Matching [for] the Lambda Calculus of Objects ⋆

Viviana Bono [1]

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

Michele Bugliesi

Dipartimento di Matematica, Università di Padova
Via Belzoni 7, I-35131 Padova, Italy
e-mail: `michele@math.unipd.it`

Abstract

A relation between recursive object types, called *matching*, has been proposed [8] to provide an adequate typing of inheritance in class-based languages.

This paper investigates the role of this relation in the design of a type system for the *Lambda Calculus of Objects* [15]. A new type system for this calculus is defined that uses implicit match-bounded quantification over type variables instead of implicit quantification over row schemes – as in [15] – to capture *MyType* polymorphic types for methods. An operational semantics is defined for the untyped calculus and type soundness for the new system is proved as a corollary of a subject reduction property. A formal analysis of the relative expressive power of the two systems is also carried out, that explains how the row schemes of [15] can be understood in terms of matching, and shows that the new system is as powerful as the original one on derivations of typing judgements for closed objects.

1 Introduction

The design of safe and flexible type systems for object-oriented languages has been a challenge for a large research community over the past years. The interest of this research has initially been centered around *class-based* languages.

⋆ A preliminary version of the present paper appeared in [4].
[1] Partially supported by ESPRIT BRA 7232, Gentzen.

More recently, in the attempt to gain deeper insight into some of the fundamental typing problems arising in these languages, researchers in the field have directed their attention to *object-based* calculi [3].

Despite the conceptual difference between the underlying object-oriented models [2], several ideas originated from the experience on class-based languages – notably, the notions of *row variable* [17] and of *recursive record types* [12,11,14] – have then proved useful in the design of type systems for object-based languages. Continuing this line of research, in this paper we investigate the role of *matching* [8] in the design of a type system for the *Lambda Calculus of Objects* [15].

The *Lambda Calculus of Objects* [15] is an object-based language defined as an extension of the simply-typed λ-calculus with object-related forms and primitives for defining (or redefining) and invoking methods on objects. The distinguishing feature of this calculus, with respect to its companion calculi in the literature [3] is the presence of primitives for changing the shape of an object by the addition of new methods. In [15] a type system for this calculus is defined, that provides for static detection of errors such as *message-not-understood*, while at the same time allowing types of methods to be specialized to the type of the inheriting objects. This mechanism, that is commonly referred to as *MyType* specialization, is rendered in the type system in terms of a form of higher-order row-polymorphism which, in turn, uses implicit quantification over *row schemes* to capture the underling notion of *protocol extension* (called *row extension* in [15]).

The type system we present in this paper inherits some of the fundamental ideas from the original system – notably the same variable-discharging pattern in the typing of methods – but takes a rather different approach in the modeling of *MyType* specialization. The new solution is based on using *matching* and implicit match-bounded quantification over type variables to characterize types of methods as type-schemes (i.e. types polymorphic in these variables), and to enforce correct instantiation of the schemes as methods are inherited.

Matching is a relation over recursive object-types that was first introduced by [8] as an alternative to F-bounded subtyping [13] in modeling the subclass relation in class-based languages. In [2], this relation was then proposed as a complement to higher-order subtyping to provide an adequate typing of inheritance of methods in related languages. The system we present in this paper shows that matching can be employed to explain the effect of row-polymorphism in the type system of [15]. A formal study of the relative ex-

[2] Briefly: in class-based languages, objects are created by class instantiation and inheritance takes place at the class level. In object-based models, instead, objects are created from existing objects used as prototypes, and inheritance occurs at the object-level.

pressive power of the two systems shows, in fact, that the new system is as powerful as the original one on typing judgements for closed objects. The proof of this result is rather technical, but at the same time interesting at the conceptual level, because it sheds considerable light on the role of row schemes in [15]. The proof is carried out in two steps. A preliminary analysis of the original system allows us to identify and isolate the "relevant" uses of row schemes in the typing derivations of that system, and to show how they can be encoded into derivations of the new system using corresponding type variables. From this result, we then show that all typing judgements for closed objects derivable in [15] are derivable also in our system.

The choice of matching and implicit match-bounded quantification in the new system has two interesting payoffs with respect to quantification over row schemes. Firstly, as shown by the comparative analysis of the two systems, it saves part of the technical overhead of the calculus of rows, and it allows simpler and more direct inductive proofs of subject reduction and type soundness. Secondly, and more importantly, it seems to provide deeper insights as to how the Lambda Calculus of Objects relates to other existing object-based and class-based languages, notably to the *Object Calculus* of [3], and to the calculi of [8,10]. The new characterization of row schemes in terms of matching may provide a simple and uniform basis for relating these systems on a formal basis.

2 The Untyped Calculus

An expression of the untyped calculus is any of the following:

$$e ::= x \mid c \mid \lambda x.e \mid e_1\, e_2 \mid \langle\rangle \mid \langle e_1 \longleftarrow\!\!\!+\, m{=}e_2\rangle \mid \langle e_1 \leftarrow m{=}e_2\rangle \mid e \Leftarrow m$$

where x is a variable, c a constant and m a method name. As usual, expressions that differ only in the names of bound variables are identified; constants are assumed to range over values of basic types. The reading of the object-related forms is as follows:

$\langle\rangle$	is the empty object,
$\langle e_1 \longleftarrow\!\!\!+\, m{=}e_2\rangle$	extends e_1 with a new method m having body e_2,
$\langle e_1 \leftarrow m{=}e_2\rangle$	replaces the current body for m, in e_1, with e_2,
$e \Leftarrow m$	sends message m to e.

The expression $\langle e_1 \longleftarrow\!\!\!+\, m{=}e_2\rangle$ is defined only when e_1 denotes an object that does not have an m method, whereas $\langle e_1 \leftarrow m{=}e_2\rangle$ is defined only when e_1 denotes an object that *does* contain an m method. As in [15], both these conditions are enforced statically by the type system.

Method invocation, denoted by $e \Leftarrow m$, requires e to be an object containing the m method, and comprises two separate actions, *search* and *self-application*. Evaluating the message $e \Leftarrow m$ requires a search of the body of m within e; once the body is found, it is applied to e itself. To formalize this behavior, we introduce a subsidiary object expression, $e \hookleftarrow m$, whose intuitive semantics is as follows: evaluating $e \hookleftarrow m$ results into a recursive traversal of the "subobjects" of e, that succeeds upon reaching the right-most addition or override of the method in question. The use of the search expression is inspired by [7], and it provides a more direct technical device than the *bookkeeping* relation originally introduced in [15].

2.1 Reduction

Table 1 below defines the top-level reduction relation for the calculus; the definition is by cases, and the notation $\leftarrow\!\!\circ$ is short for either $\longleftarrow\!\!\!+$ or $\leftarrow$.

$$
\begin{array}{llcll}
(\beta) & (\lambda x.e_1)\, e_2 & \succ & [e_2/x]\, e_1 & \\
(\Leftarrow) & e \Leftarrow m & \succ & (e \hookleftarrow m)\, e & \\
(\hookleftarrow\ \mathit{succ}) & \langle e_1 \leftarrow\!\!\circ\, m{=}e_2 \rangle \hookleftarrow m & \succ & e_2 & \\
(\hookleftarrow\ \mathit{next}) & \langle e_1 \leftarrow\!\!\circ\, n{=}e_2 \rangle \hookleftarrow m & \succ & e_1 \hookleftarrow m & (n \neq m)
\end{array}
$$

Table 1. Top-level reduction for the untyped calculus.

The evaluation relation can be defined as follows. First define a *context* $C[-]$ to be an expression with a single hole, and let $C[e]$ be the result of filling the hole with the expression e. Then define the relation of one-step evaluation as follows:

$$e_1 \xrightarrow{eval} e_2 \quad \text{if} \quad e_1 \equiv C[e_1'],\ \ e_2 \equiv C[e_2'], \text{ and } \quad e_1' \succ e_2'$$

Finally, define the evaluation relation $\xrightarrow{eval^*}$ as the reflexive and transitive closure of $\xrightarrow{eval}$.

2.2 Operational Semantics

To formalize a notion of operational semantics, we introduce an evaluation strategy. Following the standard practice, this strategy is lazy in that it does not work under λ-abstractions; similarly, it defers reducing under primitives of object formation until reduction is required to evaluate a message send.

As usual, the goal of evaluation is to reduce a *closed* expression to a value. For the purpose of the present calculus, we define a value to be either a λ-abstraction, or a constant (including empty object $\langle\rangle$), or an object expression of the form $\langle e_1 \leftarrow\!\circ\ m{=}e_2\rangle$.

The evaluation strategy is defined in terms of two mutually recursive relations: $\Downarrow$ and $\rightsquigarrow$. $\Downarrow$ is the actual evaluation relation, that reduces expressions to values. $\rightsquigarrow$ is an auxiliary relation that reduces expressions to expressions (not necessarily values) and models the search of methods within objects required to evaluate message sends: mutual recursion is needed to handle this search correctly. The two relations are axiomatized by the following rules (*val* denotes an arbitrary value, and *obj* an object expression of the form $\langle e_1 \leftarrow\!\circ\ m{=}e_2\rangle$):

Evaluation.

$$(eval\ val) \qquad \frac{}{val \Downarrow val}$$

$$(eval\ app) \qquad \frac{e_1 \Downarrow \lambda x.e_1' \qquad [e_2/x]e_1' \Downarrow val}{e_1 e_2 \Downarrow val}$$

$$(eval\ send) \qquad \frac{e \Downarrow obj \qquad obj \hookleftarrow m \rightsquigarrow e' \qquad e'\ obj \Downarrow val}{e \Leftarrow m \Downarrow val}$$

Search.

$$(search\ succ) \qquad \frac{}{\langle e_1 \leftarrow\!\circ\ m{=}e_2\rangle \hookleftarrow m \rightsquigarrow e_2}$$

$$(search\ next) \qquad \frac{e_1 \Downarrow obj \qquad obj \hookleftarrow m \rightsquigarrow e}{\langle e_1 \leftarrow\!\circ\ n{=}e_2\rangle \hookleftarrow m \rightsquigarrow e}$$

Table 2. Operational Semantics.

Observe that the operational semantics is deterministic: specifically, if $e \Downarrow val_1$ and $e \Downarrow val_2$, then $val_1 \equiv val_2$. Furthermore, both $\Downarrow$ and $\rightsquigarrow$ are contained in $\xrightarrow{eval^*}$. More formally:

Proposition 1 *If* $e \Downarrow val$, *then* $e \xrightarrow{eval^*} val$. *Similarly, if* $e \rightsquigarrow e'$, *then* $e \xrightarrow{eval^*} e'$. $\square$

3 Types and Pro[to]-Types

Objects are elements of types of the form:

$$\mathtt{pro}\, t.\langle\!\langle m_1{:}\tau_1, \ldots, m_k{:}\tau_k \rangle\!\rangle$$

where the m_i's are method names, the τ_i's are type expressions, and the *row* $\langle\!\langle m_1{:}\tau_1, \ldots, m_k{:}\tau_k \rangle\!\rangle$ defines the interface or *protocol* of the objects of this type. The binder `pro` scopes over the row, and the bound variable t may occur free within the scope of the binder, with every free occurrence referring to the `pro` type itself. As such, `pro`-types are a form of recursively-defined types, even though `pro` is not to be understood as a fixed-point operator: as in [15], the self-referential nature of these types is axiomatized by the typing rules, rather than defined in terms of an explicit unfolding rule.

3.1 Types, Rows and Judgements

The complete set of types includes type constants, type variables, function types and `pro`-types, as defined by the following productions:

$$\text{Types} \qquad \tau \ ::= b \mid t \mid \tau {\to} \tau \mid \mathtt{pro}\, t.R$$

$$\text{Rows} \qquad R ::= \langle\!\langle\rangle\!\rangle \mid \langle\!\langle R \mid m{:}\tau \rangle\!\rangle.$$

The symbol b denotes type constants like *int*, *real*, ..., while t denotes type variables. Throughout the paper we use the following conventions: types are denoted by greek letters such as τ, σ, ς, ρ, ... Type variables, instead, are denoted by t as well as by the letters u and v: typically t will be the variable associated with the binder `pro`, while u and v are used to indicate match-bound type variables declared in a context (see below).

The structure of rows, ranged over by R in the above productions, is similar but simpler than in [15], as we disregard row schemes formed around row variables and, instead, require rows to be formed only as "ground" collections of pairs "method-name:type". One advantage of this choice is a simplified notion of well-formedness for `pro`-types: instead of using the kinding judgements of [15], well-formedness for types is formalized in our system simply as follows: the type $\mathtt{pro}\, t.\langle\!\langle m_1{:}\tau_1, \ldots, m_k{:}\tau_k \rangle\!\rangle$ is well-formed if $m_i \neq m_j$ whenever $i \neq j$, and τ_i is well-formed, for i, j in $\{1, \ldots, k\}$ (cf. Appendix A).

Row expressions that differ only for the name of the bound variable, or for the order of the component $m{:}\tau$ pairs are considered identical. More formally, α-conversion of type variables bound by `pro`, as well as applications of the principle: $\langle\!\langle\langle\!\langle R \mid n{:}\tau_1\rangle\!\rangle \mid m{:}\tau_2\rangle\!\rangle = \langle\!\langle\langle\!\langle R \mid m{:}\tau_2\rangle\!\rangle \mid n{:}\tau_1\rangle\!\rangle$, are taken as syntactic conventions. The notation $\langle\!\langle m_1{:}\tau_1, \ldots, m_k{:}\tau_k\rangle\!\rangle$, used above and in the remainder of the paper, is short for $\langle\!\langle\,\langle\!\langle \ldots \langle\!\langle\langle\!\langle\rangle\!\rangle \mid m_1{:}\tau_1\rangle\!\rangle \mid \ldots\rangle\!\rangle \mid m_k{:}\tau_k\rangle\!\rangle$.

The environments, or contexts, of the type system declare term variables with their types and match-bounds for type variables:

$$\Gamma ::= \varepsilon \mid \Gamma,\, x : \tau \mid \Gamma,\, u \mathrel{<\!\!\#} \tau.$$

Contexts are lists and each term and type variable may be declared at most once in a valid context (cf. rules in Appendix A). The judgements of the type system are the following:

$\Gamma \vdash *$	Γ is a valid context,
$\Gamma \vdash \tau$	τ is well-formed in Γ,
$\Gamma \vdash e : \tau$	term has type,
$\Gamma \vdash \tau_1 \mathrel{<\!\!\#} \tau_2$	type matches type.

We denote with $Var(\tau)$, for any type τ, the set of free type variables occurring in τ, and with $Dom(\Gamma)$ the domain of the context Γ (which includes both term and type variables).

3.2 Matching

Matching is the only relation over types that we assume in the type system. It is a reflexive and transitive relation defined only over type variables and `pro`-types; for `pro`-types it formalizes the notion of *protocol extension* needed in the typing of inheritance. The relation we use here is a simplification[3] of the original matching relation [8], defined by the following rule (the notation $\vec{m{:}\tau}$ is short for $m_1{:}\tau_1,\ldots,m_k{:}\tau_k$):

$$\frac{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle \vec{m{:}\tau}, \vec{n{:}\sigma}\rangle\!\rangle}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle \vec{m{:}\tau}, \vec{n{:}\sigma}\rangle\!\rangle \mathrel{<\!\!\#} \mathtt{pro}\, t.\langle\!\langle \vec{m{:}\tau}\rangle\!\rangle}\ (\mathrel{<\!\!\#}\ pro)$$

Unlike the original definition [8], that allows the component types of a `pro`-type to be promoted by subtyping, our definition requires that these types coincide with the component types of every `pro`-type placed higher-up in the $\mathrel{<\!\!\#}$ -hierarchy. Like the original relation, our relation has the peculiarity that it is *not* used in conjunction with a subsumption rule. These two restrictions have orthogonal motivations: the first is just a matter of convenience in the

[3] The same simplification can be found in [9].

analysis of the relationships between our system and the system of [15]; the second, instead, is crucial to prevent unsound uses of type promotion in the presence of method override (as in [8,2]).

3.3 Typing Rules for Objects

For the most part, the type system is routine; the interesting rules are those pertaining to the object-related part of the calculus, which we discuss next. The first rule defines the type of the empty object: having no method, the type of this object has an empty row.

$$\frac{\Gamma \vdash *}{\Gamma \vdash \langle\rangle : \mathtt{pro}\, t.\langle\!\langle\rangle\!\rangle} \quad (\mathit{empty}).$$

The next rule defines the typing of a method invocation:

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash \sigma \mathrel{\lessdot\#} \mathtt{pro}\, t.\langle\!\langle n{:}\tau \rangle\!\rangle}{\Gamma \vdash e \Leftarrow n : [\sigma/t]\tau} \quad (\mathit{send}).$$

A call to a method n on an object e requires e to have a pro-type containing the method name n. The result of the call has the type τ listed in the pro-type of e, with σ substituted for t. The substitution $[\sigma/t]$ in τ reflects the self-referential structure of pro-types we mentioned earlier in this section. A further interesting aspect of the above rule is that the type σ may either be a pro-type matching $\mathtt{pro}\, t.\langle\!\langle n{:}\tau \rangle\!\rangle$, or else an unknown type (i.e., a type variable) occurring (match-bounded) in the context Γ. Rules like (*send*) are sometimes referred to as *structural rules* [1], and their use is critical for an adequate rendering of *MyType* polymorphism: it is the ability to refer to possibly unknown types in the type rules that allows methods to act parametrically over any $u \mathrel{\lessdot\#} \tau$, where u is the type of *self* (i.e., the type of the host object), and τ is a given pro-type.

The need for structural rules like (*send*) is further clarified in the next rule, where we define the typing of method additions.

