*Fundamenta Informaticae XX (2001) 1–40*
*IOS Press* 

# A Subtyping for Extensible, Incomplete Objects

**To Helena Rasiowa: in memoriam**

**Viviana Bono***
*Dipartimento di Informatica*
*Università di Torino*
*C.so Svizzera 185, I-10149 Torino, Italy*
*bono@di.unito.it*

**Michele Bugliesi†**
*Dipartimento di Matematica*
*Università di Padova*
*Via Belzoni 7, I-35131 Padova, Italy*
*michele@math.unipd.it*

**Mariangiola Dezani-Ciancaglini‡**
*Dipartimento di Informatica*
*Università di Torino*
*C.so Svizzera 185, I-10149 Torino, Italy*
*dezani@di.unito.it*

**Luigi Liquori§**
*Dipartimento di Matematica ed Informatica*
*Università di Udine*
*Via delle Scienze 206, I-33100 Udine, Italy*
*liquori@dimi.uniud.it*

---

**Abstract.** We extend the type system for the *Lambda Calculus of Objects* [16] with a mechanism of *width* subtyping and a treatment of incomplete objects. The main novelties over previous work are the use of subtype-bounded quantification to capture a new and more direct rendering of *MyType* polymorphism, and a uniform treatment for other features that were accounted for via different systems in subsequent extensions [7, 6] of [16]. The new system provides for ($i$) appropriate type specialization of inherited methods, ($ii$) static detection of errors, ($iii$) width subtyping compatible with object extension, and ($iv$) sound typing for partially specified objects.

**Keywords:** Objects, Type System, Subtyping, Type Soundness.

## 1. Introduction

In the last ten years, many theoretical studies have addressed the problem of deriving safe and flexible type systems for object-oriented languages. The interest of these studies has initially been centered around *class-based* languages like *Smalltalk* [18]; later it has been directed towards *object-based* languages, such as *Self* [25]. Class-based and object-based languages are distinguished by a main conceptual difference in the underlying object-oriented models. 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.

Despite this difference, the research on object-based languages has greatly benefited from the experience gained on class-based languages. For instance, the typing of extensible objects in [16, 7, 17, 6, 24, 21] relies essentially on the same notion of *row-variable* introduced by [26] to type extensible records. Similarly, the notion of *recursive record-types*, introduced to provide functional models of class-based languages [12, 9, 14, 13], has then been refined into that of *Self-types* in type systems for object calculi supporting method override and object subsumption [3].

A further notion that originated from the work of class-based languages is that of *bounded quantification*. First introduced in [12], this notion has then been refined in several papers to give a denotationally sound rendering of the class-extension mechanism underlying subclassing. In [13] and [8], subclassing was modeled in terms of the related notions of *F-bounded quantification* and *matching*. A higher-order variant of bounded quantification has subsequently been exploited to provide for a mechanism of class-extension in the Object Calculus of [3].

In this paper we study the role of bounded quantification in modeling inheritance for the *Lambda Calculus of Objects* [16], a pure object-based language, where (*i*) method extension occurs at the object level rather than at the class level, and (*ii*) inheritance is rendered in terms of delegation between (prototypical) objects. The type system we present in this paper develops on the original work of [16] and subsequent extensions [7, 6]. We next briefly review these proposals and discuss the relations with our present approach.

The *Lambda Calculus of Objects* is an untyped $\lambda$-calculus enriched with object forms and three primitive operations on objects: *method addition*, to define new methods, *method override*, to redefine methods, and *method call*, to send a message to (i.e., invoke a method on) an object. In [16] 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 the types of methods to be specialized to the types 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 polymorphism which, in turn, uses implicit quantification over *row-schemes* to capture the underlining notion of *protocol extension*.

In [7], an extension of the system of [16] is presented that gives provision for *width* subtyping. The subtype relation arises from using *labeled types* to allow methods to be "hidden" from the type of an object, provided that "hidden" methods are not referenced to by other methods in the type.

In [6], an orthogonal extension of [16] is proposed that allows objects to be typed independently of the order of their method additions (in [16], the addition of a method $m$ to an object can be typed only if all the methods that are referenced to – via message sends or method overrides – in the body of $m$ are already available from that object). That flexibility arises in [6] from introducing the notion of *completion*, a complement to *interface*, to convey information on (the types of) the methods that are not available from the object, and yet are referenced to by the methods of the interface. Besides allowing a more flexible typing of methods (in particular, of mutually recursive method definitions), this extension also allows methods to be invoked on *incomplete* objects, i.e. objects whose implementation (the set of their methods) is only partially specified.

The approach we take in this paper combines the notion of subtyping proposed in [7] with the support for incomplete objects distinctive of [6]. The combination of these two features yields a relation of subtyping in width that $(i)$ supports a safe notion of object subsumption in the presence of primitives that extend the structure of an object, and $(ii)$ allows methods to be hidden from an object without requiring the covariance constraint on the occurrences of the bound variable distinctive of the standard subtype rules for recursive record-types.

The features that are accounted for by the type systems can thus be summarized as follows:

- appropriate method specialization of inherited methods;
- static detection of errors, such as *message-not-understood*;
- width subtyping compatible with method extension;
- sound typing for partially specified objects.

The main technical novelties over previous work are a uniform treatment for the above features, that were accounted via different systems in [16, 7, 6], and a new and more direct rendering of *MyType* polymorphism for method bodies. In these proposals, *MyType* specialization is captured in terms of the notions of *higher-order row-variables* and *row-application* which, in turn, require a rule of $\beta-$reduction in the calculus of rows and types. Here, instead, method specialization is rendered directly in terms of subtyping and (implicit) bounded quantification. Technically, the new solution is based on allowing, within our contexts, occurrences of type variables that are subtypes of suitable class-types (i.e. types of objects). These type variables are used to characterize methods as polymorphic functions, while relying on the subtyping constraints to ensure correct instantiations of the method types as these methods get inherited. Although the original approach of [16] appears superior to the present one in terms of a possible encoding in *LF* [19], the new system does have the advantage of reducing the technical overhead of the calculus of rows and types in [16] and subsequent extensions, and hence to allow a simpler and more direct proof of Subject Reduction.

A few additional remarks are in order on the practical relevance of the type system. The combination of subtyping with object extension, as well as the support for sound typing of incomplete objects, are both important in modeling real languages. In particular, the ability to type partially specified objects has two interesting applications: firstly, it is clearly central for rapid prototyping, in that it allows prototypes to be defined, and operated with as well, while part of their implementation (i.e., part of their methods) is yet to be defined: secondly, as we show with an example (Example 4.2), it may be exploited in modeling language constructs such as virtual methods in class-based languages.

The rest of the paper is organized as follows. In Section 2, we briefly overview the untyped calculus of [16], and define an operational semantics. In Section 3, we present the new typing rules for objects. Some motivating examples are presented in Section 4, while, in Section 5, we give a proof of type soundness. In Section 6 we discuss the role of labeled types in our system and outline an algorithm for inferring labels for types from label-free derivations. We conclude in Section 7 with some additional remarks, and a discussion on related papers. Two separate appendices collect the set of typing rules, and the definitions needed for label inference.

A preliminary version of this paper appeared in [5].

## 2. The Untyped Calculus

An expression of the untyped calculus can be any of the following:

$$e \;::=\; x \mid \lambda x.e \mid e_1\, e_2 \mid \langle\rangle \mid e \Leftarrow m \mid \langle e_1 \leftarrow\!\circ\; m{=}e_2\rangle \mid e \hookleftarrow m,$$

where $x$ is a variable and $m$ is a method name. As usual, expressions that differ only in the names of bound variables are identified. The reading of the object-related forms is as follows:

- $\langle\rangle$ is the empty object;
- $e \Leftarrow m$ sends message $m$ to object $e$;
- $\langle e_1 \leftarrow\!\circ\; m{=}e_2\rangle$ extends $e_1$ with a new method $m$ (or redefines the existing body of $m$ in $e_1$) with body $e_2$;
- $e \hookleftarrow m$ searches the body of the method $m$ within object $e$; this form should be viewed as a subsidiary expression that does not pertain to the syntax of the calculus, but rather is functional to the definition of the operational semantics.

Unlike [16], our calculus relies on a single operator, $\leftarrow\!\circ$ , for modifying the structure of an object. The expression $\langle e_1 \leftarrow\!\circ\; m{=}e_2\rangle$ can be typed independently of whether $e_1$ has a type whose interface contains the method $m$ or not; if it does contain the method $m$, $\leftarrow\!\circ$ is operationally an override, otherwise it is an extension. As we shall see (cf. Section 3.2), the rules for subtyping allow a uniform treatment of the operations of addition and override, that were instead distinguished in [16, 7, 17, 6]. Finally, owing to subtyping, methods may be added to the same object more than once, provided that they are *hidden* from the interface of the object type prior to a new addition.

