types

Abadi and Cardelli have proved that the simply-typed ς-calculus enjoys minimum typing[2]: 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 $\Delta \vdash_\varsigma \mathbf{mintype}_\Delta(a) <: A$.

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

The meta-level function **mintype**[4] is defined as follows:

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

3.3 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 $[l_i : A_i]_{i\in 1\ldots n}$ and a stands for $[l_i = \varsigma(x_i : A)a_i]_{i\in 1\ldots n}$:

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

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

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 $\mathbf{mintype}_\Delta\ (a.l_j \Leftarrow \varsigma(x : A')u)$ is (by definition) A' but this term reduces to a value of type A.

3.4 Example

The expressivity of the ς-calculus can be illustrated by the following example from Abadi and Cardelli[2] assuming that we have a type Num for numbers and that the simply-typed λ-calculus has been encoded:

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

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

A *PromCell* is a memory cell which can be written once (*set* method), 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* implemented as an object of type *PrivateCell* can be given the type *PromCell* thanks to subsumption.

[4] Bold face is here used to distinguish the meta-level.

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

This section describes an encoding of the simply-typed ς-calculus given by a $\lambda\Pi$m-context and a translation of ς-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[7] producing Dedukti terms from ς-terms and types.

4.1 Encoding of types

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

Unit, product, Σ-types, and Leibnitz equality can all be encoded in $\lambda\Pi$m (they are special cases of inductive types, which are translated to $\lambda\Pi$m by Coqine[4]) so we will consider that they are available with the usual notations (respectively `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:

> `domain` : `Type`.
> `nil` : `domain`.
> `cons` : `label` $\to$ `domain` $\to$ `domain`.

We will use the notation $[l_1; \ldots; l_n]$ for (`cons` l_1 $(\ldots$ (`cons` l_n `nil`) $\ldots)$).

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 *positions*:

> $\bullet \in \bullet$: `label` $\to$ `domain` $\to$ `Type`.
> `at-head` : $\Pi l.\Pi d.l \in$ `cons` l d.
> `in-tail` : $\Pi l_1.\Pi l_2.\Pi d.l_1 \in d \to l_1 \in$ `cons` l_2 d.

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:

> `type` : `Type`.
> `typenil` : `type`.
> `typecons` : `label` $\to$ `type` $\to$ `type` $\to$ `type`.

The $\lambda\Pi$-term `type` should not be confused with the $\lambda\Pi$-term `Type`; the former is the $\lambda\Pi$ equivalent of ς-types and the latter is sort of all the $\lambda\Pi$-types.

A translation function $[\![\bullet]\!]$ from ς-types to $\lambda\Pi$-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 Section 3.4 are translated as follows:

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

4.1.3 Design choices

This encoding of ς-types as association lists is a bit under-specified: the type `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:

$$\begin{array}{l}
\texttt{type} : \texttt{Type}. \\
\texttt{minors} : \texttt{label} \to \texttt{type} \to \texttt{Type}. \\
\texttt{typenil} : \texttt{type}. \\
\texttt{typecons} : \Pi l : \texttt{label}.\Pi A : \texttt{type}.\Pi B : \texttt{type}.\texttt{minors}\ l\ B \to \texttt{type}. \\
\texttt{minors-nil} : \Pi l : \texttt{label}.\texttt{minors}\ l\ \texttt{typenil}. \\
\texttt{minors-cons} : \Pi l.\Pi l'.\Pi A.\Pi B.\texttt{minors}\ l'\ B \to \\
\qquad\qquad l < l' \to \texttt{minors}\ l\ (\texttt{typecons}\ l'\ A\ B).
\end{array}$$

  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:

$$\begin{array}{lcl}
\texttt{typecons}\ l_1\ A_1\ (\texttt{typecons}\ l_2\ A_2\ B) & \hookrightarrow & \texttt{typecons}\ l_2\ A_2\ (\texttt{typecons}\ l_1\ A_1\ B). \\
\texttt{typecons}\ l\ A_1\ (\texttt{typecons}\ l\ A_2\ B) & \hookrightarrow & \texttt{typecons}\ l\ A_1\ B.
\end{array}$$

  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 `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 λΠ-terms of type `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:

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

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

We will abbreviate $\texttt{assoc}\ A\ l\ p$ as $A._p l$ or $A.l$ leaving the position p implicit.

4.1.5 Subtyping relations

The subtyping relation is defined by:

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

where $=_{\texttt{type}}$ is the Leibnitz equality defined on `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 [7].

▶ **Lemma 1** (subtype-weakening). *The $\leq$ relation enjoys weakening; it means that in $\lambda\Pi$m, we can define a total 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 (as explained previously, the function ***`subtype-weakening`*** is defined by two rewrite rules, one for $B \equiv \texttt{typenil}$ and another for $B \equiv \texttt{typecons } \ldots$). ◀

▶ **Lemma 2** (subtype-refl). *The $\leq$ relation is reflexive; in $\lambda\Pi$m, we can define a total 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 $\lambda\Pi$m, we can define a total 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 (there is no rewrite rule for this case because it is an empty case)
- if B is $\texttt{typecons } l'\ B_1\ B_2$, 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 $l \equiv l'$ and p' proves the goal, or p is in tail and we conclude using the induction hypothesis.

◀

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

Proof. By induction on B.

