A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects

Viviana Bono and Luigi Liquori

Dipartimento di Informatica - Università di Torino,
C.so Svizzera 185 - 10149 Torino
E-mail: `bono,liquori@di.unito.it`

Abstract. Labeled types and a new relation between types are added to the lambda calculus of objects as described in [5]. This relation is a trade-off between the possibility of having a restricted form of *width* subtyping and the features of the delegation-based language itself. The original type inference system allows both specialization of the type of an inherited method to the type of the inheriting object and static detection of errors, such as '*message-not-understood*'. The resulting calculus is an extension of the original one. Type soundness follows from the subject reduction property.

1 Introduction

Object-oriented languages can be classified as either *class-based* or *delegation-based* languages. In class-based languages, such as Smalltalk [3] and C^{++} [4], the implementation of an object is specified by its class. Objects are created by istantiating their classes. In delegation-based languages, objects are defined directly from other objects by adding new methods via *method addition* and replacing old methods bodies with new ones via *method override*. Adding or overriding a method produces a new object that inherits all the properties of the original one. In this paper we consider the delegation-based axiomatic model developed by Fisher, Honsell and Mitchell, and, in particular, we refer to the model in [5] and [6]. This calculus offers:

- a very simple and effective inheritance mechanism,
- a straightforward *mytype* method specialization,
- dynamic lookup of methods, and
- easy definition of binary methods.

The original calculus is essentially an untyped lambda calculus enriched with object primitives. There are three operations on objects: method addition (denoted by $\langle \mathtt{e}_1 \leftarrow\!+\, \mathtt{m}{=}\mathtt{e}_2 \rangle$) to define methods, method override ($\langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2 \rangle$) to re-define methods, and method call ($\mathtt{e} \Leftarrow \mathtt{m}$) to send a message $\mathtt{m}$ to an object $\mathtt{e}$. In the system of [5], the method addition makes sense only if method $\mathtt{m}$ does not occur in the object $\mathtt{e}$, while method override can be done only if $\mathtt{m}$ occurs in $\mathtt{e}$. If the expression $\mathtt{e}_1$ denotes an object without method $\mathtt{m}$, then $\langle \mathtt{e}_1 \leftarrow\!+\, \mathtt{m}{=}\mathtt{e}_2 \rangle$ denotes a new object obtained from $\mathtt{e}_1$ by adding the method body $\mathtt{e}_2$ for $\mathtt{m}$. When the message $\mathtt{m}$ is sent to $\langle \mathtt{e}_1 \leftarrow\!+\, \mathtt{m}{=}\mathtt{e}_2 \rangle$, the result is obtained by applying $\mathtt{e}_2$ to $\langle \mathtt{e}_1 \leftarrow\!+\, \mathtt{m}{=}\mathtt{e}_2 \rangle$ (similarly for $\langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2 \rangle$).

This form of *self-application* allows to model the special symbol *self* of object oriented languages directly by lambda abstraction. Intuitively, the method body

$\mathtt{e}_2$ must be a function and the first actual parameter of $\mathtt{e}_2$ will always be the object itself. The type system of this calculus allows methods to be specialized appropriately as they are inherited.

We consider the type of an object as the collection of the types of its methods. The intuitive definition of the *width* subtyping then is: σ is a subtype of τ if σ has *more methods* than τ. The usual subsumption rule allows to use an object of type σ in any context expecting an object of type τ. In the original object calculus of [5], no *width* subtyping is possible, because the addition of the method `m` to the object `e` is allowed if and only if `m` does not occur in `e`. So, the object `e` could not be replaced by an object $\mathtt{e}'$ that already contains `m`.

Moreover, it is not possible to have *depth* subtyping, namely, to generalize the types of methods that appear in the type of the object, because with method override we can give type to an expression that produces run-time errors (a nice example of [1] is translated in the original object calculus in [7]).

In this paper, we introduce a restricted form of subtyping, informally written as $\sigma \preceq \tau$. This relation is a *width* subtyping, i.e., a type of an object is a subtype of another type if the former has more methods than the latter. Subtyping is constrained by one restriction: σ is a subtype of another type τ if and only if we can assure that the methods of σ, that are not methods of τ, are not referred to by the methods also in τ. The restriction is crucial to avoid that methods of τ will refer to the *forgotten* methods of σ, causing a run-time error. The subtyping relation allows to *forget* methods in the type without changing the shape of the object; it follows that we can type programs that accept as actual parameters objects with more methods than could be expected. The information on which methods are used is collected by introducing *labeled types*. A first consequence of this relation is that it can be possible to have an object in which a method is, via a new operation, added more than once. For this reason, we introduce a different symbol to indicate the method addition operation on objects, namely

$$\langle \mathtt{e}_1 \hookleftarrow\!\circ\, \mathtt{m}{=}\mathtt{e}_2 \rangle.$$

The operation $\hookleftarrow\!\circ$ behaves exactly as the method addition of [5], but it can be used to add the same method more than once. For example, in the object

$$\langle\langle \mathtt{e}_1 \hookleftarrow\!\circ\, \mathtt{m}{=}\mathtt{e}_2 \rangle \hookleftarrow\!\circ\, \mathtt{m}{=}\mathtt{e}_3 \rangle,$$

the first addition of the method `m` is forgotten by the type inference system via a subsumption rule. Our extension gives the following (positive) consequences:

- objects with extra methods can be used in any context where an object with fewer methods might be used,
- our subtyping relation does not cause the shortcomings described in [1],
- we do not loose any feature of the calculus of [5].

We also extend the set of objects and we present an alternative operational semantics. Our evaluation rules search method bodies more directly and deal with possible errors. This semantics was inspired by [2], where the calculus is proved to be Church-Rosser. The typing of the operator for searching methods uses the information given by labeled types in a essential way.

This paper is organized as follows. In section 2 we present our language and the evaluation strategy, in section 3 we give the type inference rules for the calculus and the subtyping relation. Some interesting examples, showing the power of this calculus with respect to the original one, are illustrated. In section 4 we prove some structural properties of the system and we give a subject reduction theorem. Moreover, we prove the type soundness, in the sense that we show that the type system prevents *message-not-understood* errors.

2 Untyped Calculus of Objects

The untyped lambda calculus enriched with object related syntactic forms is defined as follows:

$$\mathtt{e} \ ::= \ x \mid \mathtt{c} \mid \lambda x.\mathtt{e} \mid \mathtt{e}_1\mathtt{e}_2 \mid \langle\rangle \mid \langle \mathtt{e}_1 \leftarrow\!\!\circ\, \mathtt{m}{=}\mathtt{e}_2\rangle \mid \langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2\rangle \mid \mathtt{e} \Leftarrow \mathtt{m} \mid \mathtt{e} \hookleftarrow \mathtt{m} \mid \mathtt{err},$$

where x is a term variable, $\mathtt{c}$ belongs to a fixed set of constants, and $\mathtt{m}$ is a method name. The object forms are:

$\langle\rangle$	the empty object;
$\langle \mathtt{e}_1 \leftarrow\!\!\circ\, \mathtt{m}{=}\mathtt{e}_2\rangle$	extends object $\mathtt{e}_1$ with a new method $\mathtt{m}$ having body $\mathtt{e}_2$;
$\langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2\rangle$	replaces the body of method $\mathtt{m}$ in $\mathtt{e}_1$ by $\mathtt{e}_2$;
$\mathtt{e} \Leftarrow \mathtt{m}$	sends message $\mathtt{m}$ to the object $\mathtt{e}$;
$\mathtt{e} \hookleftarrow \mathtt{m}$	searches the body of the message $\mathtt{m}$ into the object $\mathtt{e}$;
$\mathtt{err}$	the error object.

Notice that the last two object forms are not present in the original calculus of [5].

Let $\leftarrow\!\!*$ denote $\leftarrow\!\!\circ$ or $\leftarrow$. The description of an object via $\leftarrow\!\!*$ operations is intensional, and the object corresponding to a sequence of $\leftarrow\!\!*$ can be extensionally defined as follows.

Definition 1. Let $\mathtt{m}_1, \ldots, \mathtt{m}_k$, and $\mathtt{n}$ be distinct method names. $\langle \mathtt{m}_1{=}\mathtt{e}_1, \ldots, \mathtt{m}_k{=}\mathtt{e}_k\rangle$ is defined as:

1. $\langle \ldots \langle\langle\rangle \leftarrow\!\!\circ\, \mathtt{m}_1{=}\mathtt{e}_1\rangle \ldots \leftarrow\!\!\circ\, \mathtt{m}_k{=}\mathtt{e}_k\rangle = \langle \mathtt{m}_1{=}\mathtt{e}_1, \ldots, \mathtt{m}_k{=}\mathtt{e}_k\rangle$,
2. $\langle\langle \mathtt{m}_1{=}\mathtt{e}_1, \ldots, \mathtt{m}_i{=}\mathtt{e}_i, \ldots, \mathtt{m}_k{=}\mathtt{e}_k\rangle \leftarrow\!\!*\ \mathtt{m}_i{=}\mathtt{e}'_i\rangle = \langle \mathtt{m}_1{=}\mathtt{e}_1, \ldots, \mathtt{m}_i{=}\mathtt{e}'_i, \ldots, \mathtt{m}_k{=}\mathtt{e}_k\rangle$,
3. $\langle\langle \mathtt{m}_1{=}\mathtt{e}_1, \ldots, \mathtt{m}_k{=}\mathtt{e}_k\rangle \leftarrow\!\!\circ\, \mathtt{n}{=}\mathtt{e}'\rangle = \langle \mathtt{m}_1{=}\mathtt{e}_1, \ldots, \mathtt{m}_k{=}\mathtt{e}_k, \mathtt{n}{=}\mathtt{e}'\rangle$.

So $\langle \mathtt{m}_1{=}\mathtt{e}_1, \ldots, \mathtt{m}_k{=}\mathtt{e}_k\rangle$ is the object in which each method body corresponds to the outermost assignment (by addition or by override) perfomed on the method.