The other main operation on objects is method invocation, whose semantics is defined by self-application: the result of sending a message $m$ to an object $e$ containing a method $m$ is the result of applying the body of $m$ to the object $e$ itself. To account for this behavior, a mechanism is needed for extracting the appropriate method out of an object. As suggested in [16], a natural way to approach this is to use a permutation rule like the following:

Table 1. Top-level Reduction Rules

| | | | |
|---|---|---|---|
| $(\beta)$ | $(\lambda x.e_1)\, e_2$ | $\longrightarrow$ | $[e_2/x]\, e_1$ |
| $(\Leftarrow)$ | $e \Leftarrow m$ | $\longrightarrow$ | $(e \hookleftarrow m)\, e$ |
| $(\hookleftarrow\ succ)$ | $\langle e_1 \longleftarrow\!\circ\ m{=}e_2\rangle \hookleftarrow m$ | $\longrightarrow$ | $e_2$ |
| $(\hookleftarrow\ next)$ | $\langle e_1 \longleftarrow\!\circ\ n{=}e_2\rangle \hookleftarrow m$ | $\longrightarrow$ | $e_1 \hookleftarrow m$ |

$$\langle\langle e \longleftarrow\!\circ\ m{=}e_1\rangle \longleftarrow\!\circ\ n{=}e_2\rangle \;=\; \langle\langle e \longleftarrow\!\circ\ n{=}e_2\rangle \longleftarrow\!\circ\ m{=}e_1\rangle \qquad (m \neq n),$$

and define the semantics of method invocation as a reduction from the message send $\langle e_1 \longleftarrow\!\circ\ m{=}e_2\rangle \Leftarrow m$ to the application $e_2\, \langle e_1 \longleftarrow\!\circ\ m{=}e_2\rangle$. In [6], it was shown that this form of method permutation can soundly be accounted for in a system without subtyping. Unfortunately, in the presence of subtyping, permuting the order of two method additions within an object may change the type of the object, thus making the above equation unsound (see Example 4.4).

Therefore, in the definition of the operational semantics, we adopt a different solution that uses the search operator '$\hookleftarrow$' to inspect the structure of objects and to perform method extraction. The core of the operational semantics is given in Table 1.
Rule $(\beta)$ is standard, while the remaining rules formalize the semantics of method invocation: evaluating $e \Leftarrow m$ leads to evaluating the application $(e \hookleftarrow m)\, e$, where $e \hookleftarrow m$ returns the body of the method $m$ that is then applied to $e$ itself. Method search is performed by a recursive traversal of the "sub-objects" of $e$ that succeeds upon reaching the rightmost addition of the method in question. The use of search expressions in our calculus is inspired by [7], and it provides a more direct technical device than the *bookkeeping* relation originally introduced in [16].

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 $\xrightarrow{eval}$ as follows:

$$e_1 \xrightarrow{eval} e_2 \qquad \text{iff} \quad e_1 \equiv C[e_1'],\ \ e_2 \equiv C[e_2'], \text{ and} \quad e_1' \longrightarrow e_2'.$$

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

## 2.1. 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 $\lambda$-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 closed $\lambda$-abstraction, or a constant (the empty object $\langle\rangle$), or a closed object expression of the form $\langle e_1 \longleftarrow\!\circ\ m{=}e_2\rangle$.

Table 2. Operational Semantics

**Evaluation**

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

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

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

**Search**

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

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

The evaluation strategy is defined in terms of two mutually recursive relations: $\Downarrow$ and $\downarrow$. The actual evaluation relation is $\Downarrow$, that reduces expressions to values; $\downarrow$ 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 rules in Table 2 (*val* denotes an arbitrary value, and *obj* an object expression of the form $\langle e_1 \hookleftarrow\!\!\circ\ m{=}e_2\rangle$).

Observe that the operational semantics is deterministic: if $e \Downarrow val_1$ and $e \Downarrow val_2$, then $val_1 \equiv val_2$. Moreover, this semantics is interesting since it immediately suggest an algorithm for reduction (i.e., an interpreter for our calculus). Furthermore, both $\Downarrow$ and $\downarrow$ are contained in $\xrightarrow{eval}\!\!\!\!\rightarrow$. More formally:

**Proposition 2.1.** If $e \Downarrow val$, then $e \xrightarrow{eval}\!\!\!\!\rightarrow val$. Similarly, if $e \downarrow e'$, then $e \xrightarrow{eval}\!\!\!\!\rightarrow e'$. □

This proposition will be useful in the proof of type soundness for the operational semantics (cf. Section 5). Given that both $\Downarrow$ and $\downarrow$ are contained in $\xrightarrow{eval}\!\!\!\!\rightarrow$, we can $(i)$ prove a single subject reduction theorem, for $\xrightarrow{eval}\!\!\!\!\rightarrow$, and then $(ii)$ use the immediate consequence that both $\Downarrow$ and $\downarrow$ preserve types to prove the absence of stuck states.

# 3. The Type System

The definition of the type system is centered around the notion of incomplete objects [6]. Incomplete objects behave operationally as "standard" objects whose methods may be invoked via corresponding messages. Their typing, instead, is different, in that an incomplete object may be typed even though it contains references (via message sends or overrides) to methods that are yet to be added to the object. The type of an incomplete object is defined by a type expression of the form:

$$\texttt{class}\, t.\langle\!\langle m_1{:}\alpha_1, \ldots, m_k{:}\alpha_k \rangle\!\rangle \circ \langle\!\langle p_1{:}\gamma_1, \ldots, p_l{:}\gamma_l \rangle\!\rangle,$$

where the $m_i$'s and $p_i$'s are method names, whereas the $\alpha_i$'s and the $\gamma_i$'s are *labeled* types (whose role is discussed below). Given the above class-type, we refer to the two *rows* $\langle\!\langle m_1{:}\alpha_1, \ldots, m_k{:}\alpha_k \rangle\!\rangle$ and $\langle\!\langle p_1{:}\gamma_1, \ldots, p_l{:}\gamma_l \rangle\!\rangle$ as, respectively, the *interface* and the *completion* of the type. The binder `class` scopes over the two rows, and the bound variable $t$ may occur free within the scope of the binder, with every free occurrence referring to the class-type itself.

The interface of a class-type describes all the methods (and their types) that have been added to the objects of that type. The completion, instead, conveys information on (the types of) the methods that have not been added to an object, and yet are (may be) referenced by the methods already available from that object. Thus, intuitively, the completion of a class-type lists all of the methods that are needed to "complete" objects with that type.

The typing of incomplete objects relies in an essential way on the information encoded in the labels associated to the types of methods. Labeled types bear the same meaning as in [7]: if $\alpha \equiv \tau_\Delta$ is the labeled type of a method $m$ within an object, then $\Delta$ lists the names of other methods of that object upon which the body of $m$ may depend. Unlike [7], however, in our system labels contain also indirect dependencies: the label $\Delta$ above contains the names of the methods referenced by $m$ in a send or an override for *self* [1], together with the methods referenced by these methods, and so on. This encoding still allows new types to be derived, by subtyping, from a given class-type. Subtyping is based on the standard idea of *hiding* methods from the type of an object (*width* subtyping). The difference, here, is that hiding is constrained by labels: a method may be hidden from the interface or the completion of a type only if it does not occur in the labeled type of any of the remaining methods of the interface of the type.

## 3.1. Types and Rows

Types include type variables, function types and class-types. Throughout the paper we use the following conventions: types are denoted by Greek letters such as $\tau$, $\sigma$, $\varsigma$, $\mu$, ..., whereas labeled types are denoted by $\alpha$, $\gamma$, $\eta$, .... Type variables are denoted by $t$ as well as by the letters $u$ and $v$: typically $t$ will be the variable associated with the binder `class`, while $u$ and $v$ are used to indicate subtype-bound type variables declared in a context

[1] Following the terminology of [10], invoking or overriding a method on the self-parameter are *self-inflicted* operations.

Table 3. Syntax of Types and Rows

| | | | | |
|---|---|---|---|---|
| Labels | $\Delta$ | $::=$ | $\{m_1, \ldots, m_k\}$ | $k \geq 0$ |
| Rows | $R$ | $::=$ | $\langle\!\langle\rangle\!\rangle \mid \langle\!\langle R \mid m{:}\tau_\Delta \rangle\!\rangle$ | with $m \notin \mathcal{M}(R), m \notin \Delta$ |
| Types | $\tau$ | $::=$ | $t \mid \tau \to \tau \mid \texttt{class}\, t.R_1 \circ R_2$ | with $\mathcal{M}(R_1) \cap \mathcal{M}(R_2) = \{\}$ and $\mathcal{M}(R_1) \cup \mathcal{M}(R_2) \supseteq \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$ |

(see below). All symbols may appear indexed by integers, and the vector notation "$\overline{\phantom{x}}$" has the usual meaning.

Labels, rows, and types are defined inductively in Table 3. $\mathcal{M}(R)$ denotes the set of method names of a row $R$, i.e.:

$$\mathcal{M}(\langle\!\langle\rangle\!\rangle)=\{\}, \qquad \text{and} \qquad \mathcal{M}(\langle\!\langle R \mid m{:}\tau_\Delta \rangle\!\rangle)=\mathcal{M}(R) \cup \{m\}.$$

$\mathcal{L}(R)$ denotes the set of method names occurring in the labels of types in $R$, i.e:

$$\mathcal{L}(\langle\!\langle\rangle\!\rangle)=\{\}, \qquad \text{and} \qquad \mathcal{L}(\langle\!\langle R \mid m{:}\tau_\Delta \rangle\!\rangle)=\mathcal{L}(R) \cup \Delta.$$

The conditions on $\texttt{class}\, t.R_1 \circ R_2$ guarantee $(i)$ that the same method name does not occur twice either in the interface or in the completion of an object type, and $(ii)$ that all the method names occurring in the labels occur in the rows as well.