- base case is trivial
- if B is `typecons` $l'\ B_1\ B_2$, we have some position $p' : l' \in \texttt{dom}\ A$ such that $A.{}_{p'} l' =_{\texttt{type}} B_1$ 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 $l \equiv l'$ and $B.{}_p l \equiv B_1$. $A.{}_{\texttt{subtype-dom}\ A\ B\ st\ l\ p} l' \equiv A.{}_{p'} l' \equiv B_1$
  - 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 λΠm, we can define a total 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`. ◀

4.2 Encoding of terms

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

These functions preserve typing in the sense that we can define, in λΠm, 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 λΠm.

We define a λΠm-context reflecting the syntax and the semantics of the ς-calculus. We start with concrete objects, we then define coercions reflecting the use of the subsumption rule. From these declarations, we define the λΠm version of selection and update, and finally we give the translation function for terms.

4.2.1 Objects

`Expr` A represents the λΠm-type of well-typed objects of type A and `Meth` $A\ B$ represents the λΠm-type of methods of A returning an object of type B.

We can declare `Expr` and `Meth`:

```
Expr : type → Type.
Meth : type → type → 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 λΠm-type `Preobj` $A\ d$ by the following declarations:

```
Preobj : type → domain → Type.
prenil : ΠA.Preobj A nil.
precons : ΠA.Πd.Πl.Πp : l ∈ dom A.
            Meth A A.p l → Preobj A d → Preobj A (cons l d).
```

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