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \mathtt{pro}\, t.\langle\!\langle \vec{m}{:}\vec{\rho} \rangle\!\rangle \\ \Gamma,\, u \mathrel{\lessdot\#} \mathtt{pro}\, t.\langle\!\langle \vec{m}{:}\vec{\rho}, n{:}\tau \rangle\!\rangle \vdash e_2 : [u/t](t {\rightarrow} \tau) \qquad n \notin \{\vec{m}\}\end{array}}{\Gamma \vdash \langle e_1 \longleftarrow\!\!+\, n{=}e_2 \rangle : \mathtt{pro}\, t.\langle\!\langle \vec{m}{:}\vec{\rho}, n{:}\tau \rangle\!\rangle} \quad (\mathit{ext}).$$

A method addition on e_1 requires e_1 to be an object of some pro-type not including the n method, to be added. The first constraint is expressed by the top judgement in the premises of the rule; the second is enforced by the side condition $n \notin \{\vec{m}\}$. Given this condition, an inductive reasoning on the formation of objects shows that the pro-type in the conclusion is well-formed, and hence that all the pro-types occurring in the rule also are well-formed. The remaining assumption is a typing for e_2, the body of the n method. There are two things to notice about this judgement. The first is the use of the type variable u, match-bounded in the context, to render the polymorphism requirement for method bodies: if this polymorphic type can be derived for e_2, then it is guaranteed that e_2 will behave consistently on any future extension of $\langle e_1 \longleftarrow\!\!+\, n{=}e_2 \rangle$. The second important aspect of this typing, that we inherit from [15], is that the method n has type τ in the pro-type of the extended object (in the conclusion of the rule); on the other hand, the body of n has type $t \to \tau$ (with u substituted for t) to conform with the self-application semantics of method invocation.

The typing rule for method overrides is defined similarly to the rule for method addition: the difference is that we need a structural rule like (*send*) to be able to carry out derivations where the (*over*) rule is applied with σ unknown (i.e., the type of *self*).

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \sigma \qquad \Gamma \vdash \sigma \mathrel{<\!\!\#} \texttt{pro}\, t.\langle\!\langle \vec{m:\rho}, n{:}\tau \rangle\!\rangle \\ \Gamma,\ u \mathrel{<\!\!\#} \texttt{pro}\, t.\langle\!\langle \vec{m:\rho}, n{:}\tau \rangle\!\rangle \vdash e_2 : [u/t](t{\to}\tau)\end{array}}{\Gamma \vdash \langle e_1 \leftarrow n{=}e_2 \rangle : \sigma}\ (\mathit{over}).$$

We conclude with the rule for typing a search expression:

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash \sigma \mathrel{<\!\!\#} \texttt{pro}\, t.\langle\!\langle n{:}\tau \rangle\!\rangle \quad \Gamma \vdash \varsigma \mathrel{<\!\!\#} \sigma}{\Gamma \vdash e \hookleftarrow n : [\varsigma/t](t{\to}\tau)}\ (\mathit{search}).$$

Once more we use a structural rule like (*send*) assuming a possibly unknown type (i.e., a type variable) for e. Typing a search requires, however, more generality than typing a method invocation because the search of a method encompasses a recursive inspection of the recipient object (i.e., of *self*). This explains the roles of the two types ς and σ in the above rule: given that $e \hookleftarrow n$ arises as a result of reducing a message send, ς is the type of the *self* object, to which the n message was sent and to which the body of n will be applied; on the other hand, σ is the type of e, the sub-object of *self* where the body of n method is eventually found while searching within *self*. This intuitive argument is formally justified in the proof of the Subject Reduction property, Theorem 10.

3.4 An Example of Type Derivations

We illustrate the use of the typing rules in typing derivations with an example borrowed from [15]. The example also shows that the type system captures the desired form of *MyType* specialization.

Consider first the following expression representing a point object with an $\mathtt{x}$ coordinate and a $\mathtt{move}$ method ($\langle \mathtt{x} = \lambda\mathtt{self}.3\rangle$ is short for $\langle\langle\rangle \longleftarrow\!\!+\ \mathtt{x} = \lambda\mathtt{self}.3\rangle$):

$$\mathtt{P} \stackrel{\triangle}{=} \langle\langle \mathtt{x} = \lambda\mathtt{self}.3\rangle \longleftarrow\!\!+\ \mathtt{move} = \lambda\mathtt{self}.\lambda\mathtt{dx}.\langle \mathtt{self} \leftarrow \mathtt{x} = (\lambda\mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx})\rangle\rangle.$$

The expected typing for this expression is $\mathtt{P} : \mathtt{pro}\, t.\langle\!\langle \mathtt{x} : int, \mathtt{move} : int \to t\rangle\!\rangle$. Table 3 below sketches a derivation for this typing from the judgement $\varepsilon \vdash \langle \mathtt{x} = \lambda\mathtt{self}.3\rangle : \mathtt{pro}\, t.\langle\!\langle \mathtt{x} : int\rangle\!\rangle$.

Contexts

$\Gamma_1 \;=\; u \lessdot\!\# \ \mathtt{pro}\, t.\langle\!\langle \mathtt{x} : int, \mathtt{move} : int \to t\rangle\!\rangle,\ \mathtt{self} : u,\ \mathtt{dx} : int$

$\Gamma_2 \;=\; \Gamma_1,\ v \lessdot\!\# \ \mathtt{pro}\, t.\langle\!\langle \mathtt{x} : int, \mathtt{move} : int \to t\rangle\!\rangle,\ \mathtt{s} : v$

Derivation

(i) $\Gamma_2 \vdash (\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} : int$
by (*send*) from $\Gamma_2 \vdash \mathtt{self} : u$ and $\Gamma_2 \vdash u \lessdot\!\# \ \mathtt{pro}\, t.\langle\!\langle \mathtt{x} : int\rangle\!\rangle$.

(ii) $\Gamma_2 - \mathtt{s} \vdash \lambda\mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} : v \to int$

(iii) $\Gamma_1 \vdash \langle \mathtt{self} \leftarrow \mathtt{x} = (\lambda\mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx})\rangle : u$
by (*over*) from $\Gamma_1 \vdash \mathtt{self} : u$ and $\Gamma_1 \vdash u \lessdot\!\# \ \mathtt{pro}\, t.\langle\!\langle \mathtt{x} : int, \mathtt{move} : int \to t\rangle\!\rangle$, and 2.

(iv) $\Gamma_1 - \mathtt{self} - \mathtt{dx} \vdash \lambda\mathtt{self}.\lambda\mathtt{dx}.\langle \mathtt{self} \leftarrow \mathtt{x} = (\lambda\mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx})\rangle : u \to int \to u$

(v) $\varepsilon \vdash \mathtt{P} : \mathtt{pro}\, t.\langle\!\langle \mathtt{x} : int, \mathtt{move} : int \to t\rangle\!\rangle$
by (*ext*) from $\varepsilon \vdash \langle \mathtt{x} = \lambda\mathtt{self}.3\rangle : \mathtt{pro}\, t.\langle\!\langle \mathtt{x} : int\rangle\!\rangle$ and 4.

Table 3. Derivation of $\varepsilon \vdash \mathtt{P} : \mathtt{pro}\, t.\langle\!\langle \mathtt{x} : int, \mathtt{move} : int \to t\rangle\!\rangle$.

If we now invoke the $\mathtt{move}$ method on $\mathtt{P}$, like in $\mathtt{P} \Leftarrow\ \mathtt{move}\, 2$, the result of the call is again an object of type $\mathtt{pro}\, t.\langle\!\langle \mathtt{x} : int, \mathtt{move} : int \to t\rangle\!\rangle$, that coincides with $\mathtt{P}$ except for the value of the $\mathtt{x}$ coordinate.

Consider then defining $\mathtt{CP}$ as a new point, obtained from $\mathtt{P}$ with the addition of a $\mathtt{color}$ method, namely: $\mathtt{CP} \stackrel{\triangle}{=} \langle \mathtt{P} \longleftarrow\!\!+\ \mathtt{color} = \lambda\mathtt{self}.\mathtt{blue}\rangle$. With a derivation similar to that for $\mathtt{P}$, we may prove that $\mathtt{CP}$ has type $\mathtt{pro}\, t.\langle\!\langle \mathtt{x} : int, \mathtt{move} : int \to t, \mathtt{color} : colors\rangle\!\rangle$. Here, although the variable t in the two types is unchanged, the meaning of this variable is changed by virtue of the self-referential nature of $\mathtt{pro}$-types. The invocation $\mathtt{CP} \Leftarrow\ \mathtt{move}\, 2$ exposes the difference: the type of this expression, namely $\mathtt{pro}\, t.\langle\!\langle \mathtt{x} : int, \mathtt{move} : int \to t, \mathtt{color} : colors\rangle\!\rangle$, shows that the type of $\mathtt{move}$ has been specialized as the method is inherited from $\mathtt{P}$ to $\mathtt{CP}$.

4 Soundness of the Type System

We continue the description of the type system with a proof of type soundness. We first show that types are preserved by the reduction process, i.e., that if e has type τ, and e reduces to e', then also e' has type τ. We then use this result to prove the absence of *stuck states* in the evaluation of well-typed expressions.

4.1 Auxiliary Lemmas

The first two lemmas are standard generation lemmas, both proved by induction on the derivation. The third lemma proves a useful structural property of well-formed types, while the last lemma shows that the following rule is admissible:

$$\frac{\Gamma_1 \vdash A \quad \Gamma_1, \Gamma_2 \vdash *}{\Gamma_1, \Gamma_2 \vdash A} \ (\textit{weakening}).$$

In the above rule, and in the rest of this section, we write $\Gamma \vdash A$ to denote any derivable judgement in the system. Throughout this section we also use the easily verified facts:

- $\Gamma \vdash *$ is derivable if so is $\Gamma \vdash A$,
- $\Gamma \vdash *$ is derivable if so is $\Gamma, \Gamma' \vdash *$.

Lemma 2 *If $\Gamma \vdash \sigma \lessdot\!\# \ \tau$ is derivable, then so are $\Gamma \vdash \sigma$ and $\Gamma \vdash \tau$.* □

Lemma 3 *If $\Gamma \vdash e : \tau$ is derivable, then so is $\Gamma \vdash \tau$.* □

Lemma 4 *If $\Gamma \vdash \tau$ is derivable, then $Var(\tau) \subseteq Dom(\Gamma)$.*

Proof. By induction on the derivation of $\Gamma \vdash \tau$. The case when the derivation ends up with (*type proj*) is obvious. The case of (*type arrow*) follows immediately by induction hypothesis. The case (*type pro*) also follows by the induction hypothesis noting that $Var(\mathtt{pro}\, t.\langle\!\langle m_1{:}\tau_1, \ldots, m_k{:}\tau_k \rangle\!\rangle) = (\bigcup_i Var(\tau_i)) \setminus \{t\}$. □

Lemma 5 *If $\Gamma_1 \vdash A$, and $\Gamma_1, \Gamma_2 \vdash *$ are both derivable, then so is $\Gamma_1, \Gamma_2 \vdash A$.*

Proof. We prove the following, more general, result:

> if $\Gamma_1, \Gamma_2 \vdash A$, and $\Gamma_1, a, \Gamma_2 \vdash *$ are derivable, then so is $\Gamma_1, a, \Gamma_2 \vdash A$, where a is either $x : \tau$ or $u \lessdot\!\# \ \sigma$.

As a consequence, we have that $\Gamma_1, a \vdash A$ is derivable whenever so are $\Gamma_1 \vdash A$, and $\Gamma_1, a \vdash *$. That (*weakening*) is admissible follows then by induction on the length of Γ_2.

The proof is in two parts. We first give a proof for judgements where A is τ: this is an easy induction on the derivation of $\Gamma_1, \Gamma_2 \vdash A$, using renaming of variable bound by `pro` in the (*type pro*) case.

Then we prove the case of judgements where A is either $\tau_1 \lessdot\!\# \, \tau_2$ or $e : \tau$. This also is by induction on the derivation of $\Gamma_1, \Gamma_2 \vdash A$. The only interesting cases are (i) when a is $u \lessdot\!\# \, \tau$ and $\Gamma_1, \Gamma_2 \vdash A$ is the conclusion of either one of (ext) or ($over$), and (ii) when a is $x : \tau_1$ and $\Gamma_1, \Gamma_2 \vdash A$ is the conclusion of (abs), with A of the form $\lambda x'.e : \tau_1 \to \tau_2$ (with x' possibly equal to x).

In the case of (i), the proof follows directly from the induction hypothesis, noting that the typeability of the method body in the premises of (ext) and ($over$) does not depend on the name of the type variable that is discharged by the rule.

In the case of (ii), the judgement in question must have been derived from a judgement of the form $\Gamma_1, \Gamma_2, x' : \tau_1 \vdash e : \tau_2$. Since this judgement is itself derivable, we have that also $\Gamma_1, \Gamma_2, y : \tau_1 \vdash [y/x']e : \tau_2$ is derivable, for any $y \neq x'$ that is not contained in Γ_1, Γ_2: this can be shown by induction on the derivation.
Then, $\Gamma_1, \Gamma_2, y : \tau_1 \vdash *$ is derivable, and hence so is $\Gamma_1, \Gamma_2 \vdash \tau_1$. From the first part of the proof, it now follows that $\Gamma_1, x : \sigma, \Gamma_2 \vdash \tau_1$ is derivable, and hence also $\Gamma_1, x : \sigma, \Gamma_2, y : \tau_1 \vdash *$ is derivable from $\Gamma_1, x : \sigma, \Gamma_2 \vdash *$ using (var) (and noting that we may always choose $y \neq x$). By the induction hypothesis $\Gamma_1, x : \sigma, \Gamma_2, y : \tau_1 \vdash [y/x']e : \tau_2$ is derivable. Finally, by a further application of (abs), we obtain a derivation of $\Gamma_1, x : \sigma, \Gamma_2 \vdash \lambda y.[y/x']e : \tau_1 \to \tau_2$, which is the judgement we wished to derive, as $\lambda y.[y/x']e$ is (considered) identical to $\lambda x'.e$. $\square$

4.2 Substitution Lemmas

We now prove two substitution lemmas for term variables and type variables: Lemma 6 proves the desired property for term variables; Lemma 9 shows that if we derive a type for an expression, and this type contains a match-bound type variable, then a corresponding type can be proved by substituting the type variable with an appropriate type expression. The proof of Lemma 9 requires Lemma 7 and Lemma 8 to show an analogous property on all the other typing judgements of the system. Lemma 7 shows the desired property for matching and type judgements under the additional hypothesis that the judgement $\Gamma_1, [\varsigma/u]\Gamma_2 \vdash *$ is derivable. Lemma 8 shows that this property is unnecessary.

Lemma 6 *If the judgements $\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau$ and $\Gamma_1 \vdash e_2 : \sigma$ are both derivable, then so is the judgement $\Gamma_1, \Gamma_2 \vdash [e_2/x]e_1 : \tau$.*

Proof. The proof is by induction on the derivation of $\Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau$; renaming of bound variables is required in the substitution $[e_2/x]$ to avoid variable capture. $\square$

Lemma 7 *If the judgement $\Gamma_1, u \lessdot\!\# \ \sigma, \Gamma_2 \vdash A$, the judgement $\Gamma_1 \vdash \varsigma \lessdot\!\# \ \sigma$, and the judgement $\Gamma_1, [\varsigma/u]\Gamma_2 \vdash *$ are all derivable, then so is the judgement $\Gamma_1, [\varsigma/u]\Gamma_2 \vdash [\varsigma/u]A$, where A is either τ or $\tau_1 \lessdot\!\# \ \tau_2$.*

Proof. The proof is by induction on the derivation of $\Gamma_1, u \lessdot\!\# \ \sigma, \Gamma_2 \vdash A$.

- (*type proj*). In this case A is τ and we distinguish two subcases, depending on whether the substituted variable is the projected one or not. In the first case $\tau = u$, and $\Gamma_1, [\varsigma/u]\Gamma_2 \vdash \varsigma$ derives from $\Gamma_1 \vdash \varsigma$ and $\Gamma_1, [\varsigma/u]\Gamma_2 \vdash *$ by (*weakening*). That $\Gamma_1 \vdash \varsigma$ is derivable follows by Lemma 2 from the hypothesis that $\Gamma_1 \vdash \varsigma \lessdot\!\# \ \sigma$ is derivable. In the second case, $\tau = v$ for some $v \neq u$, and the proof follows by substitution (vacuous on v).
- (*type arrow*), (*type pro*). By induction hypothesis, using renaming of variables bound by `pro` in the (*type pro*) case.
- ($\lessdot\!\#$ *proj*). We distinguish two subcases, as for (*type proj*). If $u \lessdot\!\# \ \sigma$ is the projected bound, then the judgement in question is $\Gamma_1, u \lessdot\!\# \ \sigma, \Gamma_2 \vdash u \lessdot\!\# \ \sigma$. From the hypothesis that $\Gamma_1 \vdash \varsigma \lessdot\!\# \ \sigma$ is derivable, by Lemma 4, it follows that $u \notin Var(\sigma)$; then the desired judgement $\Gamma_1, [\varsigma/u]\Gamma_2 \vdash [\varsigma/u]u \lessdot\!\# \ \sigma$ is just $\Gamma_1, [\varsigma/u]\Gamma_2 \vdash \varsigma \lessdot\!\# \ \sigma$, which is derivable by (*weakening*) from the hypothesis that $\Gamma_1 \vdash \varsigma \lessdot\!\# \ \sigma$ and $\Gamma_1, [\varsigma/u]\Gamma_2 \vdash *$ are derivable. If $u \lessdot\!\# \ \sigma$ is not the projected bound we further distinguish two subcases, according to the possible shapes of the judgement.
  - If it is of the form $\Gamma_1, u \lessdot\!\# \ \sigma, \Gamma_2', v \lessdot\!\# \ \tau_2, \Gamma_2'' \vdash v \lessdot\!\# \ \tau_2$, the proof follows by ($\lessdot\!\#$ *proj*) and substitution.
  - Assume then that the judgement is $\Gamma_1', v \lessdot\!\# \ \tau_2, \Gamma_1'', u \lessdot\!\# \ \sigma, \Gamma_2 \vdash v \lessdot\!\# \ \tau_2$. Observing that the context of this judgement is well-formed, it follows that $\Gamma_1', v \lessdot\!\# \ \tau_2 \vdash *$ is derivable. Then also $\Gamma_1' \vdash \tau_2$ is derivable, and this, by Lemma 4, implies that $u \notin Var(\tau_2)$ (for $u \notin Dom(\Gamma_1')$, since $\Gamma_1', v \lessdot\!\# \ \tau_2, \Gamma_1'', u \lessdot\!\# \ \sigma, \Gamma_2 \vdash *$ is derivable). Then the proof follows by substitution (vacuous on τ_2).
- ($\lessdot\!\#$ *refl*). In this case, the judgement in question is $\Gamma_1, u \lessdot\!\# \ \sigma, \Gamma_2 \vdash \tau \lessdot\!\# \ \tau$, and $\Gamma_1, u \lessdot\!\# \ \sigma, \Gamma_2 \vdash \tau$ is derivable. By the induction hypothesis $\Gamma_1, [\varsigma/u]\Gamma_2 \vdash [\varsigma/u]\tau$ is derivable and the proof follows by ($\lessdot\!\#$ *refl*).