Row expressions that differ only for the order of $m{:}\alpha$ pairs, or for the name of the type variable bound by `class` are considered identical. Although the interface and completion of a class-type are structurally equivalent, we will often find it convenient to distinguish their role by choosing different symbols, namely, $R$ and $C$ stand for arbitrary interfaces and completions, respectively.

Note that, unlike previous proposals, [16, 7, 6], the structure of our types is given independently of notions such as row variables, higher-order rows, applications of rows to types, and kinds. This allows a simplification over these proposals as, having no $\beta$-reduction for types, our type derivations are in normal form by construction.

The contexts of the type system list types for term variables and subtype bounds for type variables:

$$\Gamma \quad ::= \quad \varepsilon \mid \Gamma,\, x : \tau \mid \Gamma,\, u \preceq \texttt{class}\, t.R \circ C.$$

The bounds for type variables declared in a context must be class-types: this choice is motivated by the structure of the typing rules, where the subtype bounds are functional only to the polymorphic typing of method bodies (see the typing rules in Section 3.3 below).

The judgments of the type system are the following:

$$\Gamma \vdash * \qquad \Gamma \vdash e : \tau \qquad \Gamma \vdash \tau_1 \preceq \tau_2,$$

where $\Gamma \vdash *$ can be read as "$\Gamma$ is a well-formed context" and the meanings of the other judgments are the usual ones. Table A.1 shows the formation rules for contexts. We denote with $Var(\tau)$, for any type $\tau$, the set of type variables occurring in $\tau$, and with $Dom(\Gamma)$ the domain of the context $\Gamma$ (which includes term variables and type variables); the notation $\not\in$ stands for "does not occur in", and the notation $\equiv$ (respectively $\not\equiv$) denotes syntactic equality (resp. inequality) modulo renaming of bound variables between terms, types, labels, labeled types, or contexts.

**Remark 3.1.** In writing the typing rules for contexts and the rules for subtyping, we assume that all the types occurring in the judgments of the rules conform with the syntax of types. In contrast, in writing the typing rules for terms, only the types in the assumptions are assumed to be well-formed, while the correctness of the type in the conclusion follows from this assumption and, when needed, from the side-conditions for the rules.

### 3.2. Subtyping Rules

The subtype relation, defined in Table A.2, combines the standard rules for reflexivity, transitivity and for the arrow-type constructor (that is contravariant in its domain), with two additional rules that define subtyping over class-types.

The first rule, ($\preceq$ *shift*), states that moving a method from the interface to the completion of a given class-type generates a supertype of the given type.

The ($\preceq$ *hide*) rule, in turn, states that a given class-type can be generalized to any other type that results from hiding methods from the completion. From the formation rules for types, it follows that the methods that are "hidden" are not referenced by the remaining methods of the interface and the completion. In fact, by definition, we have that $\mathcal{M}(R) \cup \mathcal{M}(C) \supseteq \mathcal{L}(R) \cup \mathcal{L}(C)$, and this, in turn, implies that $\overline{p} \not\in \mathcal{L}(R) \cup \mathcal{L}(C)$.

Width subtyping over class-types may be accounted for by the combination of ($\preceq$ *shift*) and ($\preceq$ *hide*): methods of the interface of a class-type may be hidden by first moving them to the completion. The resulting subtype relation is the same as in [7]: a set of methods may be *hidden* from the interface of a class-types only if no method in the set occurs in the dependency set of the remaining methods. Since labels are enforced to provide a sound representation of the dependencies of a method (see the (*ext*), (*comp*), (*over*) rules in the next subsection), hiding of methods may safely be accomplished by looking at the method labels without imposing the covariance constraints on the occurrences of the bound variable distinctive to the standard subtype rules for recursive record types.

### 3.3. Typing Rules

The typing rules of the system, presented in Table A.3, include the standard rules for the untyped lambda calculus, as well as a subsumption rule ($\preceq$) used in conjunction with the subtype relation to account for type promotion. The rules for object expressions are described next.

The first rule, (*empty*), should be self-explanatory: since the empty object contains no method, it needs no further method to be completed.

The (*send*) rule types a method invocation. A call to a method $n$ on an object $e$ requires that the interface of $e$ contain the method $n$, to be called, as well as all of the $\{\overline{m}\}$ methods upon which $n$ may depend: from the definition of the subtype relation, it follows that every type $\sigma \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\tau_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$ satisfies both these constraints. The result of the invocation has the type $[\sigma/t]\tau$ that results from substituting $\sigma$, the type of $e$, for $t$ in $\tau$. As in [16], this substitution reflects the recursive nature of class-types.

The (*search*) rule, for typing a search expression, is similar to the previous, but slightly more general, because the search of a method encompasses a recursive inspection of the recipient object (i.e. of *self*). Given that $e \hookleftarrow n$ arises as a result of reducing a message send, the type $\sigma$ that is substituted for $t$ in the conclusion is the type of the recipient of the message: as such, this type is subject to the same constraint imposed in the (*send*) rule over the type of the recipient. On the other hand, the object $e$ is the sub-object of the recipient where the body of method $n$ is eventually found: for this sub-object, we require that it contain $n$ in its interface, as well as the dependencies $\overline{m{:}\alpha}$ of $n$, either in the interface or in the completion.

A further interesting aspect of both the (*send*) and (*search*) rules is that the type $\sigma$ may either be a class-type or else an unknown type (i.e., a type variable) occurring (bounded) in the context $\Gamma$. This generality is required in the polymorphic typing of method bodies where the recipient of a message may be the *self* object, whose type is a bound type variable occurring in the context $\Gamma$ (see the rules (*ext*), (*comp*), and (*over*)).

To explain the typing rule for method addition, we distinguish the case when the method $n$, to be added, does not occur in the type of the object that gets extended, from the case when it does.

In the first case, we need to determine the labeled type of $n$, and possibly to extend the completion of the resulting class-type with new methods referenced by this method. This is accomplished by the rule (*ext*). Here, $\overline{m{:}\alpha} \in R \circ C$ indicates that the $\overline{m{:}\alpha}$ methods are contained in $R \circ C$, whereas the condition $n, \overline{p} \notin \mathcal{M}(R) \cup \mathcal{M}(C)$ is required to ensure that the type in the conclusion contains no duplicate occurrences of the methods $n$ and $\overline{p}$. The fact that this type is well-formed follows then from the assumption that the type $\mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, \overline{p{:}\gamma}, n{:}\tau_{\{\overline{m},\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$ is well-formed: in fact, one has $\{\overline{m}, \overline{p}, n\} \supseteq \mathcal{L}(\langle\!\langle \overline{p{:}\gamma} \rangle\!\rangle)$ and this implies that $\mathcal{M}(R) \cup \{n\} \cup \mathcal{M}(C) \cup \{\overline{p}\} \supseteq \mathcal{L}(\langle\!\langle \overline{p{:}\gamma} \rangle\!\rangle)$.

The intuitive reading of the (*ext*) rule is as follows. First we note that $n$ may depend on methods that are already contained in the object as well as on methods that are yet to be added. This explains why the label associated to the type of $n$ includes the methods $\overline{m}$ that are already present in the type of $e_1$, and the $\overline{p}$ that are, instead, yet to be added (and hence, included in the completion with their labeled types $\overline{\gamma}$). Note, further, that all of these methods (i.e. the methods $\overline{m}$ and $\overline{p}$) are assumed to occur in the interface of the type used as a bound for $u$ in the typing of $e_2$: this guarantees that the choice of $\{\overline{m}, \overline{p}\}$ as the label of $n$ is a sound representation of the (direct and indirect) dependencies of $n$. To see this, consider the case when $e_2$ is the expression $\lambda self.(self \Leftarrow p)$, for a given method $p$. An inspection of the (*send*) rule shows that, in order for the invocation $self \Leftarrow p$ to be typeable, the interface of the type of *self* must include not only $p$, but also all of the, say, $\overline{q}$ methods in the label of the type of $p$. But then, the label of $n$ must include $p$, a direct reference, as well as the methods $\overline{q}$ that $n$ references indirectly via $p$.

As a final and important remark, we note that, as in [16], the type of $n$ has the form $t \to \tau$ (with a class-type substituted for $t$) to conform with the self-application semantics of method invocation. The difference from [16] is in the way polymorphic types of method bodies are instantiated to allow applications to extended objects. Instead of introducing row variables, we allow applications of $e_2$ to any object of type $\rho$ with $\rho \preceq \texttt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, \overline{p{:}\gamma}, n{:}\tau_{\{\overline{m},\overline{p}\}}\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$.

The other case of method addition arises when the method $n$ occurs in the type of the object $e_1$ that is being extended. There are two possible situations that may lead to this case: either $n$ has not been added to $e_1$, but it is referenced by other methods of that object, or else $n$ has been added to $e_1$. In the first case, $n$ occurs in the completion of the type of $e_1$, and the addition is typed with the (*comp*) rule; in the second, $n$ is in the interface, and the addition is, operationally, an override typed with the (*over*) rule.