2.1 The Evaluation Rules

The operational semantics of the original calculus of [5] is mainly based on the following evaluation rules for β-reduction and for message sending $\Leftarrow$-reduction:

$$\begin{array}{lll} (\beta) & (\lambda x.\mathtt{e}_1)\mathtt{e}_2 & \stackrel{eval}{\rightarrow} \ [\mathtt{e}_2/x]\mathtt{e}_1 \\ (\Leftarrow) & \langle \mathtt{e}_1 \leftarrow\!\!*\ \mathtt{m}{=}\mathtt{e}_2\rangle \Leftarrow \mathtt{m} & \stackrel{eval}{\rightarrow} \ \mathtt{e}_2\langle \mathtt{e}_1 \leftarrow\!\!*\ \mathtt{m}{=}\mathtt{e}_2\rangle. \end{array}$$

To send message $\mathtt{m}$ to the object $\mathtt{e}$ means applying the body of $\mathtt{m}$ to the object itself. In fact, the body of $\mathtt{m}$ is a lambda abstraction whose first bound variable will be substituted by the full object in the next step of β-reduction.

The problem that arises in the calculus of objects is how to extract the appropriate method out of an object. The most natural way is moving the required method in an accessible position (the most external one). This means to treat objects as sets of methods. Unfortunately, this approach is not possible: in fact, the typing rules of objects depend on the order of $\leftarrow\!\!*$ operations. For instance, the typing of $\texttt{e}_3$ in the object expression $\langle\langle \texttt{e}_1 \leftarrow\!\!* \ \texttt{m}{=}\texttt{e}_2\rangle \leftarrow\!\!* \ \texttt{n}{=}\texttt{e}_3\rangle$ depends on the typing of the "subobjects" $\texttt{e}_1$ and $\langle \texttt{e}_1 \leftarrow\!\!* \ \texttt{m}{=}\texttt{e}_2\rangle$.

The approach chosen in [5] to solve the problem of method order is to add to the $\overset{eval}{\rightarrow}$ relation a *bookeeping* relation $\overset{book}{\rightarrow}$. This relation leads to a *standard form*, in which each method is defined *exactly* once (with the extension operation), using some "dummy" bodies, and redefined *exactly* once (with the override operation), giving it the desired body.

In our system the notion of standard form is unuseful since the *subject reduction* property does not hold for the $\overset{book}{\rightarrow}$ part of the evaluation rule. On the other hand, we can use the extra information contained in types to type correctly the extraction of the bodies of methods from the objects (it will be clear how in paragraph 3.2). Therefore, we propose the following operational semantics. We list here only the most meaningful reduction rules. Appendix 1 contains the full set of rules, which includes rules of error propagation. The evaluation relation is the least congruence generated by these rules.

$$\begin{array}{lll l}
(\beta) & (\lambda x.\texttt{e}_1)\texttt{e}_2 & \overset{eval}{\rightarrow} & [\texttt{e}_2/x]\texttt{e}_1 \\
(\Leftarrow) & \texttt{e} \Leftarrow \texttt{m} & \overset{eval}{\rightarrow} & (\texttt{e} \hookleftarrow \texttt{m})\texttt{e} \\
(succ \hookleftarrow) & \langle \texttt{e}_1 \leftarrow\!\!* \ \texttt{m}{=}\texttt{e}_2\rangle \hookleftarrow \texttt{m} & \overset{eval}{\rightarrow} & \texttt{e}_2 \\
(next \hookleftarrow) & \langle \texttt{e}_1 \leftarrow\!\!* \ \texttt{n}{=}\texttt{e}_2\rangle \hookleftarrow \texttt{m} & \overset{eval}{\rightarrow} & \texttt{e}_1 \hookleftarrow \texttt{m} \\
(fail\ \langle\rangle) & \langle\rangle \hookleftarrow \texttt{m} & \overset{eval}{\rightarrow} & \texttt{err} \\
(fail\ abs) & \lambda x.\texttt{e} \hookleftarrow \texttt{m} & \overset{eval}{\rightarrow} & \texttt{err}.
\end{array}$$

To send message $\texttt{m}$ to the object $\texttt{e}$ still means applying the body of $\texttt{m}$ to the object itself. The difference is that in our semantics the body of the method is recursively searched by the $\hookleftarrow$ operator without modifying the shape of the full object; if such a method does not exist, the object evaluates to error.

Remark. The rule $(fail\ var)$ $x \hookleftarrow \texttt{m} \overset{eval}{\rightarrow} \texttt{err}$ is unsound, since the variable x could be substituted (by applying a β-reduction) by an object containing the method $\texttt{m}$.

3 Static Type System

The central part of the type system of an object oriented language consists of the types of objects. In [5], the type of an object is called a *class-type*. It has the form:

$$\mathbf{class}\, t.\langle\!\langle \texttt{m}_1{:}\tau_1, \ldots, \texttt{m}_k{:}\tau_k \rangle\!\rangle,$$

where $\langle\!\langle \texttt{m}_1{:}\tau_1, \ldots, \texttt{m}_k{:}\tau_k \rangle\!\rangle$ is called a *row* expression. This type expression describes the properties of any object $\texttt{e}$ that can receive messages $\texttt{m}_i$, $(\texttt{e} \Leftarrow \texttt{m}_i)$, producing a result of type τ_i, for $1 \leq i \leq k$. The bound variable t may appear in τ_i, referring to the object itself. Thus, a class-type is a form of recursively-defined type. As a simple example of types in [5], we consider the following `point` object:

$$\texttt{point} \overset{def}{=} \langle \texttt{x}{=}\lambda self.0,\ \texttt{mv}_\texttt{x}{=}\lambda self.\lambda dx.\langle self \leftarrow \texttt{x}{=}\lambda s.(self \Leftarrow \texttt{x}) + dx\rangle\rangle,$$

with the following class-type:

$$\mathbf{class}\, t.\langle\!\langle \texttt{x}{:}int,\ \texttt{mv}_\texttt{x}{:}int{\rightarrow}t \rangle\!\rangle.$$

A significant aspect of this type system is that the type $(int{\to}t)$ of method $\mathtt{mv}_{\mathtt{x}}$ does not change syntactically if we perform a method addition of a method `color` to build a `colored_point` object from `point`. Instead, the meaning of the type changes, since before the `color` adjuction the bound variable t referred to an object of type `point`, and after t refers to an object of type `colored_point`.

So the type of a method may change when a method is inherited: the authors of [5] called this property *method specialization* (also called *mytype* specialization in [7]). The typing rules assure that every possible type for an added method will be correct and this is done via a sort of implicit higher-order polymorphism.

To allow subtyping, we add a new sort of types, the *labeled types*, that carry on the information about the methods used to type a certain method body. This information is given by a subscript which is a set of method names. The methods used to type a body are roughly the method names which occur in the body itself. For example, suppose that the object $\mathtt{e}_1$ has a method $\mathtt{m}$ with body $\mathtt{e}_2$, that in $\mathtt{e}_2$ a message $\mathtt{n}$ is sent to the bound variable *self* and a method $\mathtt{n}'$ (of $\mathtt{e}_1$) is overriden. Then the type τ of $\mathtt{e}_2$ is subscripted by the set $\{\mathtt{n},\mathtt{n}'\}$, since $\mathtt{e}_2$ uses $\mathtt{n},\mathtt{n}'$. These labeled types are written inside the row of the class-type and they do not appear externally. Therefore, in our system the object `point` will have the following class-type:

$$\mathbf{class}\,t.\langle\!\langle \mathtt{x}{:}int_{\{\}},\ \mathtt{mv}_{\mathtt{x}}{:}(int{\to}t)_{\{\mathtt{x}\}}\rangle\!\rangle.$$

We can forget by subtyping those methods that are not used by other methods in the object, i.e., a method is *forgettable* if and only it does not appear in the labels of the types of the remaining methods. This dependency is correctly handled in the typing rules for adding and overriding methods (i.e., *(obj ext)* and *(obj over)*), where the labels of types are created. We refer to section 3.4 for some meaningful examples.

3.1 Types, Rows, and Kinds

The type expressions include type constants, type variables, function types and class-types. In this paper, a *term* will be an *object* of the calculus, or a *type*, or a *row*, or a *kind*.

The symbols σ, τ, ρ, ... are metavariables over types; ι ranges over type constants; t, *self*,... range over type variables; $\mathit{\Delta}$ ranges over labels; α, β, ... range over labeled types; R, r range over rows and row variables respectively; $\mathtt{m}$, $\mathtt{n}$, ... range over method names and κ ranges over kinds. The symbols a, b, c, ... range over term variables or constants; u, v, ... range over type and row variables; U, V, ... range over type, row, and kind expressions, and, finally, A, B, C, ... range over terms. All symbols may appear indexed. The set of types, rows and kinds are mutually defined by the following grammar:

$$\begin{array}{lll}
\text{Types} & \tau & ::= \ \iota \mid t \mid \tau{\to}\tau \mid \mathbf{class}\,t.R \\
\text{Labels} & \mathit{\Delta} & ::= \ \{\} \mid \mathit{\Delta}\cup\{\mathtt{m}\} \\
\text{Labeled Types} & \alpha & ::= \ \tau_{\mathit{\Delta}} \\
\text{Rows} & R & ::= \ r \mid \langle\!\langle\rangle\!\rangle \mid \langle\!\langle R \mid \mathtt{m}{:}\alpha\rangle\!\rangle \mid \lambda t.R \mid R\tau \\
\text{Kinds} & \kappa & ::= \ T \mid T^n{\to}[\mathtt{m}_1,\ldots,\mathtt{m}_k]\ (n\geq 0, k\geq 1).
\end{array}$$

We say that τ is the type and $\mathit{\Delta}$ is the label of the labeled type $\tau_{\mathit{\Delta}}$.