· ($\lll\#$ *trans*) and ($\lll\#$ *pro*) follow by the induction hypothesis, using renaming of bound variables for ($\lll\#$ *pro*) case. $\square$

Lemma 8 *If the judgements* $\Gamma_1, u \lll\# \sigma, \Gamma_2 \vdash *$ *and* $\Gamma_1 \vdash \varsigma \lll\# \sigma$ *are both derivable, then so is the judgement* $\Gamma_1, [\varsigma/u]\Gamma_2 \vdash *$*.*

Proof. By induction on the length of Γ_2. The basis of induction, when $\Gamma_2 = \varepsilon$, is immediate. There are two inductive cases, when $\Gamma_2 = \Gamma_2', x : \tau$ and when $\Gamma_2 = \Gamma_2', v \lll\# \tau$. In the first case, since $\Gamma_1, u \lll\# \sigma, \Gamma_2', x : \tau \vdash *$ is derivable, then $x \notin Dom(\Gamma_1, u \lll\# \sigma, \Gamma_2')$, and both the judgements $\Gamma_1, u \lll\# \sigma, \Gamma_2' \vdash \tau$ and $\Gamma_1, u \lll\# \sigma, \Gamma_2' \vdash *$ are derivable. By induction on the latter, it follows that $\Gamma_1, [\varsigma/u]\Gamma_2' \vdash *$ is derivable, and hence, by Lemma 7, also $\Gamma_1, [\varsigma/u]\Gamma_2' \vdash [\varsigma/t]\tau$ is derivable. Since $x \notin Dom(\Gamma_1, [\varsigma/u]\Gamma_2')$, the proof follows by (*var*). The second case is proven in an analogous way. $\square$

Lemma 9 *If the judgements* $\Gamma_1, u \lll\# \sigma, \Gamma_2 \vdash e : \tau$ *and* $\Gamma_1 \vdash \varsigma \lll\# \sigma$ *are both derivable, then so is the judgement* $\Gamma_1, [\varsigma/u]\Gamma_2 \vdash e : [\varsigma/u]\tau$*.*

Proof. The proof uses the following property of substitutions, which follows from Lemma 7 and Lemma 8 using the fact that $\Gamma \vdash *$ is derivable if so is $\Gamma \vdash A$:

(i) If $\Gamma_1, u \lll\# \sigma, \Gamma_2 \vdash A$ and $\Gamma_1 \vdash \varsigma \lll\# \sigma$ are both derivable, then so is the judgement $\Gamma_1, [\varsigma/u]\Gamma_2 \vdash [\varsigma/u]A$, where A is either $\tau_1 \lll\# \tau_2$ or τ.

The proof of the lemma is by induction on the derivation of the judgement $\Gamma_1, u \lll\# \sigma, \Gamma_2 \vdash e : \tau$.

Base cases. If $\Gamma_1, u \lll\# \sigma, \Gamma_2 \vdash e : \tau$ is derived by (*projection*), then either $e : \tau \in \Gamma_2$, or $e : \tau \in \Gamma_1$. If $e : \tau \in \Gamma_2$ we reason as follows. By Lemma 3, the judgement $\Gamma_1, u \lll\# \sigma, \Gamma_2 \vdash \tau$ is derivable; from this, by (*i*), it follows that $\Gamma_1, [\varsigma/u]\Gamma_2 \vdash [\varsigma/u]\tau$ is derivable, and the lemma follows by substitution (given that $e : [\varsigma/u]\tau \in [\varsigma/u]\Gamma_2$). If instead $e : \tau \in \Gamma_1$, then $\Gamma_1 \vdash e : \tau$ is derivable and then, by Lemma 3, $\Gamma_1 \vdash \tau$ is derivable. Now, by Lemma 4, it follows that $u \notin Var(\tau)$, which implies that $[\varsigma/u]\tau = \tau$. Then $\Gamma_1 \vdash e : [\varsigma/u]\tau$ is derivable, and the proof follows by (*weakening*), using Lemma 8. The case of (*empty*) follows immediately by Lemma 8.

Induction cases. The cases of (*abs*) and (*app*) follow immediately from the induction hypothesis. The object-related cases are worked out below.

In the (*ext*) case, the judgement in question is $\Gamma_1, u \lessdot\!\!\# \ \sigma, \Gamma_2 \vdash \langle e_1 \longleftarrow\!\!+\ n{=}e_2\rangle :$ $\mathtt{pro}\, t.\langle\!\langle \vec{m}{:}\vec{\rho}, n{:}\tau\rangle\!\rangle$, and both the judgements in the premise of the rule, namely,

$$\Gamma_1, u \lessdot\!\!\# \ \sigma, \Gamma_2 \vdash e_1 : \mathtt{pro}\, t.\langle\!\langle \vec{m}{:}\vec{\rho}\rangle\!\rangle$$

and

$$\Gamma_1, u \lessdot\!\!\# \ \sigma, \Gamma_2, v \lessdot\!\!\# \ \mathtt{pro}\, t.\langle\!\langle \vec{m}{:}\vec{\rho}, n{:}\tau\rangle\!\rangle \vdash e_2 : [v/t](t {\to} \tau)$$

are derivable. This also implies that $v \neq u$, for otherwise the context of the last judgement above would not be valid. Then, by the induction hypothesis, one has that both

$$\Gamma_1, [\varsigma/u]\Gamma_2 \vdash e_1 : \mathtt{pro}\, t.\langle\!\langle \vec{m}{:}[\varsigma/u]\vec{\rho}\rangle\!\rangle$$

and

$$\Gamma_1, [\varsigma/u]\Gamma_2, v \lessdot\!\!\# \ \mathtt{pro}\, t.\langle\!\langle \vec{m}{:}[\varsigma/u]\vec{\rho}, n{:}[\varsigma/u]\tau\rangle\!\rangle \vdash e_2 : v {\to} [v/t][\varsigma/u]\tau$$

are derivable, and the desired judgement is derivable by (*ext*). The case of (*over*) is similar.

In the (*send*) case, the judgement has the form $\Gamma_1, u \lessdot\!\!\# \ \sigma, \Gamma_2 \vdash e \Leftarrow n : [\rho/t]\tau$, and both $\Gamma_1, u \lessdot\!\!\# \ \sigma, \Gamma_2 \vdash e : \rho$, and $\Gamma_1, u \lessdot\!\!\# \ \sigma, \Gamma_2 \vdash \rho \lessdot\!\!\# \ \mathtt{pro}\, t.\langle\!\langle n{:}\tau\rangle\!\rangle$ are derivable judgements. Now, using induction on the first judgement it follows that $\Gamma_1, [\varsigma/u]\Gamma_2 \vdash e : [\varsigma/u]\rho$ is derivable; from the second judgement and from (*i*) we then have that $\Gamma_1, [\varsigma/u]\Gamma_2 \vdash [\varsigma/u]\rho \lessdot\!\!\# \ \mathtt{pro}\, t.\langle\!\langle n{:}[\varsigma/u]\tau\rangle\!\rangle$ is derivable. Then, by (*send*), we have a derivation of

$$\Gamma_1, [\varsigma/u]\Gamma_2 \vdash e : [[\varsigma/u]\rho/t]([\varsigma/u]\tau),$$

which is the desired judgement, being $[[\varsigma/u]\rho/t]([\varsigma/u]\tau) = [\varsigma/u]([\rho/t]\tau)$. The case of (*search*) is similar. □

4.3 Subject Reduction

Theorem 10 (Subject Reduction for $\xrightarrow{eval^*}$) *If $e \xrightarrow{eval^*} e'$ and the judgement $\Gamma \vdash e : \tau$ is derivable, then so is the judgement $\Gamma \vdash e' : \tau$.*

Proof. We give the proof for the one-step evaluation relation $\xrightarrow{eval}$. The theorem follows then by induction on the number of steps. For the one-step case, we first observe that derivations depend only on the form of types, not on the form of expressions. More precisely, if $\Gamma \vdash C[e] : \tau$ is derived from $\Gamma' \vdash e : \sigma$, and $\Gamma' \vdash e' : \sigma$ is also derivable, then so is $\Gamma \vdash C[e'] : \tau$. This fact is easily proved by an inspection of the type rules. To prove the theorem, it is then enough to show that the top-level reduction relation preserves types. This can be done by a case analysis on the definition of $\succ$.

(β). This case is standard: the redex is of the form $(\lambda x.e_1)\, e_2$ and its typing judgement is derived as follows:

$$\dfrac{\dfrac{\Gamma, x : \tau_1 \vdash e_1 : \tau_2}{\Gamma \vdash \lambda x.e_1 : \tau_1 \to \tau_2}\ (abs) \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (\lambda x.e_1)\, e_2 : \tau_2}\ (app).$$

Then the typing of the contractum is derived from the premise of (abs) and Lemma 6.

($\Leftarrow$). If the redex is of the form $e \Leftarrow n$, then its typing judgement is derived by an instance of $(send)$ of the form:

$$\dfrac{\Gamma \vdash e : \sigma \quad \Gamma \vdash \sigma \lessdot\!\# \ \mathtt{pro}\, t.\langle\!\langle n{:}\tau \rangle\!\rangle}{\Gamma \vdash e \Leftarrow n : [\sigma/t]\tau}\ (send),$$

and the two judgements in the premises are derivable. In this case, the contractum is the expression $(e \hookleftarrow n)e$, and a typing for this expression can be derived as follows. First observe that since $\Gamma \vdash \sigma \lessdot\!\# \ \mathtt{pro}\, t.\langle\!\langle n{:}\tau \rangle\!\rangle$ is derivable, σ is either a type variable or a pro-type: this can be verified by an inspection of the type rules. Then, using the premises of $(send)$, the judgement $\Gamma \vdash e \hookleftarrow n : \sigma {\to} [\sigma/t]\tau$ is derivable by $(search)$, using the fact that $\lessdot\!\#$ is reflexive, on type variables and pro-types. Then, $\Gamma \vdash (e \hookleftarrow n)e : [\sigma/t]\tau$ derives by (app).

($\hookleftarrow$ *succ*). We distinguish the two cases that arise from the possible forms of $\longleftarrow\!\circ$. If $\longleftarrow\!\circ$ is $\longleftarrow\!+$, then the typing of the redex is derived as follows:

$$\dfrac{\dfrac{\Xi}{\Gamma \vdash \langle e_1 \longleftarrow\!+\, n{=}e_2 \rangle : \sigma} \quad \Gamma \vdash \sigma \lessdot\!\# \ \mathtt{pro}\, t.\langle\!\langle n{:}\tau \rangle\!\rangle \quad \Gamma \vdash \varsigma \lessdot\!\# \ \sigma}{\Gamma \vdash \langle e_1 \longleftarrow\!+\, n{=}e_2 \rangle \hookleftarrow n : [\varsigma/t](t \to \tau)}\ (search).$$

By an inspection of the typing rules, it follows that $\sigma = \mathtt{pro}\, t.\langle\!\langle \vec{m{:}\rho}, n{:}\tau \rangle\!\rangle$ for some (possibly empty) row $\langle\!\langle \vec{m{:}\rho} \rangle\!\rangle$, and the last rule of Ξ is an instance of (ext) of the following form (for $n \notin \{\vec{m}\}$):

$$\dfrac{\Gamma \vdash e_1 : \mathtt{pro}\, t.\langle\!\langle \vec{m{:}\rho} \rangle\!\rangle \quad \Gamma, u \lessdot\!\# \ \mathtt{pro}\, t.\langle\!\langle \vec{m{:}\rho}, n{:}\tau \rangle\!\rangle \vdash e_2 : [u/t](t \to \tau)}{\Gamma \vdash \langle e_1 \longleftarrow\!+\, n{=}e_2 \rangle : \mathtt{pro}\, t.\langle\!\langle \vec{m{:}\rho}, n{:}\tau \rangle\!\rangle}\ (ext).$$

Now, given that $\sigma = \mathtt{pro}\, t.\langle\!\langle \vec{m{:}\rho}, n{:}\tau \rangle\!\rangle$ and that $\Gamma \vdash \varsigma \lessdot\!\# \ \sigma$ is derivable, the derivable judgement $\Gamma \vdash \varsigma \lessdot\!\# \ \sigma$ in the premises of $(search)$ is just the judgement:

$$\Gamma \vdash \varsigma \lessdot\!\# \ \mathtt{pro}\, t.\langle\!\langle \vec{m{:}\rho}, n{:}\tau \rangle\!\rangle$$

Then, applying Lemma 9 to the right premise of (*ext*) and to the above judgement, we have a derivation for the judgement $\Gamma \vdash e_2 : [\varsigma/t](t \to \tau)$. This concludes this case, since e_2 is the contractum.

For the case when $\leftarrow\!\circ$ is $\leftarrow$, the proof is similar. The typing judgement for the redex is derived as follows:

$$\dfrac{\dfrac{\Xi}{\Gamma \vdash \langle e_1 \leftarrow n{=}e_2 \rangle : \sigma} \quad \Gamma \vdash \sigma \lessdot\!\# \; \texttt{pro}\, t.\langle\!\langle n{:}\tau \rangle\!\rangle \quad \Gamma \vdash \varsigma \lessdot\!\# \; \sigma}{\Gamma \vdash \langle e_1 \leftarrow n{=}e_2 \rangle \hookleftarrow n : [\varsigma/t](t \to \tau)} \;(\textit{search}),$$

where Ξ, in turn, ends up with a rule of the form:

$$\dfrac{\begin{array}{c}\Gamma \vdash e_1 : \sigma \qquad \Gamma \vdash \sigma \lessdot\!\# \; \texttt{pro}\, t.\langle\!\langle \overrightarrow{m{:}\rho}, n{:}\tau \rangle\!\rangle \\ \Gamma, u \lessdot\!\# \; \texttt{pro}\, t.\langle\!\langle \overrightarrow{m{:}\rho}, n{:}\tau \rangle\!\rangle \vdash e_2 : [u/t](t \to \tau)\end{array}}{\Gamma \vdash \langle e_1 \leftarrow n{=}e_2 \rangle : \sigma} \;(\textit{over}).$$

Now, from $\Gamma \vdash \sigma \lessdot\!\# \; \texttt{pro}\, t.\langle\!\langle \overrightarrow{m{:}\rho}, n{:}\tau \rangle\!\rangle$, and $\Gamma \vdash \varsigma \lessdot\!\# \; \sigma$, we have that

$$\Gamma \vdash \varsigma \lessdot\!\# \; \texttt{pro}\, t.\langle\!\langle \overrightarrow{m{:}\rho}, n{:}\tau \rangle\!\rangle$$

is derivable, since $\lessdot\!\#$ is transitive. Then, to obtain the expected type for the contractum, apply Lemma 9 to the last judgement above and to the judgement for e_2 in the premises of (*over*).

($\hookleftarrow$ *next*). Again, we distinguish the two cases that arise from the possible forms of $\leftarrow\!\circ$. If $\leftarrow\!\circ$ is $\leftarrow\!\!+$, then the typing judgement for the redex is derived by an instance of (*search*) as follows:

$$\dfrac{\dfrac{\Xi}{\Gamma \vdash \langle e_1 \leftarrow\!\!+ m{=}e_2 \rangle : \sigma} \quad \Gamma \vdash \sigma \lessdot\!\# \; \texttt{pro}\, t.\langle\!\langle n{:}\tau \rangle\!\rangle \quad \Gamma \vdash \varsigma \lessdot\!\# \; \sigma}{\Gamma \vdash \langle e_1 \leftarrow\!\!+ m{=}e_2 \rangle \hookleftarrow n : [\varsigma/t](t \to \tau)} \;(\textit{search}).$$

Again, by an inspection of the typing rules, it follows that

$$\sigma = \texttt{pro}\, t.\langle\!\langle \overrightarrow{p{:}\eta}, m{:}\rho \rangle\!\rangle$$

for some (possibly empty) row $\langle\!\langle \overrightarrow{p{:}\eta} \rangle\!\rangle$, and some type ρ. From this, it follows that the last rule of Ξ has the form:

$$\dfrac{\Gamma \vdash e_1 : \texttt{pro}\, t.\langle\!\langle \overrightarrow{p{:}\eta} \rangle\!\rangle \quad \Gamma, u \lessdot\!\# \; \texttt{pro}\, t.\langle\!\langle \overrightarrow{p{:}\eta}, m{:}\rho \rangle\!\rangle \vdash e_2 : [u/t](t \to \tau)}{\Gamma \vdash \langle e_1 \leftarrow\!\!+ m{=}e_2 \rangle : \texttt{pro}\, t.\langle\!\langle \overrightarrow{p{:}\eta}, m{:}\rho \rangle\!\rangle} \;(\textit{ext}).$$

From $\Gamma \vdash \sigma \lessdot\!\!\# \; \texttt{pro}\, t.\langle\!\langle n{:}\tau \rangle\!\rangle$, it follows that the n method must be one of the $\vec{p}$ methods in the type of e_1, and hence the following judgement is derivable:

$$\Gamma \vdash \texttt{pro}\, t.\langle\!\langle \vec{p{:}\eta} \rangle\!\rangle \lessdot\!\!\# \; \texttt{pro}\, t.\langle\!\langle n{:}\tau \rangle\!\rangle.$$

Then, letting $\sigma' = \texttt{pro}\, t.\langle\!\langle \vec{p{:}\eta} \rangle\!\rangle$, we derive the desired type for the contractum $e_1 \hookleftarrow n$ as follows:

$$\frac{\Gamma \vdash e_1 : \sigma' \quad \Gamma \vdash \sigma' \lessdot\!\!\# \; \texttt{pro}\, t.\langle\!\langle n{:}\tau \rangle\!\rangle \quad \Gamma \vdash \varsigma \lessdot\!\!\# \; \sigma'}{\Gamma \vdash e_1 \hookleftarrow n : [\varsigma/t](t \to \tau)} \;(\mathit{search}),$$

where $\Gamma \vdash \varsigma \lessdot\!\!\# \; \sigma'$ derives by transitivity from $\Gamma \vdash \varsigma \lessdot\!\!\# \; \sigma$ and from

$$\Gamma \vdash \sigma = \texttt{pro}\, t.\langle\!\langle \vec{p{:}\eta}, m{:}\rho \rangle\!\rangle \lessdot\!\!\# \; \texttt{pro}\, t.\langle\!\langle \vec{p{:}\eta} \rangle\!\rangle = \sigma'$$

The case when $\leftarrow\!\!\circ$ is $\leftarrow$ is simpler, since the derivation for the redex has the form:

$$\frac{\Gamma \vdash \langle e_1 \leftarrow m{=}e_2 \rangle : \sigma \quad \Gamma \vdash \sigma \lessdot\!\!\# \; \texttt{pro}\, t.\langle\!\langle n{:}\tau \rangle\!\rangle \quad \Gamma \vdash \varsigma \lessdot\!\!\# \; \sigma}{\Gamma \vdash \langle e_1 \leftarrow m{=}e_2 \rangle \hookleftarrow n : [\varsigma/t](t \to \tau)} \;(\mathit{search}).$$

Then $\Gamma \vdash \langle e_1 \leftarrow m{=}e_2 \rangle : \sigma$ is derived by (over), which implies that $\Gamma \vdash e_1 : \sigma$ also is derivable. Then, the desired judgement for the contractum is derived by the following instance of (search):

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma \vdash \sigma \lessdot\!\!\# \; \texttt{pro}\, t.\langle\!\langle n{:}\tau \rangle\!\rangle \quad \Gamma \vdash \varsigma \lessdot\!\!\# \; \sigma}{\Gamma \vdash e_1 \hookleftarrow n : [\varsigma/t](t \to \tau)} \;(\mathit{search}).$$

□

4.4 Absence of Stuck States

The reduction rules for the operational semantics given in Table 2 (Section 2) can be used (almost) directly as the definition of an interpreter for the untyped calculus.