The (*comp*) rule is similar to (*ext*), with two differences: firstly, instead of *adding* the method $n$ to the interface, (*comp*) *moves* $n$ (and its type) from the completion to the interface; secondly, it does not need to add new methods to the completion, since the dependencies of $n$ are a subset of the dependencies of the method that, when added, caused $n$ to be included into the completion.

In the (*over*) rule, the bound $\texttt{class}\, t.\langle\!\langle n{:}\tau_{\{\overline{m}\}}\rangle\!\rangle \circ \langle\!\langle \overline{m{:}\alpha}\rangle\!\rangle$ for the type $\sigma$ of $e_1$ guarantees (*i*) that $e$ does contain the method $n$ in its interface, and (*ii*) the $n$ and the methods $\overline{m}$ do not depend on other methods of $R \circ C$ (this is ensured by the choice of $\overline{m}$ as the label of $n$, made when typing $e_1$). As for (*send*), $\sigma$ can be a type variable, thus allowing method overrides on the *self* object.

## 4. Examples

In this section we put forward a few examples that help illustrate the distinguishing features of our type system, and compare it with previous proposals. To ease the presentation, we use an extended syntax with (term and type) constants and predefined functions, and use the following shorthand: $\langle x = e\rangle$ for $\langle\langle\rangle \leftarrow\!\circ\, x = e\rangle$, and $\tau$ for $\tau_\Delta$ whenever $\Delta$ is empty.

### 4.1. MyType Method Specialization

The first example is borrowed from [16], and shows that the typing rules capture the desired form of method specialization. Consider the following object expression:

$$\texttt{p} \quad \stackrel{\triangle}{=} \quad \langle\langle \texttt{x} = \lambda\texttt{self}.3\rangle \leftarrow\!\circ\, \texttt{mv} = \lambda\texttt{self}.\lambda\texttt{dx}.\langle \texttt{self} \leftarrow\!\circ\, \texttt{x} = \lambda\texttt{s}.(\texttt{self} \Leftarrow \texttt{x}) + \texttt{dx}\rangle\rangle.$$

In Table 4 below, we give a derivation of the judgment:

$$\varepsilon \vdash \texttt{p} : \texttt{class}\, t.\langle\!\langle \texttt{x}{:}int, \texttt{mv}{:}(int \to t)_{\{\texttt{x}\}}\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle,$$

assuming that $\varepsilon \vdash \langle \texttt{x} = \lambda\texttt{self}.3\rangle : \texttt{class}\, t.\langle\!\langle \texttt{x}{:}int\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$ is derivable in our system. With a similar proof, we may also derive the judgment:

$$\varepsilon \vdash \langle \texttt{p} \leftarrow\!\circ\, \texttt{c} = \lambda\texttt{self}.\texttt{blue}\rangle : \texttt{class}\, t.\langle\!\langle \texttt{x}{:}int, \texttt{mv}{:}(int \to t)_{\{\texttt{x}\}}, \texttt{c}{:}col\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle,$$

that illustrates how the type of mv gets specialized as the method is inherited from a point to a colored point.

Table 4. Proof of $\varepsilon \vdash \mathtt{p} : \mathtt{class}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv}{:}(int\to t)_{\{\mathtt{x}\}}\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$ in our system

**Contexts**

$$\Gamma_0 \equiv u \preceq \mathtt{class}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv}{:}(int\to t)_{\{\mathtt{x}\}}\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \qquad \Gamma_1 \equiv \Gamma_0, \mathtt{self} : u, \mathtt{dx} : int$$
$$\Gamma_2 \equiv \Gamma_1, v \preceq \mathtt{class}\, t.\langle\!\langle \mathtt{x}{:}int\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \qquad \Gamma_3 \equiv \Gamma_2, \mathtt{s} : v$$

**Derivation**

1. $\Gamma_3 \vdash (\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} : int$
   by (*send*) from $\Gamma_3 \vdash \mathtt{self} : u$, and $\Gamma_3 \vdash u \preceq \mathtt{class}\, t.\langle\!\langle \mathtt{x}{:}int\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$.
2. $\Gamma_2 \vdash \lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} : v \to int$
3. $\Gamma_1 \vdash \langle \mathtt{self} \leftarrow\!\circ\, \mathtt{x} = \lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx}\rangle : u$
   by (*over*) from $\Gamma_1 \vdash \mathtt{self} : u$, $\Gamma_1 \vdash u \preceq \mathtt{class}\, t.\langle\!\langle \mathtt{x}{:}int\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$, and 2.
4. $\Gamma_0 \vdash \lambda \mathtt{self}.\lambda \mathtt{dx}.\langle \mathtt{self} \leftarrow\!\circ\, \mathtt{x} = \lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx}\rangle : u \to int \to u$
5. $\varepsilon \vdash \mathtt{p} : \mathtt{class}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv}{:}(int\to t)_{\{\mathtt{x}\}}\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$
   by (*ext*) from $\varepsilon \vdash \langle \mathtt{x} = \lambda \mathtt{self}.3\rangle : \mathtt{class}\, t.\langle\!\langle \mathtt{x}{:}int\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$ and 4.

To see the difference between our system and the system of [16], Table 4 should be contrasted with Table 5, where we report the original proof of the corresponding judgment in [16][2].

As it can be seen, the two derivations have essentially the same structure, but there is a key difference in the way the polymorphic type for mv is captured in the two systems. In our system, that type is polymorphic in the type variable

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

while in [16] the polymorphic type derives from the use of the extensible row inside the class-type $\mathtt{class}\, t.\langle\!\langle rt \mid \mathtt{x}{:}int, \mathtt{mv}{:}(int\to t)\rangle\!\rangle$.

## 4.2. Incomplete Objects

The next two examples show that the type system allows the typing of partially specified objects, thus also allowing complete freedom in the order of method additions.

Consider first reversing the order of the method additions in the p object of Example 4.1, and let

$$\mathtt{rp} \stackrel{\triangle}{=} \langle\langle \mathtt{mv} = \lambda \mathtt{self}.\lambda \mathtt{dx}.\langle \mathtt{self} \leftarrow\!\circ\, \mathtt{x} = \lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx}\rangle\rangle \leftarrow\!\circ\, \mathtt{x} = \lambda \mathtt{s}.3\rangle.$$

This object *cannot* be typed in the system of [16], because the sub-object

$$\mathtt{ip} \stackrel{\triangle}{=} \langle \mathtt{mv} = \lambda \mathtt{self}.\lambda \mathtt{dx}.\langle \mathtt{self} \leftarrow\!\circ\, \mathtt{x} = \lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx}\rangle\rangle$$

[2] In Table 5, $r{:}T \to [\mathtt{x}, \mathtt{mv}]$ indicates that $r$ is a higher-order row variable which, when applied to a type, returns a row not containing the methods $\mathtt{x}, \mathtt{mv}$. So $r$ can be applied to the type variable $t$.

Table 5. Proof of $\varepsilon \vdash \mathtt{p} : \mathtt{class}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv}{:}(int\to t)_{\{\mathtt{x}\}}\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$ in [14]

**Contexts**

$$\begin{array}{lll} \Gamma_0 \equiv r{:}T \to [\mathtt{x},\mathtt{mv}] & \Gamma_1 \equiv \Gamma_0, \mathtt{self}{:}\mathtt{class}\, t.\langle\!\langle rt \mid \mathtt{x}{:}int, \mathtt{mv}{:}(int\to t)\rangle\!\rangle, \mathtt{dx} : int \\ \Gamma_2 \equiv \Gamma_1, r'{:}T \to [\mathtt{x},\mathtt{mv}] & \Gamma_3 \equiv \Gamma_2, \mathtt{s}{:}\mathtt{class}\, t.\langle\!\langle r't \mid \mathtt{x}{:}int, \mathtt{mv}{:}(int\to t)\rangle\!\rangle \end{array}$$

**Derivation**

1. $\Gamma_3 \vdash (\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} : int$
2. $\Gamma_2 \vdash \lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} : \mathtt{class}\, t.\langle\!\langle r't \mid \mathtt{x}{:}int, \mathtt{mv}{:}(int\to t)\rangle\!\rangle \to int$
3. $\Gamma_1 \vdash \langle \mathtt{self} \leftarrow\!\circ\, \mathtt{x} = \lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx}\rangle : \mathtt{class}\, t.\langle\!\langle rt \mid \mathtt{x}{:}int, \mathtt{mv}{:}(int\to t)\rangle\!\rangle$
4. $\Gamma_0 \vdash \lambda \mathtt{self}.\lambda \mathtt{dx}.\langle \mathtt{self} \leftarrow\!\circ\, \mathtt{x} = \lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx}\rangle$
   $:[\mathtt{class}\, t.\langle\!\langle rt \mid \mathtt{x}{:}int, \mathtt{mv}{:}(int\to t)\rangle\!\rangle / t](t \to int \to t)$
5. $\varepsilon \vdash \mathtt{p} : \mathtt{class}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv}{:}(int\to t)\rangle\!\rangle$

is not well typed in that system. In contrast, `rp` is typeable in our system, as the judgment

$$\varepsilon \vdash \mathtt{ip} : \mathtt{class\, t}.\langle\!\langle \mathtt{mv}{:}(\mathtt{int}\to\mathtt{t})_{\{\mathtt{x}\}}\rangle\!\rangle \circ \langle\!\langle \mathtt{x}{:}\mathtt{int}\rangle\!\rangle$$

is derivable. To see this, apply steps 1–4 of Table 4, and conclude from $\varepsilon \vdash \langle\rangle : \mathtt{class}\, t.\langle\!\langle\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$ by (*ext*) and 4. Then the judgment $\varepsilon \vdash \mathtt{rp} : \mathtt{class\, t}.\langle\!\langle \mathtt{mv}{:}(\mathtt{int}\to\mathtt{t})_{\{\mathtt{x}\}}, \mathtt{x}{:}\mathtt{int}\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$ is derivable using the (*comp*) rule.

