An Interpretation of Extensible Objects

Gérard Boudol (*) and *Silvano Dal-Zilio*

INRIA Sophia Antipolis
BP 93 – 06902 Sophia Antipolis Cedex

Abstract

We provide a translation of Fisher-Honsell-Mitchell's delegation-based object calculus with subtyping into a λ-calculus with extensible records. The target type system is an extension of the system $\mathcal{F}^\omega$ of dependent types with recursion, extensible records and a form of bounded universal quantification. We show that our translation is computationally adequate, that the typing rules of Fisher-Honsell-Mitchell's calculus can be derived in a rather simple and natural way, and that our system enjoys the standard subject reduction property.

Keywords

objects, delegation, extensible records, λ-calculi, type systems

1. Introduction

The theoretical foundations of object-oriented programming have been intensively explored in the recent past, the main purpose being to design type systems that would be safe, allowing in particular to prevent some run-time errors like *message not understood*, while being at the same time flexible enough to support object-oriented programming idioms. This has proven to be actually quite difficult. Some early works, following Cardelli's pioneering paper [8], developed *encodings* of object-oriented constructions into calculi with records. For instance, Cook & al. in [11] proposed an interpretation of class-based programming. In their model, an object is the fixed-point of a function (of self) returning a record. An object can only be invoked, while inheritance acts on the object's "generator", by adding or updating fields of the resulting record, and extending the scope of the self parameter. The types, based on F-bounded quantification (or else higher-order quantification and recursive types, see [18]), allow in particular to address the property identified as *method specialization* by Mitchell [19], by which is meant the fact that the type of a method is updated whenever the hosting class is inherited.

Mitchell's work deals with delegation-based object-oriented programming, where a single entity, called a *prototype*, may be both invoked, by messages calling for some methods to be executed, and inherited, to build a new prototype by adding a new method or modifying an existing one. Although this looks simpler than the class-based approach, the property of method specialization is not easy to obtain in this setting. For this reason, Mitchell & al. proposed in [12] a primitive object calculus, with a specific typing construct for prototypes, that is not derived from an encoding. Indeed,

(*) `phone: (+33/0)4 92 38 79 40  fax: (+33/0)4 92 38 79 98   email: gbo@sophia.inria.fr`

the authors conclude that "[*method specialization*] *seems very difficult to achieve directly with any calculus of records*". Several other primitive object calculi were developed at about the same time by Abadi and Cardelli (see for instance [1, 2]) to formalize abstractly what can be expected from the typing of objects, while avoiding the difficulties of discovering adequate encodings in λ-calculi with records. Later on however, these authors, together with R. Viswanathan [3], solved the encoding problem for a delegation-based calculus without object extension (hence with no non-trivial method specialization), and Viswanathan improved their results in [20], where the target of the translation is a first-order calculus.

Our purpose in this paper is to introduce an encoding of Mitchell & al. calculus, extended with subtyping [14], into a λ-calculus with extensible records equipped with a rich but fairly standard type system, that has been recognized as a suitable framework for studying the typing of objects, see [18, 7]. We show in particular how the typing rules for prototypes can be derived in a natural way. The idea of the encoding is very simple. To explain it, let us first come back to the difficulty of typing method specialization for prototypes: there are two seemingly opposite requirements for types in this setting. One is that, when the prototype is invoked, its type should not tell too much, because the prototype can be revealed upon invocation, for instance by an identity method. In particular, the type should only exhibit the method names that are actually available, since otherwise runtime errors could occur. On the other hand, when the prototype is inherited, its type – or more precisely the type of the self parameter – should be "open" to potential extensions (see [14] where the authors distinguish a "client interface" from the "inheritance interface" for an object). This tension is solved in Cook's model, simply by fulfilling separately these two requirements.

Then the idea of our encoding is to *separate the two usages* of a prototype by means of record field selection: in our interpretation, a prototype is a recursive record with two fields. The first one, that we call inht, contains the current value of the prototype generator, that is the function of self that returns the record of methods. The other field, called invk, contains the application of the generator to the prototype itself. The first field is selected to inherit the prototype, while the second is selected to invoke it. We show how to type such a prototype in a system involving extensible record types, types depending on types and a limited form of bounded universal quantification, thus deriving naturally the typing rules of [12] – or more precisely rules given by Fisher in her thesis [15] (1). We also show that the subtyping of prototypes proposed in [14] arises in a very simple way: while a "pro" type of a prototype is a record with two fields, the "obj" type only contains the field invk, thus allowing a "sealed prototype" to be only invoked. The standard subtyping rule for recursive types allows to derive the subtyping relations of [14].

We think that our interpretation validates, and even justifies the (non-trivial) rules designed for primitive type constructors in Fisher and Mitchell's systems, and allows to reuse results that can be established in the target calculus. Bruce in [6] has shown that, from a semantical point of view, Cook and Mitchell's models are equivalent. Our interpretation confirms this view from a syntactical perspective, and also confirms that Bruce's matching – that is width subtyping of the record of methods – is the form of subtyping we need to type extensible objects, as far as the goal is to retrieve Fisher and Mitchell's systems. Our interpretation also shows that it is possible to do a little better, allowing the type of an updated method to be specialized to a subtype, by using the standard subtyping relation, together with width subtyping which is useful to type self-inflicted method update.

Note

For lack of space, we have omitted most of the formal description of Fisher-Honsell-Mitchell's typing system. We hope that the reader unfamiliar with it can grasp some idea of it from our interpretation.

(1) we could perhaps have used Cook's typing, involving F-bounded quantification, but for the purpose of translating Fisher-Honsell-Mitchell's systems this looks less convenient.

2. The Calculus of Prototypes

The calculus of prototypes introduced by Fisher, Honsell and Mitchell in [12] is a λ-calculus extended with constructions for building prototypes from the empty one, namely prototype extension, denoted $\langle P \leftarrow\!\!+\, \ell = Q\rangle$, and method update $\langle P \leftarrow \ell = Q\rangle$. There is also an operation of method invocation, written $P \Leftarrow \ell$. In [12] the notation $\langle P \leftarrow\!\!\circ\, \ell = Q\rangle$ is introduced, to mean either $\langle P \leftarrow\!\!+\, \ell = Q\rangle$ or $\langle P \leftarrow \ell = Q\rangle$. Then the evaluation of prototypes mainly consists in the following transition:

$$\langle P \leftarrow\!\!\circ\, \ell = Q\rangle \Leftarrow \ell \;\rightarrow\; Q\langle P \leftarrow\!\!\circ\, \ell = Q\rangle$$

There are actually several possible ways to define the evaluation mechanism for prototypes. In [12] a subsidiary "bookkeeping" transition system is used to set the construction of a prototype into an appropriate shape. In [15] an evaluation strategy is defined, that uses an auxiliary "get method" function, to extract the method of a given name from a prototype. A similar technique is used in [4] where the language is enriched with a notation for method extraction. Another auxiliary operation is used in [16], which is a combination of method extraction and self-application. Moreover, in that paper no distinction is made between the two ways of building prototypes, and the notation $\langle P \leftarrow \ell = Q\rangle$ stands for prototype extension as well as method update – or override. The two facets of this operation are revealed in the typing system.

Since from the operational point of view our translation mimics the prototypes exactly as they are described in [16], we adopt the syntax of this paper. We assume given a denumerable set $\mathcal{X}$ of *variables*, and a denumerable set $\mathcal{K}$, disjoint from $\mathcal{X}$, of *keys*, or labels, used as method names. We use x, y, $z\ldots$ and ℓ, k to range over $\mathcal{X}$ and $\mathcal{K}$ respectively. The grammar for terms is as follows:

$$\begin{array}{rcll}
P,\,Q & ::= & x \;\mid\; W \;\mid\; (PQ) & \lambda\textit{-calculus} \\
 & \mid & (P \Leftarrow \ell) & \textit{method invocation} \\
 & \mid & \mathsf{S}(P,\ell,Q) & \textit{subsidiary operation} \\
W & ::= & \lambda xP \;\mid\; O & \textit{values} \\
O & ::= & \langle\rangle & \textit{empty prototype} \\
 & \mid & \langle P \leftarrow \ell = Q\rangle & \textit{prototype extension/update}
\end{array}$$

As usual we denote the term $\langle\langle\langle\rangle \leftarrow \ell_1 = P_1\rangle \cdots \leftarrow \ell_n = P_n\rangle$ by $\langle \ell_1 = P_1, \ldots, \ell_n = P_n\rangle$. We assume the reader is familiar with the notions of free and bound variables, α-conversion and substitution. This applies to terms, types, typing contexts, and so on. Given any such item X we denote by $\mathsf{fv}(X)$ the set of its free variables, and $[X/x]Y$ denotes the result of substituting X for x in Y.