```
Obj A ↪ Preobj A (dom A).
```

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

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

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

$$\begin{aligned}&\texttt{expr-of-obj} : \Pi A.\texttt{Obj}\ A \to \texttt{Expr}\ A.\\ &\texttt{expr-of-obj}\ a \hookrightarrow (A, a, \texttt{subtype-refl}\ A).\end{aligned}$$

We would like to define $\texttt{Meth}\ A\ B$ as $\texttt{Expr}\ A \to \texttt{Expr}\ B$ 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 $\texttt{Meth}\ A\ B$ is equivalent to $\texttt{Expr}\ A \to \texttt{Expr}\ B$:

$$\begin{aligned}&\texttt{Eval-meth} \ : \Pi A.\Pi B.\texttt{Meth}\ A\ B \to \texttt{Expr}\ A \to \texttt{Expr}\ B.\\ &\texttt{Make-meth} \ : \Pi A.\Pi B.(\texttt{Expr}\ A \to \texttt{Expr}\ B) \to \texttt{Meth}\ A\ B.\end{aligned}$$

The key point here is that `Eval-meth` and `Make-meth` will freeze reduction. For example 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.

To get a reduction-preserving encoding, we just have to add some rewrite rules; either the rule $\texttt{Eval-meth}\ A\ B\ (\texttt{Make-meth}\ A\ B\ f)\ a \hookrightarrow f\ a$ or the following one $\texttt{Meth}\ A\ B \hookrightarrow \texttt{Expr}\ A \to \texttt{Expr}\ B$ (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 β 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 `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 $\leq$:

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

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

4.2.3 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 using 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 $\lambda\Pi m$, we can define a total function* `preobj-subset` *of type* $\Pi A.\Pi d.\texttt{Preobj}\ A\ d \to d \subset \texttt{dom}\ A$.

Proof. Straightforward by induction on d. ◀

The definition of update is straightforward:

$$
\begin{array}{l}
\texttt{preupdate} : \Pi A.\Pi d.\Pi l.\Pi p : l \in d.\Pi po : \texttt{Preobj}\ A\ d. \\
\qquad\qquad \texttt{Meth}\ A\ A._{\texttt{preobj-subset}\ A\ d\ po\ l\ p} l \to \texttt{Preobj}\ A\ d. \\
\texttt{obj-update} : \Pi A.\Pi l.\Pi p : l \in \texttt{dom}\ A.\texttt{Obj}\ A \to \texttt{Meth}\ A\ A._p l \to \texttt{Obj}\ A. \\
\texttt{update} : \Pi A.\Pi l.\Pi p : l \in \texttt{dom}\ A.\texttt{Expr}\ A \to \texttt{Meth}\ A\ A._p l \to \texttt{Expr}\ A. \\
\\
\texttt{preupdate}\ A\ (\texttt{cons}\ l\ d)\ l\ (\texttt{at-head}\ l\ d)\ (\texttt{precons}\ A\ d\ l\ p'\ m'\ po)\ m \\
\quad \hookrightarrow \texttt{precons}\ A\ d\ l\ p'\ m\ po. \\
\texttt{preupdate}\ A\ (\texttt{cons}\ l'\ d)\ l\ (\texttt{in-tail}\ l\ l'\ d\ p)\ (\texttt{precons}\ A\ d\ l'\ p'\ m'\ po)\ m \\
\quad \hookrightarrow \texttt{precons}\ A\ d\ l'\ p'\ m'\ (\texttt{preupdate}\ A\ d\ l\ p\ po\ m). \\
\\
\texttt{obj-update}\ A\ l\ p\ a\ m \quad \hookrightarrow \quad \texttt{preupdate}\ A\ (\texttt{dom}\ A)\ l\ p\ a\ m.
\end{array}
$$

`obj-update` 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 $\texttt{Meth}\ B\ A.l$ (for some l) where `obj-update` needs a $\texttt{Meth}\ A\ A.l$. This can be solved by a substitution of the $self$ variable by its coercion $self \uparrow_A^B$ in the method body, which is easy to write as $(\texttt{Make-meth}\ A\ A.l\ ((\lambda(self : \texttt{Expr}\ A)\ (\texttt{Eval-meth}\ B\ A.l\ m\ (self \uparrow_A^B)))))$. Hence we can define `update` as follows:

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

Selection is a bit more subtle because we need both the selected method, which is found by inductively destructing the object, and the full object which should be substituted for the self variable. The `preselect` function doesn't return an object but the method associated with the label. The `select` function duplicates its argument a, one copy is passed to `preselect` and the other is used with the returned method to build a blocked redex using the `Eval-meth` axiom.

$$
\begin{array}{l}
\texttt{preselect} : \Pi A.\Pi d.\Pi l.\Pi p : l \in d.\texttt{Preobj}\ A\ d \to \texttt{Meth}\ A\ (A._p l). \\
\texttt{obj-select} : \Pi A.\Pi l.\Pi p : l \in \texttt{dom}\ A.\texttt{Obj}\ A \to \texttt{Meth}\ A\ (A._p l). \\
\texttt{select} : \Pi A.\Pi l.\Pi p : l \in \texttt{dom}\ A.\texttt{Expr}\ A \to \texttt{Expr}\ A._p l. \\
\\
\texttt{preselect}\ A\ (\texttt{cons}\ l\ d)\ l\ (\texttt{at-head}\ l\ d)\ (\texttt{precons}\ A\ d\ l\ p'\ m\ po) \\
\quad \hookrightarrow m. \\
\texttt{preselect}\ A\ (\texttt{cons}\ l'\ d)\ l\ (\texttt{in-tail}\ l\ l'\ d\ p)\ (\texttt{precons}\ A\ d\ l'\ p'\ m'\ po) \\