Run-time errors for this interpreter correspond to pattern-matching failures (i.e., stuck states) when using the rules to evaluate a closed expression. An inspection of the rules shows that there are only three ways in which evaluation may get stuck; (i) when evaluating a send expression $e \Leftarrow m$, and evaluating e does not yield an *obj* expression of the form $\langle e_1 \leftarrow\!\!\circ\; n{=}e_2 \rangle$; (ii) when evaluating an application $e_1\, e_2$, and the evaluation of e_1 does not return a λ-abstraction; (iii) when searching an m $(m \neq n)$ method within an object $\langle e_1 \leftarrow\!\!\circ\; n{=}e_2 \rangle$, and evaluating e_1 does not yield an object in the same form. Stuck states like *(ii)* correspond to the standard run-time errors of an interpreter of the λ-calculus. Instead, stuck states like *(i)* and *(iii)* correspond to run-time errors of the sort *message-not-understood* arising in object-oriented languages as a consequence of sending a message to an object that does not have a corresponding method.

The following theorem proves the absence of such errors in the evaluation of a well-typed closed expression: type soundness follows from this result.

Theorem 11 (Absence of stuck states) *Let e be a closed expression such that the judgement $\varepsilon \vdash e : \tau$ is derivable for some type τ. Then:*

(i) *if $e = e_1\, e_2$ and $e_1 \Downarrow val$, then $val = \lambda x.e'$ for some x and e';*
(ii) *if $e = e_1 \Leftarrow m$ and $e_1 \Downarrow val$, then $val = obj$ for some object expression obj;*
(iii) *if $e = \langle e_1 \hookleftarrow\!\!\circ\, n{=}e_2 \rangle \hookleftarrow m$, with $m \neq n$, and $e_1 \Downarrow val$, then $val = obj$ for some object expression obj.*

Proof. The proof uses a subject reduction property for $\Downarrow$ and $\rightsquigarrow$. For $\Downarrow$, the property states that if $\varepsilon \vdash e : \tau$ is derivable, and $e \Downarrow val$, then $\varepsilon \vdash val : \tau$ is derivable. This follows by Theorem 10, since $e \xrightarrow{eval^*} val$ when $e \Downarrow val$ (cf. Proposition 1). A corresponding property holds for $\rightsquigarrow$.

The proof of (i) is standard. For (ii), if $\varepsilon \vdash e : \tau$, is derivable, then $\tau = \texttt{pro}\, t.\langle\!\langle R \mid m{:}\sigma \rangle\!\rangle$ for some possibly empty row R (we may safely assume that τ is not a variable, since the fact that $\varepsilon \vdash e : \tau$ is derivable implies that also $\varepsilon \vdash \tau$ is derivable). Then, $\varepsilon \vdash val : \texttt{pro}\, t.\langle\!\langle R \mid m{:}\sigma \rangle\!\rangle$ and that $val = obj$ follows by observing that no other values (i.e. constants, the empty object, λ-abstractions) have this type. The proof of (iii) follows in exactly the same way. □

5 Relationship with the *Fisher-Honsell-Mitchell* Type System

As we mentioned already, the fundamental difference between the system of this paper and the original system of [15] is in the rendering of method polymorphism. In [15] the polymorphism of method bodies is modeled using pro-types formed around extensible rows, or *row schemes*. A typical example of such types is $\texttt{pro}\, t.\langle\!\langle r\sigma \mid \vec{m{:}\tau} \rangle\!\rangle$: the polymorphic part of this type is the sub-row $r\sigma$, where r is a row variable and σ a type, and the application $r\sigma$ constitutes the extensible part of the row. Applications of rows to types of the sort just described are formalized in [15] with a calculus of rows that includes higher-order rows, and two operations of, respectively, type-abstraction on rows, and application of rows to types. Specifically, the calculus of rows requires a rule of β-reduction together with the associated equational theory, as well as a kind assignment system for axiomatizing the notion of well-formedness for types. Briefly, kinds of rows have the general form $T^n \rightarrow [m_1, \ldots, m_k]$ where $n \geq 0$, and T is the kind of types. When $n = 0$, these kinds degenerate to $[m_1, \ldots, m_k]$, and denote rows that do not contain any of the m_i methods mentioned in the kind. This form of negative information is critical to ensure the well-formedness of types formed around row schemes: a type like $\mathtt{pro}\, t.\langle\!\langle r\sigma \mid \vec{m{:}\tau} \rangle\!\rangle$ is well formed in a context Γ only if r has kind $T \rightarrow [\vec{m}]$ in this context, thus ensuring that $r\sigma$ has kind $[\vec{m}]$ for any type $\sigma : T$. Any row $R : [\vec{m}]$ can then be used to instantiate the row scheme, yielding the type $\mathtt{pro}\, t.\langle\!\langle R \mid \vec{m{:}\tau} \rangle\!\rangle$.

Having given this informal description, the relationship between our system and the system of [15] is best illustrated by looking at the derivation of Section 3 (Table 3), and contrasting it with the derivation of the corresponding judgement in [15], reported in Table 4 below[4]. To avoid ambiguities, throughout this section we use the symbol $\vdash\!\!\div$ for judgements from [15], and the symbol $\vdash$ for judgements in our system.

Contexts

$\Gamma_1 \;=\; r{:}T \rightarrow [\mathtt{x}, \mathtt{move}],\ \mathtt{self}{:}\mathtt{pro}\, t.\langle\!\langle rt \mid \mathtt{x}{:}int, \mathtt{move}{:}int \rightarrow t \rangle\!\rangle,\ \mathtt{dx} : int$

$\Gamma_2 \;=\; \Gamma_1,\ r'{:}T \rightarrow [\mathtt{x}, \mathtt{move}], \mathtt{s}{:}\mathtt{pro}\, t.\langle\!\langle r't \mid \mathtt{x}{:}int, \mathtt{move}{:}int \rightarrow t \rangle\!\rangle$

Derivation

(i) $\Gamma_2 \vdash\!\!\div (\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} : int$

(ii) $\Gamma_2 - \mathtt{s} \vdash\!\!\div \lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} : \mathtt{pro}\, t.\langle\!\langle r't \mid \mathtt{x}{:}int, \mathtt{move}{:}int \rightarrow t \rangle\!\rangle \rightarrow int$

(iii) $\Gamma_1 \vdash\!\!\div \langle \mathtt{self} \longleftarrow\!\!\!+\ \mathtt{x} = (\lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx})\rangle\rangle : \mathtt{pro}\, t.\langle\!\langle rt \mid \mathtt{x}{:}int, \mathtt{move}{:}int \rightarrow t \rangle\!\rangle$

(iv) $\Gamma_1 - \mathtt{dx} - \mathtt{self} \vdash\!\!\div \lambda \mathtt{self}.\lambda \mathtt{dx}.\langle \mathtt{self} \longleftarrow\!\!\!+\ \mathtt{x} = (\lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx})\rangle$
$\qquad : [\mathtt{pro}\, t.\langle\!\langle rt \mid \mathtt{x}{:}int, \mathtt{move}{:}int \rightarrow t \rangle\!\rangle / t](t \rightarrow int \rightarrow t)$

(v) $\varepsilon \vdash\!\!\div \mathtt{P} : \mathtt{pro}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{move}{:}int \rightarrow t \rangle\!\rangle$

Table 4. Derivation of $\varepsilon \vdash\!\!\div \mathtt{P} : \mathtt{pro}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{move}{:}int \rightarrow t \rangle\!\rangle$ in [FHM94].

Although the two derivations have essentially the same structure, there is a fundamental difference in the polymorphic typing of the `move` method. In the original system, polymorphism is placed *inside* the row of the type $\mathtt{pro}\, t.\langle\!\langle rt \mid \mathtt{x}{:}int, \mathtt{move}{:}int \rightarrow t \rangle\!\rangle$: extensions of the `pro`-type may be obtained by substituting any well-kinded row expression for the row variable r. As an example, r could be substituted by, say, the expression $\lambda t_1.\langle\!\langle \mathtt{m}{:}t_1 \rangle\!\rangle$, yielding the new type $\mathtt{pro}\, t.\langle\!\langle \mathtt{m}{:}t, \mathtt{x}{:}int, \mathtt{move}{:}int \rightarrow t \rangle\!\rangle$ as the result of β-reducing the application $[\lambda t_1.\langle\!\langle \mathtt{m}{:}t_1 \rangle\!\rangle / r]rt$. Notice that the combination of abstraction and application is critical to capture the expected form of type specialization.

[4] In the example we have only polymorphic pro-types in which $\sigma = t$, but, in principle, any well-formed type can be applied to a row of kind $T \rightarrow [\vec{m}]$.

In our system, instead, we place polymorphism outside rows: the type of `move` is polymorphic in the type variable u, and extensions of this type may be obtained by substituting u with any well-formed type that matches the bound $\mathtt{pro}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{move}{:}int{\rightarrow}t \rangle\!\rangle$ for u. Placing polymorphism outside rows simplifies the mechanism of `pro`-type extension, in that it saves the need for abstraction and application used in the original system.

What remains to be discussed is whether the above correspondence between type derivations in the two systems carries over to the general case. In the rest of this section we give a proof of this result in the case of derivations of *first-order* judgements, i.e., of judgements whose context does not contain row variables. Restricting to first-order judgements is critical in the proof: the reason is rather technical and will be discussed in the next subsection. Below, we present an informal argument that should help motivate the restriction and give some intuition about the subsequent formal development.

As we already noticed, the correspondence between the typing judgements in the two systems is essentially a correspondence between the row schemes used in the polymorphic `pro`-types of [15] and our match-bound type variables. This intuition works correctly as long as the row schemes (and the corresponding `pro`-types) are used in a "disciplined", or "uniform" way in typing judgements and derivations from [15]; it breaks, instead, in other cases. Consider, for instance, the following judgement:

$$r : T{\rightarrow}[m],\, e : \mathtt{pro}\, t.\langle\!\langle rt \rangle\!\rangle \vdash\!\!\div \langle e \longleftarrow\!\!+\, m{=}\lambda s.s \rangle : \mathtt{pro}\, t.\langle\!\langle rt \mid m{:}t \rangle\!\rangle$$

While this judgement is derivable in [15], replacing the polymorphic `pro`-types with corresponding type variables fails to produce a derivable judgement in our system. There are several reasons for this, the most evident being that our (*ext*) rule requires a `pro`-type (not a variable) in the type of the object to be extended.

More generally, the correspondence between row schemes and match-bound type variables breaks whenever the same row variable occurs in different polymorphic `pro`-types in a judgement of a derivation. In the previous case, for example, we are left with two only possibilities: either use the same match-bound type variable to encode both the `pro`-types, or use different variables. Both choices are clearly incorrect: with the first we identify two types that are different; with the second we fail to render the fact that the two types share the "sub-row" rt.

Fortunately, however, it can be shown that uses of polymorphic `pro`-types can be made "uniform" in typing derivations for first-order judgements. Given this, it is then possible to define a translation from "uniform" derivations of [15] to corresponding derivations in our system, and to prove that the translation is "complete", in that the translation of a derivable judgement from [15] is derivable in our system.

The rest of this section is dedicated to the proof of these properties. We first formalize the definition of "uniform derivations", then we define the translation we just mentioned, and finally we give a proof that every first-order judgement from [15] that has a derivation has also a "uniform" derivation.

5.1 Uniform Derivations

We restrict attention to derivations in the system of [15] that satisfy the conditions stated in the following definition. Below, and in the sequel of the paper, we use the following terminology. We say that a judgement $\Gamma \vdash\!\div A$ from [15] is *first-order* if Γ does not contain any row-variable declaration. Following [15], we say that a type or row expression is in τnf (for *type normal form*) if they do not contain occurrences of β-redexes; similarly, a derivation is in normal form if it does not contain occurrences of rules for type and row equality (cf. Appendix B).

Definition 12 (Simple Derivations) *A derivation Ξ is* simple *if:*

- *Ξ is rooted at a first-order judgement;*
- *every instance of an (ext) and (over) rule of Ξ discharges a different row variable;*
- *Ξ is in normal form, and every type and row occurring in Ξ is in τnf.*

The first condition enforces the restriction we discussed above; the remaining two conditions, instead, are not further restrictive. The second condition can be stated equivalently by saying that there is a one-to-one correspondence between row variables and instances of (*ext/over*) in the derivation: this may always be enforced by a consistent renaming of the row variables of the derivation. The third condition requires types of a derivation to be in τnf. That this condition is not restrictive follows from observing that: (*i*) every expression that has a type has a type in τnf; (*ii*) any judgement $\Gamma \vdash\!\div A$ is derivable if and only if the corresponding normal-form judgement $\tau nf(\Gamma) \vdash\!\div \tau nf(A)$ has a normal-form derivation (cf. [15], Lemma 4.10); (*iii*) occurrences of type-equality rules may be eliminated from normal-form derivations because two row or type expressions related via β-reduction must have the same τnf.

The following proposition follows from the above observations.

Proposition 13 *If a first-order judgement is derivable, then it has a simple derivation.*

A few additional remarks are in order. Rows and types arising in simple derivations, as we have defined them, have a considerably simpler structure than rows and types arising in arbitrary derivations. Specifically:

· *Row variables arising in simple derivations have kinds* $T \to [m_1, \ldots, m_k]$.
  This follows from an inspection of the type rules of [15] (cf. Appendix B): the only rules that discharge row variables are (*ext*) and (*over*), and they discharge variables with the indicated kinds. Since all the row variables of a simple derivation must eventually be discharged, they must all have the indicated kinds.
· *In applications of rows to types of the form* $R\tau$*, the row* R *is a row variable.*
  This follows from the previous observation and the argument we discuss next. Consider an application of the form $R\tau$. Since rows and types of simple derivations are in normal form, they do not contain β-redexes. Then, in the above application, R cannot be a λ-abstraction, for otherwise $R\tau$ would be a β-redex. Then, it may only be a variable or an application of the form $R_1\tau_1$, for some row R_1 and type τ_1. Now, R_1 itself cannot be a λ-abstraction (because otherwise $R_1\tau_1$ would be a β-redex), and hence it can only be an application or a row variable. Reasoning in this way, we see that in the application $R\tau$ we only have two possibilities: either R is a row variable, or an application of the form $(\ldots(R_n\tau_n)\ldots)\tau_1$ for $n \geq 1$, and R_n is a row variable. But then the kind of R_n would be $T^{n+1} \to [m_1, \ldots, m_k]$, for $n > 0$, while we just observed that the row variables of simple derivations have kind $T \to [m_1, \ldots, m_k]$. Hence, R is indeed a row variable.

When restricting to simple derivations, it is therefore possible to simplify the syntax of types, rows and kinds from [15] as follows:

Types	$\tau ::= b \mid t \mid \tau \to \tau \mid \mathtt{pro}\, t.R$	
Rows	$R ::= \langle\!\langle\rangle\!\rangle \mid r\tau \mid \langle\!\langle R \mid m{:}\tau \rangle\!\rangle$	
Kinds	$::= T \mid \kappa \mid T \to \kappa$	$(\kappa = [m_1, \ldots, m_k])$

with r denoting row variables of kind $T \to \kappa$.

Definition 14 (Row-variable References) *Let* Ξ *be a simple derivation, and let* r *be a row variable used in* Ξ*. The* reference *for* r *(or* r-reference*) in* Ξ *is the type* $\mathtt{pro}\, t.\langle\!\langle rt \mid \vec{m{:}\tau} \rangle\!\rangle$ *that occurs at the instance of (*ext/over*) that discharges* r *in* Ξ*.*

Definition 15 (Uniform Derivations) *Let* Ξ *be a simple derivation.* Ξ *is* uniform *if for every row variable* r *used in* Ξ*, if* r *occurs within the type* $\mathtt{pro}\, t.\langle\!\langle rv \mid \vec{m{:}\tau} \rangle\!\rangle$*, then this type is an occurrence of the* r*-reference in* Ξ *(hence, in particular* $v = t$*).*

We now present a translation for uniform derivations from [15] and we show that the translation of a derivable first-order judgement is a derivable judgement in our system. While our main goal is to show that this property holds for uniform derivations of first-order term judgements of the form $\Gamma \mathrel{\dot{\vdash}} e : \tau$, the inductive reasoning used in the proofs requires more generality, because the derivation of term judgements may depend on subderivations of kinding judgements and, more generally, of judgements that are not first-order (for example, the typing judgements of method bodies).

As a consequence, definitions and results in this section are stated and proved for the more general case of types, contexts and judgements occurring in a given uniform derivation.

5.2.1 Translation of Types and Contexts

The translation of a type τ is the type τ^* that results from replacing every occurrence of a polymorphic `pro`-type in τ with a corresponding type variable. The correspondence between the polymorphic `pro`-types of τ and the type variables of τ^* may be established using any injective map from row variables to corresponding type variables: this is possible because in every uniform derivation there is a one-to-one correspondence between row variables and `pro`-types built around these variables (as row variables occur only within their references). Given a row variable r, we then denote with $\xi(r)$ the uniquely determined type variable associated with r.