The row expressions appear as subexpressions of class-type expressions, with rows and types distinguished by kinds. Intuitively, the elements of kind $[\mathtt{m}_1, \ldots, \mathtt{m}_k]$ are rows that do not include method names $\mathtt{m}_1, \ldots, \mathtt{m}_k$. We need this information to guarantee statically that methods are not multiply defined. In what follows, we use the notation $\vec{\mathtt{m}}{:}\vec{\tau}_{\Delta}$ as short for $\mathtt{m}_1{:}\tau^1_{\Delta_1}, \ldots, \mathtt{m}_k{:}\tau^k_{\Delta_k}$ and $\vec{\mathtt{m}}{:}\vec{\alpha}$ as short for $\mathtt{m}_1{:}\alpha_1, \ldots, \mathtt{m}_k{:}\alpha_k$.

We say that a row R' is a *subrow* of a row R if and only if $R{=}\langle\!\langle R' \mid \vec{\mathtt{m}}{:}\vec{\alpha}\rangle\!\rangle$, for suitable $\vec{\mathtt{m}}$ and $\vec{\alpha}$.

The set $\mathcal{S}(R)$ of labels of a row R is inductively defined by:
$\mathcal{S}(\langle\!\langle\rangle\!\rangle){=}\mathcal{S}(r){=}\{\}$, $\mathcal{S}(\langle\!\langle R \mid \mathtt{m}{:}\tau_{\Delta}\rangle\!\rangle){=}\mathcal{S}(R) \cup \Delta$, $\mathcal{S}(\lambda t.R){=}\mathcal{S}(R)$, and $\mathcal{S}(R\tau){=}\mathcal{S}(R)$.
In our system, the contexts are defined as follows:

$$\Gamma \quad ::= \quad \varepsilon \mid \Gamma, x{:}\tau \mid \Gamma, t{:}T \mid \Gamma, r{:}\kappa \mid \Gamma, \iota_1 \preceq \iota_2,$$

and the judgement forms are:

$$\Gamma \vdash * \qquad \Gamma \vdash R : \kappa \qquad \Gamma \vdash \tau : T \qquad \Gamma \vdash \tau_1 \preceq \tau_2 \qquad \Gamma \vdash \mathtt{e} : \tau.$$

The judgement $\Gamma \vdash *$ can be read as "Γ is a well-formed context". The meaning of the other judgements is the usual one.

3.2 Typing Rules

In this subsection we discuss all the typing rules which are new with respect to [5], except for the subsumption rule which will be discussed in the next section. More precisely, we present the rules for extending an object with a new method or for re-defining an existing one with a new body, the rule for searching method bodies and the rule for sending messages. The remaining rules of the type system are presented in Appendix 2.

We can assume, without loss of generality, that the order of methods inside rows can be arbitrarily modified: this assumption allows to write the method $\mathtt{m}$ as the last method listed in the class-type. A formal definition of type equality is given in Appendix 2.

The *(obj ext)* rule performs a method addition, producing the new object $\langle \mathtt{e}_1 \leftarrow\!\!\!\circ\, \mathtt{n} {=} \mathtt{e}_2\rangle$. This rule always adds the method to the syntactic object in case the latter is not present or it is present in the object but it was previously forgotten in the type by an application of the subtyping rule *(sub $\preceq$)*. Another task performed by this rule is to build the labeled type $\tau_{\{\vec{\mathtt{m}}\}}$ for the new method $\mathtt{n}$, where the label $\{\vec{\mathtt{m}}\}$ represents the set of all methods of $\mathtt{e}_1$ that are useful to type $\mathtt{n}$'s body.

$$\textit{(obj ext)} \quad \frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_1 : \mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}\rangle\!\rangle \qquad \Gamma, t{:}T \vdash R : [\vec{\mathtt{m}}, \mathtt{n}] \qquad \mathtt{n} \notin \mathcal{S}(\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}\rangle\!\rangle) \\ \Gamma, r{:}T \rightarrow [\vec{\mathtt{m}}, \mathtt{n}] \vdash \mathtt{e}_2 : [\mathbf{class}\, t.\langle\!\langle rt \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle / t](t \rightarrow \tau) \quad r \text{ not in } \tau\end{array}}{\Gamma \vdash \langle \mathtt{e}_1 \leftarrow\!\!\!\circ\, \mathtt{n} {=} \mathtt{e}_2\rangle : \mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle.}$$

The condition $\mathtt{n} \notin \mathcal{S}(\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}\rangle\!\rangle)$ prevents unmeaningful labels, since the typings of the previously added methods cannot use $\mathtt{n}$. This condition is not derivable, since $\mathtt{e}_1$ could be a term variable.

The *(obj over)* rule types an object in which method $\mathtt{n}$ is overrided as in the original rule of [5]. The label Δ is changed to $\{\vec{\mathtt{m}}\}$, because Δ represents the dependences of the previous body of $\mathtt{n}$, and these ones could not hold anymore for the new body.

$$\textit{(obj over)}\quad \frac{\begin{array}{c}\mathit{\Gamma} \vdash \mathtt{e}_1 : \mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\Delta}\rangle\!\rangle \\ \mathit{\Gamma}, r{:}T{\to}[\vec{\mathtt{m}}, \mathtt{n}] \vdash \mathtt{e}_2 : [\mathbf{class}\, t.\langle\!\langle rt \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle/t](t{\to}\tau) \quad r \text{ not in } \tau\end{array}}{\mathit{\Gamma} \vdash \langle \mathtt{e}_1 \leftarrow \mathtt{n}{=}\mathtt{e}_2\rangle : \mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle.}$$

In the *(obj ext)* and the *(obj over)* rules we say that the method $\mathtt{n}$ *uses* all the methods belonging to the label $\{\vec{\mathtt{m}}\}$ associated with the labeled type of $\mathtt{n}$.

The *(meth search)* rule asserts that the type of the extracted method body is an instance of the type we deduced for it when the method was added (by an *(obj ext)* rule) or overrided (by an *(obj over)* rule). Note that the labels are used in an essential way in this rule.

$$\textit{(meth search)}\quad \frac{\mathit{\Gamma} \vdash \mathtt{e} : \mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle \quad \mathit{\Gamma}, t{:}T \vdash R' : [\vec{\mathtt{m}}, \mathtt{n}]}{\mathit{\Gamma} \vdash \mathtt{e} \hookleftarrow \mathtt{n} : [\mathbf{class}\, t.\langle\!\langle R' \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle/t](t{\to}\tau).}$$

The *(meth appl)* rule is a sort of *unfolding* rule; in fact the class-type is a form of recursive type, and it does not need any futher explanation.

$$\textit{(meth appl)}\quad \frac{\mathit{\Gamma} \vdash \mathtt{e} : \mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle}{\mathit{\Gamma} \vdash \mathtt{e} \Leftarrow \mathtt{n} : [\mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle/t]\tau.}$$

3.3 The Subtyping Relation and the Subsumption Rule

The subtyping relation is based on the information given by the labeled types of methods in rows. Looking at rules *(obj ext)* and *(obj over)* it is clear that if the body of the method $\mathtt{n}$ in $\mathtt{e}$ has type τ, and its typing uses the methods $\vec{\mathtt{m}}$ of $\mathtt{e}$, then the type of $\mathtt{e}$ will be $\mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle$, for suitable R and $\vec{\alpha}$. In other words, we label the types of the methods in rows by the sets of methods the typing of their bodies depends on.

The *(width $\preceq$)* rule says that a type is a subtype of another type if the forgotten methods (i.e., the methods not occurring) in the second type are not in the union of the sets of labels of the remaining methods. The condition $\vec{\mathtt{n}} \notin \mathcal{S}(R)$ formally assures that the remaining methods do not use the methods $\vec{\mathtt{n}}$.

$$\textit{(width }\preceq\textit{)}\quad \frac{\mathit{\Gamma} \vdash \mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{n}}{:}\vec{\alpha}\rangle\!\rangle : T \quad \vec{\mathtt{n}} \notin \mathcal{S}(R)}{\mathit{\Gamma} \vdash \mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{n}}{:}\vec{\alpha}\rangle\!\rangle \preceq \mathbf{class}\, t.R.}$$

Clearly, we can forget groups of mutually recursive methods with this rule.

We have also the usual subtyping rules for constants, reflexivity, transivity and for the arrow type constructor (that behaves controvariantly in its domain with respect to the $\preceq$ relation). The full subtyping system is given in Appendix 2.

Let two class-types τ_1 and τ_2 be given, such that the judgement $\mathit{\Gamma} \vdash \tau_1 \preceq \tau_2$ is derivable and the object $\mathtt{e}$ is of type τ_1. The *(sub $\preceq$)* rule says that we can derive also type τ_2 for $\mathtt{e}$. It follows that the object $\mathtt{e}$ can be used in any context in which an object of type τ_2 is required. The possibility of giving more types to the same object makes our calculus more expressive than the original one.

$$\textit{(sub }\preceq\textit{)}\quad \frac{\mathit{\Gamma} \vdash \mathtt{e} : \tau_1 \qquad \mathit{\Gamma} \vdash \tau_1 \preceq \tau_2}{\mathit{\Gamma} \vdash \mathtt{e} : \tau_2.}$$

Using the *(sub $\preceq$)* rule we can obtain judgements of the shape $\mathit{\Gamma} \vdash \mathtt{e} : \mathbf{class}\, t.R$, where $\mathtt{n}$ is a method of $\mathtt{e}$ but $\mathit{\Gamma} \vdash R : [\mathtt{n}]$. In this case we say that this rule *forgets* the method $\mathtt{n}$. It is important to remark that, when a method is forgotten in the type of an object, it is like it was never added to the object.

3.4 Examples

In this section we will present two examples: the first shows how our subtyping relation works on a critical example of [1]. The second example gives a simple object, typable in our calculus, but not typable in the original calculus.