As a further example, consider the following expression, defining an object with an `x` field and a method that returns the logarithm of this field, whenever it contains a non-negative integer:

$$\begin{array}{lll} \mathtt{abs} & \stackrel{\triangle}{=} & \langle\langle\ \mathtt{x} = \lambda\mathtt{self}.\ 0\rangle \leftarrow\!\circ\, \mathtt{safe\_log} = \lambda\mathtt{self}.\ \mathtt{if}\ (\mathtt{self} \Leftarrow \mathtt{x}) \geq 0 \\ & & \qquad\qquad\qquad\qquad\qquad \mathtt{then}\ \mathtt{log}\,(\mathtt{self} \Leftarrow \mathtt{x}) \\ & & \qquad\qquad\qquad\qquad\qquad \mathtt{else}\ \mathtt{self} \Leftarrow \mathtt{handle\_ex}\ \rangle. \end{array}$$

Given this definition, the following typing is derivable in our system:

$$\varepsilon \vdash \mathtt{abs} : \mathtt{class}\, t.\langle\!\langle \mathtt{x} : int, \mathtt{safe\_log} : real_{\{\mathtt{handle\_ex}\}}\rangle\!\rangle \circ \langle\!\langle \mathtt{handle\_ex} : real\rangle\!\rangle.$$

Note that `abs` does not specify how exceptions should be handled, assuming that this method will be defined in objects derived from `abs`. Objects like `abs` are reminiscent of the notion of abstract classes in class-based languages like C++, with methods like `handle_ex` playing the role of what's referred to as *pure virtual* functions in the C++ jargon. Given the above typing, the call $\mathtt{abs} \Leftarrow \mathtt{safe\_log}$ does not type check, for the method `safe_log` depends on the method `handle_ex`, yet to be added. However, new objects can be derived from `abs`, as for instance,

$$\varepsilon \vdash \mathtt{handler} \stackrel{\triangle}{=} \langle \mathtt{abs} \leftarrow\!\circ\, \mathtt{handle\_ex} = ...\rangle$$

providing a definition of the handle_ex method such that the judgment

$$\varepsilon \vdash \mathtt{handler} \quad : \quad \mathtt{class}\, t.\langle\!\langle \mathtt{x} : int, \mathtt{safe\_log} : real_{\{\mathtt{handle\_ex}\}}, \mathtt{handle\_ex} : real \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle,$$

is derivable. Given this typing, the call $\mathtt{handler} \Leftarrow \mathtt{safe\_log}$ does type check, for the judgment $\varepsilon \vdash \mathtt{handler} \Leftarrow \mathtt{safe\_log} : real$ is derivable.

### 4.3. Transitivity of Labels

The next example motivates the requirement that the labels of method types contain both direct and indirect dependencies. Consider extending the ip object of Example 4.2 as shown below:

$$\mathtt{newip} \stackrel{\triangle}{=} \langle \mathtt{ip} \leftarrow\!\circ\, \mathtt{foo} = \lambda \mathtt{s}.\lambda \mathtt{dx}.(\mathtt{s} \Leftarrow \mathtt{move})\, \mathtt{dx} \rangle.$$

The following judgment may be derived:

$$\varepsilon \vdash \mathtt{newip} : \mathtt{class}\, t.\langle\!\langle \mathtt{move}{:}(int \to t)_{\{\mathtt{x}\}}, \mathtt{foo}{:}(int \to t)_{\{\mathtt{move},\mathtt{x}\}} \rangle\!\rangle \circ \langle\!\langle \mathtt{x}{:}int \rangle\!\rangle.$$

Note that the typing rules force the label of foo to include the method x, although foo depends on x indirectly via move. The reason why indirect references in the labels are needed should now be clear: if we only had move in the label of foo, we would be able to type the invocation $(\mathtt{newip} \Leftarrow \mathtt{foo})3$ which, instead, causes a *message-not-understood* error because newip does not contain x.

### 4.4. Permutation of Methods

The next example shows that the permutation rule of [6] for method additions does not preserve types in presence of width subtyping. As a counterexample, consider the expression

$$\mathtt{e} \stackrel{\triangle}{=} \langle \mathtt{x} = \lambda \mathtt{s}.3 \rangle.$$

Then, $\varepsilon \vdash \mathtt{e} : \mathtt{class}\, t.\langle\!\langle \mathtt{x}{:}int \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$ is derivable, and hence, by ($\preceq$ *hide*), $\varepsilon \vdash \mathtt{e} : \mathtt{class}\, t.\langle\!\langle\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$ is also derivable. Now, we can extend e with a method y of type $bool_{\{\mathtt{x}\}}$ that requires x to have the type *bool*. For instance, the following judgment is derivable:

$$\varepsilon \vdash \langle \mathtt{e} \leftarrow\!\circ\, \mathtt{y} = \lambda \mathtt{s}.\mathtt{not}(\mathtt{s} \Leftarrow \mathtt{x}) \rangle : \mathtt{class}\, t.\langle\!\langle \mathtt{y}{:}bool_{\{\mathtt{x}\}} \rangle\!\rangle \circ \langle\!\langle \mathtt{x}{:}bool \rangle\!\rangle,$$

where not is the logical negation. Clearly, it is impossible to permute the two method additions (of x and y) preserving the final type.

### 4.5. Comparison with the System of Fisher & Mitchell

A final example illustrates one interesting difference between our system and a related extension of the system of [16] presented in [17]. In [17], the subtype relation arises from introducing two distinguished sorts of object types: pro-types, and obj-types. Objects having pro-types may be freely operated with (they may be sent messages, or extended with new methods, or modified by overriding existing methods), but only trivial subtyping is allowed over pro-types. On the other hand, objects having obj-types may only respond to messages, or modify their own structure from the "inside" (i.e. via overrides on *self* within their own methods), whereas they may *not* be modified or extended from the outside.

pro-type and obj-types are ordered by the subtype relation, so as to allow pro-typed objects to be "sealed" by promoting their types to corresponding obj-types by subsumption. Type promotion from pro-types to obj-types is allowed provided that the type variable bound by the obj-type does not occur in contravariant position.

This distinction between pro- and obj-types has several interesting consequences: first it gives insights into the different nature of the inheritance and client interfaces of objects and classes in object-oriented languages; secondly, as shown in [17], it allows a quite natural modeling of method encapsulation. However, the resulting type discipline does not allow the typing of some expressions that, instead, we can deal with. To illustrate the problem, consider the following function:

$$\texttt{plot} \stackrel{\triangle}{=} \lambda\texttt{p}.\langle \texttt{p} \hookleftarrow \texttt{c} = \lambda \texttt{s}.\texttt{white} \rangle,$$

mapping one-dimensional points to colored points. The following judgment is easily derived in our type system:

$$\varepsilon \vdash \texttt{plot} : \texttt{class}\, t.\langle\!\langle \texttt{x}{:}int \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \to \texttt{class}\, t.\langle\!\langle \texttt{x}{:}int, \texttt{c}{:}col \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle.$$

Then, given a colored point cp of type, say, $\texttt{class}\, t.\langle\!\langle \texttt{x}{:}int, \texttt{c}{:}col \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$, we may safely apply plot to cp because, by subtyping, we have $\varepsilon \vdash \texttt{cp} : \texttt{class}\, t.\langle\!\langle \texttt{x}{:}int \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$.

This simple property is lost in the system of [17]. In fact, having distinguished obj- and pro-types, one may prove that:

$$\varepsilon \vdash \texttt{plot} : \texttt{pro}\, t.\langle\!\langle \texttt{x}{:}int \rangle\!\rangle \to \texttt{probj}\, t.\langle\!\langle \texttt{x} : int, \texttt{c}{:}col \rangle\!\rangle,$$

where probj is either pro or obj. On the other hand, one *cannot* prove that:

$$\varepsilon \vdash \texttt{plot} : \texttt{obj}\, t.\langle\!\langle \texttt{x}{:}int \rangle\!\rangle \to \texttt{probj}\, t.\langle\!\langle \texttt{x} : int, \texttt{c}{:}col \rangle\!\rangle.$$

This is because plot modifies its input argument with a method addition, an operation that is only allowed on pro-types. But, then, there is no way to type an application of plot to the colored point cp. In fact, one may either take cp to have type $\texttt{obj}\, t.\langle\!\langle \texttt{x}{:}int \rangle\!\rangle$ or type $\texttt{pro}\, t.\langle\!\langle \texttt{x}{:}int, \texttt{c}{:}col \rangle\!\rangle$, but, according to the subtype relation of [17], neither of these types is a subtype of $\texttt{pro}\, t.\langle\!\langle \texttt{x}{:}int \rangle\!\rangle$ (since pro-types are subtypes of obj-types, but not vice versa, and width subtyping is not allowed over pro-types).