\quad \hookrightarrow \texttt{preselect}\ A\ d\ l\ p\ po. \\
\\
\texttt{obj-select}\ A\ l\ p\ a \quad \hookrightarrow \quad \texttt{preselect}\ A\ (\texttt{dom}\ A)\ l\ p\ a. \\
\\
\texttt{select}\ B\ l\ p\ (A, a, st) \quad \hookrightarrow \quad \texttt{Eval-meth}\ A\ A._p\ l \\
\qquad\qquad\qquad\qquad\qquad\qquad (\texttt{obj-select}\ A\ l\ p\ a)\ (A, a, st).
\end{array}
$$

4.2.4 Translation function for expressions

We now have all we need to define a translation function from simply-typed ς-terms to λΠm.

The same ς-term a may have to be translated to different λΠ-terms of different types because λΠm lacks subtyping and subsumption. Hence we have to parameterize our translation function by the targeted type A of a in the ς-calculus. Fortunately, it is enough to define the translation function for the minimum type of a, written $[\![a]\!]_\Delta$. We can then define the general translation function for type A as $[\![a]\!]_{\Delta,A} := [\![a]\!]_\Delta \uparrow^{[\![A]\!]}_{[\![\mathbf{mintype}_\Delta(a)]\!]}$ where the proof of $[\![\mathbf{mintype}_\Delta(a)]\!] \leq [\![A]\!]$ is computed by a meta-level[5] function **decide-subtype** (omitted here).

The $[\![\bullet]\!]_\Delta$ function, the $[\![\bullet]\!]_{\Delta,A}$ function and the translation function for methods are mutually defined by:

$$[\![a]\!]_{\Delta,A} := [\![a]\!]_\Delta \uparrow^{[\![A]\!]}_{[\![\mathbf{mintype}_\Delta(a)]\!]}$$

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

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

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

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 $\texttt{in-tail}\ l_n\ l_1\ (\ldots(\texttt{in-tail}\ l_n\ l_{n-1}\ (\texttt{at-head}\ l_n\ [\,])\ )\ldots)$, 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). This technique is generally known as Higher-Order Abstract Syntax (HOAS)[25].

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

[5] The function **decide-subtype** is easy to define at the meta-level but could also be defined in λΠm.

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

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

$$
\begin{array}{ll}
[\![[l : [\,]]]\!]_{\emptyset} & \equiv \texttt{typecons}\ l\ \texttt{typenil}\ \texttt{typenil} \\
[\![x.l]\!]_{x:[l:[\,]]} & \equiv \texttt{select}\ [\![[l : [\,]]]\!]_{x:[l:[\,]]}\ l\ p_1\ x \\
[\![\varsigma(x : [l : [\,]])x.l]\!]_{\emptyset,[]} & \equiv \texttt{Make-meth}\ [\![[l : [\,]]]\!]_{\emptyset}\ [\,]\ (\lambda x : \texttt{Expr}\ [\![[l : [\,]]]\!]_{\emptyset}.[\![x.l]\!]_{x:[l:[\,]]}) \\
[\![[l = \varsigma(x : [l : [\,]])x.l]]\!]_{\emptyset} & \equiv ([\![[l : [\,]]]\!]_{\emptyset}, \\
 & \qquad \texttt{precons}\ [\![[l : [\,]]]\!]_{\emptyset}\ [\,]\ l\ p_1\ [\![\varsigma(x : [l : [\,]])x.l]\!]_{\emptyset,[]}\ \texttt{prenil}\ [\![[l : [\,]]]\!]_{\emptyset}, \\
 & \qquad \texttt{subtype-refl}\ [\![[l : [\,]]]\!]_{\emptyset}) \\
[\![[l = \varsigma(x : [l : [\,]])x.l].l]\!]_{\emptyset} & \equiv \texttt{select}\ [\![[l : [\,]]]\!]_{\Delta}\ l\ p_1\ [\![[l = \varsigma(x : [l : [\,]])x.l]]\!]_{\emptyset} \\
 & \hookrightarrow \texttt{Eval-meth}\ [\![[l : [\,]]]\!]_{\emptyset}\ [\,]\ l \\
 & \qquad (\texttt{obj-select}\ [\![[l : [\,]]]\!]_{\emptyset}\ l\ p_1 \\
 & \qquad\quad (\texttt{precons}\ \ldots)) \\
 & \qquad\quad\quad [\![[l = \varsigma(x : [l : [\,]])x.l]]\!]_{\emptyset} \\
 & \hookrightarrow \texttt{Eval-meth}\ [\![[l : [\,]]]\!]_{\emptyset}\ [\,]\ l \\
 & \qquad (\texttt{preselect}\ [\![[l : [\,]]]\!]_{\emptyset}\ [l]\ l\ p_1 \\
 & \qquad\quad (\texttt{precons}\ \ldots)) \\
 & \qquad\quad\quad [\![[l = \varsigma(x : [l : [\,]])x.l]]\!]_{\emptyset} \\
 & \hookrightarrow \texttt{Eval-meth}\ [\![[l : [\,]]]\!]_{\emptyset}\ [\,]\ l \\
 & \qquad\quad (\texttt{Make-meth}\ [\![[l : [\,]]]\!]_{\emptyset}\ [\,]\ (\lambda x : \texttt{Expr}\ [\![[l : [\,]]]\!]_{\emptyset}.[\![x.l]\!]_{x:[l:[\,]]})) \\
 & \qquad\quad\quad [\![[l = \varsigma(x : [l : [\,]])x.l]]\!]_{\emptyset}
\end{array}
$$

4.3 Properties of the encoding

Let Γ_0 be the λΠm-context composed of the declarations and rewrite rules presented previously in this section. We investigate properties of the rewrite system $\mathbf{R}_0$ associated with

Γ_0 and of translated ς-terms in contexts of the form Γ_0, Λ where Λ is a $\lambda\Pi$-context (a $\lambda\Pi$m-context without rewrite rule) so the rewrite system associated with Γ_0, Λ is $\mathbf{R}_0$.