Example 1. Given the following objects:

$$\begin{aligned}
\mathtt{p}_1 &\overset{def}{=} \langle \mathtt{x}{=}\lambda \mathit{self}.0,\ \mathtt{mv}_{\mathtt{x}}{=}\lambda \mathit{self}.\lambda dx.\langle \mathit{self} \leftarrow \mathtt{x}{=}\lambda s.(\mathit{self} \Leftarrow \mathtt{x}) + dx\rangle\rangle\\
\mathtt{p}_2 &\overset{def}{=} \langle \mathtt{x}{=}\lambda \mathit{self}.1,\ \mathtt{y}{=}\lambda \mathit{self}.0,\ \mathtt{mv}_{\mathtt{x}}{=}\lambda \mathit{self}.\lambda dx.\langle \mathit{self} \leftarrow \mathtt{x}{=}\lambda s.(\mathit{self} \Leftarrow \mathtt{x}) + dx\rangle,\\
&\qquad \mathtt{mv}_{\mathtt{y}}{=}\lambda \mathit{self}.\lambda dy.\langle \mathit{self} \leftarrow \mathtt{y}{=}\lambda s.(\mathit{self} \Leftarrow \mathtt{y}) + dy\rangle\rangle,
\end{aligned}$$

we can derive $\vdash \mathtt{p}_1 : P_1$ and $\vdash \mathtt{p}_2 : P_2$, where

$$\begin{aligned}
P_1 &\overset{def}{=} \mathbf{class}\, t.\langle\!\langle \mathtt{x}{:}\mathit{int},\ \mathtt{mv}_{\mathtt{x}}{:}(\mathit{int}{\to}t)_{\{\mathtt{x}\}}\rangle\!\rangle\\
P_2 &\overset{def}{=} \mathbf{class}\, t.\langle\!\langle \mathtt{x}{:}\mathit{int},\ \mathtt{y}{:}\mathit{int},\ \mathtt{mv}_{\mathtt{x}}{:}(\mathit{int}{\to}t)_{\{\mathtt{x}\}},\ \mathtt{mv}_{\mathtt{y}}{:}(\mathit{int}{\to}t)_{\{\mathtt{y}\}}\rangle\!\rangle,
\end{aligned}$$

and *int* stands for $\mathit{int}_{\{\}}$.

It is easy to verify that in our system $P_2 \preceq P_1$. This relation between P_2 and P_1 is the one we want to have, since it is the intuitive relation between a one-dimensional point and a two-dimensional point. If we modify $\mathtt{p}_1$ and $\mathtt{p}_2$ as follows:

$$\begin{aligned}
\mathtt{p}'_1 &\overset{def}{=} \langle \mathtt{p}_1 \leftarrow \mathtt{mv}_{\mathtt{x}}{=}\lambda \mathit{self}.\lambda dx.\mathit{self}\,\rangle\\
\mathtt{p}'_2 &\overset{def}{=} \langle\langle \mathtt{p}_2 \leftarrow \mathtt{x}{=}\lambda \mathit{self}.((\mathit{self} \Leftarrow \mathtt{mv}_{\mathtt{x}}\ 1) \Leftarrow \mathtt{y})\rangle \leftarrow \mathtt{mv}_{\mathtt{x}}{=}\lambda \mathit{self}.\lambda dx.\mathit{self}\rangle,
\end{aligned}$$

we can derive $\vdash \mathtt{p}'_1 : P'_1$ and $\vdash \mathtt{p}'_2 : P'_2$, where

$$\begin{aligned}
P'_1 &\overset{def}{=} \mathbf{class}\, t.\langle\!\langle \mathtt{x}{:}\mathit{int},\ \mathtt{mv}_{\mathtt{x}}{:}(\mathit{int}{\to}t)\rangle\!\rangle\\
P'_2 &\overset{def}{=} \mathbf{class}\, t.\langle\!\langle \mathtt{x}{:}\mathit{int}_{\{\mathtt{mv}_{\mathtt{x}},\mathtt{y}\}},\ \mathtt{y}{:}\mathit{int},\ \mathtt{mv}_{\mathtt{x}}{:}(\mathit{int}{\to}t),\ \mathtt{mv}_{\mathtt{y}}{:}(\mathit{int}{\to}t)_{\{\mathtt{y}\}}\rangle\!\rangle.
\end{aligned}$$

Now $P'_2 \not\preceq P'_1$, because we cannot forget the `y` method since the `x` method uses it. Therefore, we are unable to assign type P'_1 to the object $\mathtt{p}'_2$. In this way, we avoid the so called *message-not-understood* error. In fact, if we allowed $P'_2 \preceq P'_1$, we would get $\vdash \mathtt{p}'_2 : P'_1$ by subtyping. Then, it would be possible to override the $\mathtt{mv}_{\mathtt{x}}$ method of $\mathtt{p}'_2$ by a body that has an output of type P'_1. Since the `x` method of $\mathtt{p}'_2$ uses `y`, this would produce a run-rime error. Let us formalize this situation (the original pattern appears in [1], paragraph 5.4). Suppose to override the object $\mathtt{p}'_2$ as follows:

$$\mathtt{p}''_2 \overset{def}{=} \langle \mathtt{p}'_2 \leftarrow \mathtt{mv}_{\mathtt{x}}{=}\lambda \mathit{self}.\lambda dx.\mathtt{p}'_1\rangle.$$

If we send message `x` to $\mathtt{p}''_2$, then an error occurs, since the body of `x` sends the message `y` to the object $(\mathit{self} \Leftarrow \mathtt{mv}_{\mathtt{x}}\, 1)$, but this object does not have any `y` method.

Example 2. Consider the object `draw` that can receive two messages: `figure`, that describes a geometrical figure, and `plot`, that, given a point, colors it black or white, depending on the position of the point with respect to the figure. The object `draw` accepts as input both a colored point or a point. This would be impossible in the original system of [5], since there one would have to write two different objects, one for colored points and one for points, with different bodies for the method `plot`. In fact, for colored points we need an override instead of an extension. For the object `draw`:

$$\begin{gathered}
\mathtt{draw} \overset{def}{=}\\
\langle \mathtt{figure}{=}\lambda \mathit{self}.\lambda dx.\lambda dy.(dy{=}f(dx)),\ \mathtt{plot}{=}\lambda \mathit{self}.\lambda p.\mathit{if}\ (\mathit{self} \Leftarrow \mathtt{figure})\\
(p \Leftarrow x)(p \Leftarrow y)\ \ \mathit{then}\ \langle p {\leftarrow\!\!\circ}\ \mathtt{col}{=}\lambda \mathit{self}.\mathit{black}\rangle\ \ \mathit{else}\ \langle p {\leftarrow\!\!\circ}\ \mathtt{col}{=}\lambda \mathit{self}.\mathit{white}\rangle\rangle,
\end{gathered}$$

we can derive $\vdash \texttt{draw} : DR$, where

$$\begin{array}{lll}
DR & \overset{def}{=} & \mathbf{class}\, t.\langle\!\langle \texttt{figure}{:}int{\to}int{\to}bool,\ \texttt{plot}{:}(P{\to}CP)_{\{\texttt{figure}\}}\rangle\!\rangle \\
P & \overset{def}{=} & \mathbf{class}\, t.\langle\!\langle \texttt{x}{:}int,\ \texttt{y}{:}int,\ \texttt{mv}_{\texttt{x}}{:}(int{\to}t)_{\{\texttt{x}\}},\ \texttt{mv}_{\texttt{y}}{:}(int{\to}t)_{\{\texttt{y}\}}\rangle\!\rangle \\
CP & \overset{def}{=} & \mathbf{class}\, t.\langle\!\langle \texttt{x}{:}int,\ \texttt{y}{:}int,\ \texttt{mv}_{\texttt{x}}{:}(int{\to}t)_{\{\texttt{x}\}},\ \texttt{mv}_{\texttt{y}}{:}(int{\to}t)_{\{\texttt{y}\}},\ \texttt{color}{:}colors\rangle\!\rangle.
\end{array}$$

4 Properties of the System

In this section we will show that our extension has all the good properties of the original system. We follow the same pattern of [6]: first we introduce some substitution lemmas and, then, the notion of derivation in *normal form* that simplifies the proofs of technical lemmas and the proof of the *subject reduction* theorem. Proofs of lemmas that easily extend the ones of the original system in [6] are omitted.

4.1 Substitution Properties

The following lemmas are useful to show both a substitution property on type and kind derivations and to specialize class-types with additional methods. Let $U \bullet V$ stands for $U : V$ or $U \preceq V$.

Lemma 2. *1. If* $\Gamma, u_2{:}V_2, \Gamma' \vdash U_1 \bullet V_1$ *and* $\Gamma \vdash U_2 : V_2$ *and* $\Gamma, [U_2/u_2]\Gamma' \vdash *$ *then* $\Gamma, [U_2/u_2]\Gamma' \vdash [U_2/u_2]U_1 \bullet [U_2/u_2]V_1$.
2. If $\Gamma, u_2{:}V_2, \Gamma' \vdash *$ *and* $\Gamma \vdash U_2 : V_2$ *then* $\Gamma, [U_2/u_2]\Gamma' \vdash *$.
3. If $\Gamma, u_2{:}V_2, \Gamma' \vdash U_1{\bullet}V_1$ *and* $\Gamma \vdash U_2 : V_2$ *then* $\Gamma, [U_2/u_2]\Gamma' \vdash [U_2/u_2]U_1{\bullet}[U_2/u_2]V_1$.

Proof. 1. By induction on the structure of a derivation of $\Gamma, u_2{:}V_2, \Gamma' \vdash U_1 \bullet V_1$.
2. By induction on the length of Γ', using part (1).
3. By (1) and (2). ■

Lemma 3. *If* $\Gamma, r{:}T^n{\to}[\vec{\texttt{m}}], \Gamma' \vdash \texttt{e} : \tau$ *and* $\Gamma \vdash R : T^n{\to}[\vec{\texttt{m}}]$ *then* $\Gamma, [R/r]\Gamma' \vdash \texttt{e} : [R/r]\tau$.

Proof. By induction on the structure of a derivation of $\Gamma, r{:}T^n{\to}[\vec{\texttt{m}}], \Gamma' \vdash \texttt{e} : \tau$. ■

4.2 Normal Form

It is well known that equality rules in proof systems usually complicate derivations, and make theorems and lemmas more difficult to prove. These rules introduce many unessential judgement derivations. In this subsection, we introduce the notion of *normal form derivation* and of type and row in *normal form*, respectively denoted by $\vdash_N$ and τnf in [6]. Although it is not possible to derive all judgements of the system by means of these derivations, we will show that all judgements whose rows and type expressions are in τnf are $\vdash_N$ -derivable. Using this, we can prove the subject reduction theorem using only $\vdash_N$ derivations.