The meaning of the auxiliary operation $\mathsf{S}(P,\ell,Q)$ is that it extracts from the prototype P the method of name ℓ and applies it to Q. Then the (lazy) evaluation relation over prototypes is given by the following rules:

$$\begin{array}{rcll}
(\lambda xP)Q & \rightarrow & [Q/x]P & \\
(O \Leftarrow \ell) & \rightarrow & \mathsf{S}(O,\ell,O) & \\
\mathsf{S}(\langle P \leftarrow \ell = Q\rangle,\ell,P') & \rightarrow & (QP') & \\
\mathsf{S}(\langle P \leftarrow k = Q\rangle,\ell,P') & \rightarrow & \mathsf{S}(P,\ell,P') & k \neq \ell \\
P \rightarrow P' & \Rightarrow & \left\{\begin{array}{l} (PQ) \rightarrow (P'Q) \\ (P \Leftarrow \ell) \rightarrow (P' \Leftarrow \ell) \\ \mathsf{S}(P,\ell,Q) \rightarrow \mathsf{S}(P',\ell,Q) \end{array}\right. &
\end{array}$$

As usual we denote by $\overset{*}{\rightarrow}$ the reflexive and transitive closure of this relation, and we denote by $P\Downarrow$ the fact that there exists a value W such that $P \overset{*}{\rightarrow} W$. Since evaluation is deterministic, this value, if it exists, is unique, and we may denote it $\mathsf{eval}(P)$.

There are several versions of the type system for the calculus of prototypes. The system presented in [14] is essentially the one of [12], enriched with a form of subtyping. Here we use a simplified version of the system given in Fisher's thesis [15]. The simplifications are as follows: we adopt the system of Chapter 3 of [15], which does not involve variance annotation nor existential quantification, with a restriction on the assumptions regarding row variables given below. Moreover, we do not take subsumption into account in the simplified system (this will be dealt with in a next section). Then the typing system we consider here is defined as follows. The grammar of kinds, types and rows is

$$\begin{array}{rcll} \mathit{kind} & ::= & \mathrm{T} \mid \kappa & \\ \kappa & ::= & \mathrm{M} \mid \mathrm{T} \to \mathrm{M} & \text{where } \mathrm{M} = \{\ell_1, \ldots, \ell_n\} \\ \tau & ::= & t \mid (\tau_1 \to \tau_2) \mid \mathsf{pro}\, t.\rho & \\ \rho & ::= & r \mid [] \mid [\rho, \ell : \tau] \mid (\lambda t.\rho) \mid (\rho\tau) & \end{array}$$

The kind T is that of well-formed types (the only constraint is on $\mathsf{pro}\, t.\rho$). The kind $\{\ell_1, \ldots, \ell_n\}$ is that of a row which does not contain the keys $\ell_1, \ldots, \ell_n$. The typing contexts are

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau \mid \Gamma, t :: \mathrm{T} \mid \Gamma, r <:_w \rho$$

Notice that an assumption about a row variable r only involves the "width subtyping" relation $<:_w$. Moreover, in Fisher's thesis, there is a kind annotation, that is the assumption is $r <:_w \rho :: \kappa$; here we omit this annotation since in our simplified system this κ is always $\mathrm{T} \to \emptyset$. The type judgements are

$\Gamma \vdash *$	*well-formed context*	$\Gamma \vdash \rho <:_w \rho'$	*subrow*
$\Gamma \vdash \tau :: \mathrm{T}$	*type is well-formed*	$\Gamma \vdash P : \tau$	*term has type*
$\Gamma \vdash \rho :: \kappa$	*row has kind*		

For lack of space, we omit most of the rules of the system, which are quite standard. For instance the only interesting rule for well-formedness of types is

$$\frac{\Gamma, t :: \mathrm{T} \vdash \rho :: \mathrm{M}}{\Gamma \vdash \mathsf{pro}\, t.\rho :: \mathrm{T}}$$

where the well-formedness of the context $\Gamma, t :: \mathrm{T}$ implies that t is not free in Γ. The interesting rules for typing terms are:

$$\frac{\Gamma \vdash *}{\Gamma \vdash \langle\rangle : \mathsf{pro}\, t.[]}\ (\mathit{empty\ pro}) \qquad \frac{\Gamma \vdash P : \mathsf{pro}\, t.\rho \quad , \quad \Gamma, t :: \mathrm{T} \vdash \rho <:_w [\ell : \tau]}{\Gamma \vdash P \Leftarrow \ell : [\mathsf{pro}\, t.\rho/t]\tau}\ (\mathit{pro} \ \Leftarrow)$$

where $[\ell : \tau]$ denotes $[[], \ell : \tau]$,

$$\frac{\begin{array}{l}\Gamma \vdash P : \mathsf{pro}\, t.\rho \\ \Gamma, t :: \mathrm{T} \vdash \rho :: \{\ell\} \\ \Gamma, r <:_w \lambda t.[\rho, \ell : \tau] \vdash Q : [\mathsf{pro}\, t.(rt)/t](t \to \tau)\end{array}}{\Gamma \vdash \langle P \leftarrow \ell = Q\rangle : \mathsf{pro}\, t.[\rho, \ell : \tau]}\ r \notin \mathsf{fv}(\tau) \ \ (\mathit{pro\ ext})$$

and

$$\frac{\begin{array}{l}\Gamma \vdash P : \mathsf{pro}\, t.\rho \\ \Gamma, t :: \mathrm{T} \vdash \rho <:_w [\ell : \tau] \\ \Gamma, r <:_w \lambda t.\rho \vdash Q : [\mathsf{pro}\, t.(rt)/t](t \to \tau)\end{array}}{\Gamma \vdash \langle P \leftarrow \ell = Q\rangle : \mathsf{pro}\, t.\rho}\ r \notin \mathsf{fv}(\tau) \ \ (\mathit{pro\ over})$$

We also need a rule for the subsidiary operation:

$$\frac{\Gamma \vdash P : \mathsf{pro}\, t.\rho \quad , \quad \Gamma \vdash Q : \mathsf{pro}\, t.\rho' \quad , \quad \Gamma\,,\, t :: \mathrm{T} \vdash \rho' <:_w \rho <:_w [\ell : \tau]}{\Gamma \vdash \mathsf{S}(P, \ell, Q) : [\mathsf{pro}\, t.\rho'/t]\tau} \;(subsid)$$

As we said, there are two distinct rules for typing the construction $\langle P \leftarrow \ell = Q \rangle$. In the first one, the premise $\Gamma\,,\, t :: \mathrm{T} \vdash \rho :: \{\ell\}$ indicates that in the prototype P there is no method named ℓ, and then this rule allows to type a true extension. On the other hand, the premise $\Gamma\,,\, t :: \mathrm{T} \vdash \rho <:_w [\ell : \tau]$ says the a method with name ℓ is already provided by P, with result type τ. Then one may override this method, with another one having the same result type. We refer to [12] for examples of typing.