The translation of types is defined compositionally over the types' components, with the exception of polymorphic `pro`-types: a type of the form $\texttt{pro}\, t.\langle\!\langle rt \mid \vec{m:\tau}\rangle\!\rangle$ is encoded by the type variable $\xi(r)$ associated with the row variable r.

Definition 16 (Translation of Types) *The translation of a type τ is the type τ^* defined as follows:*

- $\tau^* = \tau$, *for every type variable or base type τ;*
- $(\tau_1 \to \tau_2)^* = \tau_1^* \to \tau_2^*$
- $(\texttt{pro}\, t.\langle\!\langle m_1{:}\tau_1, \ldots, m_k{:}\tau_k\rangle\!\rangle)^* = \texttt{pro}\, t.\langle\!\langle m_1{:}\tau_1^*, \ldots, m_k{:}\tau_k^*\rangle\!\rangle$
- $(\texttt{pro}\, t.\langle\!\langle rt \mid m_1{:}\tau_1, \ldots, m_k{:}\tau_k\rangle\!\rangle)^* = \xi(r)$

From the definition, it follows that the translation of a "ground" type, i.e., one that does not contain any occurrence of row variables, leaves the type unchanged: in other words, τ^* is equal to τ, whenever τ is ground in the sense we just explained.

Definition 17 (Translation of Contexts) *Let Γ be a context of a uniform derivation Ξ, and let* $\texttt{pro}\, t.\langle\!\langle rt \mid \vec{m:\tau}\rangle\!\rangle$ *denote the reference for the row variable r of Ξ. The translation of Γ, denoted by Γ^*_Ξ, is defined as follows:*

- $\varepsilon^*_\Xi = \varepsilon$
- $(\Gamma, x : \tau)^*_\Xi = \Gamma^*_\Xi,\ x : \tau^*.$
- $(\Gamma, t : T)^*_\Xi = \Gamma^*_\Xi, t \lessdot\!\# \ \texttt{pro}\, t.\langle\!\langle\rangle\!\rangle;$
- $(\Gamma, r : \kappa)^*_\Xi = \Gamma^*_\Xi,\ \xi(r) \lessdot\!\# \ \texttt{pro}\, t.\langle\!\langle \vec{m:\tau^*}\rangle\!\rangle$

The first two cases should be self-explanatory. In the third case, $\texttt{pro}\, t.\langle\!\langle\rangle\!\rangle$ plays the same role as the type Top in presentations of type systems based on subtyping. For the last case, the translation is obtained by encoding the declaration of a row variable in [15] by a corresponding match-bound declaration for the type variable associated to the row variable via ξ; the choice of the bound is determined by the shape of the reference for the row variable in the derivation Ξ where the context occurs. The translation is well-defined since the choice of the r-reference is univocally determined in a uniform derivation. Finally, observe that if Γ is valid and occurs in a first-order judgement, the translation of Γ is simply Γ, for all types of Γ do not contain any row variable.

5.2.2 Translation of Judgements

The translation of judgements is also given with respect to the uniform derivation where they occur.

Definition 18 (Translation of Judgements) *Let $\Gamma \vdash\!\!\div A$ be a judgement of a uniform derivation Ξ, where A is either $e : \tau$ or $\tau : T$ for some expression e and type τ. The translation of this judgement is $\Gamma^*_\Xi \vdash A^*$, where $A^* = e : \tau^*$ if $A = e : \tau$, and $A^* = \tau^*$ if $A = \tau : T$.*

The translation is not defined for kinding judgements of the form $\Gamma \vdash\!\!\div R : \kappa$, as these judgements have no counterpart in our system. As for the previous definition, it is useful to single out a few properties of the translation for derivable first-order judgements: if $\Gamma \vdash\!\!\div A$ is one such judgement, and it is derivable, then its translation is simply the judgement $\Gamma \vdash A$ itself. To see this, note that since the judgement is first-order and derivable, then the context Γ must be first-order and valid, and hence, the types occurring in A must be ground.

5.2.3 Derivable Judgements are Translated into Derivable Judgements

The outline of the proof is as follows: given a uniform derivation Ξ, take a subderivation rooted at a judgement of Ξ, and prove that a corresponding subderivation for the translation of this judgement exists in our system.

The proof is given in two steps. Lemma 22 proves the desired property for kinding judgements of the form $\Gamma \vdash\!\!\div \tau : T$. Lemma 23 handles the case of term judgements.

The proof of Lemma 22, in turn, uses two preliminary results. Lemma 19 proves the desired property for kinding judgements under the additional assumption that a uniform derivation exists in our system for $\Gamma^*_\Xi \vdash *$. Lemma 21 uses Lemma 19, and the generation lemmas from [15] (stated in Proposition 20) to show that the additional assumption is not needed.

All the derivations and judgements mentioned in the following lemmas are intended to be subderivations, and judgements, of a given uniform derivation. Abusing the terminology, we will say that a subderivation of a uniform derivation is itself uniform. As we already observed, such derivations may not be simple (because they may be rooted at not first-order judgements), and hence, strictly speaking, not uniform. However, they do have the property that all the occurrences of polymorphic pro-types are occurrences of the reference for some row variable: this is what really matters in the proofs of this section.

Lemma 19 *Let $\Gamma \vdash\!\!\div \tau : T$ be a judgement occurring in a uniform derivation Ξ. If the judgement $\Gamma^*_\Xi \vdash *$ is derivable in our system, then also $\Gamma^*_\Xi \vdash \tau^*$ is derivable in our system.*

Proof. The proof is by induction on the subderivation of $\Gamma \vdash\!\!\div \tau : T$. The only non trivial case is when τ is a pro-type: we distinguish two subcases, depending on whether the pro-type is polymorphic or not. If it is not polymorphic, the judgement in question is $\Gamma \vdash\!\!\div \mathtt{pro}\, t.\langle\!\langle \vec{m:\tau} \rangle\!\rangle : T$, and its (uniform) subderivation in Ξ is of the form:

$$\dfrac{\begin{array}{c}\Gamma, t : T \vdash\!\!\div \tau_i : T \\ \vdots \;\; (row\ empty\ +\ row\ ext's) \\ \Gamma, t : T \vdash\!\!\div \langle\!\langle \vec{m:\tau} \rangle\!\rangle : \kappa\end{array}}{\Gamma \vdash\!\!\div \mathtt{pro}\, t.\langle\!\langle \vec{m:\tau} \rangle\!\rangle : T}\;(type\ pro).$$

Then, the translation of the judgement $\Gamma \vdash\!\!\div \mathtt{pro}\, t.\langle\!\langle \vec{m:\tau} \rangle\!\rangle : T$ can be derived, as follows:

$$\dfrac{\Gamma^*_\Xi, t \mathrel{<\!\#} \mathtt{pro}\, t.\langle\!\langle\rangle\!\rangle \vdash \tau_i^*}{\Gamma^*_\Xi \vdash \mathtt{pro}\, t.\langle\!\langle \vec{m:\tau^*} \rangle\!\rangle}\;(type\ pro),$$

since the premises of this rule are derivable by induction hypothesis. If instead the pro-type is polymorphic, the judgement may be written as follows:

$$\Gamma_1, r : T \rightarrow [\vec{m}], \Gamma_2 \vdash\!\!\div \mathtt{pro}\, t.\langle\!\langle rt \mid \vec{m:\tau} \rangle\!\rangle : T$$

Then, $\mathtt{pro}\, t.\langle\!\langle rt \mid \vec{m:\tau} \rangle\!\rangle$ is the r-reference in Ξ, and the translation of this judgement is

$$\Gamma_{1\Xi}^{*}, u \lessdot\!\!\# \ \mathtt{pro}\, t.\langle\!\langle \vec{m{:}\tau^*} \rangle\!\rangle, \Gamma_{2\Xi}^{*} \vdash u$$

where $u = \xi(r)$. This judgement is derivable by (*type proj*), since the judgement $\Gamma_{1\Xi}^{*}, u \lessdot\!\!\# \ \mathtt{pro}\, t.\langle\!\langle \vec{m{:}\tau^*} \rangle\!\rangle, \Gamma_{2\Xi}^{*} \vdash *$ is derivable by hypothesis. □

The following generation lemmas are stated and proved in [15] for general derivations in [15]. That they hold for uniform derivations follows from the results we prove in Section 5.3.

Proposition 20

(i) If $\Gamma \dot{\vdash}\ A$ *has a uniform derivation, so does the judgement* $\Gamma \dot{\vdash} *$ *;*
(ii) If $\Gamma \dot{\vdash}\ \mathtt{pro}\, t.\langle\!\langle R \mid \vec{m{:}\tau} \rangle\!\rangle : T$ *has a uniform derivation, so does the judgement* $\Gamma, t : T \dot{\vdash}\ \tau_i : T$*, for each* τ_i *in* $\vec{\tau}$*.*
(iii) If $\Gamma \dot{\vdash}\ e : \tau$ *has a uniform derivation, so does the judgement* $\Gamma \dot{\vdash}\ \tau : T$*.*

Lemma 21 *If* $\Gamma \dot{\vdash} *$ *has a uniform derivation, then* $\Gamma_{\Xi}^{*} \vdash *$ *is derivable in our system.*

Proof. The proof is by induction on the length of Γ. The basis of induction is trivial. The case when Γ is extended with $t : T$ follows immediately by the induction hypothesis. The case when Γ is extended with $x : \tau$ also follows by induction, using Lemma 19 to deduce that $\Gamma_{\Xi}^{*} \vdash \tau^*$ is derivable. The case when Γ is extended with $r : \kappa$ requires the fact that the judgement occurs in a uniform derivation. The argument is as follows. Let $\mathtt{pro}\, t.\langle\!\langle rt \mid \vec{m{:}\tau} \rangle\!\rangle$ be the reference for r in the uniform derivation Ξ. Then the conclusion of the (*ext/over*) rule that discharges r contains a typing judgement of the form $\Gamma_1 \dot{\vdash}\ e : \mathtt{pro}\, t.\langle\!\langle R \mid \vec{m{:}\tau} \rangle\!\rangle$, for some row R. Since this judgement has a uniform derivation (just because it occurs in Ξ), then a uniform derivation also exists for the judgement $\Gamma_1 \dot{\vdash}\ \mathtt{pro}\, t.\langle\!\langle R \mid \vec{m{:}\tau} \rangle\!\rangle : T$ (by Proposition 20(*iii*)). Then, for each τ_i in $\vec{\tau}$, the judgements $\Gamma_1, t : T \dot{\vdash}\ \tau_i$ also have a uniform derivation (by Proposition 20 (*ii*)). Now, the translation of $\Gamma, r : \kappa \dot{\vdash} *$, which is the judgement $\Gamma_{\Xi}^{*}, u \lessdot\!\!\# \ \mathtt{pro}\, t.\langle\!\langle \vec{m{:}\tau} \rangle\!\rangle \vdash *$ for $u = \xi(r)$, is derived as follows:

$$\dfrac{\dfrac{\Gamma_{\Xi}^{*}, t \lessdot\!\!\# \ \mathtt{pro}\, t.\langle\!\langle\rangle\!\rangle \vdash \tau_i^*}{\Gamma_{\Xi}^{*} \vdash \mathtt{pro}\, t.\langle\!\langle \vec{m{:}\tau^*} \rangle\!\rangle \qquad u \notin Dom(\Gamma_{\Xi}^{*})}\ (\mathit{type\ pro})}{\Gamma_{\Xi}^{*}, u \lessdot\!\!\# \ \mathtt{pro}\, t.\langle\!\langle \vec{m{:}\tau^*} \rangle\!\rangle \vdash *}\ (\lessdot\!\!\# \ \ \mathit{var})$$

That the judgements in the premise of the rule (*type pro)* are derivable follows from Lemma 19. □

Lemma 22 follows from Lemmas 19 and 21 observing that $\Gamma \dot{\vdash} *$ has a uniform derivation whenever so does $\Gamma \mathrel{\vdash\!\!\div} \tau : T$.

Lemma 22 *Let $\Gamma \mathrel{\vdash\!\!\div} \tau : T$ be a judgement occurring in a uniform derivation Ξ. Then the judgement $\Gamma^*_\Xi \vdash \tau^*$ is derivable in our system.*

Using Lemma 22, we can then prove the corresponding property for term-judgements of the form $\Gamma \mathrel{\vdash\!\!\div} e : \tau$.

Lemma 23 *If $\Gamma \mathrel{\vdash\!\!\div} e : \tau$ occurs in a uniform derivation Ξ, then the judgement $\Gamma^*_\Xi \vdash e : \tau^*$ is derivable in our system.*

Proof. The proof is by induction on the subderivation Ξ' of $\Gamma \mathrel{\vdash\!\!\div} e : \tau$. The basis of induction is when Ξ' ends up with (*projection*). This follows from Lemma 21, using the fact that $\Gamma \mathrel{\vdash\!\!\div} *$ is derivable (by Proposition 20 (*i*)). Lemma 21 is used similarly in the case (*empty*).

The case when Ξ' ends up with (*abs*) or (*app*) follows immediately from the induction hypothesis.

In the (*ext*) case, Ξ' ends up with a rule of the following form:

$$\frac{\begin{array}{ll}\Gamma \mathrel{\vdash\!\!\div} e_1 : \mathtt{pro}\,t.\langle\!\langle \vec{p{:}\sigma}, \vec{m{:}\tau} \rangle\!\rangle & \\ \Gamma,\, t : T \mathrel{\vdash\!\!\div} \langle\!\langle \vec{p{:}\sigma} \rangle\!\rangle : [\vec{m}, n] & r \textit{ not in } \tau \\ \Gamma,\, r : T \to [\vec{m}, n] \mathrel{\vdash\!\!\div} e_2 : [\mathtt{pro}\,t.\langle\!\langle rt \mid \vec{m{:}\tau}, n{:}\tau \rangle\!\rangle / t](t \to \tau) & \end{array}}{\Gamma \mathrel{\vdash\!\!\div} \langle e_1 \longleftarrow\!\!+\, n{=}e_2 \rangle : \mathtt{pro}\,t.\langle\!\langle \vec{p{:}\sigma}, \vec{m{:}\tau}, n{:}\tau \rangle\!\rangle}$$

Then, the judgement in the conclusion of the rule may be derived by the following rule in our system:

$$\frac{\begin{array}{ll}\Gamma^*_\Xi \vdash e_1 : \mathtt{pro}\,t.\langle\!\langle \vec{p{:}\sigma}^*, \vec{m{:}\tau}^* \rangle\!\rangle & n \notin \{\vec{p}, \vec{m}\} \\ \Gamma^*_\Xi,\, u \mathrel{\lessdot\!\!\#}\, \mathtt{pro}\,t.\langle\!\langle \vec{m{:}\tau}^*, n{:}\tau^* \rangle\!\rangle \vdash e_2 : [u/t](t \to \tau)^* & \end{array}}{\Gamma^*_\Xi \vdash \langle e_1 \longleftarrow\!\!+\, n{=}e_2 \rangle : \mathtt{pro}\,t.\langle\!\langle \vec{p{:}\sigma}^*, \vec{m{:}\tau}^*, n{:}\tau^* \rangle\!\rangle}$$

The context $\Gamma^*_\Xi,\, u \mathrel{\lessdot\!\!\#}\, \mathtt{pro}\,t.\langle\!\langle \vec{m{:}\tau}^*, n{:}\tau \rangle\!\rangle$ is the translation of the context $\Gamma,\, r : T \to [\vec{m}, n]$, given that $\mathtt{pro}\,t.\langle\!\langle rt \mid \vec{m{:}\tau}, n{:}\tau \rangle\!\rangle$ is the reference for r in the derivation Ξ. The premises of this rule are derivable by the induction hypothesis; hence also the conclusion is derivable in our system. There is no loss of generality in restricting to instances of (*ext*) as the one we have used, even though the (*ext*) rule from [15] has the following, more general, structure:

$$\frac{\begin{array}{ll}\Gamma \Vdash e_1 : \mathtt{pro}\, t.\langle\!\langle R \mid \vec{m{:}\tau}\rangle\!\rangle & \\ \Gamma, t:T \Vdash R : [\vec{m}, n] & r \textit{ not in } \tau \\ \Gamma,\, r : T \to [\vec{m}, n] \Vdash e_2 : [\mathtt{pro}\, t.\langle\!\langle rt \mid \vec{m{:}\tau}, n{:}\tau\rangle\!\rangle/t](t{\to}\tau) & \end{array}}{\Gamma \Vdash \langle e_1 \leftarrow\!\!+\, n{=}e_2\rangle : \mathtt{pro}\, t.\langle\!\langle R \mid \vec{m{:}\tau}, n{:}\tau\rangle\!\rangle}\ .$$

This formulation allows applications of the (*ext*) rule in cases when the type of e_1 is polymorphic. However, it is not difficult to see that instances of the (*ext*) rule arising in a uniform derivations must have the restricted form we have assumed above. To see this we reason as follows. The only other possibility would be that $R = \langle\!\langle r't \mid \ldots\rangle\!\rangle$ for some row variable $r' \neq r$ (that $r' \neq r$ follows from the fact that the type $\mathtt{pro}\, t.\langle\!\langle R \mid \vec{m{:}\tau}, n{:}\tau\rangle\!\rangle$ is well-formed in Γ, which in turn follows from the fact that the judgement in the conclusion of the rule is derivable (cf. [15]). But then the derivation would contain an occurrence of r' that is not within its reference (for the reference is either $\mathtt{pro}\, t.\langle\!\langle R \mid \vec{m{:}\tau}\rangle\!\rangle$ or $\mathtt{pro}\, t.\langle\!\langle R \mid \vec{m{:}\tau}, n{:}\tau\rangle\!\rangle$, not both). Hence the derivation would not be uniform.