Definition 4. 1. $\Gamma \vdash_N \texttt{e} : \tau$ is a normal form derivation only if the only appearance of an equality rule is as *(row β)* immediately following an occurrence of a *(row fn appl)* rule.

2. The τnf of a type and of a row are their β-normal form.

It is easy to show that τnf satisfies the following identities:

Fact 5. *$\tau nf(\mathbf{class}\,t.R) \equiv \mathbf{class}\,t.\tau nf(R)$, $\tau nf(\langle\!\langle R \mid \mathtt{m}{:}\tau_\Delta\rangle\!\rangle) \equiv \langle\!\langle \tau nf(R) \mid \mathtt{m}{:}\tau nf(\tau_\Delta)\rangle\!\rangle$, and $\tau nf(\tau_\Delta) \equiv \tau nf(\tau)_\Delta$.*

In [6], the type and row parts of the calculus are translated into the typed λ-calculus with function types over an assigned signature Σ (called $\lambda^{\rightarrow}(\Sigma)$). In Appendix 3, we present an extension of the translation function tr of [6] that takes into account labeled types. This extension is done by deleting the labels in labeled types.

The following theorem states that the row and type fragment of our calculus is strongly normalizing and confluent. For the proof we can refer to [6], since the target calculus $\lambda^{\rightarrow}(\Sigma)$ of tr is unchanged.

Theorem 6. *1. If $\Gamma \vdash U : V$ then there is no infinite sequence of $\rightarrow_\beta$ reductions out of U.*
2. If $\Gamma \vdash U_1 : V_1$ and $U_1 \twoheadrightarrow_\beta U_2$ and $U_1 \twoheadrightarrow_\beta U_3$, then there exists U_4, such that $U_2 \twoheadrightarrow_\beta U_4$ and $U_3 \twoheadrightarrow_\beta U_4$. ■

The following lemma states that, for each derivation in our system, it is possible to find a corresponding derivation in normal form. Let $A \bullet B$ stands for any statement in the calculus, and let $\tau nf(\mathtt{e}){=}\mathtt{e}$.

Lemma 7. *If $\Gamma \vdash A \bullet B$ then $\tau nf(\Gamma) \vdash_N \tau nf(A) \bullet \tau nf(B)$.*

Proof. By induction on the structure of a derivation of $\Gamma \vdash A \bullet B$. We distinguish three cases:

1. $A \bullet B$ is a statement of the form $R : k$ or $\tau : T$. The proof goes as in Lemma 4.10 of [6]. Equality rules may be eliminated in the $\vdash_N$ -derivations, because of the unicity of the τnf. Most cases follow directly from the induction hypothesis, while an interesting case is the *(row fn app)* rule.
2. $A \bullet B$ is a statement of the form $\tau \preceq \sigma$. Most cases follow from the induction hypothesis. The only interesting case is when the last rule applied is *(width $\preceq$)*:
$$\textit{(width } \preceq\textit{)}\ \frac{\Gamma \vdash \mathbf{class}\,t.\langle\!\langle R \mid \vec{\mathtt{n}}{:}\vec{\alpha}\rangle\!\rangle : T \quad \vec{\mathtt{n}} \notin \mathcal{S}(R)}{\Gamma \vdash \mathbf{class}\,t.\langle\!\langle R \mid \vec{\mathtt{n}}{:}\vec{\alpha}\rangle\!\rangle \preceq \mathbf{class}\,t.R.}$$
By case (1) we get $\tau nf(\Gamma) \vdash_N \tau nf(\mathbf{class}\,t.\langle\!\langle R \mid \vec{\mathtt{n}}{:}\vec{\alpha}\rangle\!\rangle) : \tau nf(T)$, that is equal, by Fact 5, to: $\tau nf(\Gamma) \vdash_N \mathbf{class}\,t.\langle\!\langle \tau nf(R) \mid \vec{\mathtt{n}}{:}\tau nf(\vec{\alpha})\rangle\!\rangle : T$. The hypothesis gives us $\vec{\mathtt{n}} \notin \mathcal{S}(R)$, so $\vec{\mathtt{n}} \notin \mathcal{S}(\tau nf(R))$, by Fact 5. We can apply a *(width $\preceq$)* rule to get $\tau nf(\Gamma) \vdash_N \mathbf{class}\,t.\langle\!\langle \tau nf(R) \mid \vec{\mathtt{n}}{:}\tau nf(\vec{\alpha})\rangle\!\rangle \preceq \mathbf{class}\,t.\tau nf(R)$, that is, again by Fact 5, $\tau nf(\Gamma) \vdash_N \tau nf(\mathbf{class}\,t.\langle\!\langle R \mid \vec{\mathtt{n}}{:}\vec{\alpha}\rangle\!\rangle) \preceq \tau nf(\mathbf{class}\,t.R)$.
3. $A \bullet B$ is a statement of the form $\mathtt{e} : \tau$. If the last applied rule is *(sub $\preceq$)*, then the thesis follows from case (2). All other cases are straightforward by induction. ■

4.3 Technical Lemmas

We are going to show some technical lemmas, necessary to prove some parts of the *subject reduction* theorem. They essentially say that each component of a judgement is well-formed.

The following lemma shows that the contexts are well-formed in every judgement and allows to treat contexts, which are lists, more like sets.

Lemma 8. *1. If* $\Gamma \vdash_N A \bullet B$ *then* $\Gamma \vdash_N *$.
2. If $\Gamma, \Gamma' \vdash_N A \bullet B$ *and* $\Gamma, c{\bullet}C, \Gamma' \vdash_N *$ *then* $\Gamma, c{\bullet}C, \Gamma' \vdash_N A \bullet B$. ■

The second lemma allows us to build well-formed row expressions.

Lemma 9. *1. If* $\Gamma \vdash_N \lambda t_1 \ldots t_p.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\tau}_\Delta \rangle\!\rangle : T^p {\rightarrow} [\vec{\mathtt{n}}]$*, then* $\Gamma, t_1{:}T, \ldots, t_p{:}T \vdash_N \tau_i : T$ *for each* τ_i *in* $\vec{\tau}_\Delta$ *and* $\Gamma, t_1{:}T, \ldots, t_p{:}T \vdash_N R : [\vec{\mathtt{m}}, \vec{\mathtt{n}}]$.
2. If $\Gamma \vdash_N \textbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\tau}_\Delta \rangle\!\rangle : T$ *then* $\Gamma, t{:}T \vdash_N \tau_i : T$ *for each* τ_i *in* $\vec{\tau}_\Delta$ *and* $\Gamma, t{:}T \vdash_N R : [\vec{\mathtt{m}}]$. ■

The proofs of the above two lemmas are easy extensions of the proofs of Lemmas 4.11 and 4.12 in [6]. The last lemma of this section assures us that we can deduce only well-formed types for objects.

Lemma 10. *1. If* $\Gamma \vdash_N \sigma \preceq \tau$ *then* $\Gamma \vdash_N \sigma : T$ *and* $\Gamma \vdash_N \tau : T$.
2. If $\Gamma \vdash_N \mathsf{e} : \tau$ *then* $\Gamma \vdash_N \tau : T$.

Proof. By induction on the structure of derivations.

1. Most cases are trivial, or come from the induction hypothesis. We consider the case in which the last applied rule is *(width* $\preceq$*)*. The hypothesis of the rule gives us: $\Gamma \vdash_N \textbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{n}}{:}\vec{\alpha} \rangle\!\rangle : T$. Let R be $\langle\!\langle \vec{\mathtt{m}}{:}\vec{\tau}_\Delta \rangle\!\rangle$, for some $\vec{\mathtt{m}}$, $\vec{\tau}_\Delta$. By lemmas 9 (2) and 8 (1), we have that $\Gamma, t{:}T \vdash_N *$. We can apply *(empty row)* rule to get $\Gamma, t{:}T \vdash_N \langle\!\langle\rangle\!\rangle : [\vec{\mathtt{m}}, \vec{\mathtt{n}}]$. By lemma 9 (2), we have that $\Gamma, t{:}T \vdash_N \tau_i : T$ for each τ_i in $\vec{\tau}_\Delta$. By applying a sequence of *(row ext)* rules to the above judgements we get $\Gamma, t{:}T \vdash_N \langle\!\langle \vec{\mathtt{m}}{:}\vec{\tau}_\Delta \rangle\!\rangle : [\vec{\mathtt{n}}]$. Finally, using the *(class)* rule, we can conclude $\Gamma \vdash_N \textbf{class}\, t.\langle\!\langle \vec{\mathtt{m}}{:}\vec{\tau}_\Delta \rangle\!\rangle : T$.
2. The only case not treated in [6] is when the last applied rule is *(sub* $\preceq$*)*. This case immediately follows from part (1). ■

4.4 Subject Reduction Theorem

We are going to prove the *subject reduction* property for our calculus, by a case analysis of the $\overset{eval}{\rightarrow}$ rules.

The next lemma is used to prove that β-reduction preserves types.

Lemma 11. *1. If* $\Gamma, x{:}\tau_1, \Gamma' \vdash_N \mathsf{e}_2 : \tau_2$ *and* $\Gamma \vdash_N \mathsf{e}_1 : \tau_1$ *then* $\Gamma, \Gamma' \vdash_N [\mathsf{e}_1/x]\mathsf{e}_2 : \tau_2$.
2. If $\Gamma \vdash_N \mathsf{e} : \tau$ *and* $\mathsf{e} {\rightarrow_\beta} \mathsf{e}'$ *then* $\Gamma \vdash_N \mathsf{e}' : \tau$.

Proof. 1. By induction on the structure of a derivation of $\Gamma, x{:}\tau_1, \Gamma' \vdash_N \mathsf{e}_2 : \tau_2$. For each case we distinguish whether the x variable occurs free in e_2 or does not occur at all.