3. Encoding Prototypes with Extensible Records

As a target for our encoding of prototypes, we use a λ-calculus enriched with extensible records. We also find it convenient to use an explicit fixpoint construct. The syntax of the calculus is as follows:

$$\begin{array}{rcll}
M,\, N \ldots & ::= & x \mid V \mid (MN) & \textit{λ-calculus} \\
& \mid & \mathsf{fix}\, x.M & \textit{fixpoint} \\
& \mid & (M.\ell) & \textit{field selection} \\
V & ::= & \lambda x M \mid R & \textit{values} \\
R & ::= & [] & \textit{empty record} \\
& \mid & [M, \ell = N] & \textit{record extension}
\end{array}$$

where $x \in \mathcal{X}$ and $\ell \in \mathcal{K}$. For ease of readability, one quite often writes $(\lambda x M)$ for $\lambda x M$, and MN for (MN), and similarly $M.\ell$ for $(M.\ell)$. We denote by $[\ell_1 = M_1, \ldots, \ell_n = M_n]$ the record $[\cdots[[], \ell_1 = M_1] \cdots \ell_n = M_n]$. The evaluation relation $\rightarrow$ for the λ-calculus with records is the least one satisfying the following rules:

$$\begin{array}{rcll}
(\lambda x M)N & \rightarrow & [N/x]M & \\
\mathsf{fix}\, x.M & \rightarrow & [\mathsf{fix}\, x.M/x]M & \\
[M, \ell = N].\ell & \rightarrow & N & \\
[M, k = N].\ell & \rightarrow & M.\ell & k \neq \ell \\
M \rightarrow M' & \Rightarrow & \left\{ \begin{array}{l} MN \rightarrow M'N \\ M.\ell \rightarrow M'.\ell \end{array} \right. &
\end{array}$$

Again, since evaluation is deterministic, we may denote by $\mathsf{eval}(M)$ the unique value V, if it exists, such that $M \xrightarrow{*} V$.

Now we turn to the interpretation of the calculus of prototypes. To represent a prototype, say $\langle \ell_1 = P_1, \ldots, \ell_n = P_n \rangle$, where the P_i's are functions of the prototype itself (usually in the form of a self parameter), we shall use a function returning a record

$$G = \lambda\, \mathsf{self}\, [\ell_1 = M_1 \mathsf{self}\,, \ldots\,,\, \ell_n = M_n \mathsf{self}\,]$$

This is an "object definition" in Cook's model (see [11, 13]), also called "object generator" (see [2, 3]). We cannot however use directly this representation with a "self-application semantics", because of well-known typing problems (see for instance [2]), nor a "recursive-record semantics" [8], since it has difficulties in modelling the method update operation. As we said in the introduction, in our interpretation of a prototype, we separate the two usages we have of it, namely invocation and inheritance. This idea of separating the usages of an object is embodied in the "split-method" of [3, 20], but we cannot use this specific approach here because the set of methods of a prototype is not fixed once for all. To give our translation, let us introduce a notation: we define

$$\begin{array}{rcl} \mathsf{proto} & =_{\mathrm{def}} & \mathsf{fix}\, p.\lambda z.[\, \mathsf{inht} = z \,,\, \mathsf{invk} = z(pz)\,] \\ & \rightarrow & \lambda z.[\, \mathsf{inht} = z \,,\, \mathsf{invk} = z(\mathsf{proto}\, z)\,] \end{array}$$

Then the translation $[\![.]\!]$ from the λ-calculus of prototypes to the λ-calculus of extensible records is given by – omitting the part of the translation that regards the λ-calculus, which is trivial:

$$\boxed{\begin{array}{rcl} [\![\langle\rangle]\!] & = & \mathsf{proto}(\lambda\,\mathsf{self}\,[\,]) \\ [\![\langle P \leftarrow \ell = Q\rangle]\!] & = & \mathsf{proto}(\lambda\,\mathsf{self}\,[[\![P]\!].\mathsf{inht}\,\mathsf{self}, \ell = [\![Q]\!]\,\mathsf{self}]) \\ [\![P \Leftarrow \ell\,]\!] & = & [\![P]\!].\mathsf{invk}.\ell \\ [\![\mathsf{S}(P,\ell,Q)]\!] & = & ([\![P]\!].\mathsf{inht}\,[\![Q]\!]).\ell \end{array}}$$

In the translation we obviously assume that the name self is not free in the source terms. To shorten the notation, we shall replace self by the ordinary λ-variable x in the sequel. Notice that our translation also allows us to define a "(pre)method extraction" operation, that may be denoted $(P \hookrightarrow \ell)$, as follows:

$$[\![P \hookrightarrow \ell]\!] = \lambda\,\mathsf{self}\,(([\![P]\!].\mathsf{inht}\,\mathsf{self}\,).\ell)$$

As one can see, our interpretation is very close to Cook's model, separating objects from object generators, with inheritance acting on the generators; in particular, extending and updating a prototype are operationally the same. There is a difference with Cook's model, however, since prototypes feature runtime extension. Let us see an example of how the translation works. We have, if we let $G = \lambda x[[\![P]\!].\mathsf{inht}\,x, \ell = [\![Q]\!]x]$

$$\begin{array}{rcl} [\![\langle P \leftarrow \ell = Q\rangle \Leftarrow \ell]\!] & = & (\mathsf{proto}\,G).\mathsf{inht}.\ell \\ & \xrightarrow{*} & [\,\mathsf{inht} = G \,,\, \mathsf{invk} = G(\mathsf{proto}\,G)\,].\mathsf{invk}.\ell \\ & \xrightarrow{*} & [\![Q]\!](\mathsf{proto}\,G) \\ & = & [\![Q\langle P \leftarrow \ell = Q\rangle]\!] \end{array}$$

Our aim for the rest of this section is to show the soundness of our translation from an operational point of view. More precisely, we aim at establishing that a prototype converges – i.e. evaluates to a value – if and only if its translation converges. To this end, we define a relation $\mathcal{S}$ between the source and target calculi defined as follows: $\mathcal{S}$ is the least relation satisfying

$$\begin{gathered} x\,\mathcal{S}\,x \\ \lambda x P\,\mathcal{S}\,\lambda x[\![P]\!] \\ P\,\mathcal{S}\,M \ \Rightarrow\ (PQ)\,\mathcal{S}\,(M[\![Q]\!]) \\ [\![O]\!] \xrightarrow{*} M \ \Rightarrow\ O\,\mathcal{S}\,M \\ P\,\mathcal{S}\,M \quad \& \quad M.\mathsf{invk} \xrightarrow{*} N \ \Rightarrow\ (P \Leftarrow \ell)\,\mathcal{S}\,(N.\ell) \\ P\,\mathcal{S}\,M \quad \& \quad M.\mathsf{inht}\,[\![Q]\!] \xrightarrow{*} M' \ \Rightarrow\ \mathsf{S}(P,\ell,Q)\,\mathcal{S}\,(M'.\ell) \end{gathered}$$

It is easy to see that the following holds:

LEMMA 3.1. *$P\,\mathcal{S}\,[\![P]\!]$ for any P, and if $O\,\mathcal{S}\,M$ then $M \xrightarrow{*} [\,\mathsf{inht} = G \,,\, \mathsf{invk} = G(\mathsf{proto}\,G)\,]$ for some G.*

Notice also that $P\,\mathcal{S}\,M \;\Rightarrow\; [\![P]\!] \xrightarrow{*} M$. Now we prove that the relation $\mathcal{S}$ is a "simulation", that is:

LEMMA 3.2.
(i) $P\,\mathcal{S}\,M \quad \& \quad P \rightarrow P' \;\Rightarrow\; \exists M'.\, M \xrightarrow{*} M' \quad \& \quad P'\,\mathcal{S}\,M'$
(ii) $P\,\mathcal{S}\,M \quad \& \quad M \rightarrow M' \;\Rightarrow\; \exists P'.\, P \xrightarrow{*} P' \quad \& \quad P'\,\mathcal{S}\,M'$

PROOF: (i) we prove this point by induction on the definition of $P \rightarrow P'$, and on the definition of $\mathcal{S}$. If $P = (\lambda x Q)S$ and $P' = [S/x]Q$, then we have $M = (\lambda x[\![Q]\!])[\![S]\!]$, and it is easy to see that $[\![P']\!] = [[\![S]\!]/x][\![Q]\!]$, therefore we may let $M' = [\![P']\!]$, since $P'\,\mathcal{S}\,M'$ by the previous lemma.

If $P = (O \Leftarrow \ell)$ and $P' = \mathsf{S}(O,\ell,O)$, then we have $M = N.\ell$ with $[\![O]\!] \xrightarrow{*} L$ and $L.\mathsf{invk} \xrightarrow{*} N$ for some L. Since $[\![O]\!] = (\mathsf{proto}\,\lambda x R)$ for some R, it is easy to see that $N \xrightarrow{*} [[\![O]\!]/x]R$. Now let $M' = [[\![O]\!]/x]R.\ell$. We have $M \xrightarrow{*} M'$, and $[\![O]\!].\mathsf{inht}\,[\![O]\!] \xrightarrow{*} [[\![O]\!]/x]R$, therefore $P'\,\mathcal{S}\,M'$ by definition of $\mathcal{S}$.