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

4.3.1 Normalization and confluence

The rewrite system $\mathbf{R}_0$ 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[23].

In order to be extra-confident in these properties, we implemented the definitions of Γ_0 in the Calculus of Inductive Constructions, which is known to be strongly normalizing and confluent[12], and type-checked this implementation with Coq.

Our code is available at `http://sigmaid.gforge.inria.fr`. However this translation to Coq uses axioms (`Meth`, `Make-meth`, and `Eval-meth`) which are *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 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* $l \equiv l_j$ *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 $l_1 > \ldots > l_n$. $\texttt{dom}\ [\![[l_i : A_i]_{i\in 1\ldots n}]\!]$ is $[l_n; \ldots; l_1]$. We prove that $l_j \in [l_n; \ldots; l_1]$ by induction on n:

- case $n \equiv 0$: the hypothesis $j \in 1\ldots n$ is a contradiction.
- case $n \equiv p+1$: if $j \equiv p+1$ then $\texttt{at-head}\ j\ [l_p; \ldots; l_1]$ 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.** *If* $j \in 1\ldots n$, *then* $[\![[l_i : A_i]_{i\in 1\ldots n}]\!].{}_{pos} l_j \equiv [\![A_j]\!]$ *where pos is the proof of the previous lemma.*

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

▶ **Theorem 10.** *For every type* A *and* B, *if* $\Delta \vdash_\varsigma A <: B$ *then* $[\![A]\!] \leq [\![B]\!]$.

Proof. We proceed by induction on the derivation of $\Delta \vdash_\varsigma A <: B$, there are three cases:

- case (subtype)
  A is some $[l_i : A_i]_{i\in 1\ldots n+m}$ with $B \equiv [l_i : A_i]_{i\in 1\ldots n}$. Without loss of generality, we may assume $l_n < l_{n-1} < \ldots < l_2 < l_1$. We proceed by induction on n:
  - case $n \equiv 0$: $[\![B]\!] \equiv \texttt{typenil}$ hence $[\![A]\!] \leq [\![B]\!]$.
  - case $n \equiv p+1$: $[\![B]\!] \equiv \texttt{typecons}\ l_{p+1}\ [\![A_{p+1}]\!]\ [\![[l_i : A_i]_{i\in 1\ldots p}]\!]$.

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

    pos and the equality proof are given by Lemma 8 and Lemma 9. The proof of $[\![A]\!] \leq [\![[l_i : A_i]_{i\in 1\ldots p}]\!]$ is given by the induction hypothesis.

- case (refl)
  This is trivial by Lemma 2.
- case (trans)
  This is trivial by Lemma 5.

◀

▶ **Theorem 11.** *The translation function on types is injective: if* $[\![A]\!] =_{\mathit{type}} [\![B]\!]$ *then* $A \equiv B$.

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

▶ **Theorem 12.** *For every type* A *and* B*, well-formed in context* Δ*, if* $[\![A]\!] \leq [\![B]\!]$ *then* $\Delta \vdash_\varsigma A <: B$.

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

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

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

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 $l_{p+1} \equiv l'_j$ 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 $A \equiv [l_i : B_i; l'_j : A_j]_{i \in 1 \ldots p, j \in 1 \ldots m}$ so $\Delta \vdash_\varsigma A <: B$ by rule (subtype). ◀

4.3.3 Type preservation

We want to prove the following type preservation theorem:

▶ **Theorem 13.** *If, in the simply typed* ς*-calculus, the judgment* $\Delta \vdash_\varsigma a : A$ *is valid, then the encoded judgment* $[\![\Delta]\!] \vdash_d [\![a]\!]_{\Delta,A} : \mathit{Expr}\ [\![A]\!]$*, is valid in* $\lambda\Pi m$.

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

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

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

Proof. From $\Delta \vdash_\varsigma a : A$ we get, by minimality, $\Delta \vdash_\varsigma \mathbf{mintype}_\Delta\,(a) <: A$ hence $[\![\mathbf{mintype}_\Delta\,(a)]\!] \leq [\![A]\!]$ by Theorem 10. Therefore $[\![a]\!]_{\Delta,A} \equiv [\![a]\!]_\Delta \uparrow^{[\![A]\!]}_{[\![\mathbf{mintype}_\Delta(a)]\!]}$ has type $\mathtt{Expr}\ [\![A]\!]$. ◀

▶ **Lemma 15.** *If, in* $\lambda\Pi m$*, the judgment* $[\![\Delta]\!], x : \mathit{Expr}\ [\![A]\!] \vdash_d [\![b]\!]_{(\Delta, x:A), B} : \mathit{Expr}\ [\![B]\!]$ *is valid, then so is the judgment* $[\![\Delta]\!] \vdash_d [\![\varsigma(x : A)b]\!]_{\Delta,B} : \mathit{Meth}\ [\![A]\!]\ [\![B]\!]$.

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: $[\![\Delta]\!] \vdash_d \lambda x : \texttt{Expr}\ [\![A]\!].[\![b]\!]_{(\Delta,x:A),B} : \texttt{Expr}\ [\![A]\!] \to \texttt{Expr}\ [\![B]\!]$.

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