The remaining cases are when Ξ' ends up with (*send*) or (*over*). If Ξ' ends up with (*send*), then it has the form:

$$\frac{\Gamma \Vdash e : \mathtt{pro}\, t.\langle\!\langle R \mid m{:}\tau\rangle\!\rangle}{\Gamma \Vdash e \Leftarrow m : [\mathtt{pro}\, t.\langle\!\langle R \mid m{:}\tau\rangle\!\rangle/t]\tau}\ ,$$

and the judgement in the premise is derivable. Then the translation of this judgement may be derived as follows.

$$\frac{\begin{array}{l}\Gamma^*_{\Xi} \vdash e : (\mathtt{pro}\, t.\langle\!\langle R \mid m{:}\tau\rangle\!\rangle)^* \\ \Gamma^*_{\Xi} \vdash (\mathtt{pro}\, t.\langle\!\langle R \mid m{:}\tau\rangle\!\rangle)^* \mathrel{<\!\!\!\#} \mathtt{pro}\, t.\langle\!\langle m{:}\tau^*\rangle\!\rangle\end{array}}{\Gamma^*_{\Xi} \vdash e \Leftarrow m : ([\mathtt{pro}\, t.\langle\!\langle R \mid m{:}\tau\rangle\!\rangle/t]\tau)^*}\ .$$

The type $(\mathtt{pro}\, t.\langle\!\langle R \mid m{:}\tau\rangle\!\rangle)^*$ may either be a $\mathtt{pro}$-type in our system, if $R = \langle\!\langle \vec{p{:}\sigma}\rangle\!\rangle$, or the type variable $u = \xi(r)$ if $R = \langle\!\langle rt \mid \ldots\rangle\!\rangle$. The two cases correspond respectively to messages to an object, and messages to *self*. As in [15], they are treated uniformly in our type system.

That $\Gamma^*_{\Xi} \vdash e : (\mathtt{pro}\, t.\langle\!\langle R \mid m{:}\tau\rangle\!\rangle)^*$ is derivable follows from the induction hypothesis. For the other premise, instead, we distinguish the two possible subcases.

If $R = \langle\!\langle \vec{p{:}\sigma}\rangle\!\rangle$, then $(\mathtt{pro}\, t.\langle\!\langle R \mid m{:}\tau\rangle\!\rangle)^* = \mathtt{pro}\, t.\langle\!\langle \vec{p{:}\sigma}^*, m{:}\tau^*\rangle\!\rangle$ and the judgement

$$\Gamma^*_{\Xi} \vdash \mathtt{pro}\, t.\langle\!\langle \vec{p{:}\sigma}^*, m{:}\tau^*\rangle\!\rangle \mathrel{<\!\!\!\#} \mathtt{pro}\, t.\langle\!\langle m{:}\tau^*\rangle\!\rangle$$

is derivable by ($\lessdot\#$ *pro*) using the fact that $\Gamma^*_\Xi \vdash \mathtt{pro}\, t.\langle\!\langle p\vec{:}\sigma^*, m{:}\tau^*\rangle\!\rangle$ is derivable by Lemma 22 (to apply Lemma 22, observe that $\Gamma \mathrel{\vdash\!\div} \mathtt{pro}\, t.\langle\!\langle p\vec{:}\sigma, m{:}\tau\rangle\!\rangle : T$ has a uniform derivation, since so does $\Gamma \mathrel{\vdash\!\div} e : \mathtt{pro}\, t.\langle\!\langle R \mid m{:}\tau\rangle\!\rangle$: this follows from Proposition 20(*iii*)). If, instead, $(\mathtt{pro}\, t.\langle\!\langle R \mid m{:}\tau\rangle\!\rangle)^*$ is a type variable, say $\xi(r)$, then it must be the case that $R = \langle\!\langle rt \mid p\vec{:}\sigma\rangle\!\rangle$ for some $p\vec{:}\sigma$. But then $r \in Dom(\Gamma)$ and $\mathtt{pro}\, t.\langle\!\langle rt \mid p\vec{:}\sigma, m{:}\tau\rangle\!\rangle$ is the r-reference in Ξ. Hence, from Definition 17, it follows that $\xi(r) \lessdot\# \mathtt{pro}\, t.\langle\!\langle p\vec{:}\sigma^*, m{:}\tau^*\rangle\!\rangle \in \Gamma^*_\Xi$, and then $\Gamma^*_\Xi \vdash \xi(r) \lessdot\# \mathtt{pro}\, t.\langle\!\langle m{:}\tau^*\rangle\!\rangle$ is derivable by ($\lessdot\#$ *proj*), since $\Gamma^*_\Xi \vdash *$ is derivable by 21, and by ($\lessdot\#$ *trans*).

In the case of (*over*), the derivation Ξ' ends up with a rule like the following:

$$\frac{\Gamma \mathrel{\vdash\!\div} e_1 : \mathtt{pro}\, t.\langle\!\langle R \mid m\vec{:}\tau\rangle\!\rangle \qquad \Gamma,\, r : T \to [\vec{m}] \mathrel{\vdash\!\div} e_2 : [\mathtt{pro}\, t.\langle\!\langle rt \mid m\vec{:}\tau\rangle\!\rangle/t](t{\to}\tau)}{\Gamma \mathrel{\vdash\!\div} \langle e_1 \leftarrow m_i{=}e_2\rangle : \mathtt{pro}\, t.\langle\!\langle R \mid m\vec{:}\tau\rangle\!\rangle}$$

Then, the translation of the judgement in the conclusion can be derived as follows:

$$\frac{\Gamma^*_\Xi \vdash e_1 : (\mathtt{pro}\, t.\langle\!\langle R \mid m\vec{:}\tau\rangle\!\rangle)^* \qquad \Gamma^*_\Xi \vdash (\mathtt{pro}\, t.\langle\!\langle R \mid m\vec{:}\tau\rangle\!\rangle)^* \lessdot\# \mathtt{pro}\, t.\langle\!\langle m\vec{:}\tau^*\rangle\!\rangle \qquad \Gamma^*_\Xi,\, u \lessdot\# \mathtt{pro}\, t.\langle\!\langle m\vec{:}\tau^*\rangle\!\rangle \vdash e_2 : [u/t](t{\to}\tau)^*}{\Gamma^*_\Xi \vdash \langle e_1 \leftarrow m_i{=}e_2\rangle : (\mathtt{pro}\, t.\langle\!\langle R \mid m\vec{:}\tau\rangle\!\rangle)^*}\ .$$

Here, again, $u = \xi(r)$. The first and last premise of the rule are derivable by the induction hypothesis; that the second premise is derivable follows as in the (*send*) case above. $\square$

From Lemma 23 we then have the desired result.

Theorem 24 *Let $\Gamma \mathrel{\vdash\!\div} e : \tau$ be a first-order judgement. If this judgement is derivable in the system of [15], then the judgement $\Gamma \vdash e : \tau$ is derivable in our system.*

Proof. Let Ξ be the uniform derivation for $\Gamma \mathrel{\vdash\!\div} e : \tau$. The existence of such derivation follows from Proposition 13 (Section 5.1), and Theorem 36 (Section 5.3). Then apply Lemma 23 to $\Gamma \mathrel{\vdash\!\div} e : \tau$, and observe that the translation of this judgement is simply $\Gamma \vdash e : \tau$, since $\Gamma \mathrel{\vdash\!\div} e : \tau$ is derivable and first-order. $\square$

Remark. As a final remark, we note that the converse of Theorem 24 could also be proved. We could, in fact, define notions of "simple" and "uniform" for derivations and judgements in our system as well, and then show that for every derivation of a first-order judgement in our system a corresponding "uniform" derivation exists for that judgement in the system of [15]. Then, an inverse translation could be defined, that given any derivation of a first-order judgement in our system, yields a corresponding uniform derivation of [15], for that same judgement. Using such inverse translation, the same reasoning in the proof of Theorem 24 could then be used to show that every first-order judgement has a derivation in [15] if and only if it has a derivation in our system.

5.3 First-order derivations can be made uniform

We conclude our analysis showing that every first-order judgement that has a simple derivation has also a uniform derivation. The existence of non-uniform derivations is easily shown. Consider, as an example, the following non-uniform derivation where the reference for r is the type $\mathtt{pro}\, t.\langle\!\langle rt \mid n{:}t \rangle\!\rangle$, and id is the identity function $\lambda s.s$. Occurrences of the r-references are boxed.

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{\dfrac{\Xi_1}{r : T \to [n], s : \boxed{\mathrm{pro}\ t.\langle rt \mid n{:}t\rangle}, x : \mathrm{pro}\, t.\langle rt\rangle \vdash \langle x \leftarrow\!\!+\ n{=}id\rangle : \mathrm{pro}\, t.\langle rt \mid n{:}t\rangle}}{r : T \to [n], s : \boxed{\mathrm{pro}\ t.\langle rt \mid n{:}t\rangle} \vdash \lambda x.\langle x \leftarrow\!\!+\ n{=}id\rangle : \mathrm{pro}\, t.\langle rt\rangle \to \mathrm{pro}\, t.\langle rt \mid n{:}t\rangle}
\quad
\dfrac{\Xi_2}{r : T \to [n], s : \boxed{\mathrm{pro}\ t.\langle rt \mid n{:}t\rangle} \vdash \lambda f.s : (\mathrm{pro}\, t.\langle rt\rangle \to \mathrm{pro}\, t.\langle rt \mid n{:}t\rangle) \to \boxed{\mathrm{pro}\ t.\langle rt \mid n{:}t\rangle}}
}{r : T \to [n], s : \boxed{\mathrm{pro}\ t.\langle rt \mid n{:}t\rangle} \vdash (\lambda f.s)(\lambda x.\langle x \leftarrow\!\!+\ n{=}id\rangle) : \boxed{\mathrm{pro}\ t.\langle rt \mid n{:}t\rangle}}
}{r : T \to [n] \vdash \lambda s.(\lambda f.s)(\lambda x.\langle x \leftarrow\!\!+\ n{=}id\rangle) : \boxed{\mathrm{pro}\ t.\langle rt \mid n{:}t\rangle} \to \boxed{\mathrm{pro}\ t.\langle rt \mid n{:}t\rangle}}
\quad \vdash \langle\rangle : \mathrm{pro}\, t.\langle\rangle \quad \vdash \langle\rangle : [n]
}{\vdash \langle \langle\rangle \leftarrow\!\!+\ n{=}\lambda s.(\lambda f.s)(\lambda x.\langle x \leftarrow\!\!+\ n{=}id\rangle)\rangle : \mathrm{pro}\, t.\langle n{:}t\rangle}
\ (ext)
$$

Lack of uniformity in this derivation is a consequence of the choice of $\mathtt{pro}\, t.\langle\!\langle rt \rangle\!\rangle$ as the type of the bound variable x in the abstraction $\lambda x.\langle x \longleftarrow\!\!+\ n{=}id\rangle$. However, it is easily verified that this choice is not critical in the derivation, as *any* $\mathtt{pro}$-type could be assumed for this variable, as long as it does not contain the n method to be added. In particular, the same typing for the body of the n method in the lower application of the (ext) rule (hence, for the whole expression) could be derived by assuming the monomorphic type $\mathtt{pro}\, t.\langle\!\langle\rangle\!\rangle$ for x, and by systematically substituting the row $\lambda t.\langle\!\langle\rangle\!\rangle$ for all the occurrences of r in the $\mathtt{pro}$-types that are not the r-reference. A similar reasoning applies to types like $\mathtt{pro}\, t.\langle\!\langle r\sigma \mid \ldots\rangle\!\rangle$, with $\sigma \neq t$, which are not occurrences of row-variable references in a simple derivation (see Definition 14).

In the rest of this section we show that the transformation we have just outlined can be generalized to arbitrary simple derivations from [15], and that the result of the transformation is still a derivation in that system.

Clearly, the transformation relies critically on the ability to identify the occurrences of the r-reference in the derivation. The subtlety is that we need to distinguish occurrences of types that are *equal* to the r-reference from occurrences *of the* r-reference. As the example shows, this may require some work: notice, in fact, that there are several occurrences of the type $\mathtt{pro}\, t.\langle\!\langle rt \mid n{:}t \rangle\!\rangle$ that are equal to the r-reference, but are *not* occurrences of the r-reference (since they arise as extensions of types that are not occurrences of the r-reference).

While the notion of type occurrence seems to be relatively well-understood in proof-theoretic presentations of type (and logical) systems, we do not know any formal technique that could be instantiated directly to identify the desired type occurrences in the type system we are considering. Below we describe an ad-hoc technique for our specific purposes. The description is somewhat informal, as the intuition is simple; instead, a formal definition would be considerably long and cumbersome, given the large number of type rules and their complex structure.

5.3.1 Locating row variables within their references

The following technique allows us to directly locate the occurrences of the row variables within their references in a derivation. Given a simple derivation Ξ, we construct a non-directed graph whose nodes are the row variables that occur in the $\mathtt{pro}$-types arising in Ξ, and whose edges are links between pairs of such nodes.

In describing the construction of the graph, we use the following terminology. Given two occurrences τ' and τ'' of a type arising in Ξ, the action of *connecting* τ' *and* τ'' consists of drawing an edge between the row variables occurring at corresponding positions within τ' and τ'' (note: given the assumption that τ' and τ'' are occurrences of the same type, row variables occurring at corresponding positions in τ' and τ'' are, in fact, the same row variable). A similar terminology is used for rows and contexts: *connecting* two occurrences of a row (context) consists of connecting every pair of row variables occurring at corresponding positions in the two occurrences of the row (context).

The construction of the graph is defined by specifying, at every rule of the derivation, which occurrences of the types, rows and contexts arising in the rule in question should be connected. Below, we give examples of how this should be accomplished depending on the structure of the rule. For future reference, we note that the declarations of a row variable r in the contexts of the derivation are never connected to the occurrences of r in the types and rows of the derivation.

($projection$). We first consider the case of projection of term variables. In this case the rule has the form:

$$\frac{\Gamma', x : \tau, \Gamma'' \vdash\!\!\div *}{\Gamma', x : \tau, \Gamma'' \vdash\!\!\div x : \tau}$$

Then connect (i) the two occurrences of Γ' and the two occurrences of Γ'', (ii) the two occurrences of τ in the conclusion of the rule, and (iii) the occurrence of τ in the context of the premise with the occurrence in the context of the conclusion.

In case of projection of row and type variables, the rule has the form given below, where $U : V$ is either $r : \kappa$ or $t : T$:

$$\frac{\Gamma', U : V, \Gamma'' \vdash\!\!\div *}{\Gamma', U : V, \Gamma'' \vdash\!\!\div U : V}$$

Here, connect the two occurrences of Γ' and the two occurrences of Γ''. Notice that no link is generated between the occurrences of U even when $U : V$ is $r : \kappa$. As we noted, the occurrences of a row variables r in the contexts of the derivation are *never* connected to any of occurrences of this variable on the right-hand side of $\vdash\!\!\div$ in the judgement of the derivation (see also the cases of (ext) and ($over$) below).

(app) and (abs). In the case of (app), the rule has the form:

$$\frac{\Gamma \vdash\!\!\div e_1 : \sigma \to \tau \quad \Gamma \vdash\!\!\div e_2 : \sigma}{\Gamma \vdash\!\!\div e_1 e_2 : \tau}$$

Then connect (i) the occurrence of Γ in the conclusion with, respectively, the occurrence of Γ in the left and right premises, and (ii) the two occurrences of τ as well as the two occurrences of σ. The case of (abs) is handled in a similar way.

($send$). This case is subtler due the type substitution arising at this rule. It is best to distinguish two subcases, depending on the result of the substitution. If $t \notin Var(\tau)$, the rule has the form:

$$\frac{\Gamma \vdash\!\!\div e : \texttt{pro}\, t.\langle\!\langle R \mid m{:}\tau \rangle\!\rangle}{\Gamma \vdash\!\!\div e \Leftarrow n : \tau}$$

Then connect the two occurrences of Γ and the two occurrences of τ. If instead $t \in Var(\tau)$ the rule is as follows:

$$\frac{\Gamma \mathrel{\vdash\!\!\div} e : \texttt{pro}\, t.\langle\!\langle R \mid m{:}\tau \rangle\!\rangle}{\Gamma \mathrel{\vdash\!\!\div} e \Leftarrow n : [\texttt{pro}\, t.\langle\!\langle R \mid m{:}\tau \rangle\!\rangle/t]\tau}$$

Then, connect the two occurrences of Γ, connect the occurrence of $\texttt{pro}\, t.\langle\!\langle R \mid m{:}\tau \rangle\!\rangle$ in the premise with the occurrence of this type in the substitution $[\texttt{pro}\, t.\langle\!\langle R \mid m{:}\tau \rangle\!\rangle/t]$, and finally connect the outer occurrence of τ in the conclusion with the occurrence of this type in the premise of the rule. (This is subtler than it appears from what we just said: if we denote with τ' the result of the substitution $[\texttt{pro}\, t.\langle\!\langle R \mid m{:}\tau \rangle\!\rangle/t]\tau$, then connecting $\texttt{pro}\, t.\langle\!\langle R \mid m{:}\tau \rangle\!\rangle$ with its occurrences in τ' requires a preliminary analysis to identify these occurrences. This can be done by contrasting τ and τ': the occurrences of $\texttt{pro}\, t.\langle\!\langle R \mid m{:}\tau \rangle\!\rangle$ in τ' that result from the substitution are all and only the occurrences that correspond to occurrences of the type variable t in τ).

(*ext*) and (*over*). The actions are defined similarly to the previous cases, using the idea described for the case of (*send*) to handle the type-substitution. As in the (*projection*) case, the occurrence of r in the context is *not* connected to the occurrence of this variable in the r-reference arising at the right-hand side of $\mathrel{\vdash\!\!\div}$ in (*ext*/*over*) rule under consideration.

The actions at the rules for types and rows are defined similarly to the previous cases. As an example of connecting two rows, consider case of (*type pro*). In this case the rule has the form:

$$\frac{\Gamma, t : T \mathrel{\vdash\!\!\div} R : [m_1, \ldots, m_k]}{\Gamma \mathrel{\vdash\!\!\div} \texttt{pro}\, t.R : T}$$

Then, connect the two occurrences of Γ as well as the two occurrences of R. As a further example, consider the (*row ext*) case:

$$\frac{\Gamma \mathrel{\vdash\!\!\div} R : [n, m_1, \ldots, m_k] \qquad \Gamma \mathrel{\vdash\!\!\div} \tau : T}{\Gamma \mathrel{\vdash\!\!\div} \langle\!\langle R \mid n{:}\tau \rangle\!\rangle : [m_1, \ldots, m_k]}$$

In this case, connect the occurrence of Γ in the conclusion with the two occurrences Γ in the premises, and the two occurrences of R and, respectively, of τ.

Let $\mathcal{G}_\Xi$ be the graph arising from iterating at all the rules of Ξ the process we just outlined, until no further action would create any new edge between two nodes (row variables). Having constructed $\mathcal{G}_\Xi$, we now have a simple mechanism to identify the occurrences of the row-variable references in the derivation Ξ. Namely, for every row variable r of Ξ, proceed as follows:

(i) underline the occurrence of r in the r-reference found at the (*ext*) or

($over$) rule that discharges r in Ξ;

(ii) then underline every non-underlined occurrence of r that is connected to an underlined occurrence of the same row variable in $\mathcal{G}_\Xi$.