If $P = \mathsf{S}(\langle P_0 \leftarrow \ell = Q\rangle,\ell,P_1)$ and $P' = (QP_1)$ then there exist M_0 and N such that $[\![O]\!] \xrightarrow{*} M_0$ where $O = \langle P_0 \leftarrow \ell = Q\rangle$ and $M_0.\mathsf{inht}\,[\![P_1]\!] \xrightarrow{*} N$ with $M = N.\ell$, and it is easy to see that

$$N \xrightarrow{*} [[\![P_0]\!].\mathsf{inht}\,[\![P_1]\!], \ell = [\![Q]\!][\![P_1]\!]]$$

Therefore we may let $M' = [\![Q]\!][\![P_1]\!]$. The case where $P = \mathsf{S}(\langle P_0 \leftarrow k = Q\rangle,\ell,P_1)$ and $P' = \mathsf{S}(P_0,\ell,P_1)$ is similar: again there exist M_0 and N such that $[\![O]\!] \xrightarrow{*} M_0$ where $O = \langle P_0 \leftarrow k = Q\rangle$, and $M_0.\mathsf{inht}\,[\![P_1]\!] \xrightarrow{*} N$ with $M = N.\ell$. Then

$$N \xrightarrow{*} [[\![P_0]\!].\mathsf{inht}\,[\![P_1]\!], \ell = [\![Q]\!][\![P_1]\!]]$$

and therefore we have $M \xrightarrow{*} ([\![P_0]\!].\mathsf{inht}\,[\![P_1]\!]).\ell = M'$, and $P'\,\mathcal{S}\,M'$ by definition of $\mathcal{S}$.

If $P = (Q \Leftarrow \ell)$ and $P' = (Q' \Leftarrow \ell)$ with $Q \rightarrow Q'$, then $M = N.\ell$ for some N such that there exists L with $Q\,\mathcal{S}\,L$ and $L.\mathsf{invk} \xrightarrow{*} N$. By induction hypothesis, there exists L' such that $L \xrightarrow{*} L'$ and $Q'\,\mathcal{S}\,L'$. Moreover one can check that, since Q is not a value, we must have $N = L''.\mathsf{invk}$ with $L \xrightarrow{*} L''$. Since evaluation is deterministic, we have $L' \xrightarrow{*} L''$ or $L'' \xrightarrow{*} L'$, and in any case we can find N' such that $N \xrightarrow{*} N'$ and $L'.\mathsf{invk} \xrightarrow{*} N'$, and we let $M' = N'.\ell$. The case where $P = \mathsf{S}(P_0,\ell,P_1)$ and $P' = \mathsf{S}(P_0',\ell,P_1)$ with $P_0 \rightarrow P_0'$ is similar: there exist L and N such that $M = N.\ell$ with $L.\mathsf{inht}\,[\![P_1]\!] \xrightarrow{*} N$ and $P_0\,\mathcal{S}\,L$. Again one uses the induction hypothesis, and observes that since P_0 is not a value, $N = L''.\mathsf{inht}\,[\![P_1]\!]$ for some L'' such that $L \xrightarrow{*} L''$.

In the case where $P = (P_0Q)$ and $P' = (P_0'Q)$ with $P_0 \rightarrow P_0'$, we simply use the induction hypothesis.

(ii) We proceed by induction on the definition of $\mathcal{S}$. The case where $M = (N[\![Q]\!])$ with $P = (P_0Q)$ and $P_0\,\mathcal{S}\,N$ is easy: we have either $N = \lambda x[\![P_0']\!]$ and $P_0 = \lambda x P_0'$ with $M' = [[\![Q]\!]/x][\![P_0']\!] = [\![[Q/x]P_0']\!]$, and we use the previous lemma, or $M' = (N'[\![Q]\!])$ with $N \rightarrow N'$, and we use the induction hypothesis. If $O\,\mathcal{S}\,M$ and $M \rightarrow M'$ then $O\,\mathcal{S}\,M'$.

If $M = N.\ell$ and $P = (Q \Leftarrow \ell)$ with $Q\,\mathcal{S}\,N'$ for some N' such that $N'.\mathsf{invk} \xrightarrow{*} N$, then we have either $M' = N''.\ell$ with $N \rightarrow N''$, in which case $P\mathcal{S}\,M'$, or $N = [N_0, k = N_1]$, with $M' = N_1$ if $k = \ell$ and $M' = N_0.\ell$ if $k \neq \ell$. Then one can see that one must have $Q = \langle Q_0 \leftarrow k = Q_1\rangle$ with $N_0 = [\![Q_0]\!].\mathsf{inht}\,[\![Q]\!]$ and $N_1 = [\![Q_1]\!][\![Q]\!]$. If $k = \ell$ we let $P' = (Q_1Q)$, and otherwise we let $P' = \mathsf{S}(Q_0,\ell,Q)$, and we use the previous lemma.

The case where $M = N'.\ell$ and $P = \mathsf{S}(P_0,\ell,P_1)$ with $P_0\,\mathcal{S}\,N$ and $N.\mathsf{inht}\,[\![P_1]\!] \xrightarrow{*} N'$ is similar: either $M' = N''.\ell$ with $N' \rightarrow N''$, in which case $P\mathcal{S}\,M'$, or $N' = [N_0, k = N_1]$, with $M' = N_1$ if $k = \ell$ and $M' = N_0.\ell$ if $k \neq \ell$. Again one must have $P_0\langle Q_0 \leftarrow k = Q_1\rangle$ with $N_0 = [\![Q_0]\!].\mathsf{inht}\,[\![P_1]\!]$ and $N_1 = [\![Q_1]\!][\![P_1]\!]$, and it is easy to conclude □

As a corollary, we can now prove the soundness of our translation:

Proposition (Computational Adequacy) 3.3. *A prototype P converges if and only if its translation converges, that is $P\Downarrow \;\Leftrightarrow\; [\![P]\!]\Downarrow$. More precisely,*

$$\mathsf{eval}([\![P]\!]) = \mathsf{eval}([\![\mathsf{eval}(P)]\!])$$

Proof: if $P \xrightarrow{*} W$ there exists N such that $[\![P]\!] \xrightarrow{*} N$ and $W\,\mathcal{S}\,N$ by the Lemma 3.2(i), since $P\,\mathcal{S}\,[\![P]\!]$ by Lemma 3.1. Now if $W = \lambda x Q$ then $N = \lambda x[\![Q]\!] = [\![W]\!]$, and if W is $\langle\rangle$ or $\langle Q \leftarrow \ell = T\rangle$ then $[\![W]\!] \xrightarrow{*} N \xrightarrow{*} [\,\mathsf{inht} = G\,,\ \mathsf{invk} = G(\mathsf{proto}\,G)\,]$ for some G by the definition of $\mathcal{S}$ and the Lemma 3.1.

Conversely if $[\![P]\!] \xrightarrow{*} V$, by the Lemma 3.2(ii) there exists Q such that $P \xrightarrow{*} Q$ and $Q\,\mathcal{S}\,V$. It is easy to see, from the definition of $\mathcal{S}$, that this can only hold if Q is a value, with $[\![Q]\!] \xrightarrow{*} V$ ❑

4. Deriving the Typing Rules

In this section we show how to derive the typing rules for prototypes. We first introduce a type system $\mathcal{T}$ for our λ-calculus with extensible records. This system features arrow types and extensible record types – also called row expressions, following [21] –, as well as recursive types, types depending on types [17] and a limited form of bounded universal quantification [9]. Moreover, we will only allow types which are well-formed with respect to a system of kinds. There are two basic kinds, $\Box$, the kind of record types, and $\Diamond$, the kind of types. Since we also have type operators, the syntax of kinds and types is the following:

$$\begin{array}{rcl} \kappa & ::= & \Box \;\mid\; \Diamond \;\mid\; (\kappa \rightarrow \kappa) \\ \tau,\ \sigma\ldots & ::= & t \;\mid\; (\tau \rightarrow \sigma) \;\mid\; [] \;\mid\; [\sigma, \ell\colon\tau] \;\mid\; (\mu t^{\kappa}.\tau) \;\mid\; (\Lambda t^{\kappa}.\tau) \;\mid\; (\tau\sigma) \;\mid\; (\forall t \lhd \tau.\sigma) \end{array}$$

For instance $\Diamond \rightarrow \Box$ is the kind of interfaces, that is functions from types to records. In $(\forall t \lhd \tau.\sigma)$, the variable t is bound in σ, but not in τ (hence we do not have F-bounded polymorphism). Dependent types are used as prototype's *interface* (see [13, 18]), which typically are of the form $\Lambda t^{\Diamond}.[\ell_1\colon\tau_1, \ldots, \ell_n\colon\tau_n]$. Bounded quantification is used to model the fact that such an interface may be extended.