▶ **Theorem 16.** *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 $\lambda\Pi m$.*

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 $\mathbf{mintype}_\Delta\,(a) \equiv \mathbf{mintype}_\Delta\,(x) \equiv \Delta(x)$. By definition of $[\![\Delta]\!]$, $x \in [\![\Delta]\!]$ and $[\![\Delta]\!](x) \equiv \texttt{Expr}\ [\![\Delta(x)]\!] \equiv \texttt{Expr}\ [\![\mathbf{mintype}_\Delta\,(a)]\!]$.
- case (obj): a is $[l_i = \varsigma(x_i : A)\ a_i]_{l_1<\ldots<l_n}$ with $\mathbf{mintype}_\Delta\,(a) \equiv A \equiv [l_i : A_i]_{l_1<\ldots<l_n}$.
  $$\begin{aligned}[\![a]\!]_\Delta &\equiv [\![[l_i = \varsigma(x_i : A)\ a_i]_{l_1<\ldots<l_n}]\!]_\Delta \\ &\equiv \texttt{expr-of-obj}\ (\\ &\qquad \texttt{precons}\ [\![A]\!]\ [l_2;\ldots;l_n]\ l_1\ p_1\ [\![\varsigma(x : A)a_1]\!]_{\Delta,A.p_1 l_1}\ (\\ &\qquad \ldots(\texttt{precons}\ [\![A]\!]\ [\ ]\ l_n\ p_n\ [\![\varsigma(x : A)a_n]\!]_{\Delta,A.p_n l_n}\ \texttt{prenil}\ [\![A]\!])))\end{aligned}$$
  The term $\texttt{expr-of-obj}$ has type $\texttt{Obj}\ [\![A]\!] \to \texttt{Expr}\ [\![A]\!]$ 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 $\texttt{dom}\ [\![A]\!]$:
    $$\begin{aligned}\texttt{dom}\ [\![A]\!] &\equiv \texttt{dom}\ [\![[l_i : A_i]_{l_1<\ldots<l_n}]\!] \\ &\equiv \texttt{dom}\ (\texttt{typecons}\ l_1\ [\![A_1]\!]\ (\ldots(\texttt{typecons}\ l_n\ [\![A_n]\!]\ \texttt{typenil})\ldots)) \\ &\equiv [l_1;\ldots;l_n]\end{aligned}$$
    hence $\texttt{Obj}\ [\![A]\!] \equiv \texttt{Preobj}\ [\![A]\!]\ (\texttt{dom}\ [\![A]\!]) \equiv \texttt{Preobj}\ [\![A]\!]\ [l_1;\ldots;l_n]$.
  - We show by induction that each built preobject is well-typed with the expected type. For all $i \in 1\ldots n$,
    $$\begin{aligned}[\![\Delta]\!] \vdash_d\ &\texttt{precons}\ [\![A]\!]\ [l_{i+1};\ldots;l_n]\ l_i\ p_i\ [\![\varsigma(x : A)a_i]\!]_{\Delta,A.p_i l_i}\ (\\ &\ldots(\texttt{precons}\ [\![A]\!]\ [\ ]\ l_n\ p_n\ [\![\varsigma(x : A)a_n]\!]_{\Delta,A.p_n l_n}\ \texttt{prenil}\ [\![A]\!])) \\ &: \texttt{Preobj}\ [\![A]\!]\ [l_i;\ldots;l_n]\end{aligned}$$
    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 $a'.l_j$ 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'[6]: $\mathbf{mintype}_\Delta\,(a') \equiv [l_i : A_i]_{i\in 1\ldots n}$ so $\mathbf{mintype}_\Delta\,(a) \equiv A_j$.
  Lemma 8 gives us a position $p : l_j \in \texttt{dom}\ [\![\mathbf{mintype}_\Delta\,(a')]\!]$ hence by Lemma 9, $[\![\mathbf{mintype}_\Delta\,(a')]\!].p l_j \equiv [\![A_j]\!] \equiv [\![\mathbf{mintype}_\Delta\,(a)]\!]$.
  Moreover, $\Delta \vdash_\varsigma [\![a']\!]_\Delta : \texttt{Expr}\ [\![\mathbf{mintype}_\Delta\,(a')]\!]$ by induction hypothesis thus $[\![a]\!]_\Delta \equiv \texttt{select}\ [\![\mathbf{mintype}_\Delta\,(a')]\!]\ l_j\ p\ [\![a']\!]_\Delta$ has type $\texttt{Expr}\ [\![\mathbf{mintype}_\Delta\,(a)]\!]$.
- case (update): a is of the form $a'.l_j \Leftarrow \varsigma(x : A)b$ with $j \in 1\ldots n$, $\Delta \vdash_\varsigma a' : A$, and $\Delta, x : A \vdash_\varsigma b : A_j$ where $A \equiv [l_i : A_i]_{i\in 1\ldots n}$.
  By induction hypothesis and Lemma 14, $[\![\Delta]\!] \vdash_d [\![a']\!]_{\Delta,A} : \texttt{Expr}\ A$. By Lemma 15, $[\![\Delta]\!] \vdash_d [\![\varsigma(x : A)b]\!]_{\Delta,A_j} : \texttt{Meth}\ [\![A]\!]\ [\![A_j]\!]$.

[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.

Like in the previous case, Lemma 8 gives us a position $p : l_j \in \texttt{dom}\ [\![A]\!]$ and by Lemma 9, $[\![A]\!].{}_p l_j \equiv [\![A_j]\!]$.

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

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

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

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

◀

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 ς-calculus is not.

However, we may want the following weaker result:

▶ Statement 1. If $\Delta \vdash_\varsigma a : A$ and $a \rightarrowtail a'$ then $[\![a]\!]_{\Delta,A} =_{\texttt{Expr}\ [\![A]\!]} [\![a']\!]_{\Delta,A}$ is inhabited in context $[\![\Delta]\!]$.

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

Hence we would need

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

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[7]:

▶ **Theorem 17.** *For any label l, the type* $\big(\Pi B.\Pi C.\Pi f.\Pi b.\textit{Eval-meth}\ B\ C\ (\textit{Make-meth}\ B\ C\ f)\ b =_{\textit{Expr}\ C} f\ b\big) \to ([\,] =_{\textit{type}} [l : [\,]])$ *is inhabited.*

Proof.

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