Let now $\underline{\Xi}$ be the derivation that results from completing the two steps above. The following properties are easily verified for $\underline{\Xi}$.

Proposition 25 *For every occurrence of a row variable r in Ξ, this occurrence is underlined in $\underline{\Xi}$ if and only if it is connected to the occurrence of r in the r-reference found at the (ext) or ($over$) rule that discharges r.*

Proposition 26 *Let α' and α'' be two occurrences of a given type (or row) at any of the typing rules of $\underline{\Xi}$. Then, given any occurrence of a row variable r in α', this occurrence is underlined if and only if so is the corresponding occurrence of r in α''.*

Proposition 27 *Let $\texttt{pro}\, t.\langle\!\langle r\sigma \mid \vec{m{:}\tau} \rangle\!\rangle$ be a type occurring in Ξ. This type is an occurrence of the r-reference (hence, in particular $\sigma = t$) if and only if the row variable r is underlined in the corresponding occurrence of this type in $\underline{\Xi}$.*

All of these properties are relevant to the transformation of simple derivations into uniform derivations we have briefly explained earlier in this section. As we said, given a simple derivation Ξ, the intention is to obtain a uniform derivation from Ξ by systematically substituting the row $\lambda t.\langle\!\langle\rangle\!\rangle$ for all the occurrences of the row variable r found in pro-types that are not occurrences of the r-reference.

Then, Proposition 25 tells us that the row occurrences that should be substituted are all and only the occurrences that are *not* underlined in $\underline{\Xi}$; Proposition 26, in turn, ensures that applying the substitution to the occurrences of a type (row) in a rule generates equal types (rows); finally, Proposition 27 guarantees that if the result of the substitution process is a derivation, then it is also a uniform derivation (for the only polymorphic pro-types that outlive the substitution process are the r-references).

A final remark is in order for the row-variable declarations occurring in the contexts of the derivation. As we said, and as it follows from the construction of the graph $\mathcal{G}_\Xi$, the declarations of a row variable r are never connected to other occurrences of r in the graph. This is necessary, for otherwise we would end up connecting *all* the occurrences of r in the derivation, thus falling short of identifying the occurrences of this variable within its reference. As a result, row-variable declarations are not underlined in $\underline{\Xi}$, and deserve separate treatment in the substitution process we describe next.

5.3.2 Substitutions

In formalizing the transformation, we realize that the process of substitution should affect not only the occurrences of a row variable within the `pro`-types of Ξ, but also the corresponding occurrences of this variable within the rows arising in the kinding judgements of the derivation.

Using the meta-syntactic variables R to indicate (possibly) underlined rows, we then denote with $R_\sharp$ the row that results from the substitution. More precisely: if R is a (possibly underlined) row, $R_\sharp$ denotes the row that results (i) from substituting the row $\lambda t.\langle\!\langle\rangle\!\rangle$ for every non-underlined occurrence of a row variable r within R, and (ii) from erasing the underline from all the underlined occurrences. We define $R_\sharp$ directly as the result of applying the substitution and reducing the β-redex that arises from the substitution. We first give a definition for rows of kind κ. Rows of kind $T{\to}\kappa$ are treated separately, in definitions 30 and 31 below.

Definition 28 (Substitution for Rows) *The definition is given by induction on the structure of rows. The cases of underlined and non-underlined row variables are distinguished.*

$\langle\!\langle\rangle\!\rangle_\sharp = \langle\!\langle\rangle\!\rangle$;
$\langle\!\langle R \mid m{:}\tau\rangle\!\rangle_\sharp = \langle\!\langle R_\sharp \mid m{:}\tau_\sharp\rangle\!\rangle$
$(r\tau)_\sharp = \langle\!\langle\rangle\!\rangle$;
$(\underline{r}\tau)_\sharp = r(\tau_\sharp)$;

Accordingly, given a row R, every occurrence of a row scheme $\langle\!\langle r\tau \mid \vec{m{:}\tau}\rangle\!\rangle$ within this row is replaced in $R_\sharp$ by the row $\langle\!\langle \vec{m{:}\tau_\sharp}\rangle\!\rangle$ that results from recursively applying the substitution to the rows nested within the types $\vec{\tau}$. Instead, row schemes such as $\langle\!\langle \underline{r}\tau \mid \vec{m{:}\tau}\rangle\!\rangle$ occurring in R (with r underlined) are transformed into the corresponding row scheme $\langle\!\langle r\tau_\sharp \mid \vec{m{:}\tau_\sharp}\rangle\!\rangle$.

Definition 29 (Substitution for Types)

$t_\sharp = t$;
$(\texttt{pro}\, t.R)_\sharp = \texttt{pro}\, t.R_\sharp$;
$(\sigma{\to}\tau)_\sharp = \sigma_\sharp{\to}\tau_\sharp$

Definition 30 (Substitution for Contexts)

$\varepsilon_\sharp = \varepsilon$;
$(\Gamma, x:\tau)_\sharp = \Gamma_\sharp, x:\tau_\sharp$;
$(\Gamma, t:T)_\sharp = \Gamma_\sharp, t:T$;
$(\Gamma, r:T{\to}\kappa)_\sharp = \Gamma_\sharp, r:T{\to}\kappa$;

Note that the row-variable declarations of a context are not affected by the substitution. This is required to ensure that the derivations of kinding judgements can be made uniform (see the (*projection*) case in the proof of Lemma 32 below).

Definition 31 (Substitution for Judgements) *For any given judgement $\Gamma \mathrel{|\!\div} A$, the result of the substitution is the judgement $\Gamma_\sharp \mathrel{|\!\div} A_\sharp$, where $A_\sharp$ is defined by cases as follows (the last two clauses handle the case of underlined and non-underlined occurrences):*

$(e : \tau)_\sharp = e : \tau_\sharp$;
$(\tau : T)_\sharp = \tau_\sharp : T$;
$(R : \kappa)_\sharp = R_\sharp : \kappa$;
$(\underline{r} : T {\to} \kappa)_\sharp = r : T {\to} \kappa$.
$(r : T {\to} \kappa)_\sharp = \lambda t.\langle\!\langle\rangle\!\rangle : T {\to} \kappa$.

5.3.3 From First-order Derivations to Uniform derivations

Given a simple derivation we give a proof of the existence of a corresponding uniform derivation. The proof is constructive, and it implicitly defines a systematic method for finding the desired uniform derivation.

The outline of the proof is as in Section 5.2: given a simple derivation, pick out a judgement $\Gamma \mathrel{|\!\div} A$ in this derivation, and let Ξ be the subderivation rooted at this judgement. We show that a corresponding uniform subderivation $\Xi_\sharp$ for the judgement $\Gamma_\sharp \mathrel{|\!\div} A_\sharp$ can be obtained from Ξ.

Again, the proof is given in two steps. Lemma 34 proves the desired property for the case when A is a kinding judgement of the form $\tau : T$ or $R : \kappa$. Lemma 35 handles the case when A is a typing judgement of the form $e : \tau$.

The proof of Lemma 34, in turn, uses two preliminary results. Lemma 32 proves the desired property for kinding judgements under the additional assumption that a uniform derivation exists for $\Gamma_\sharp \mathrel{|\!\div} *$. Lemma 33 uses Lemma 32 to show that the additional assumption is, in fact, unnecessary.

All the derivations and judgements mentioned in the hypotheses of the following lemmas are intended to be subderivations, and judgements, of a given simple derivation resulting from the underlining described earlier in this section.

Lemma 32 *Let Ξ be the derivation of a judgement $\Gamma \mathrel{|\!\div} U : V$, where $U : V$ is either $\tau : T$ or $R : \kappa$. If a uniform derivation exists for $\Gamma_\sharp \mathrel{|\!\div} *$, then a uniform derivation also exists for the judgement $\Gamma_\sharp \mathrel{|\!\div} U_\sharp : V$.*

Proof. By induction on the derivation Ξ of $\Gamma \mathrel{\vdash\!\!\div} U : V$, by a case analysis on last rule of this derivation.

- (*projection*). In this case, the judgement in question is $\Gamma \mathrel{\vdash\!\!\div} U : V$, where $U : V$ is either $t : T$, or $r : T \to \kappa$, or $\underline{r} : T \to \kappa$, and Γ is either the context $\Gamma', t : T, \Gamma''$, or the context $\Gamma', r : T \to \kappa, \Gamma''$ for some choice of Γ' and Γ''.

  This case is the basis of induction, since the derivation of this judgement depends only on the derivation of the judgement $\Gamma \mathrel{\vdash\!\!\div} *$, with $\Gamma_\sharp \mathrel{\vdash\!\!\div} *$ derivable by hypothesis. The desired uniform derivation can be constructed in either of the following ways, depending on whether $U : V$ is, respectively, $r : T \to \kappa$, or $\underline{r} : T \to \kappa$, or $t : T$:

$$\dfrac{\dfrac{\dfrac{\Xi^*_\sharp}{\Gamma'_\sharp, r : T\to\kappa, \Gamma''_\sharp \mathrel{\vdash\!\!\div} *}}{\Gamma'_\sharp, r : T\to\kappa, \Gamma''_\sharp, t : T \mathrel{\vdash\!\!\div} \langle\rangle : \kappa}}{\Gamma'_\sharp, r : T\to\kappa, \Gamma''_\sharp \mathrel{\vdash\!\!\div} \lambda t.\langle\rangle : T\to k}\ , \qquad \dfrac{\dfrac{\Xi^*_\sharp}{\Gamma'_\sharp, r : T\to\kappa, \Gamma''_\sharp \mathrel{\vdash\!\!\div} *}}{\Gamma'_\sharp, r : T\to\kappa, \Gamma''_\sharp \mathrel{\vdash\!\!\div} r : T\to\kappa}\ , \qquad \dfrac{\dfrac{\Xi^*_\sharp}{\Gamma'_\sharp, t : T, \Gamma''_\sharp \mathrel{\vdash\!\!\div} *}}{\Gamma'_\sharp, t : T, \Gamma''_\sharp \mathrel{\vdash\!\!\div} t : T}\ .$$

  $\Xi^*_\sharp$ is the uniform derivation for $\Gamma \mathrel{\vdash\!\!\div} *$, that exists by hypothesis. That the derivations are uniform follows from the observation, mentioned already, that the only polymorphic `pro`-types that outlive in $\Gamma_\sharp$ are the row-variable references.
- (*type pro*). In this case, the judgement in question is $\Gamma \mathrel{\vdash\!\!\div} \mathtt{pro}\, t.R : T$, and the premise of the rule is $\Gamma, t : T \mathrel{\vdash\!\!\div} R : \kappa$. Then $\Gamma, t : T \mathrel{\vdash\!\!\div} *$ is derivable, which implies that $t \notin Dom(\Gamma)$. From this, and from the hypothesis that a uniform derivation exists for $\Gamma_\sharp \mathrel{\vdash\!\!\div} *$, it follows that a uniform derivation also exists for $\Gamma_\sharp, t : T \mathrel{\vdash\!\!\div} *$. Then the claim follows by the induction hypothesis.
- The cases (*type arrow*), (*empty row*) and (*row label*), follow immediately from the induction hypothesis. The (*row ext*) case also follows from the induction hypothesis, using the fact that $\langle\!\langle R \mid m{:}\tau \rangle\!\rangle_\sharp = \langle\!\langle R_\sharp \mid m{:}\tau_\sharp \rangle\!\rangle$ by definition. The (*row fn abs*) case is vacuous (there are no such rows in a simple derivation).
- (*row fn app*). In this case the judgement in question is $\Gamma \mathrel{\vdash\!\!\div} R : \kappa$, and either $R = \underline{r}\tau$ or $R = r\tau$ (reminder: applications have this restricted form in simple derivations). If $R = \underline{r}\tau$, the derivation Ξ has the following structure:

$$\dfrac{\dfrac{\Xi'}{\Gamma \mathrel{\vdash\!\!\div} \underline{r} : T\to\kappa} \qquad \dfrac{\Xi''}{\Gamma \mathrel{\vdash\!\!\div} \tau : T}}{\Gamma \mathrel{\vdash\!\!\div} \underline{r}\tau : \kappa}\ .$$

  Then the claim follows by induction on the subderivations Ξ' and Ξ'', using the fact that $(\underline{r}\tau)_\sharp = r(\tau_\sharp)$ by definition. The desired uniform derivation is simply:

$$\dfrac{\dfrac{\Xi'_\sharp}{\Gamma_\sharp \mathrel{|\!\!\div} r : T \to \kappa} \qquad \dfrac{\Xi''_\sharp}{\Gamma_\sharp \mathrel{|\!\!\div} \tau_\sharp : T}}{\Gamma_\sharp \mathrel{|\!\!\div} r\tau_\sharp : \kappa} \ (\textit{row fn app}).$$

If instead $R = r\tau$, by definition we have $(r\tau)_\sharp = \langle\!\langle\rangle\!\rangle$, and the desired uniform derivation can be constructed as follows:

$$\dfrac{\dfrac{\Xi^*_\sharp}{\Gamma_\sharp \mathrel{|\!\!\div} *}}{\Gamma_\sharp \mathrel{|\!\!\div} \langle\!\langle\rangle\!\rangle : \kappa} \ (\textit{row empty}),$$

where $\Xi^*_\sharp$ is the uniform derivation for $\Gamma_\sharp \mathrel{|\!\!\div} *$ in the hypothesis of the lemma. □

Lemma 33 *If the judgement $\Gamma \mathrel{|\!\!\div} *$ is derivable, then $\Gamma_\sharp \mathrel{|\!\!\div} *$ has a uniform derivation.*

Proof. By induction on the length of Γ. The base case is immediate, for $\varepsilon_\sharp \equiv \varepsilon$ and $\varepsilon \mathrel{|\!\!\div} *$ is itself the uniform derivation. There are three inductive cases: again we use the fact that $\Gamma \mathrel{|\!\!\div} *$ is derivable if so is $\Gamma \mathrel{|\!\!\div} A$.

- (*type var*). In this case, the judgement in question is $\Gamma, t : T \mathrel{|\!\!\div} *$, $t \notin \mathit{Dom}(\Gamma)$, and $\Gamma \mathrel{|\!\!\div} *$ is derivable. By induction hypothesis, $\Gamma_\sharp \mathrel{|\!\!\div} *$ has a uniform derivation, and since $\mathit{Dom}(\Gamma) = \mathit{Dom}(\Gamma_\sharp)$, it follows that also $\Gamma_\sharp, t : T \mathrel{|\!\!\div} *$ has the desired uniform derivation.
- (*exp var*). In this case, the judgement in question is $\Gamma, x : \tau \mathrel{|\!\!\div} *$. Then $x \notin \mathit{Dom}(\Gamma)$ and $\Gamma \mathrel{|\!\!\div} \tau : T$ is derivable, which implies that $\Gamma \mathrel{|\!\!\div} *$ also is derivable. By induction hypothesis $\Gamma_\sharp \mathrel{|\!\!\div} *$ has a uniform derivation, and then $\Gamma_\sharp \mathrel{|\!\!\div} \tau_\sharp : T$ has a uniform derivation by Lemma 32. Now, the uniform derivation for $\Gamma_\sharp, x : \tau_\sharp \mathrel{|\!\!\div} *$ can be completed with an application of (*exp var*).
- (*row var*). In this case, the judgement in question is $\Gamma, r : T \to \kappa \mathrel{|\!\!\div} *$, with $r \notin \mathit{Dom}(\Gamma)$, and $\Gamma \mathrel{|\!\!\div} *$ is derivable. The claim follows directly from the induction hypothesis using the fact that $(\Gamma, r : T \to \kappa)_\sharp = \Gamma_\sharp, r : T \to \kappa$ by definition. □

Combining Lemma 32 and Lemma 33 we have the desired property for kinding judgements.

Lemma 34 *Let Ξ be the derivation of a judgement $\Gamma \vdash\!\div U : V$, where $U : V$ is either $\tau : T$ or $R : \kappa$. Then a uniform derivation exists for the judgement $\Gamma_\sharp \vdash\!\div U_\sharp : V$.*

Proof. Since $\Gamma \vdash\!\div U : V$ has a derivation, then $\Gamma \vdash\!\div *$ also is derivable. From Lemma 33, $\Gamma_\sharp \vdash\!\div *$ has a uniform derivation, and the proof follows by Lemma 32. $\square$

Next, we prove the corresponding result for typing judgements of the form $\Gamma \vdash\!\div e : \tau$.

Lemma 35 *Let Ξ be the derivation for the judgement $\Gamma \vdash\!\div e : \tau$. Then a uniform derivation exists for the judgement $\Gamma_\sharp \vdash\!\div e : \tau_\sharp$.*

Proof. The proof is by induction on the derivation Ξ: the construction of the uniform derivation for $\Gamma_\sharp \vdash\!\div e : \tau_\sharp$ results from the inductive reasoning exactly as in Lemma 32.

The (*projection*) case uses the fact that $\Gamma \vdash\!\div *$ is derivable, and Lemma 33 to derive that $\Gamma_\sharp \vdash\!\div *$ has a uniform derivation. Lemma 33 is used similarly to handle the case of (*empty*). The cases (*abs*), (*app*) and (*send*) follow immediately from the induction hypothesis. The case of (*ext*) also follows by the induction hypothesis, using Lemma 34 on the kinding judgement (i.e., the second premise of the rule), and observing that none of the occurrences of r in the rule is substituted (the declaration of r in the context of the typing judgement of the method body is not substituted by definition; the occurrences of r in the r-reference are not substituted, because underlined by construction). The case of (*over*) is similar to (*ext*), with the difference that it does not require Lemma 34. $\square$

From Lemma 35, we finally have the result we wished to prove.

Theorem 36 *Let $\Gamma \vdash\!\div e : \tau$ be a first-order judgement from [15]. This judgement is derivable if and only if it has a uniform derivation.*

Proof. The "if" part of the claim is obvious, since every uniform derivation is a derivation. For the "only if" part we reason as follows. Since $\Gamma \vdash\!\div e : \tau$ is first-order and derivable, then by Proposition 13, it has a simple derivation. Then, by Lemma 35, it follows that $\Gamma_\sharp \vdash\!\div e : \tau_\sharp$ has a uniform derivation. Finally, observe that $\Gamma_\sharp \vdash\!\div e : \tau_\sharp$ is just the judgement $\Gamma \vdash\!\div e : \tau$ itself, given that neither Γ nor τ contain any occurrence of a row variable, since $\Gamma \dot{\vdash} e : \tau$ is first-order. $\square$

6 Conclusions

We have presented a new type system for the *Lambda Calculus of Objects*, that gives provision for *MyType* method specialization, while at the same time allowing static detection of *message-not-understood* run-time errors. The new system relies on matching and implicit match-bounded quantification to render the effect of the row schemes used in the original type system for this calculus [15].