## 5. Soundness of the Type System

The proof of soundness for the type system follows the standard pattern: we first show that types are preserved by the reduction process, i.e., that if $e_1$ has type $\tau$, and $e_1$ reduces to $e_2$, then also $e_2$ has type $\tau$; then we use this result to prove absence of stuck states in the evaluation of well-typed expressions.

### 5.1. Preliminary Lemmas

Let $A$ denote any right-hand side of a judgment in the type system, (i.e. $A$ is one of $*$, or $\sigma \preceq \tau$, or $e : \tau$), and let $a$ denote any declaration that may can occur in a context (i.e. $a$ is either $u \preceq \tau$, or $x : \tau$). Throughout this section we use the following easily verified facts:

- if $\Gamma \vdash A$ is derivable, then so is $\Gamma \vdash *$;
- if $\Gamma, \Gamma' \vdash *$ is derivable, then so is $\Gamma \vdash *$;
- if $\Gamma, u \preceq \sigma \vdash *$ is derivable, then $u \notin \Gamma$ and $u \notin \sigma$.

We first show that the following two rules are admissible:

$$\frac{\Gamma_1 \vdash A \quad \Gamma_1, \Gamma_2 \vdash *}{\Gamma_1, \Gamma_2 \vdash A}\ (\mathit{weakening}) \qquad \frac{\Gamma_1, x : \sigma, \Gamma_2 \vdash A \quad \Gamma_1 \vdash \tau \preceq \sigma}{\Gamma_1, x : \tau, \Gamma_2 \vdash A}\ (\preceq \mathit{strengthening}).$$

**Lemma 5.1.**

1. If $\Gamma_1 \vdash A$, and $\Gamma_1, \Gamma_2 \vdash *$ are both derivable, then so is $\Gamma_1, \Gamma_2 \vdash A$.
2. If $\Gamma_1, x : \sigma, \Gamma_2 \vdash A$, and $\Gamma_1 \vdash \tau \preceq \sigma$ are both derivable, then so is $\Gamma_1, x : \tau, \Gamma_2 \vdash A$.

**Proof:**

(1) We prove the following, more general statement:

($\star$) if $\Gamma_1, \Gamma_2 \vdash A$, and $\Gamma_1, a, \Gamma_2 \vdash *$ are both derivable, then so is $\Gamma_1, a, \Gamma_2 \vdash A$.

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 $\Gamma_2$.

The proof of ($\star$) is by induction on the derivation of $\Gamma_1, \Gamma_2 \vdash A$. The only interesting cases are (*i*) when $a$ is $u \preceq \tau$ and $\Gamma_1, \Gamma_2 \vdash A$ is the conclusion of either one of (*ext*), (*comp*), or (*over*), and (*ii*) when $a$ is $x : \sigma$ and $\Gamma_1, \Gamma_2 \vdash A$ is the conclusion of (*exp abs*), with $A$ of the form $\lambda x.e : \tau_1 \to \tau_2$.

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*), (*comp*), and (*over*) does not depend on the name of the type variable that is discharged by the rule.

In the case of (*ii*), the judgment in question must be derived from a judgment of the form $\Gamma_1, \Gamma_2, x : \tau_1 \vdash e : \tau_2$. Since this judgment is itself derivable, we can adapt the proof of the renaming result in [23] (developed for Pure Type Systems) to the present calculus. From this, we have that $\Gamma_1, \Gamma_2, y : \tau_1 \vdash [y/x]e : \tau_2$ is derivable for any $y \not\equiv x$ that is not contained in $\Gamma_1, \Gamma_2$.

Then, also $\Gamma_1, x : \sigma, \Gamma_2, y : \tau_1 \vdash *$ is derivable from $\Gamma_1, x : \sigma, \Gamma_2 \vdash *$ using (*exp var*). By the induction hypothesis, $\Gamma_1, x : \sigma, \Gamma_2, y : \tau_1 \vdash [y/x]e : \tau_2$ is derivable; by a further application of (*exp abs*), we then obtain a derivation of $\Gamma_1, x : \sigma, \Gamma_2 \vdash \lambda y.[y/x]e : \tau_1 \to \tau_2$, which is the judgment we wished to derive, as $\lambda y.[y/x]e \equiv \lambda x.e$.

(2) By induction on the derivation of $\Gamma_1, x : \sigma, \Gamma_2 \vdash A$. The only interesting case is when $A$ is $x : \sigma$ and $\Gamma_1, x : \sigma, \Gamma_2 \vdash x : \sigma$ is the conclusion of (*proj*). Then, $\Gamma_1, x : \tau, \Gamma_2 \vdash x : \tau$ is derivable by (*proj*) and $\Gamma_1, x : \tau, \Gamma_2 \vdash \tau \preceq \sigma$ is derivable from $\Gamma_1 \vdash \tau \preceq \sigma$ by (*weakening*), which is admissible by Lemma 5.1(1). Hence, $\Gamma_1, x : \tau, \Gamma_2 \vdash x : \sigma$ is derivable by ($\preceq$). $\square$

### 5.2. Substitution and Generation Lemmas

To prove the subject reduction theorem, we need the following substitution property for typing derivations and a generation lemma (Lemma 5.4).

**Lemma 5.2. (Substitution)** If the judgments $\Gamma_1, u \preceq \sigma, \Gamma_2 \vdash A$ and $\Gamma_1 \vdash \tau \preceq \sigma$ are both derivable, and the type $\tau$ is such that $Var(\tau) \cap Dom(\Gamma_2) = \emptyset$, then also the judgment $\Gamma_1, [\tau/u]\Gamma_2 \vdash [\tau/u]A$ is derivable.

**Proof:**

By induction on the derivation of $\Gamma_1, u \preceq \sigma, \Gamma_2 \vdash A$, and by a cases analysis on the last rule of the derivation. Most of the cases follow easily by induction: below, we give the more interesting ones.

- (*type var*) If $u$ is not the introduced variable, the rule is as follows:

$$\frac{\Gamma_1, u \preceq \sigma, \Gamma_2 \vdash * \qquad v \notin (\Gamma_1, u \preceq \sigma, \Gamma_2) \qquad v \notin \rho}{\Gamma_1, u \preceq \sigma, \Gamma_2, v \preceq \rho \vdash *} \ (\textit{type var}).$$

  Let $\tau$ be any type such that $Var(\tau) \cap Dom(\Gamma_2, v \preceq \rho) = \emptyset$. Then $Var(\tau) \cap Dom(\Gamma_2) = \emptyset$, and $v \notin \tau$. Now, by the induction hypothesis $\Gamma_1, [\tau/u]\Gamma_2 \vdash *$ is derivable, and from $v \notin \tau$ together with $v \notin (\Gamma_1, u \preceq \sigma, \Gamma_2)$, it also follows that $v \notin (\Gamma_1, [\tau/u]\Gamma_2)$. Then, the desired judgment may be derived by (*type var*).

  If $u$ is the introduced variable, the rule is as follows:

$$\frac{\Gamma_1 \vdash * \qquad u \notin \Gamma_1 \qquad u \notin \sigma}{\Gamma_1, u \preceq \sigma \vdash *} \ (\textit{type var}),$$

  and the desired judgment is simply the premise $\Gamma_1 \vdash *$.
- ($\preceq$ *proj*). We distinguish three possible subcases. If the projected subtype bound belongs to $\Gamma_1$, the rule is as follows:

$$\frac{\Gamma_1, u \preceq \sigma, \Gamma_2 \vdash * \qquad v \preceq \rho \in \Gamma_1, u \preceq \sigma, \Gamma_2}{\Gamma_1, u \preceq \sigma, \Gamma_2 \vdash v \preceq \rho} \ (\preceq \textit{proj}).$$

  By the induction hypothesis, we have that $\Gamma_1, [\tau/u]\Gamma_2 \vdash *$ is derivable. Then, $v \preceq \rho \in \Gamma_1$ implies $v \preceq \rho \in \Gamma_1, [\tau/u]\Gamma_2$ and by ($\preceq$ *proj*) $\Gamma_1, [\tau/u]\Gamma_2 \vdash v \preceq \rho$.

  If the projected subtype bound belongs to $\Gamma_2$, the rule is as follows:

$$\frac{\Gamma_1, u \preceq \sigma, \Gamma_2 \vdash * \qquad v \preceq \rho \in \Gamma_1, u \preceq \sigma, \Gamma_2}{\Gamma_1, u \preceq \sigma, \Gamma_2 \vdash v \preceq \rho} \ (\preceq \textit{proj}),$$

  and $\Gamma_1, [\tau/u]\Gamma_2 \vdash *$ is derivable by the induction hypothesis. Furthermore, since $\Gamma_1, u \preceq \sigma, \Gamma_2$ is a well-formed context, it follows that $u \not\equiv v$, and hence $v \preceq [\tau/u]\rho \in [\tau/u]\Gamma_2$. Then, the desired judgment may be derived by ($\preceq$ *proj*).