The preorder in $(\forall t \lhd \tau.\sigma)$ is width subtyping (of rows, not records), as this is precisely the notion of subtyping one needs in order to derive the typing of method specialization (see the comments at the end of this section). We use the notation $\lhd$ since our axiomatization of this preorder makes it slightly more generous than $<:_w$ of [15]. For instance we have $[\ell\colon\tau', \ell\colon\tau] \lhd [\ell\colon\tau]$, but the first of these record types is not even well-formed in Fisher's system. In our setting, we allow a row $[[\sigma, \ell\colon\tau'], \ell\colon\tau]$ to be well-formed, and this will simplify the type system, but the reader should notice that this is a (width) subtype of $[\sigma, \ell\colon\tau']$ only if $\tau' = \tau$. This means that, by contrast with Fisher's system, $[\sigma, \ell\colon\tau] \lhd \sigma$ is not valid in general – if σ exhibits a field ℓ with a type different from τ.

Our type system is given in Curry's style, because types are implicit in the source calculus of prototypes, and also because this simplifies the presentation. The typing contexts of $\mathcal{T}$ are as follows:

$$\Gamma ::= \emptyset \;\mid\; \Gamma, x\colon\tau \;\mid\; \Gamma, t \lhd \tau \;\mid\; \Gamma, t :: \kappa$$

and the type judgements are $\Gamma \vdash *$, $\Gamma \vdash \tau :: \kappa$, $\Gamma \vdash \tau \lhd \sigma$, $\Gamma \vdash M\colon\tau$, with the same meaning as above, plus $\Gamma \vdash \tau <: \sigma$ and $\Gamma \vdash \sigma \sim \tau$ where $\sim$ is our notion of type equality. We omit the rules by which this is a congruence, that contains $\beta\mu$-conversion, that is:

$$\begin{array}{rcl} ((\Lambda t^{\kappa}.\sigma)\tau) & \sim & [\tau/t]\sigma \\ (\mu t^{\kappa}.\tau) & \sim & [\mu t^{\kappa}.\tau/t]\tau \end{array}$$

All the rules of the system, for well-formedness of contexts, kinding of types, subtyping judgements $\Gamma \vdash \tau \lhd \sigma$ and for typing terms, collected in the appendices, are fairly standard (except perhaps for what regards width subtyping of rows). More precisely, system $\mathcal{T}$ consists in the rules collected in the first three appendices. Let us comment on some of the rules: regarding the subtyping of rows,

$$\frac{\Gamma \vdash \sigma \lhd [\ell : \tau]}{\Gamma \vdash [\sigma, \ell : \tau] \lhd \sigma}$$

expresses the fact that if we know that a row σ contains an ℓ field with type τ, then extending it with $\ell : \tau$ actually does not modify it. This rule will be used for typing method update. One should remark that, although in $\mathcal{T}$ we have the usual *subsumption rule*

$$\frac{\Gamma \vdash M : \tau \ , \ \Gamma \vdash \tau <: \sigma}{\Gamma \vdash M : \sigma}$$

this is of limited use, since the only way to infer $\Gamma \vdash \tau <: \sigma$ in $\mathcal{T}$ is by means of the rule

$$\frac{\Gamma \vdash \tau \lhd \sigma}{\Gamma \vdash \tau <: \sigma}$$

That is, in $\mathcal{T}$ we only have "width subsumption". One should also observe that we have no subtyping rule for bounded quantification. However, using the rules of instanciation, subsumption and generalization, one can achieve the same effect. That is, the following inference is valid:

$$\frac{\Gamma \vdash M : (\forall t \lhd \tau . \sigma) \quad , \quad \Gamma \vdash \tau' \lhd \tau \quad , \quad \Gamma \vdash \sigma \lhd \sigma'}{\Gamma \vdash M : (\forall t \lhd \tau' . \sigma')} \tag{0}$$

Our type system is quite close to the one of [7] (see also [18] for a complete description of the same system without recursion), with some differences: we do not use maximal types $\top(\kappa)$, nor existentially quantified types, but we use extensible records rather than just records, and recursion at any kind. Our system is also very close, apart from what concerns subtyping, to the one sketched by Mitchell in [19]. At least superficially, it does not look much more complicated than the system for prototypes sketched above.

To see how the typing rules for prototypes are mimicked in our system, let us first give some valid inferences. To this end, we introduce some notations. Let us define:

$$\begin{array}{rcl}
\delta & =_{\text{def}} & \Diamond \to \Box \\
\gamma & =_{\text{def}} & (\Diamond \to \Box) \to \Box \\
\boldsymbol{\pi} & =_{\text{def}} & \mu p^{\gamma}.\Lambda s^{\delta}.\mu o^{\Box}.[\,\mathsf{inht} : (\forall s' \lhd s.ps' \to s(ps'))\,,\ \mathsf{invk} : so] \\
\boldsymbol{\varsigma} & =_{\text{def}} & \Lambda t^{\delta}.(\forall s' \lhd t.\boldsymbol{\pi} s' \to t(\boldsymbol{\pi} s'))
\end{array}$$

It is easy to see that, for any σ:

$$(\boldsymbol{\pi}\sigma) \sim [\,\mathsf{inht} : (\boldsymbol{\varsigma}\sigma)\,,\ \mathsf{invk} : \sigma(\boldsymbol{\pi}\sigma)\,] \tag{1}$$

and therefore the following rule is admissible:

$$\frac{\Gamma \vdash G : (\boldsymbol{\varsigma}\sigma)}{\Gamma \vdash (\mathsf{proto}\, G) : (\boldsymbol{\pi}\sigma)} \tag{2}$$

This holds in particular whenever $G = \lambda\,\mathsf{self}\,[\ell_1 = M_1\mathsf{self}\,, \dots\,,\ \ell_n = M_n\mathsf{self}]$ is an object generator, which can be given the type $\boldsymbol{\varsigma}\,(\Lambda t^{\Diamond}.\rho)$ where $\rho = [\ell_1 : \tau_1, \dots, \ell_n : \tau_n]$, using the assumption that self

has type $(\boldsymbol{\pi} s')$ with constraint $s' \lhd (\Lambda t^{\diamond}.\rho)$, and that the M_i's have type $t \to \tau_i$. In this case the type of $(\mathsf{proto}\, G)$ is

$$\boldsymbol{\pi}(\Lambda t^{\diamond}.\rho) \sim [\,\mathsf{inht} : (\forall s' \lhd \sigma.\boldsymbol{\pi} s' \to [\boldsymbol{\pi} s'/t]\rho)\,,\,\mathsf{invk} : [\boldsymbol{\pi}\sigma/t]\rho\,] \qquad \text{with} \quad \sigma = \Lambda t^{\diamond}.\rho$$

One can see that invk is a record of type $[\ell_1 : \tau'_1, \ldots, \ell_n : \tau'_n]$, thus exhibiting only the method names that are actually available in the prototype, while inht is a function that takes arguments of type $\boldsymbol{\pi}\sigma'$ with $\sigma' \lhd \sigma$, that is any possible extension to the original prototype, and returns the record of methods "specialized" to this extension. Another admissible rule, derived using the law (1), is:

$$\frac{\Gamma \vdash M : (\boldsymbol{\pi}\sigma)\ ,\ \Gamma \vdash \sigma \lhd \Lambda t^{\diamond}.[\ell : \tau]}{\Gamma \vdash M.\mathsf{invk}.\ell : [\boldsymbol{\pi}\sigma/t]\tau} \tag{3}$$