```
underlying-type : ΠB.Expr B → type.
underlying-type B (A, a, st) ↪ A.
```

- Let A_0 be the type $[l : [\,]]$ and a_0 an inhabitant of $\texttt{Expr}\ A_0$ (for instance $a_0 : \texttt{Expr}\ A_0 := [l = [\,]]$). $t_0 := a_0 \uparrow^{[\,]}_{A_0}$ is an inhabitant of $\texttt{Expr}\ [\,]$ which we can distinguish from the empty expression $[\,]$ because they have different underlying types.
- We define a function $\texttt{swap} : \texttt{Expr}\ [\,] \to \texttt{Expr}\ [\,]$ returning an expression different from its argument:

```
swap-aux : type → Expr [ ].
swap-aux typenil ↪ t0.
swap-aux (typecons l' B C) ↪ [ ].
swap : Expr [ ] → Expr [ ].
swap b ↪ swap-aux (underlying-type b).
```

- We remark that $\texttt{Expr}\ A_0$ is isomorphic to $\texttt{Expr}\ A_0 \to \texttt{Expr}\ [\,]$:

- We can define a function $\texttt{elim-A}_0 : \texttt{Expr}\ A_0 \to \texttt{Expr}\ A_0 \to \texttt{Expr}\ [\,]$ by

  $$\texttt{elim-A}_0\ [l = \varsigma(x)f(x)] := f$$

- and a function $\texttt{intro-A}_0 : (\texttt{Expr}\ A_0 \to \texttt{Expr}\ [\,]) \to \texttt{Expr}\ A_0$ by

  $$\texttt{intro-A}_0\ f := [l = \varsigma(x)f(x)]$$

- let $E_0 : \texttt{Expr}\ A_0 \to \texttt{Expr}\ [\,]$ be the function defined by $E_0\ a := \texttt{swap}(\texttt{elim-A}_0\ a)$. Then $b_0 : \texttt{Expr}\ [\,] := E_0\ (\texttt{intro-A}_0\ E_0)$ is such that we can prove, using the `reduce-meth` axiom, $b_0 =_{\texttt{Expr}\ [\,]} \texttt{swap}\ b_0$, hence $\texttt{underlying-type}\ (\texttt{swap}\ b_0) = \texttt{underlying-type}\ (\texttt{swap}\ (\texttt{swap}\ b_0))$ but $\texttt{swap}\ b_0$ is either $[\,]$ or t_0 and we get $([\,] =_{\texttt{type}} [l : [\,]])$ 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 $([\,] =_{\texttt{type}} [l : [\,]])$ 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:

$$\texttt{Meth}\ A\ B \hookrightarrow \texttt{Expr}\ A \to \texttt{Expr}\ B.$$
$$\texttt{Eval-meth}\ A\ B\ m \hookrightarrow m.$$
$$\texttt{Make-meth}\ A\ B\ f \hookrightarrow f.$$

From this, the `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 $\texttt{Expr}\ B$ as $\Sigma A : \texttt{type}.(\texttt{Obj}\ A) \times (A \leq B)$, we rewrite $\texttt{Expr}\ A$ to $\texttt{Obj}\ A$ and change the definitions of the functions that destructed expressions: `update`, `select`, and `coerce`:

$$\texttt{Expr}\ A \hookrightarrow \texttt{Obj}\ A.$$
$$\texttt{expr-of-obj}\ a \hookrightarrow a.$$
$$\texttt{update}\ \bullet\ \bullet\ \bullet\ (\texttt{precons}\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet)\ \bullet \hookrightarrow \texttt{obj-update}\ \bullet\ \bullet\ \bullet\ (\texttt{precons}\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet)\ \bullet.$$
$$\texttt{update}\ B\ l\ p\ (\texttt{coerce}\ A\ B\ st\ a)\ m \hookrightarrow \texttt{coerce}\ A\ B\ st\ \left(\texttt{update}\ A\ l\ (\texttt{subtype-dom}\ A\ B\ st\ l\ p)\ a\ (\lambda(self).m\ (self \uparrow_A^B))\right).$$
$$\texttt{select}\ \bullet\ \bullet\ \bullet\ (\texttt{precons}\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet) \hookrightarrow \texttt{obj-select}\ \bullet\ \bullet\ \bullet\ (\texttt{precons}\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet).$$
$$\texttt{select}\ B\ l\ p\ (\texttt{coerce}\ A\ B\ st\ a) \hookrightarrow \texttt{select}\ A\ l\ (\texttt{subtype-dom}\ A\ B\ st\ l\ p)\ a.$$
$$\texttt{coerce}\ B\ C\ st_{BC}\ (\texttt{coerce}\ A\ B\ st_{AB}\ a) \hookrightarrow \texttt{coerce}\ A\ C\ (\texttt{subtype-trans}\ st_{AB}\ st_{BC})\ a$$

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

We call Γ_1 this new λΠm-context and $\mathbf{R}_1$ the new rewrite system. We believe that $\mathbf{R}_1$ is confluent because the non-orthogonal part reflects the simply-typed ς-calculus known to be confluent[2], but have not formally checked it. However, $\mathbf{R}_1$ 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 `update`, `select`, and `coerce`. The new encoding has the additional property of semantics preservation:

▶ **Theorem 18.** *If* $\Delta \vdash_\varsigma a : A$ *and* $a \rightarrowtail a'$ *then* $[\![a]\!]_{\Delta,A} \hookrightarrow^+ [\![a']\!]_{\Delta,A}$.

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:* $[\![a]\!]_{(\Delta_1,x:B),A}\{[\![b]\!]_{(\Delta_1,\Delta_2),B}/x\} \equiv [\![a\{b/x\}]\!]_{(\Delta_1,\Delta_2),A}$.

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

▶ **Lemma 20.** *Unicity of subtype proofs: if* st_1 *and* st_2 *both have type* $[\![A]\!] \leq [\![B]\!]$ *then* $st_1 =_{[\![A]\!]\leq[\![B]\!]} st_2$.

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 β[8] of type `type`: if st_1 and st_2 both have type $[\![A]\!] \leq \beta$ then $st_1 =_{[\![A]\!]\leq\beta} st_2$.

We proceed by induction on β.

- base case: $\beta \equiv$ `typenil`
  $[\![A]\!] \leq \beta \equiv [\![A]\!] \leq$ `typenil` $\equiv$ `unit`. The type `unit` has only one inhabitant so $st_1 =_{[\![A]\!]\leq\beta} st_2$.

[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.

[8] We use the greek letter β here to distinguish the ς-term B and the λΠ-term β which abstracts $[\![B]\!]$.

- inductive case: $\beta \equiv \texttt{typecons}\ l\ \beta_1\ \beta_2$
  A is some $[l_i : A_i]_{l_1<\ldots<l_n}$.
  By definition of $\leq$,

$$[\![A]\!] \leq \texttt{typecons}\ l\ \beta_1\ \beta_2 \equiv \Sigma p : l \in [l_1; \ldots; l_n].([\![A]\!].{}_p l =_{\texttt{type}} \beta_1) \times ([\![A]\!] \leq \beta_2)$$

  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.
  $[\![A]\!] \leq \texttt{typecons}\ l\ \beta_1\ \beta_2$ is isomorphic to $([\![A]\!].{}_{p_0} l =_{\texttt{type}} \beta_1) \times ([\![A]\!] \leq \beta_2)$.