We have also carried out a formal study of the relative expressive power of the two systems, showing that they are equivalent on derivations of first-order typing judgements. This implies that the new system and the original one are equally expressive in the typing of closed expressions (or programs).

The result does not generalize to arbitrary expressions and judgements, as we have shown giving examples of typing judgements derivable in [15] that cannot be meaningfully encoded into our system. In fact, although we have not proved it, we are convinced that the original system is strictly more expressive than the one we have presented, if we consider arbitrary judgements. The fundamental reason for this is that the kinding system of [15] is strictly more informative than ours, as it conveys information on method absence, a form of negative information that cannot be accounted for with matching.

On the other hand, the additional expressive power of the original system has a price, which is reflected in the complexity of the calculus of rows in [15]. Also, in practice, the gap between the two systems is less severe than it appears. Notice, in fact, that both system rely on an implicit form of quantification, whereby the only polymorphic expressions are method bodies, which reside within closed objects and may not be used as polymorphic functions. Adding explicit quantification, and hence polymorphic functions, would be possible for both systems, yielding a tradeoff. The extension would be more effective for the system of [15] than for our system: with that extension, it would be possible, for instance, to write *mixins*, i.e., polymorphic functions that add a method to their parameter (see for instance [6], where an imperative version of [15] is presented with explicit quantification), a feature that our system could not account for. On the other hand, the addition of universal types in our system would allow an elegant form of match-bounded polymorphism, which seems to be more naturally expressed and more easily understood that the form of row-polymorphism that would arise in the extension of [15].

Given these observations, the equivalence we have established seems to be a quite satisfactory measure of the expressive power of the new system, especially when one considers the reduced complexity of proofs and typing derivations with respect to the original system. Furthermore, as we already pointed out, the new approach to the rendering of method polymorphism seems to bring a new perspective on the problem of relating type systems for the *Lambda Calculus of Objects* and companion systems for related calculi.

6.1 Related Work

The type systems for the language PolyToil of [10], and for the suite of Object Calculi described in [3] are closely related to the system we have presented. There are, however, two fundamental differences. The first is in the presence of typing rules for primitives that change the shape of an object by the addition of methods, an operation that is only allowed on classes in [10] and [3]. On the other hand, these other proposals give provision for subtyping and subsumption over the types of objects, two issues that we have deliberately disregarded here to privilege the comparative analysis with [15].

These two aspects left aside, all systems, including ours, share the same fundamental ideas in the main typing rules. While this was somehow expected, given the similarities among the untyped calculi, having exposed this relationships at the typing level is one of the payoffs of the system we have presented.

As for subtyping, it is worth mentioning that the system of this paper is amenable to the extension of the Lambda Calculus of Objects with subtyping presented in [16]. Subtyping could be accounted for in our system by distinguishing two "classes" of object types, namely `pro`-types and `obj`-types, exactly as it is done in [16]. The distinction would still reflect the different uses of objects in the type system, either as prototypes, as we have described in the paper, or as "proper" objects created from prototypes by promoting their types to corresponding `obj`-types, via subtyping. Type promotion for a prototype would then have the same effect as in [16] of "sealing" the prototype, by disallowing method additions and "external" redefinitions (in such a way the subsumption rule can be used soundly), but still allowing objects to respond to messages, and to modify their structure via "internal" overrides on *self*.

As a final remark, we should mention that a similar approach to the rendering of method polymorphism for the Lambda Calculus of Objects is proposed in [5]. The key difference, with respect to the system of this paper, is that [5] uses subtyping in place of matching, and subtype-bounded quantification. Again, there are fundamental tradeoffs between the two solutions. On one side, the use of subtyping in [5] allows object subsumption, as well as unrestricted combinations of subsumption with the typing of method additions and overrides, giving that system more flexibility than ours. On the other side, matching appears to be superior to subtyping in the rendering of the desired typing of methods. In fact, in order to ensure safe uses of subsumption, the system of [5] allows type promotion for an object type only when the methods in the promoted type do not reference any of the methods of the original type. As in [7], labeled types are used to encode the inter-dependencies among methods in the methods' types. In [5], however, labeled types involve some additional limitations over [7], and additional complexity in the typing rules for method addition and redefinition.

Acknowledgements. The authors would like to thank Mariangiola Dezani-Ciancaglini for inspiring this work and countless insightful suggestions. Comments from the referees of the submitted version are gratefully acknowledged: they were of great help in rectifying some of the proofs and the technical presentation of the paper.

References

[1] M. Abadi and L. Cardelli. An Imperative Objects Calculus. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *Proc. of TAPSOFT'95*, volume 915 of *Lecture Notes in Computer Science*, pages 471–485. Springer–Verlag, May 1995.

[2] M. Abadi and L. Cardelli. On Subtyping and Matching. In *Proc. of ECOOP'95*, volume 952 of *Lecture Notes in Computer Science*, pages 145–167. Springer–Verlag, August 1995.

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

[4] V. Bono and M. Bugliesi. Matching Constraints for the Lambda Calculus of Objects. In P. De Groote and R. J. Hindley, editors, *Proc. of TLCA'97*, volume 1210 of *Lecture Notes in Computer Science*, pages 46–62. Springer-Verlag, 1997.

[5] V. Bono, M. Bugliesi, M. Dezani, and L. Liquori. Subtyping Constraints for Incomplete Objects. In M. Bidoit and M. Dauchet, editors, *Proc. of CAAP'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 465–477. Springer-Verlag, 1997.

[6] V. Bono and K. Fisher. An Imperative, First-Order Calculus with Object Extension. In *Proc. of ECOOP'98*, Lecture Notes in Computer Science. Springer-Verlag, 1998. To Appear.

[7] V. Bono and L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In *Proc. of CSL'94*, volume 933 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 1995.

[8] K.B. Bruce. A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics. *Journal of Functional Programming*, 1(4):127–206, 1994.

[9] K.B. Bruce, L. Petersen, and A. Fiech. Subtyping is not a Good "Match" for Object-Oriented Languages. In *Proceedings of ECOOP'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 104–127. Springer–Verlag, 1997.

[10] K.B. Bruce, A. Shuett, and R. van Gent. PolyTOIL: a Type-Safe Polymorphic Object-Oriented Language. In *Proceedings of ECOOP'95*, volume 952 of *Lecture Notes in Computer Science*, pages 27–51. Springer–Verlag, 1995.

[11] L. Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, 76:138–164, 1988.

[12] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[13] W. Cook, W. Hill, and P. Canning. Inheritance is not Subtyping. In *Proc. of ACM Symp. POPL'90*, pages 125–135. ACM Press, 1990.

[14] W.R. Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

[15] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[16] K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proc. of FCT'95*, volume 965 of *Lecture Notes in Computer Science*, pages 42–61. Springer-Verlag, 1995.

[17] M. Wand. Complete Type Inference for Simple Objects. In *Proc. of IEEE Symp. LICS'87*, pages 37–44. Silver Spring, 1987.

A Typing Rules of Our System

General Rules: $\Gamma \vdash A$ stands for any derivable judgement in the system.

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

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

($\Lleftarrow\!\#$ *var*)
$$\frac{\Gamma \vdash \tau \quad u \notin Dom(\Gamma)}{\Gamma,\, u \Lleftarrow\!\# \ \tau \vdash *}$$

Rules for Types: In the degenerate case $k = 0$, the premise of rule (*type pro*) is $\Gamma \vdash *$.

(*type proj*)
$$\frac{\Gamma', u \Lleftarrow\!\# \ \tau, \Gamma'' \vdash *}{\Gamma', u \Lleftarrow\!\# \ \tau, \Gamma'' \vdash u}$$

(*type arrow*)
$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 {\rightarrow} \tau_2}$$

(*type pro*)
$$\frac{\Gamma, t \Lleftarrow\!\# \ \mathtt{pro}\, t.\langle\!\langle\rangle\!\rangle \vdash \tau_i}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle m_1{:}\tau_1, \ldots, m_k{:}\tau_k \rangle\!\rangle}$$

Rules for Terms

(*projection*)
$$\frac{\Gamma', x : \tau, \Gamma'' \vdash *}{\Gamma', x : \tau, \Gamma'' \vdash x : \tau}$$

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

(*app*)
$$\frac{\Gamma \vdash e_1{:}\tau_1 {\rightarrow} \tau_2 \quad \Gamma \vdash e_2{:}\tau_1}{\Gamma \vdash e_1 e_2{:}\tau_2}$$

$$(empty) \qquad \frac{\Gamma \vdash *}{\Gamma \vdash \langle\rangle : \mathtt{pro}\, t.\langle\!\langle\rangle\!\rangle}$$

$$(ext) \qquad \frac{\begin{array}{l}\Gamma \vdash e_1 : \mathtt{pro}\, t.\langle\!\langle \vec{m}{:}\vec{\rho} \rangle\!\rangle \\ \Gamma, u \lll\!\# \ \mathtt{pro}\, t.\langle\!\langle \vec{m}{:}\vec{\rho}, n{:}\tau \rangle\!\rangle \vdash e_2 : [u/t](t \to \tau) \qquad n \notin \{\vec{m}\}\end{array}}{\Gamma \vdash \langle e_1 \longleftarrow\!\!\!+ \ n{=}e_2 \rangle : \mathtt{pro}\, t.\langle\!\langle \vec{m}{:}\vec{\rho}, n{:}\tau \rangle\!\rangle}$$

$$(over) \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : \sigma \qquad \Gamma \vdash \sigma \lll\!\# \ \mathtt{pro}\, t.\langle\!\langle \vec{m}{:}\vec{\rho}, n{:}\tau \rangle\!\rangle \\ \Gamma, u \lll\!\# \ \mathtt{pro}\, t.\langle\!\langle \vec{m}{:}\vec{\rho}, n{:}\tau \rangle\!\rangle \vdash e_2 : [u/t](t \to \tau)\end{array}}{\Gamma \vdash \langle e_1 \leftarrow n{=}e_2 \rangle : \sigma}$$

$$(send) \qquad \frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash \sigma \lll\!\# \ \mathtt{pro}\, t.\langle\!\langle n{:}\tau \rangle\!\rangle}{\Gamma \vdash e \Leftarrow n : [\sigma/t]\tau}$$

$$(search) \qquad \frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash \sigma \lll\!\# \ \mathtt{pro}\, t.\langle\!\langle n{:}\tau \rangle\!\rangle \quad \Gamma \vdash \varsigma \lll\!\# \ \sigma}{\Gamma \vdash e \hookleftarrow n : [\varsigma/t](t \to \tau)}$$

Rules for Matching: Rule ($\lll\!\#$ *refl*) has the proviso that τ is either a `pro`-type or a type variable.

$$(\lll\!\# \ \ proj) \qquad \frac{\Gamma', u \lll\!\# \ \tau, \Gamma'' \vdash *}{\Gamma', u \lll\!\# \ \tau, \Gamma'' \vdash u \lll\!\# \ \tau}$$

$$(\lll\!\# \ \ refl) \qquad \frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \lll\!\# \ \tau}$$

$$(\lll\!\# \ \ trans) \qquad \frac{\Gamma \vdash \sigma \lll\!\# \ \tau \quad \Gamma \vdash \tau \lll\!\# \ \rho}{\Gamma \vdash \sigma \lll\!\# \ \rho}$$

$$(\lll\!\# \ \ pro) \qquad \frac{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \mid m{:}\tau \rangle\!\rangle}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \mid m{:}\tau \rangle\!\rangle \lll\!\# \ \mathtt{pro}\, t.R}$$

B Typing Rules of [FHM94]

General Rules: $\Gamma \vdash A$ stands for any derivable judgement in the system; $U : V$ stands for either $t : T$, or $r : \kappa$, or $x : \tau$.

(*start*)
$$\frac{}{\varepsilon \vdash\!\!\div\; *}$$

(*exp var*)
$$\frac{\Gamma \vdash\!\!\div\; \tau \quad x \notin Dom(\Gamma)}{\Gamma,\, x{:}\tau \vdash\!\!\div\; *}$$

(*type var*)
$$\frac{\Gamma \vdash\!\!\div\; * \quad t \notin Dom(\Gamma)}{\Gamma,\, t{:}T \vdash\!\!\div\; *}$$

(*row var*)
$$\frac{r \notin Dom(\Gamma) \quad \Gamma \vdash\!\!\div\; *}{\Gamma, r : T^n \rightarrow [m_1, \ldots, m_k] \vdash\!\!\div\; *}$$

(*projection*)
$$\frac{\Gamma \vdash\!\!\div\; * \quad U{:}V \in \Gamma}{\Gamma \vdash\!\!\div\; U{:}V}$$

(*weakening*)
$$\frac{\Gamma \vdash\!\!\div\; A \quad \Gamma, \Gamma' \vdash\!\!\div\; *}{\Gamma, \Gamma' \vdash\!\!\div\; A}$$

Rules for Types

(*type arrow*)
$$\frac{\Gamma \vdash\!\!\div\; \tau_1 : T \quad \Gamma \vdash\!\!\div\; \tau_2 : T}{\Gamma \vdash\!\!\div\; \tau_1 \rightarrow \tau_2 : T}$$

(*type pro*)
$$\frac{\Gamma,\, t{:}T \vdash\!\!\div\; R : [m_1, \ldots, m_k]}{\Gamma \vdash\!\!\div\; \mathtt{pro}\, t.R : T}$$

Type and Row Equality

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

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

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

Rules for Rows

$(row\ empty)$ $$\frac{}{\Gamma \mathrel{\vdash\!\!\div} \langle\!\langle\rangle\!\rangle : [m_1, \ldots, m_k]}$$

$(row\ label)$ $$\frac{\Gamma \mathrel{\vdash\!\!\div} R : T^n \!\rightarrow\! [m_1, \ldots, m_k] \quad \{n_1, \ldots, n_l\} \subseteq \{m_1, \ldots, m_k\}}{\Gamma \mathrel{\vdash\!\!\div} R : T^n \!\rightarrow\! [n_1, \ldots, n_l]}$$

$(row\ ext)$ $$\frac{\Gamma \mathrel{\vdash\!\!\div} R : [n, m_1, \ldots, m_k] \quad \Gamma \mathrel{\vdash\!\!\div} \tau : T}{\Gamma \mathrel{\vdash\!\!\div} \langle\!\langle R \mid n{:}\tau \rangle\!\rangle : [m_1, \ldots, m_k]}$$

$(row\ abs)$ $$\frac{\Gamma, t : T \mathrel{\vdash\!\!\div} R : T^n \!\rightarrow\! [m_1, \ldots, m_k]}{\Gamma \mathrel{\vdash\!\!\div} \lambda t.R : T^{n+1} \!\rightarrow\! [m_1, \ldots, m_k]}$$

$(row\ app)$ $$\frac{\Gamma \mathrel{\vdash\!\!\div} R : T^{n+1} \!\rightarrow\! [m_1, \ldots, m_k] \quad \Gamma \mathrel{\vdash\!\!\div} \tau : T}{\Gamma \mathrel{\vdash\!\!\div} R\tau : T^n \!\rightarrow\! [m_1, \ldots, m_k]}$$

Rules for Terms

(abs) $$\frac{\Gamma,\, x{:}\tau_1 \vdash\!\!\div\; e{:}\tau_2}{\Gamma \vdash\!\!\div\; \lambda x.e{:}\tau_1 \to \tau_2}$$

(app) $$\frac{\Gamma \vdash\!\!\div\; e_1{:}\tau_1 \to \tau_2 \quad \Gamma \vdash\!\!\div\; e_2{:}\tau_1}{\Gamma \vdash\!\!\div\; e_1 e_2{:}\tau_2}$$

$(empty)$ $$\frac{\Gamma \vdash\!\!\div\; \langle\!\langle\rangle\!\rangle \; : [m_1, \ldots, m_k]}{\Gamma \vdash\!\!\div\; \langle\rangle : \mathtt{pro}\, t.\langle\!\langle\rangle\!\rangle}$$

(ext) $$\frac{\begin{array}{l}\Gamma \vdash\!\!\div\; e_1 : \mathtt{pro}\, t.\langle\!\langle R \mid \vec{m{:}\tau} \rangle\!\rangle \\ \Gamma, t : T \vdash\!\!\div\; R : [m_1, \ldots, m_k, n] \\ \Gamma,\, r : T \to [\vec{m}, n] \vdash\!\!\div\; e_2 : [\mathtt{pro}\, t.\langle\!\langle rt \mid \vec{m{:}\tau}, n{:}\tau \rangle\!\rangle / t](t \to \tau) \qquad r \textit{ not in } \tau\end{array}}{\Gamma \vdash\!\!\div\; \langle e_1 \leftarrow\!\!+\, n{=}e_2 \rangle : \mathtt{pro}\, t.\langle\!\langle R \mid \vec{m{:}\tau}, n{:}\tau \rangle\!\rangle}$$

$(over)$ $$\frac{\begin{array}{l}\Gamma \vdash\!\!\div\; e_1 : \mathtt{pro}\, t.\langle\!\langle R \mid \vec{m{:}\tau} \rangle\!\rangle \\ \Gamma,\, r : T \to [\vec{m}] \vdash\!\!\div\; e_2 : [\mathtt{pro}\, t.\langle\!\langle rt \mid \vec{m{:}\tau} \rangle\!\rangle / t](t \to \tau)\end{array}}{\Gamma \vdash\!\!\div\; \langle e_1 \leftarrow m_i{=}e_2 \rangle : \mathtt{pro}\, t.\langle\!\langle R \mid \vec{m{:}\tau} \rangle\!\rangle}$$

$(send)$ $$\frac{\Gamma \vdash\!\!\div\; e : \mathtt{pro}\, t.\langle\!\langle R \mid m{:}\tau \rangle\!\rangle}{\Gamma \vdash\!\!\div\; e \Leftarrow m : [\mathtt{pro}\, t.\langle\!\langle R \mid m{:}\tau \rangle\!\rangle / t]\tau}$$