Again using (1), one can check that for $G = \lambda x[M.\mathsf{inht}\, x, \ell = Nx]$ the following is a valid inference:

$$\frac{\Gamma \vdash M : (\boldsymbol{\pi}\sigma) \quad , \quad \Gamma \vdash \sigma' \lhd \sigma \quad , \quad \Gamma \vdash N : (\forall s \lhd \sigma'.\boldsymbol{\pi} s \to [\boldsymbol{\pi} s/t]\tau)}{\Gamma \vdash G : (\forall s \lhd \sigma'.\boldsymbol{\pi} s \to [\sigma(\boldsymbol{\pi} s), \ell : [\boldsymbol{\pi} s/t]\tau])}$$

Now if we try to exploit the rule (2), to build a prototype out of G, we have to find a way to replace $[\sigma(\boldsymbol{\pi} s), \ell : [\boldsymbol{\pi} s/t]\tau]$ by $\sigma'(\boldsymbol{\pi} s)$. An obvious solution is $\sigma' = \Lambda t^{\diamond}.[\sigma t, \ell : \tau]$, since then $[\sigma(\boldsymbol{\pi} s), \ell : [\boldsymbol{\pi} s/t]\tau] \sim \sigma'(\boldsymbol{\pi} s)$, and this gives us another admissible rule:

$$\frac{\Gamma \vdash M : (\boldsymbol{\pi}\sigma)\ ,\ \Gamma \vdash \Lambda t^{\diamond}.[\sigma t, \ell : \tau] \lhd \sigma\ ,\ \Gamma \vdash N : (\forall s \lhd \Lambda t^{\diamond}.[\sigma t, \ell : \tau].\boldsymbol{\pi} s \to [\boldsymbol{\pi} s/t]\tau)}{\Gamma \vdash (\mathsf{proto}\, \lambda x[M.\mathsf{inht}\, x, \ell = Nx]) : \boldsymbol{\pi}(\Lambda t^{\diamond}.[\sigma t, \ell : \tau])} \tag{4}$$

There is another possibility, however, which is $\sigma' = \sigma$ and $\sigma \lhd \Lambda t^{\diamond}.[\ell : \tau]$, since in this case the following inference is valid:

$$\cfrac{\cfrac{\Gamma \vdash \sigma \lhd \Lambda t^{\diamond}.[\ell : \tau]}{\Gamma \vdash [\sigma(\boldsymbol{\pi} s), \ell : [\boldsymbol{\pi} s/t]\tau] \lhd \sigma(\boldsymbol{\pi} s)}}{\Gamma \vdash \boldsymbol{\pi} s \to [\sigma(\boldsymbol{\pi} s), \ell : [\boldsymbol{\pi} s/t]\tau] \lhd \boldsymbol{\pi} s \to \sigma(\boldsymbol{\pi} s)}$$

Using the inference (0) above, this gives us the following admissible rule:

$$\frac{\Gamma \vdash M : (\boldsymbol{\pi}\sigma) \quad , \quad \Gamma \vdash \sigma \lhd \Lambda t^{\diamond}.[\ell : \tau] \quad , \quad \Gamma \vdash N : (\forall s \lhd \sigma.\boldsymbol{\pi} s \to [\boldsymbol{\pi} s/t]\tau)}{\Gamma \vdash (\mathsf{proto}\, \lambda x[M.\mathsf{inht}\, x, \ell = Nx]) : (\boldsymbol{\pi}\sigma)} \tag{5}$$

Finally it is easy to see – still using (1) – that the following is admissible:

$$\frac{\Gamma \vdash M : (\boldsymbol{\pi}\sigma) \quad , \quad \Gamma \vdash \sigma' \lhd \sigma \lhd \Lambda t^{\diamond}.[\ell : \tau] \quad , \quad \Gamma \vdash N : (\boldsymbol{\pi}\sigma')}{\Gamma \vdash (M.\mathsf{inht}\, N).\ell : [\boldsymbol{\pi}\sigma'/t]\tau} \tag{6}$$

Now the translation from the simplified version of Fisher's system sketched in Section 2 to our type system should be clear. As far as types are concerned, it is given by

$$\boxed{[\![\mathsf{pro}\, t.\rho]\!] = \boldsymbol{\pi}(\Lambda t^{\diamond}.[\![\rho]\!])}$$

(the rest is trivial, e.g. $[\![\lambda t.\rho]\!] = \Lambda t^{\diamond}.[\![\rho]\!]$). Regarding the kinds, we let $[\![\mathrm{M}]\!] = \square$ and $[\![\mathrm{T}]\!] = \diamond$. Translating the contexts in the obvious way (where $r <:_w \rho$ is translated into $r \lhd [\![\rho]\!]$), we have:

THEOREM. *If $\Gamma \vdash M : \tau$ can be inferred in the type system for prototypes, then $[\![\Gamma]\!] \vdash [\![M]\!] : [\![\tau]\!]$ can be proved in the type system $\mathcal{T}$ for the λ-calculus with extensible records.*

The derived inferences (2) to (6) allow us to retrieve the rules (*empty pro*), (*pro* $\Leftarrow$), (*pro ext*), (*pro over*) and (*subsid*) respectively, with $\sigma = \Lambda t^{\diamond}.[\![\rho]\!]$ in the last four cases. One should notice that the only use of subsumption we made is in deriving the rule (*pro over*), via the inference (0). One should also observe that our inference (4) is actually slightly more general than the corresponding rule (*pro ext*), since for the premise $\Gamma \vdash \sigma \lhd \Lambda t^{\diamond}.[\sigma t, \ell : \tau]$ to be valid with $\sigma = \Lambda t^{\diamond}.[\![\rho]\!]$ there may actually be two cases: either ρ does not contain the field ℓ, in which case we are typing a true extension, as in the rule (*pro ext*), or ρ does contain the field ℓ, but then it must be, thanks to the axiomatization of $\lhd$, of type τ, and this is actually a particular case of inference (5). Remark that in the latter the premise regarding N is stronger (a bounded quantification is contravariant in the bound) than in (4), and this is needed to type self-inflicted update, as in the standard "movable point" object – see [12].

Our result is a *completeness* result similar to the one obtained by Abadi & al. in [3]. Its converse does not hold, for some interesting reasons: there is a slight difference in the modelling of prototypes between Fisher-Honsell-Mitchell's calculus and our encoding – or Cardelli's [8] and Cook's [11] models, for that matter –, which is that the underlying record representing the prototype is, in the former, a record of pre-methods, which are functions of self, while it is a record of methods, where the self parameter is free, in the latter. In other word, the self parameter is "late bound" in the recursive record or generator model, while it is bound earlier – that is, before extension – in Fisher-Honsell-Mitchell's prototypes. Then for instance the prototype

$$\langle\langle\langle\rangle \leftarrow k = \lambda\,\mathsf{self}\,(\mathsf{self}.\ell)\rangle \leftarrow \ell = \lambda\,\mathsf{self}\,(\mathsf{self}.\ell)\rangle$$

is not typable in Fisher-Honsell-Mitchell's type system (see [12] for a similar example), while its translation is typable in our system. In fact, this prototype has "semantically" (modulo permutation of fields of different names in a record) the same translation as

$$\langle\langle\langle\rangle \leftarrow \ell = \lambda\,\mathsf{self}\,(\mathsf{self}.\ell)\rangle \leftarrow k = \lambda\,\mathsf{self}\,(\mathsf{self}.\ell)\rangle$$

because both are represented by the "same" generator, and the latter is typable in Fisher-Honsell-Mitchell's calculus.

Finally, one may notice that, when we specialize the inferences (3), (4) and (5) with $\sigma = \Lambda t^{\diamond}.[\![\rho]\!]$, where ρ is supposed to be of record kind, we could reformulate these inferences using a preorder introduced by Bruce in [6] and now called *matching*, denoted $<\#$, which may be defined as follows:

$$\frac{\Gamma \vdash \rho \lhd \rho'}{\Gamma \vdash \mathsf{pro}\, t.\rho <\# \mathsf{pro}\, t.\rho'}$$

Then the reformulated admissible rules would be nothing else than the rules recently proposed for Fisher-Honsell-Mitchell's calculus of prototypes by Bono and Bugliesi in [5]. As shown by Bruce in [6], matching is precisely the kind of subtyping we need in typing the operations of extension and update on prototypes (though the paper [6] actually deals with classes). This explains why we need, at the lower level of records, to use width subtyping.