2. By (1). ■

Now we can state the Subject Reduction Theorem.

Theorem 12. *If* $\Gamma \vdash_N \mathtt{e} : \sigma$ *and* $\mathtt{e} \stackrel{eval}{\to} \mathtt{e}'$ *then* $\Gamma \vdash_N \mathtt{e}' : \sigma$.

Proof. It is enough to give that each of the basic evaluation steps preserves the type of the expression being reduced. We show the derivation for the left-hand side of each rule (considering the most difficult cases, in which the *(sub* $\preceq$*)* rule is applied after each other rule) and then we build the correct derivation for the right-hand side. For the rules (*succ* $\hookleftarrow$) and (*next* $\hookleftarrow$), we consider the $\leftarrow\!\circ$ cases only. The $\leftarrow$ cases are similar.

- (β) $(\lambda x.\mathtt{e}_1)\mathtt{e}_2 \stackrel{eval}{\to} [\mathtt{e}_2/x]\mathtt{e}_1$. This case follows from Lemma 11.
- ($\Leftarrow$) $\mathtt{e} \Leftarrow \mathtt{n} \stackrel{eval}{\to} (\mathtt{e} \hookleftarrow \mathtt{n})\mathtt{e}$. In this case the left-hand side derivation is:

$$\begin{array}{l} \textit{(meth appl)} \\ \textit{(sub} \preceq\textit{)} \end{array} \quad \begin{array}{c} \Gamma \vdash_N \mathtt{e} : \mathbf{class}\,t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle \\ \hline \Gamma \vdash_N \mathtt{e} \Leftarrow \mathtt{n} : [\mathbf{class}\,t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle/t]\tau \\ \hline \Gamma \vdash_N \mathtt{e} \Leftarrow \mathtt{n} : \sigma. \end{array}$$

We can build the derivation for the right-hand side as follows. The judgement $\Gamma \vdash_N \mathtt{e} : \mathbf{class}\,t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle$ implies, by Lemma 10 (2), $\Gamma \vdash_N \mathbf{class}\,t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle : T$. So Lemma 9 (2) produces $\Gamma, t{:}T \vdash_N R : [\vec{\mathtt{m}}, \mathtt{n}]$. Therefore, we can conclude as follows:

$$\begin{array}{l} \textit{(exp app)} \\ \textit{(sub} \preceq\textit{)} \end{array} \quad \begin{array}{c} \mathcal{D} \qquad \Gamma \vdash_N \mathtt{e} : \mathbf{class}\,t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle \\ \hline \Gamma \vdash_N (\mathtt{e} \hookleftarrow \mathtt{n})\mathtt{e} : [\mathbf{class}\,t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle/t]\tau \\ \hline \Gamma \vdash_N (\mathtt{e} \hookleftarrow \mathtt{n})\mathtt{e} : \sigma, \end{array}$$

where $\mathcal{D}$ is the following derivation:

$$\textit{(meth search)} \quad \begin{array}{c} \Gamma \vdash_N \mathtt{e} : \mathbf{class}\,t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle \quad \Gamma, t{:}T \vdash_N R : [\vec{\mathtt{m}}, \mathtt{n}] \\ \hline \Gamma \vdash_N \mathtt{e} \hookleftarrow \mathtt{n} : [\mathbf{class}\,t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle/t](t \to \tau). \end{array}$$

- (*succ* $\hookleftarrow$) $\langle \mathtt{e}_1 \leftarrow\!\circ\, \mathtt{n}{=}\mathtt{e}_2 \rangle \hookleftarrow \mathtt{n} \stackrel{eval}{\to} \mathtt{e}_2$. In this case the left-hand side is:

$$\begin{array}{l} \\ \\ \textit{(obj ext)} \\ \textit{(sub} \preceq\textit{)} \\ \textit{(meth search)} \\ \textit{(sub} \preceq\textit{)} \end{array} \quad \begin{array}{c} \Gamma \vdash_N \mathtt{e}_1 : \mathbf{class}\,t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}\rangle\!\rangle \quad \Gamma, t{:}T \vdash_N R : [\vec{\mathtt{m}}, \mathtt{n}] \quad \mathtt{n} \notin \mathcal{S}(\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}\rangle\!\rangle) \\ \Gamma, r{:}T \to [\vec{\mathtt{m}}, \mathtt{n}] \vdash_N \mathtt{e}_2 : [\mathbf{class}\,t.\langle\!\langle rt \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle/t](t \to \tau) \quad r \text{ not in } \tau \\ \hline \Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow\!\circ\, \mathtt{n}{=}\mathtt{e}_2 \rangle : \mathbf{class}\,t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle \\ \hline \Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow\!\circ\, \mathtt{n}{=}\mathtt{e}_2 \rangle : \mathbf{class}\,t.\langle\!\langle R' \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle \quad \Gamma, t{:}T \vdash_N R'' : [\vec{\mathtt{m}}, \mathtt{n}] \\ \hline \Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow\!\circ\, \mathtt{n}{=}\mathtt{e}_2 \rangle \hookleftarrow \mathtt{n} : [\mathbf{class}\,t.\langle\!\langle R'' \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle/t](t \to \tau) \\ \hline \Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow\!\circ\, \mathtt{n}{=}\mathtt{e}_2 \rangle \hookleftarrow \mathtt{n} : \sigma, \end{array}$$

Observe that it is not possible to forget $\mathtt{n}$ in the first application of *(sub* $\preceq$*)* rule, since afterwards we have to type its search. Moreover, it is not possible to forget any of the $\vec{\mathtt{m}}$ methods, since $\mathtt{n}$ uses them.

From $\Gamma, t{:}T \vdash_N R'' : [\vec{\mathtt{m}}, \mathtt{n}]$, by applying *(row fn abs)* rule, we get $\Gamma \vdash_N \lambda t.R'' : T \to [\vec{\mathtt{m}}, \mathtt{n}]$. This, together with $\Gamma, r{:}T \to [\vec{\mathtt{m}}, \mathtt{n}] \vdash_N \mathtt{e}_2 : [\mathbf{class}\,t.\langle\!\langle rt \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle/t](t \to \tau)$ implies, by Lemma 3, $\Gamma \vdash \mathtt{e}_2 : [\mathbf{class}\,t.\langle\!\langle (\lambda t.R'')t \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle/t](t \to \tau)$. So, by Lemma 7, we can conclude $\Gamma \vdash_N \mathtt{e}_2 : [\mathbf{class}\,t.\langle\!\langle R'' \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle/t](t \to \tau)$. Then we build a derivation for the right-hand side as follows:

$$\textit{(sub} \preceq\textit{)} \quad \begin{array}{c} \Gamma \vdash_N \mathtt{e}_2 : [\mathbf{class}\,t.\langle\!\langle R'' \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{m}}\}}\rangle\!\rangle/t](t \to \tau) \\ \hline \Gamma \vdash_N \mathtt{e}_2 : \sigma. \end{array}$$

This proof shows the usefulness of labeled types. Thanks to the label $\{\vec{\mathtt{m}}\}$ (that contains all the methods used by $\mathtt{n}$) it is possible to reconstruct the correct type of $\mathtt{e}_2$ in the derivation of the left-hand side of the rule, and therefore the correct type of the right-hand side.

- $(\textit{next} \hookleftarrow)$ $\langle \mathtt{e}_1 \leftarrow\!\circ\, \mathtt{m}{=}\mathtt{e}_2 \rangle \hookleftarrow \mathtt{n} \overset{eval}{\rightarrow} \mathtt{e}_1 \hookleftarrow \mathtt{n}$.

There are two possible cases, according to whether the labeled type of $\mathtt{m}$ contains $\mathtt{n}$ or not. We consider the first case, the second being similar. The left-hand side is:

$$\begin{array}{l} (\textit{meth search}) \\ (\textit{sub} \preceq) \end{array} \quad \dfrac{\dfrac{\mathcal{D} \qquad\qquad \Gamma, t{:}T \vdash_N R'' : [\vec{\mathtt{q}}, \mathtt{n}]}{\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow\!\circ\, \mathtt{m}{=}\mathtt{e}_2 \rangle \hookleftarrow \mathtt{n} : [\mathbf{class}\, t.\langle\!\langle R'' \mid \vec{\mathtt{q}}{:}\vec{\beta}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{q}}\}} \rangle\!\rangle / t](t \to \tau)}}{\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow\!\circ\, \mathtt{m}{=}\mathtt{e}_2 \rangle \hookleftarrow \mathtt{n} : \sigma,}$$

where $\mathcal{D}$ is the following derivation:

$$\begin{array}{l} (\textit{obj ext}) \\ (\textit{sub} \preceq) \end{array} \quad \dfrac{\dfrac{\begin{array}{l} \Gamma \vdash_N \mathtt{e}_1 : \mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{p}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{q}}\}} \rangle\!\rangle \quad \Gamma, t{:}T \vdash_N R : [\vec{\mathtt{p}}, \mathtt{n}, \mathtt{m}] \\ \Gamma, r{:}T \to [\vec{\mathtt{p}}, \mathtt{m}, \mathtt{n}] \vdash_N \mathtt{e}_2 : [\mathbf{class}\, t.\langle\!\langle rt \mid \vec{\mathtt{p}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{q}}\}}, \mathtt{m}{:}\rho_{\{\vec{\mathtt{p}},\mathtt{n}\}} \rangle\!\rangle / t](t \to \rho) \\ \mathtt{m} \notin \mathcal{S}(\langle\!\langle R \mid \vec{\mathtt{p}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{q}}\}} \rangle\!\rangle) \quad r \text{ not in } \rho \end{array}}{\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow\!\circ\, \mathtt{m}{=}\mathtt{e}_2 \rangle : \mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{p}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{q}}\}}, \mathtt{m}{:}\rho_{\{\vec{\mathtt{p}},\mathtt{n}\}} \rangle\!\rangle}}{\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow\!\circ\, \mathtt{m}{=}\mathtt{e}_2 \rangle : \mathbf{class}\, t.\langle\!\langle R' \mid \vec{\mathtt{q}}{:}\vec{\beta}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{q}}\}} \rangle\!\rangle.}$$