If the projected bound is $u \preceq \sigma$, the rule is as follows:

$$\frac{\Gamma_1, u \preceq \sigma, \Gamma_2 \vdash *}{\Gamma_1, u \preceq \sigma, \Gamma_2 \vdash u \preceq \sigma} \quad (\preceq proj).$$

$\Gamma_1, [\tau/u]\Gamma_2 \vdash *$ is derivable by the induction hypothesis, and $u \notin \sigma$, as $\Gamma_1, u \preceq \sigma$ is a well-formed context. Then $\Gamma_1, [\tau/u]\Gamma_2 \vdash \tau \preceq \sigma$ is derivable from $\Gamma_1 \vdash \tau \preceq \sigma$ using (*weakening*), which is admissible (by Lemma 5.1(1)).

· (*proj*). Again, we need a case analysis, depending on whether the projected declaration belongs to $\Gamma_1$ or to $\Gamma_2$. The proofs of these cases are similar to those of the corresponding cases when the last applied rule is ($\preceq proj$).

· (*send*). In this case, the rule is as follows:

$$\frac{\Gamma_1, u \preceq \sigma, \Gamma_2 \vdash e : \rho \qquad \Gamma_1, u \preceq \sigma, \Gamma_2 \vdash \rho \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\mu_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle}{\Gamma_1, u \preceq \sigma, \Gamma_2 \vdash e \Leftarrow n : [\rho/t]\mu} \quad (send).$$

We may assume $t \notin \tau$, for otherwise we may rename $t$ ($t$ being bound), so that this condition be satisfied. Then, by the induction hypothesis, we have that both $\Gamma_1, [\tau/u]\Gamma_2 \vdash e : [\tau/u]\rho$ and

$$\Gamma_1, [\tau/u]\Gamma_2 \vdash [\tau/u]\rho \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}[\tau/u]\alpha}, n{:}[\tau/u]\mu_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$$

are derivable. Then by (*send*), we have a derivation of

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

which is the judgment we wished to derive, being $[[\tau/u]\rho/t]([\tau/u]\mu) \equiv [\tau/u]([\rho/t]\mu)$. The case of (*search*) is similar to the one just proved.

· (*ext*). We prove this case as representative of the similar cases of (*comp*) and (*over*). The rule is of the following form:

$$\frac{\begin{array}{c}\Gamma_1, u \preceq \sigma, \Gamma_2 \vdash e_1 : \mathtt{class}\, t.R{\circ}C \quad \overline{m{:}\alpha} \in R \circ C \quad n, \overline{p} \notin \mathcal{M}(R) \cup \mathcal{M}(C) \\ \Gamma_1, u \preceq \sigma, \Gamma_2, v \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, \overline{p{:}\gamma}, n{:}\rho_{\{\overline{m},\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \vdash e_2 : [v/t](t {\to} \rho)\end{array}}{\Gamma_1, u \preceq \sigma, \Gamma_2 \vdash \langle e_1 \leftarrow\!\circ\, n{=}e_2 \rangle : \mathtt{class}\, t.\langle\!\langle R \mid n{:}\rho_{\{\overline{m},\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle C \mid \overline{p{:}\gamma} \rangle\!\rangle} \quad (ext).$$

Again, we may safely assume that $t \notin \tau$. By the induction hypothesis, we have that $\Gamma_1, [\tau/u]\Gamma_2 \vdash e_1 : \mathtt{class}\, t.[\tau/u]R{\circ}[\tau/u]C$, and

$$\Gamma_1, [\tau/u]\Gamma_2, v \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}[\tau/u]\alpha}, \overline{p{:}[\tau/u]\gamma}, n{:}[\tau/u]\rho_{\{\overline{m},\overline{p}\}} \rangle\!\rangle {\circ} \langle\!\langle\rangle\!\rangle \vdash e_2 : [\tau/u]([v/t](t {\to} \rho))$$

are derivable. The induction on the second premise works for we can always choose $v$ such that $v \notin \tau$, and use the renaming result of [23], since the typing of $e_2$ does not depend on the variable's name. Since $u \not\equiv v$, being $\Gamma_1, u \preceq \sigma, \Gamma_2, v \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, \overline{p{:}\gamma}, n{:}\rho_{\{\overline{m},\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \vdash *$, we get $[\tau/u]([v/t](t {\to} \rho)) \equiv [v/t]([\tau/u](t {\to} \rho))$, for $t \notin \tau$[3]. From $\overline{m{:}\alpha} \in R \circ C$ we then have $\overline{m{:}[\tau/u]\alpha} \in [\tau/u]R \circ [\tau/u]C$, while from $n, \overline{p} \notin \mathcal{M}(R) \cup \mathcal{M}(C)$ we have that $n, \overline{p} \notin \mathcal{M}([\tau/u]R) \cup \mathcal{M}([\tau/u]C)$. Then, by an application of $(ext)$ we have a derivation of

$$\Gamma_1, [\tau/u]\Gamma_2 \vdash \langle e_1 \leftarrow\!\circ\; n{=}e_2\rangle : \mathtt{class}\, t.\langle\!\langle [\tau/u]R \mid n{:}[\tau/u]\rho_{\{\overline{m},\overline{p}\}}\rangle\!\rangle \circ \langle\!\langle [\tau/u]C \mid \overline{p{:}[\tau/u]\gamma}\rangle\!\rangle,$$

which is the judgment we wished to derive, being

$$\begin{array}{c}\mathtt{class}\, t.\langle\!\langle [\tau/u]R \mid n{:}[\tau/u]\rho_{\{\overline{m},\overline{p}\}}\rangle\!\rangle \circ \langle\!\langle [\tau/u]C \mid \overline{p{:}[\tau/u]\gamma}\rangle\!\rangle \\ \equiv \\ [\tau/u](\mathtt{class}\, t.\langle\!\langle R \mid n{:}\rho_{\{\overline{m},\overline{p}\}}\rangle\!\rangle \circ \langle\!\langle C \mid \overline{p{:}\gamma}\rangle\!\rangle).\end{array}$$

□

**Remark 5.1.** The condition $Var(\tau) \cap Dom(\Gamma_2) = \emptyset$ is needed in the lemma, as it does not follow from the remaining hypotheses. Specifically, it is not a consequence of the fact that the judgment $\Gamma_1 \vdash \tau \preceq \sigma$ is derivable: to see this, observe that choosing $\Gamma_1 \equiv \emptyset$, $\tau \equiv \mathtt{class}\, t.\langle\!\langle n{:}int\rangle\!\rangle \circ \langle\!\langle m{:}v\rangle\!\rangle$ and $\sigma \equiv \mathtt{class}\, t.\langle\!\langle n{:}int\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$, the judgment $\Gamma_1 \vdash \tau \preceq \sigma$ is derived by $(\preceq\ hide)$. But we can have $\Gamma_2 \equiv v \preceq \rho$ and then $Var(\tau) \cap Dom(\Gamma_2) = \{v\}$.

The typing rules could be modified so as to ensure that $Var(\tau) \in Dom(\Gamma)$ whenever $\Gamma \vdash \tau \preceq \sigma$ is derivable. However, this would require either introducing kinding judgments to establish well-formedness for types, or cluttering the typing rules with side conditions. Neither option seems worthwhile, given that the substitution lemma is only used in the proof of subject reduction, in cases when the additional condition is trivially satisfied. (cf. Theorem 5.1, case (*search succ*)).

The proof of the Generation Lemma requires the following properties of subtyping (subtyping rules are in Table A.2).

**Lemma 5.3. (Properties of $\preceq$)**

1. If $\Gamma \vdash \sigma \to \tau \preceq \rho$ is derivable, then $\rho \equiv \sigma' \to \tau'$ for some $\sigma', \tau'$ such that $\Gamma \vdash \sigma' \preceq \sigma$ and $\Gamma \vdash \tau \preceq \tau'$ also are derivable.
2. If $\Gamma \vdash \mathtt{class}\, t.R \circ C \preceq \rho$ is derivable, then $\rho \equiv \mathtt{class}\, t.R' \circ C'$ for some $R', C'$ such that
   - $m : \alpha \in R$ whenever $m : \alpha \in R'$ and
   - $m : \alpha \in R \circ C$ whenever $m : \alpha \in C'$.
3. If $\Gamma \vdash \sigma \preceq \mathtt{class}\, t.R_1 \circ C_1$ and $\Gamma \vdash \sigma \preceq \mathtt{class}\, t.R_2 \circ C_2$ are both derivable, then for every $m$ such that $m{:}\alpha_1 \in R_1 \circ C_1$, and $m{:}\alpha_2 \in R_2 \circ C_2$, it is the case that $\alpha_1 \equiv \alpha_2$.

**Proof:**
Points (1) and (2) are proved by straightforward induction on the derivation of the judgments in the hypotheses.

[3] The condition $v \notin \tau$ is not necessary to verify the equivalence, even though we need it here to have well-formed substitutions.

For point (3), we distinguish two cases, depending on whether $\sigma$ is a class-type, or a type variable.

If $\sigma$ is a class-type, say $\mathtt{class}\,t.R{\circ}C$, the claim follows from point (2), for $m{:}\alpha_1 \in R_1{\circ}C_1$ implies that $m{:}\alpha_1 \in R \circ C$, and $m{:}\alpha_2 \in R_2 \circ C_2$ implies that $m{:}\alpha_2 \in R \circ C$. Then $\alpha_1 \equiv \alpha_2$ follows from the formation rules of class-types (that require every method to occur at most once in the component rows).