5. Objects and Subtyping

As we said, our type system $\mathcal{T}$ for the λ-calculus of extensible records only makes use of a particular form of subtyping. It has been remarked by Fisher and Mitchell in [13] that adding the subsumption rule in the typing of prototypes, one actually does not gain anything, since "*in pure delegation based languages, no subtyping is possible*". This is easy to explain looking at our translation: one easily sees that $\boldsymbol{\varsigma}$ is invariant in its type parameter (the bound τ in $\forall t \lhd \tau.\sigma$ is in a contravariant position), and therefore $\boldsymbol{\pi}$ is invariant too. For this reason we cannot deal with the preorder proposed by Bono and Liquori [4] for the calculus of prototypes, nor with the width subtyping of prototypes of fixed size as it is done by Abadi and Cardelli in [2], at least if we stick to the translation given here – it might be possible to combine it with the translation of [3, 20], allowing their "assembled" objects to be part of the syntax, but this looks a bit ad hoc.

In a subsequent paper [14], Fisher and Mitchell enriched their calculus by distinguishing, at the level of types, prototypes from *objects*, which are "sealed prototypes". The distinction is that objects feature only method invocation, but no extension or update. Then a new type construct $\mathsf{obj}\,t.\rho$ is introduced together with three new rules, which are essentially the following, where the typing contexts are enriched with new constraints $t <: \tau$:

$$\frac{\Gamma\,,\,t <: t' \vdash \rho <: \rho'}{\Gamma \vdash \mathsf{pro}\,t.\rho <: \mathsf{obj}\,t'.\rho'} \qquad \frac{\Gamma\,,\,t <: t' \vdash \rho <: \rho'}{\Gamma \vdash \mathsf{obj}\,t.\rho <: \mathsf{obj}\,t'.\rho'} \qquad \frac{\Gamma \vdash P : \mathsf{obj}\,t.\rho \quad , \quad \Gamma\,,\,t : \mathrm{T} \vdash \rho <:_w [\ell : \tau]}{\Gamma \vdash P \Leftarrow \ell : [\mathsf{obj}\,t.\rho/t]\tau}$$

Now consider our typing system $\mathcal{T}$ enriched with subtyping. That is, we introduce an extension, called $\mathcal{T}_{<:}$, of $\mathcal{T}$ in which we have a new kind of contexts, namely $\Gamma, t <: \tau$, and new rules for inferring the judgements $\Gamma \vdash \tau <: \sigma$ – the rules are given in the fourth appendix. If we let

$$\boxed{[\![\mathsf{obj}\,t.\rho]\!] = \boldsymbol{\omega}(\Lambda t^{\diamond}.\rho) \qquad \text{where} \quad \boldsymbol{\omega} =_{\mathrm{def}} \Lambda s^{\delta}.\mu o^{\square}.[\,\mathsf{invk} : so\,]}$$

then it is easy to see, using the standard rule for subtyping recursive types, that the three rules above can be derived in our system with subtyping. Notice that $[\![\mathsf{obj}\,t.\rho]\!] \sim \mu t^{\diamond}.[\mathsf{invk} : \rho]$ and therefore this is essentially what Bruce & al. [7] call the "classical recursive record encoding" (see also [13] § 6.3), which is the natural typing of Cardelli's recursive records [8].

Regarding the full type system with subtyping $\mathcal{T}_{<:}$ for our λ-calculus with extensible records, our main result is *type safety*. Indeed, we can prove the subject reduction property:

THEOREM (SUBJECT REDUCTION). *If $\Gamma \vdash M : \tau$ is provable in $\mathcal{T}_{<:}$ and $M \to M'$ then $\Gamma \vdash M : \tau'$ is provable in $\mathcal{T}_{<:}$.*

Our proof, which, not surprisingly, is quite long and technical, differs from Compagnoni's one [10]; in particular, since we have recursive types, we cannot rely on a strong normalization result. However, since we have no subtyping rule for bounded quantification, we can make a direct proof, analysing the typing of a compound term in typings of its components.

It is possible to exploit subsumption to gain another typing rule for method update, where the type of the method is specialized to a subtype (this is valid in Cook's system, as observed in [13]). To this end one would use a more standard bounded quantification ($\forall t <: \tau.\sigma$), and then the derived inference (4) could be generalized to one involving the premiss $\Gamma \vdash \Lambda t^{\diamond}.[\sigma t, \ell : \tau] <: \sigma$ (a similar remark is made by Bruce in [6]). To type "self-inflicted" method update however, we still need width subtyping $\lessdot$. One may regard this improved system as the right one to adopt for typing prototypes.

References

[1] M. ABADI, *Baby Modula-3 and a theory of objects,* J. of Functional Programming Vol 4, No 2 (1994) 249-283.

[2] M. ABADI, L. CARDELLI, *A theory of primitive objects: untyped and first-order systems,* TACS'94, Lecture Notes in Comput. Sci. 789 (1994) 296-320.

[3] M. ABADI, L. CARDELLI, R. VISWANATHAN, *An interpretation of objects and object types,* POPL'96 (1996) 396-409.

[4] V. Bono, L. Liquori, *A subtyping for the Fisher-Honsell-Mitchell lambda calculus of objects,* CSL'94, Lecture Notes in Comput. Sci. 933 (1994) 16-30.

[5] V. Bono, M. Bugliesi, *Matching constraints for the lambda calculus of objects,* TLCA'97, Lecture Notes in Comput. Sci. 1210 (1997) 46-63.

[6] K. Bruce, *The equivalence of two semantic definitions for inheritance in object-oriented languages,* MFPS'92, Lecture Notes in Comput. Sci. 598 (1992) 102-124.

[7] K. Bruce, L. Cardelli, B. Pierce, *Comparing object encodings,* TACS'97, Lecture Notes in Comput. Sci. 1281 (1997) 415-438.

[8] L. Cardelli, *A semantics of multiple inheritance,* Semantics of Data Types, Lecture Notes in Comput. Sci. 173 (1984) 51-67. Also published in Information and Computation, Vol 76 (1988).

[9] L. Cardelli, P. Wegner, *On understanding types, data abstraction, and polymorphism,* Computing Surveys 17 (1985) 471-522.

[10] A. Compagnoni, *Subject reduction and minimal types for higher order subtyping,* Tech. Rep. ECS-LFCS-97-363, University of Edinburgh (1997).

[11] W. Cook, W. Hill, P. Canning, *Inheritance is not subtyping,* POPL'90 (1990) 125-135.

[12] K. Fisher, F. Honsell, J. Mitchell, *A lambda calculus of objects and method specialization,* LICS'93 (1993) 26-38.

[13] K. Fisher, J. Mitchell, *Notes on typed object-oriented programming,* TACS'94, Lecture Notes in Comput. Sci. 789 (1994) 844-885.

[14] K. Fisher, J. Mitchell, *A delegation-based object calculus with subtyping,* FCT'95, Lecture Notes in Comput. Sci. 965 (1995) 42-61.

[15] K. Fisher, *Types Systems for Object-Oriented Programming Languages,* PhD Thesis, Stanford University (1996).

[16] P. Di Gianantonio, F. Honsell, L. Liquori, *A lambda-calculus of objects with self-inflicted extension,* OOPSLA'98, ACM SIGPLAN Notices Vol 33, No 10 (1998) 166-178.

[17] J.-Y. Girard, *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur,* Thèse d'État, Université Paris 7 (1972).

[18] M. Hofmann, B. Pierce, *A unifying type-theoretic framework for objects,* J. of Functional Programming Vol. 5, No 4 (1995) 593-635.

[19] J. Mitchell, *Toward a typed foundation for method specialization and inheritance,* POPL'90 (1990) 109-124.

[20] R. Viswanathan, *Full abstraction for first-order objects with recursive types and subtyping,* LICS'98 (1998).

[21] M. Wand, *Complete type inference for simple objects,* LICS'87 (1987) 37-44.