The correctness of the application of *(sub $\preceq$)* rule in $\mathcal{D}$ implies that $\langle\!\langle \vec{\mathtt{q}}{:}\vec{\beta}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{q}}\}} \rangle\!\rangle$ is a subrow of $\langle\!\langle R \mid \vec{\mathtt{p}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{q}}\}}, \mathtt{m}{:}\rho_{\{\vec{\mathtt{p}},\mathtt{n}\}} \rangle\!\rangle$. Moreover, the condition $\mathtt{m} \notin \mathcal{S}(\langle\!\langle R \mid \vec{\mathtt{p}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{q}}\}} \rangle\!\rangle)$ implies $\mathtt{m} \notin \{\vec{\mathtt{q}}\}$. Therefore, $\langle\!\langle R \mid \vec{\mathtt{p}}{:}\vec{\alpha}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{q}}\}} \rangle\!\rangle \equiv \langle\!\langle R''' \mid \vec{\mathtt{q}}{:}\vec{\beta}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{q}}\}} \rangle\!\rangle$, for a suitable R'''. So we can build the derivation for the right-hand side as follows:

$$\begin{array}{l} (\textit{meth search}) \\ (\textit{sub} \preceq) \end{array} \quad \dfrac{\dfrac{\Gamma \vdash_N \mathtt{e}_1 : \mathbf{class}\, t.\langle\!\langle R''' \mid \vec{\mathtt{q}}{:}\vec{\beta}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{q}}\}} \rangle\!\rangle \quad \Gamma, t{:}T \vdash_N R'' : [\vec{\mathtt{q}}, \mathtt{n}]}{\Gamma \vdash_N \mathtt{e}_1 \hookleftarrow \mathtt{n} : [\mathbf{class}\, t.\langle\!\langle R'' \mid \vec{\mathtt{q}}{:}\vec{\beta}, \mathtt{n}{:}\tau_{\{\vec{\mathtt{q}}\}} \rangle\!\rangle / t](t \to \tau)}}{\Gamma \vdash_N \mathtt{e}_1 \hookleftarrow \mathtt{n} : \sigma.}$$

Clearly, the left-hand sides of all other evaluation rules cannot be typed. ■

4.5 Type Soundness

The subject reduction proof shows the power of our typing system. Labeled types not only allow a restricted form of subtyping that enriches the set of types of typable objects, but they also lead us to find a simpler and more natural operational semantics, in which no transformations on the objects are necessary to get the body of a message. In fact, the typing rule for the $\hookleftarrow$ operation is strictly based on the information given by the labels. Moreover, since an $\overset{eval}{\rightarrow}$ produces the object `err` when a message $\mathtt{m}$ is sent to an expression which does not define an object with a method $\mathtt{m}$, the type soundness follows from Theorem 12.

Theorem 13. *If $\Gamma \vdash_N \mathtt{e} : \tau$ is derivable for some Γ and τ, then the evaluation of* $\mathtt{e}$ *cannot produce* `err`*, i.e.* $\mathtt{e} \overset{eval}{\not\rightarrow} \mathtt{err}$*.* ■

Notice that in [6] the type soundness was proved by introducing a structural operational semantics and showing suitable properties.

5 Conclusions

This paper extends the delegation-based calculus of objects of [5] with a subtyping relation between types. This new relation states that two types of objects with different numbers of methods can be subsumed under certain restrictions. This restricted form of subsumption is conservative with respect to the features of delegation-based languages which are present in the original system.

Among the other proposals, that allow width specialization we have studied, one solution is to explicitely coerce an object with more methods into an object with less methods, by expanding each call of a method that does not belong to the smaller type with the proper body of that method. This solution forces a new subsumption rule that performs explicitely this job; this rule creates a quite different object but allows to eliminate the labels and the restrictions on them.

Further goals of our research are:

- finding a denotational model for the calculus,
- adding some mechanism to model *encapsulation*,
- determining if the type-checking of this calculus is decidable.

Acknowledgements

The authors wish to thank Mariangiola Dezani-Ciancaglini for her precious technical support and encouragement, Furio Honsell and Kathleen Fisher for their helpful comments, Steffen van Bakel and Paola Giannini for their careful reading of the preliminary versions of the paper.

References

1. Abadi, M., Cardelli, L., A Theory of Primitive Objects. Manuscript, 1994. Also in *Proc. Theoretical Aspect of Computer Software*, LNCS 789, Springer-Verlag, 1994, pp. 296–320.
2. Bellè, G. Some Remarks on Lambda Calculus of Objects. Internal Report, Dipartimento di Matematica ed Informatica, Università di Udine, 1994.
3. Borning, A.H., Ingalls, D.H., A Type Declaration and Inference System for Smalltalk. In *Proc. ACM Symp. Principles of Programming Languages*, ACM Press, 1982, pp. 133–141.
4. Ellis, E., Stroustrop, B., *The Annotated* C^{++} *Reference Manual.* Addison Wesley, 1990.
5. Fisher, K., Honsell, F., Michell, J. C. A Lambda Calculus of Objects and Method Specialization. In *Proc. 8th Annual IEEE Symposium on Logic in Computer Science*, Computer Society Press, 1993, pp. 26–38.
6. Fisher, K., Honsell, F., Mitchell, J. C. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, **1** (1), 1994, pp. 3–37.
7. Fisher, K., Michell, J. C. Notes on Typed Object-Oriented Programming. In *Proc. Theoretical Aspect of Computer Software*, LNCS 789, Springer-Verlag, 1994, pp. 844–885.

Appendix 1: Evaluation Rules

(β)	$(\lambda x.\mathsf{e}_1)\mathsf{e}_2$	$\overset{eval}{\rightarrow}$	$[\mathsf{e}_2/x]\mathsf{e}_1$	$(err\ abs)$	$\lambda x.\mathtt{err}$	$\overset{eval}{\rightarrow}$	$\mathtt{err}$
$(\Leftarrow)$	$\mathsf{e} \Leftarrow \mathtt{m}$	$\overset{eval}{\rightarrow}$	$(\mathsf{e} \hookleftarrow \mathtt{m})\mathsf{e}$	$(err\ appl)$	$\mathtt{err}\ \mathsf{e}$	$\overset{eval}{\rightarrow}$	$\mathtt{err}$
$(succ \hookleftarrow)$	$\langle \mathsf{e}_1 \leftarrow\!\!* \ \mathtt{m} = \mathsf{e}_2 \rangle \hookleftarrow \mathtt{m}$	$\overset{eval}{\rightarrow}$	e_2	$(err \hookleftarrow)$	$\mathtt{err} \hookleftarrow \mathtt{n}$	$\overset{eval}{\rightarrow}$	$\mathtt{err}$
$(next \hookleftarrow)$	$\langle \mathsf{e}_1 \leftarrow\!\!* \ \mathtt{n} = \mathsf{e}_2 \rangle \hookleftarrow \mathtt{m}$	$\overset{eval}{\rightarrow}$	$\mathsf{e}_1 \hookleftarrow \mathtt{m}$	$(fail\ \langle\rangle)$	$\langle\rangle \hookleftarrow \mathtt{m}$	$\overset{eval}{\rightarrow}$	$\mathtt{err}$
$(err \leftarrow\!\!* \ )$	$\langle \mathtt{err} \leftarrow\!\!* \ \mathtt{m} = \mathsf{e} \rangle$	$\overset{eval}{\rightarrow}$	$\mathtt{err}$	$(fail\ abs)$	$\lambda x.\mathsf{e} \hookleftarrow \mathtt{m}$	$\overset{eval}{\rightarrow}$	$\mathtt{err}$

Appendix 2: Typing rules

General Rules

(start) $$\frac{}{\varepsilon \vdash *}$$

(proj) $$\frac{\Gamma \vdash * \quad A \bullet B \in \Gamma}{\Gamma \vdash A \bullet B}$$

(weak) $$\frac{\Gamma \vdash A \bullet B \quad \Gamma, \Gamma' \vdash *}{\Gamma, \Gamma' \vdash A \bullet B}$$

Rules for type expressions

(type var) $$\frac{\Gamma \vdash * \quad t \notin dom(\Gamma)}{\Gamma, t{:}T \vdash *}$$

(type arr) $$\frac{\Gamma \vdash \tau_1 : T \quad \Gamma \vdash \tau_2 : T}{\Gamma \vdash \tau_1 \rightarrow \tau_2 : T}$$

(class) $$\frac{\Gamma, t{:}T \vdash R : [\vec{\mathtt{m}}]}{\Gamma \vdash \mathbf{class}\, t.R : T}$$

Type and row equality
We consider α-conversion of type variables bound by λ or **class** and application of the principle:

$$\langle\!\langle \langle\!\langle R \mid \mathtt{n}{:}\tau^1_{\Delta_1} \rangle\!\rangle \mid \mathtt{m}{:}\tau^2_{\Delta_2} \rangle\!\rangle = \langle\!\langle \langle\!\langle R \mid \mathtt{m}{:}\tau^2_{\Delta_2} \rangle\!\rangle \mid \mathtt{n}{:}\tau^1_{\Delta_1} \rangle\!\rangle$$

within type or row expression to be conventions of syntax, rather than explicit rules of the system. Additional equations between types and rows arise as result of β-reduction, written $\rightarrow_\beta$, or $\leftrightarrow_\beta$.