  The left type $[\![A]\!].{}_{p_0} l =_{\texttt{type}} \beta_1$ has at most one inhabitant thanks to Hedberg Theorem[17] because equality on `type` is decidable; the right type $[\![A]\!] \leq \beta_2$ has only one element by induction hypothesis so $st_1 =_{[\![A]\!] \leq \beta} st_2$.

◀

We can now prove Theorem 18:

Proof. The simply-typed ς-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 $\texttt{Expr}\ [\![A]\!]$ 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 $a'' := [l_i = \varsigma(x_i : A'')a_i]_{i\in 1\ldots n}$ and $A'' := [l_i : A_i]_{i=1\ldots n}$.
  So $a \equiv a''.l_j$ and $a' \equiv a_j\{a''/x_j\}$.
  We look at the minimum types of a and a':
  - $\mathbf{mintype}_\Delta\,(a'') \equiv A'' \equiv [l_i : A_i]_{i\in 1\ldots n}$ so $\mathbf{mintype}_\Delta\,(a) \equiv \mathbf{mintype}_\Delta\,(a''.l_j) \equiv A_j$
  - We call A' the minimum type of a', by minimality we know that $\Delta \vdash_\varsigma A' <: A_j$.

  $[\![a'']\!]_\Delta$ is of the form $(\ldots(\texttt{precons}\ [\![A'']\!]\ [l_{j+1}; \ldots; l_n]\ l_j\ [\![\varsigma(x_j : A'')a_j]\!]_{\Delta,A_j} \ldots\ )\ )$, we abbreviate it as α.

$$\begin{array}{lcl}
[\![a]\!]_\Delta & \equiv & \texttt{select}\ [\![A'']\!]\ l_j\ p\ \alpha \\
& \hookrightarrow & \texttt{obj-select}\ [\![A'']\!]\ l_j\ p\ \alpha\ \alpha \\
& \hookrightarrow & \texttt{preselect}\ [\![A'']\!]\ [l_1; \ldots; l_n]\ l_j\ p\ \alpha\ \alpha \\
& \hookrightarrow^* & [\![\varsigma(x_j : A'')a_j]\!]_{\Delta,A_j}\ \alpha \\
& \equiv & \left(\lambda x_j : \texttt{Expr}\ [\![A'']\!].[\![a_j]\!]_{(\Delta,x_j:A''),A_j}\right)\ \alpha \\
& \longrightarrow_\beta & [\![a_j]\!]_{(\Delta,x_j:A''),A_j}\{[\![a'']\!]_\Delta/x\}
\end{array}$$

  Hence, by Lemma 19, we get exactly $[\![a]\!]_\Delta \hookrightarrow^+ [\![a']\!]_{\Delta,A_j}$.
  Finally,

$$\begin{array}{lcl}
[\![a]\!]_{\Delta,A} & \equiv & [\![a]\!] \uparrow^{[\![A]\!]}_{[\![\mathbf{mintype}_\Delta(a)]\!]} \\
& \hookrightarrow^+ & [\![a']\!]_{\Delta,A_j} \uparrow^{[\![A]\!]}_{[\![A_j]\!]} \\
& \equiv & \left([\![a']\!]_\Delta \uparrow^{[\![A_j]\!]}_{[\![\mathbf{mintype}_\Delta(a')]\!]}\right) \uparrow^{[\![A]\!]}_{[\![A_j]\!]} \\
& \hookrightarrow & [\![a']\!]_\Delta \uparrow^{[\![A]\!]}_{[\![\mathbf{mintype}_\Delta(a')]\!]} \\
& \equiv & [\![a']\!]_{\Delta,A}
\end{array}$$

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

◀

6 Conclusion

We defined an embedding of the simply-typed ς-calculus to λΠm and implemented it in Dedukti as a compiler named sigmaid (SIGMA-calculus In Dedukti)[7]. This implementation has been tested on the following original examples from Abadi and Cardelli:

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

Despite non-termination of the ς-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[21] in λΠm 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 λΠm.

References

1 Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *In Proc. TACS'94, Theoretical Aspects of Computing Sofware*, pages 296–320, 1994.

2 Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer New York, 1996.

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

4 Mathieu Boespflug and Guillaume 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 Mathieu Boespflug, Quentin Carbonneaux, Olivier Hermant, and Ronan Saillard. Dedukti: a universal proof checker. `http://dedukti.gforge.inria.fr`.

6 Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999.

7 Raphaël Cauderlier. Sigmaid. `http://sigmaid.gforge.inria.fr`.

8 Alberto Ciaffaglione, Luigi Liquori, and Marino Miculan. Reasoning about object-based calculi in (co)inductive type theory and the theory of contexts. *J. Autom. Reason.*, 39(1):1–47, July 2007.

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

10 Horatiu Cirstea, Luigi Liquori, and Benjamin Wack. Rewriting calculus with fixpoints: Untyped and first-order systems. In *TYPES*, volume 3085. Springer, 2003.

11 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.

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

13 Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *TLCA*, volume 4583 of *LNCS*, pages 102–117. Springer, 2007.

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

15 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.

16 Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993.

17 Michael Hedberg. A coherence theorem for martin-löf's type theory. *J. Functional Programming*, pages 4–8, 1998.

18 Ludovic Henrio and Florian Kammüller. A mechanized model of the theory of objects. In MarcelloM. Bonsangue and EinarBroch Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 4468 of *Lecture Notes in Computer Science*, pages 190–205. Springer Berlin Heidelberg, 2007.

19 Ludovic Henrio, Florian Kammüller, Bianca Lutz, and Henry Sudhof. Locally nameless sigma calculus. *Archive of Formal Proofs*, April 2010. `http://afp.sf.net/entries/Locally-Nameless-Sigma.shtml`, Formal proof development.

20 focalize-devel@inria.fr. Focalize. `http://focalize.inria.fr`.

21 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.

22 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.

23 Fritz Müller. Confluence of the lambda calculus with left-linear algebraic rewriting. *Information Processing Letters*, 41(6):293–299, 1992.

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

25 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Programming Language Design and Implementation*, 1988.

26 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.

27 Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

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