Appendix: well-formedness of contexts and kinding

$$\emptyset \vdash * \qquad \frac{\Gamma \vdash \tau :: \Diamond}{\Gamma\,,\, x : \tau \vdash *}\ (\dagger) \qquad \frac{\Gamma \vdash *}{\Gamma\,,\, t :: \kappa \vdash *}\ (\ddagger) \qquad \frac{\Gamma \vdash \tau :: \kappa}{\Gamma\,,\, t \lessdot \tau \vdash *}\ (\ddagger)$$

$$\frac{\Gamma \vdash \sigma :: \Box}{\Gamma \vdash \sigma :: \Diamond} \qquad \frac{\Gamma \vdash *}{\Gamma \vdash t :: \kappa}\ t :: \kappa \in \Gamma \qquad \frac{\Gamma \vdash \tau :: \kappa}{\Gamma \vdash t :: \kappa}\ t \lessdot \tau \in \Gamma$$

$$\frac{\Gamma \vdash \tau :: \Diamond\ ,\ \Gamma \vdash \sigma :: \Diamond}{\Gamma \vdash (\tau \to \sigma) :: \Diamond} \qquad \frac{\Gamma \vdash *}{\Gamma \vdash [] :: \Box} \qquad \frac{\Gamma \vdash \sigma :: \Box\ ,\ \Gamma \vdash \tau :: \Diamond}{\Gamma \vdash [\sigma, \ell : \tau] :: \Box}$$

$$\frac{\Gamma\,,\, t :: \kappa \vdash \tau :: \kappa}{\Gamma \vdash (\mu t^{\kappa}.\tau) :: \kappa}\ (\ddagger) \qquad \frac{\Gamma\,,\, t :: \kappa \vdash \tau :: \chi}{\Gamma \vdash (\Lambda t^{\kappa}.\tau) :: \kappa \to \chi}\ (\ddagger)$$

$$\frac{\Gamma \vdash \tau :: \chi \to \kappa\ ,\ \Gamma \vdash \sigma :: \chi}{\Gamma \vdash (\tau\sigma) :: \kappa} \qquad \frac{\Gamma\,,\, t \lessdot \tau \vdash \sigma :: \Diamond}{\Gamma \vdash (\forall t \lessdot \tau.\sigma) :: \Diamond}\ (\ddagger)$$

$(\dagger)$ $x \notin \mathsf{dom}(\Gamma)$, $(\ddagger)$ $t \notin \mathsf{fv}(\Gamma)$

Appendix: width subtyping

$$\frac{\Gamma \vdash \tau, \sigma :: \kappa}{\Gamma \vdash \tau \lessdot \sigma}\ \tau \sim \sigma \qquad \frac{\Gamma \vdash \tau \lessdot \theta\ ,\ \Gamma \vdash \theta \lessdot \sigma}{\Gamma \vdash \tau \lessdot \sigma}$$

$$\frac{\Gamma \vdash *}{\Gamma \vdash t \lessdot \tau}\ t \lessdot \tau \in \Gamma \qquad \frac{\Gamma \vdash \tau \lessdot \sigma}{\Gamma \vdash \tau <: \sigma} \qquad \frac{\Gamma \vdash \tau, \sigma :: \Diamond\ ,\ \Gamma \vdash \tau' \lessdot \tau\ ,\ \Gamma \vdash \sigma \lessdot \sigma'}{\Gamma \vdash \tau \to \sigma \lessdot \tau' \to \sigma'}$$

$$\frac{\Gamma \vdash \sigma :: \Box}{\Gamma \vdash \sigma \lessdot []} \qquad \frac{\Gamma \vdash \sigma \lessdot \sigma' \quad , \quad \Gamma \vdash \sigma :: \Box, \tau :: \Diamond}{\Gamma \vdash [\sigma, \ell : \tau] \lessdot [\sigma', \ell : \tau]}$$

$$\frac{\Gamma \vdash \sigma \lessdot [\ell : \tau]}{\Gamma \vdash [\sigma, \ell : \tau] \lessdot \sigma} \qquad \frac{\Gamma \vdash \sigma :: \Box \quad , \quad \Gamma \vdash \tau, \tau' :: \Diamond}{\Gamma \vdash [[\sigma, k : \tau'], \ell : \tau] \lessdot [[\sigma, \ell : \tau], k : \tau']}\ k \neq \ell$$

$$\frac{\Gamma\,,\, t :: \kappa \vdash \tau \lessdot \sigma}{\Gamma \vdash (\Lambda t^{\kappa}.\tau) \lessdot (\Lambda t^{\kappa}.\sigma)}\ t \notin \mathsf{fv}(\Gamma) \qquad \frac{\Gamma \vdash (\tau\sigma) :: \kappa\ ,\ \Gamma \vdash \tau \lessdot \tau'}{\Gamma \vdash (\tau\sigma) \lessdot (\tau'\sigma)}$$

Appendix: typing

$$\frac{\Gamma \vdash *}{\Gamma \vdash x : \tau}\ x : \tau \in \Gamma \qquad \frac{\Gamma\,,\, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x M : \tau \to \sigma} \qquad \frac{\Gamma \vdash M : \tau \to \sigma\ ,\ \Gamma \vdash N : \tau}{\Gamma \vdash (MN) : \sigma}$$

$$\frac{\Gamma\,,\,x\!:\tau \vdash M\!:\tau}{\Gamma \vdash \mathsf{fix}\,x.M\!:\tau} \qquad \frac{\Gamma \vdash M\!:[\ell\!:\tau]}{\Gamma \vdash M.\ell\!:\tau}$$

$$\frac{\Gamma \vdash *}{\Gamma \vdash []\!:[]} \qquad \frac{\Gamma \vdash R\!:\sigma \;,\; \Gamma \vdash M\!:\tau}{\Gamma \vdash [R,\ell = M]\!:[\sigma,\ell\!:\tau]}$$

$$\frac{\Gamma \vdash \tau' \lessdot \tau \quad , \quad \Gamma \vdash M\!:(\forall t \lessdot \tau.\sigma)}{\Gamma \vdash M\!:[\tau'/t]\sigma} \qquad \frac{\Gamma\,,\,t \lessdot \tau \vdash M\!:\sigma}{\Gamma \vdash M\!:(\forall t \lessdot \tau.\sigma)}$$

$$\frac{\Gamma \vdash M\!:\tau \;,\; \Gamma \vdash \tau <: \sigma}{\Gamma \vdash M\!:\sigma} \qquad \frac{\Gamma \vdash M\!:\tau \;,\; \Gamma,\Gamma' \vdash *}{\Gamma\,,\,\Gamma' \vdash M\!:\tau}$$

Appendix: subtyping

$$\frac{\Gamma \vdash \tau :: \kappa}{\Gamma\,,\,t <: \tau \vdash *}\; t \notin \mathsf{fv}(\Gamma) \qquad \frac{\Gamma \vdash \tau :: \kappa}{\Gamma \vdash t :: \kappa}\; t <: \tau \in \Gamma$$

$$\frac{\Gamma \vdash *}{\Gamma \vdash t <: \tau}\; t <: \tau \in \Gamma \qquad \frac{\Gamma \vdash \tau <: \theta \;,\; \Gamma \vdash \theta <: \sigma}{\Gamma \vdash \tau <: \sigma}$$

$$\frac{\Gamma \vdash \tau,\sigma :: \diamond \;,\; \Gamma \vdash \tau' <: \tau \;,\; \Gamma \vdash \sigma <: \sigma'}{\Gamma \vdash \tau \to \sigma <: \tau' \to \sigma'} \qquad \frac{\Gamma \vdash \sigma :: \Box, \tau :: \diamond \;,\; \Gamma \vdash \sigma <: \sigma' \;,\; \Gamma \vdash \tau <: \tau'}{\Gamma \vdash [\sigma,\ell\!:\tau] <: [\sigma',\ell\!:\tau']}$$

$$\frac{\Gamma\,,\,t <: s \vdash \tau <: \sigma \quad , \quad \Gamma\,,\,t :: \kappa \vdash \tau :: \kappa \quad , \quad \Gamma\,,\,s :: \kappa \vdash \sigma :: \kappa}{\Gamma \vdash \mu t^{\kappa}.\tau <: \mu s^{\kappa}.\sigma}\; t \neq s$$

$$\frac{\Gamma\,,\,t :: \kappa \vdash \tau <: \sigma}{\Gamma \vdash (\Lambda t^{\kappa}.\tau) <: (\Lambda t^{\kappa}.\sigma)} \qquad \frac{\Gamma \vdash (\tau\sigma) :: \kappa \;,\; \Gamma \vdash \tau <: \tau'}{\Gamma \vdash (\tau\sigma) <: (\tau'\sigma)}$$