(type β) $$\frac{\Gamma \vdash \tau : T \quad \tau \rightarrow_\beta \tau'}{\Gamma \vdash \tau' : T}$$

(row β) $$\frac{\Gamma \vdash R : \kappa \quad R \rightarrow_\beta R'}{\Gamma \vdash R' : \kappa}$$

(type eq) $$\frac{\Gamma \vdash \mathsf{e} : \tau \quad \tau \leftrightarrow_\beta \tau' \quad \Gamma \vdash \tau' : T}{\Gamma \vdash \mathsf{e} : \tau'}$$

Rules for rows

(empty row) $$\frac{\Gamma \vdash *}{\Gamma \vdash \langle\!\langle\rangle\!\rangle : [\vec{\mathtt{m}}]}$$

(row var) $$\frac{\Gamma \vdash * \quad r \notin dom(\Gamma)}{\Gamma, r{:}T^n \rightarrow [\vec{\mathtt{m}}] \vdash *}$$

(row label) $$\frac{\Gamma \vdash R : T^n \rightarrow [\vec{\mathtt{m}}] \quad \{\vec{\mathtt{n}}\} \subseteq \{\vec{\mathtt{m}}\}}{\Gamma \vdash R : T^n \rightarrow [\vec{\mathtt{n}}]}$$

(row fn abs) $$\frac{\Gamma, t{:}T \vdash R : T^n \rightarrow [\vec{\mathtt{m}}]}{\Gamma \vdash \lambda t.R : T^{n+1} \rightarrow [\vec{\mathtt{m}}]}$$

(row fn app) $$\frac{\Gamma \vdash R : T^{n+1} \rightarrow [\vec{\mathtt{m}}] \quad \Gamma \vdash \tau : T}{\Gamma \vdash R\tau : T^n \rightarrow [\vec{\mathtt{m}}]}$$

(row ext) $$\frac{\Gamma \vdash R : [\vec{\mathtt{m}}, \mathtt{n}] \quad \Gamma \vdash \tau : T}{\Gamma \vdash \langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle : [\vec{\mathtt{m}}]}$$

Rules for assigning types to terms

(exp var) $$\frac{\Gamma \vdash \tau : T \quad x \notin dom(\Gamma)}{\Gamma, x{:}\tau \vdash *}$$

(exp abs) $$\frac{\Gamma, x{:}\tau_1 \vdash \mathsf{e} : \tau_2}{\Gamma \vdash \lambda x.\mathsf{e} : \tau_1 \rightarrow \tau_2}$$

(empty obj) $$\frac{\Gamma \vdash *}{\Gamma \vdash \langle\rangle : \mathbf{class}\, t.\langle\!\langle\rangle\!\rangle}$$

(exp app) $$\frac{\Gamma \vdash \mathsf{e}_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash \mathsf{e}_2 : \tau_1}{\Gamma \vdash \mathsf{e}_1 \mathsf{e}_2 : \tau_2}$$

$$(\mathit{sub} \preceq)\quad \frac{\Gamma \vdash \mathsf{e} : \tau_1 \quad \Gamma \vdash \tau_1 \preceq \tau_2}{\Gamma \vdash \mathsf{e} : \tau_2} \qquad (\mathit{meth\ appl})\quad \frac{\Gamma \vdash \mathsf{e} : \mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathrm{n}{:}\tau_{\{\vec{\mathtt{m}}\}} \rangle\!\rangle}{\Gamma \vdash \mathsf{e} \Leftarrow \mathrm{n} : [\mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathrm{n}{:}\tau_{\{\vec{\mathtt{m}}\}} \rangle\!\rangle / t]\tau}$$

$$(\mathit{meth\ search})\quad \frac{\Gamma \vdash \mathsf{e} : \mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathrm{n}{:}\tau_{\{\vec{\mathtt{m}}\}} \rangle\!\rangle \quad \Gamma, t{:}T \vdash R' : [\vec{\mathtt{m}}, \mathrm{n}]}{\Gamma \vdash \mathsf{e} \hookleftarrow \mathrm{n} : [\mathbf{class}\, t.\langle\!\langle R' \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathrm{n}{:}\tau_{\{\vec{\mathtt{m}}\}} \rangle\!\rangle / t](t \rightarrow \tau)}$$

$$(\mathit{obj\ ext})\quad \frac{\begin{array}{c}\Gamma \vdash \mathsf{e}_1 : \mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha} \rangle\!\rangle \qquad \Gamma, t{:}T \vdash R : [\vec{\mathtt{m}}, \mathrm{n}] \qquad \mathrm{n} \notin \mathcal{S}(\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha} \rangle\!\rangle) \\ \Gamma, r{:}T \rightarrow [\vec{\mathtt{m}}, \mathrm{n}] \vdash \mathsf{e}_2 : [\mathbf{class}\, t.\langle\!\langle rt \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathrm{n}{:}\tau_{\{\vec{\mathtt{m}}\}} \rangle\!\rangle / t](t \rightarrow \tau) \quad r \text{ not in } \tau\end{array}}{\Gamma \vdash \langle \mathsf{e}_1 \leftarrow\!\!\circ\ \mathrm{n} = \mathsf{e}_2 \rangle : \mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathrm{n}{:}\tau_{\{\vec{\mathtt{m}}\}} \rangle\!\rangle}$$

$$(\mathit{obj\ over})\quad \frac{\begin{array}{l}\Gamma \vdash \mathsf{e}_1 : \mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathrm{n}{:}\tau_{\Delta} \rangle\!\rangle \\ \Gamma, r{:}T \rightarrow [\vec{\mathtt{m}}, \mathrm{n}] \vdash \mathsf{e}_2 : [\mathbf{class}\, t.\langle\!\langle rt \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathrm{n}{:}\tau_{\{\vec{\mathtt{m}}\}} \rangle\!\rangle / t](t \rightarrow \tau) \quad r \text{ not in } \tau\end{array}}{\Gamma \vdash \langle \mathsf{e}_1 \leftarrow \mathrm{n} = \mathsf{e}_2 \rangle : \mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathtt{m}}{:}\vec{\alpha}, \mathrm{n}{:}\tau_{\{\vec{\mathtt{m}}\}} \rangle\!\rangle}$$

Rules of subtyping

$$(\mathit{width} \preceq)\quad \frac{\Gamma \vdash \mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathrm{n}}{:}\vec{\alpha} \rangle\!\rangle : T \quad \vec{\mathrm{n}} \notin \mathcal{S}(R)}{\Gamma \vdash \mathbf{class}\, t.\langle\!\langle R \mid \vec{\mathrm{n}}{:}\vec{\alpha} \rangle\!\rangle \preceq \mathbf{class}\, t.R}$$

$$(\mathit{const} \preceq)\quad \frac{\Gamma \vdash * \quad \Gamma \vdash \iota_1 : T \quad \Gamma \vdash \iota_2 : T}{\Gamma, \iota_1 \preceq \iota_2 \vdash *} \qquad (\mathit{trans} \preceq)\quad \frac{\Gamma \vdash \sigma \preceq \tau \quad \Gamma \vdash \tau \preceq \rho}{\Gamma \vdash \sigma \preceq \rho}$$

$$(\mathit{refl} \preceq)\quad \frac{\Gamma \vdash \sigma : T}{\Gamma \vdash \sigma \preceq \sigma} \qquad (\mathit{arrow} \preceq)\quad \frac{\Gamma \vdash \sigma' \preceq \sigma \quad \Gamma \vdash \tau \preceq \tau'}{\Gamma \vdash \sigma \rightarrow \tau \preceq \sigma' \rightarrow \tau'}$$

Appendix 3: Translation Function $\boldsymbol{tr} : \boldsymbol{Row} \cup \boldsymbol{Types} \rightarrow \boldsymbol{\lambda}^{\rightarrow}(\boldsymbol{\Sigma})$

Given the following signature Σ:

$$\begin{array}{lll} \mathit{Type\ Constant} & : & \mathit{typ}, \mathit{meth} \\ \mathit{Term\ Constant} & : & \mathtt{iot}_i : \mathit{typ},\ \text{for each constant type } \iota_i \\ & & \mathtt{er} : \mathit{meth} \\ & & \mathtt{ar} : \mathit{typ} \rightarrow \mathit{typ} \rightarrow \mathit{typ} \\ & & \mathtt{cl} : (\mathit{typ} \rightarrow \mathit{meth}) \rightarrow \mathit{typ} \\ & & \mathtt{br}_{\mathtt{m}} : \mathit{meth} \rightarrow \mathit{typ} \rightarrow \mathit{meth},\ \text{for each method name } \mathtt{m}, \end{array}$$

the function $tr : Row \cup Types \rightarrow \lambda^{\rightarrow}(\Sigma)$ is inductively defined as follows:

$$\begin{array}{ll} tr(\iota_i) = \mathtt{iot}_i & tr(r) = r \\ tr(t) = t & tr(\langle\!\langle\rangle\!\rangle) = \mathtt{er} \\ tr(\tau_1 \rightarrow \tau_2) = \mathtt{ar}\ tr(\tau_1)\ tr(\tau_2) & tr(\langle\!\langle R \mid \mathtt{m}{:}\tau_{\Delta} \rangle\!\rangle) = \mathtt{br}_{\mathtt{m}}\ tr(R)\ tr(\tau) \\ tr(\mathbf{class}\, t.R) = \mathtt{cl}\ (\lambda t{:}\mathit{typ}.tr(R)) & tr(\lambda t.R) = \lambda t{:}\mathit{typ}.tr(R) \\ tr(R\tau) = tr(R)\ tr(\tau). & \end{array}$$

We extend tr to kinds and contexts in the standard way.

$$\begin{array}{ll} tr(T) = \mathit{typ} & tr(T^n \rightarrow [\vec{\mathtt{m}}]) = \mathit{typ}^n \rightarrow \mathit{meth} \\ tr(\epsilon) = \emptyset & tr(\Gamma, \iota{:}T) = tr(\Gamma) \cup \{tr(\iota) : tr(T)\} \\ tr(\Gamma, \mathtt{x}{:}\tau) = tr(\Gamma) & tr(\Gamma, \iota_1 \preceq \iota_2) = tr(\Gamma) \cup \{tr(\iota_1) \preceq tr(\iota_2)\} \\ tr(\Gamma, t{:}T) = tr(\Gamma) \cup \{tr(t){:}tr(T)\} & tr(\Gamma, r{:}\kappa) = tr(\Gamma) \cup \{tr(t){:}tr(\kappa)\}. \end{array}$$

This article was processed using the LaTeX macro package with LLNCS style