If $\sigma$ is a type variable, the proof follows as in the previous case, using the following property: if $\Gamma \vdash u \preceq \mathtt{class}\,t.R \circ C$ is derivable, then there exists $u \preceq \mathtt{class}\,t.R' \circ C' \in \Gamma$ such that also the judgment $\Gamma \vdash \mathtt{class}\,t.R' \circ C' \preceq \mathtt{class}\,t.R \circ C$ is derivable. The proof of this property is by induction on the derivation of the judgment $\Gamma \vdash u \preceq \mathtt{class}\,t.R \circ C$, using the easily verified fact that if $\Gamma \vdash u \preceq \tau$, then either $\tau \equiv u$ or $\tau \equiv \mathtt{class}\,t.R \circ C$ for some $R$ and $C$. □

**Lemma 5.4. (Generation Lemma)**

1. If $\Gamma \vdash x : \tau$ is derivable, then $x : \sigma \in \Gamma$, and $\Gamma \vdash \sigma \preceq \tau$ is derivable for some $\sigma$.
2. If $\Gamma \vdash \lambda x.e : \tau$ is derivable, then $\Gamma, x : \sigma_1 \vdash e : \sigma_2$ and $\Gamma \vdash \sigma_1 \to \sigma_2 \preceq \tau$ are both derivable for some $\sigma_1, \sigma_2$.
3. If $\Gamma \vdash e_1 e_2 : \tau$ is derivable, then $\Gamma \vdash e_1 : \tau_1{\to}\tau_2$, $\Gamma \vdash e_2 : \tau_1$, and $\Gamma \vdash \tau_2 \preceq \tau$ are all derivable for some $\tau_1, \tau_2$.
4. If $\Gamma \vdash \langle\rangle : \tau$ is derivable, then $\tau$ is $\mathtt{class}\,t.\langle\!\langle\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$.
5. If $\Gamma \vdash e \Leftarrow n : \tau$ is derivable, then

   · $\Gamma \vdash e : \sigma$, and
   · $\Gamma \vdash \sigma \preceq \mathtt{class}\,t.\langle\!\langle \overline{m{:}\alpha}, n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$, and
   · $\Gamma \vdash [\sigma/t]\rho \preceq \tau$

   are all derivable for some $\sigma, \rho, \overline{m}, \overline{\alpha}$.
6. If $\Gamma \vdash \langle e_1 \leftarrow\!\circ\, n{=}e_2 \rangle : \tau$ is derivable, then $\Gamma, u \preceq \mathtt{class}\,t.\langle\!\langle \overline{m{:}\alpha}, n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \vdash e_2 : [u/t](t{\to}\rho)$ is derivable for some $u, \rho, \overline{m}, \overline{\alpha}$, and one of the following is true:

   (a) · $\Gamma \vdash e_1 : \mathtt{class}\,t.R \circ C$ is derivable, and
   · $\overline{m{:}\alpha} \equiv \overline{q{:}\eta}, \overline{p{:}\gamma}$, and
   · $\overline{q{:}\eta} \in R \circ C$, and
   · $\Gamma \vdash \mathtt{class}\,t.\langle\!\langle R \mid n{:}\rho_{\{\overline{q},\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle C \mid \overline{p{:}\gamma} \rangle\!\rangle \preceq \tau$ is derivable
   for some $R, C, \overline{q}, \overline{p}, \overline{\eta}, \overline{\gamma}$;

   (b) · $\Gamma \vdash e_1 : \mathtt{class}\,t.R \circ \langle\!\langle C \mid n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle$ is derivable, and
   · $\overline{m{:}\alpha} \in R \circ C$, and
   · $\Gamma \vdash \mathtt{class}\,t.\langle\!\langle R \mid n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ C \preceq \tau$ is derivable
   for some $R, C$;

   (c) · $\Gamma \vdash e_1 : \tau$, and
   · $\Gamma \vdash \sigma \preceq \mathtt{class}\,t.\langle\!\langle n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\alpha} \rangle\!\rangle$, and
   · $\Gamma \vdash \sigma \preceq \tau$
   are all derivable for some $\sigma$.

7. If $\Gamma \vdash e \hookleftarrow n : \tau$ is derivable, then

   - $\Gamma \vdash e : \texttt{class}\, t.\langle\!\langle n : \rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\alpha} \rangle\!\rangle$, and
   - $\Gamma \vdash \sigma \preceq \texttt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n : \rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$, and
   - $\Gamma \vdash [\sigma/t](t \to \rho) \preceq \tau$

   are all derivable for some $\sigma, \rho, \overline{m}, \overline{\alpha}$.

**Proof:**
All cases are proved by induction on the derivation of the judgment in the hypothesis: if the last rule in this derivation is not $(\preceq)$, the proof follows directly by the induction hypothesis and an inspection of the typing rules. We give the proof when the last rule is $(\preceq)$, and it immediately follows an application of a rule other than $(\preceq)$ itself. The cases when the proof is concluded by consecutive uses of $(\preceq)$ follows similarly, noting that consecutive uses of $(\preceq)$ can be coalesced, using $(\preceq\ trans)$.

Cases (1) and (3) are immediate, while case (4) requires Lemma 5.3(2) to show that $\rho \equiv \texttt{class}\, t.\langle\!\langle\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$ whenever $\Gamma \vdash \texttt{class}\, t.\langle\!\langle\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \preceq \rho$ is derivable. The remaining cases are worked out below.

(2). In this case, the derivation has the following shape:

$$\dfrac{\dfrac{\Gamma, x{:}\sigma_1 \vdash e : \sigma_2}{\Gamma \vdash \lambda x.e : \sigma_1 \to \sigma_2}\,(exp\ abs) \qquad \Gamma \vdash \sigma_1 \to \sigma_2 \preceq \tau}{\Gamma \vdash \lambda x.e : \tau}\,(\preceq).$$

(5). It suffices to observe that a derivation for $e \Leftarrow n$ ending up with $(\preceq)$ has the following shape:

$$\dfrac{\dfrac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash \sigma \preceq \texttt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle}{\Gamma \vdash e \Leftarrow n : [\sigma/t]\rho}\,(send) \quad \Gamma \vdash [\sigma/t]\rho \preceq \tau}{\Gamma \vdash e \Leftarrow n : \tau}\,(\preceq).$$

(6). We distinguish three cases depending on whether $\langle e_1 \leftarrow\!\circ\, n = e_2 \rangle$ is typed using $(ext)$, $(comp)$, or $(over)$. The derivations have respectively the following shapes:

- Case $(ext)$:

$$\dfrac{\dfrac{\begin{array}{c}\Gamma \vdash e_1 : \texttt{class}\, t.R \circ C \quad \overline{q{:}\eta} \in R \circ C \quad n, \overline{p} \notin \mathcal{M}(R) \cup \mathcal{M}(C) \\ \Gamma, u \preceq \texttt{class}\, t.\langle\!\langle \overline{q{:}\eta}, \overline{p{:}\gamma}, n{:}\rho_{\{\overline{q},\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \vdash e_2 : [u/t](t \to \rho)\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\circ\, n = e_2 \rangle : \texttt{class}\, t.\langle\!\langle R \mid n{:}\rho_{\{\overline{q},\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle C \mid \overline{p{:}\gamma} \rangle\!\rangle}\,(ext) \quad \Gamma \vdash \texttt{class}\, t.\langle\!\langle R \mid n{:}\rho_{\{\overline{q},\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle C \mid \overline{p{:}\gamma} \rangle\!\rangle \preceq \tau}{\Gamma \vdash \langle e_1 \leftarrow\!\circ\, n = e_2 \rangle : \tau}\,(\preceq);$$

· Case (*comp*):

$$\dfrac{\dfrac{\Gamma \vdash e_1 : \mathtt{class}\, t.R\circ\langle\!\langle C \mid n{:}\rho_{\{\overline{m}\}}\rangle\!\rangle \qquad \overline{m{:}\alpha} \in R \circ C \qquad \Gamma, u \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\rho_{\{\overline{m}\}}\rangle\!\rangle\circ\langle\!\langle\rangle\!\rangle \vdash e_2 : [u/t](t\to\rho)}{\Gamma \vdash \langle e_1 \leftarrow\!\!\circ\, n{=}e_2\rangle : \mathtt{class}\, t.\langle\!\langle R \mid n{:}\rho_{\{\overline{m}\}}\rangle\!\rangle\circ C}\,(comp) \qquad \Gamma \vdash \mathtt{class}\, t.\langle\!\langle R \mid n{:}\rho_{\{\overline{m}\}}\rangle\!\rangle\circ C \preceq \tau}{\Gamma \vdash \langle e_1 \leftarrow\!\!\circ\, n{=}e_2\rangle : \tau}\,(\preceq);$$

· Case (*over*):

$$\dfrac{\dfrac{\Gamma \vdash e_1 : \sigma \qquad \Gamma \vdash \sigma \preceq \mathtt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{m}\}}\rangle\!\rangle\circ\langle\!\langle \overline{m{:}\alpha}\rangle\!\rangle \qquad \Gamma, u \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\rho_{\{\overline{m}\}}\rangle\!\rangle\circ\langle\!\langle\rangle\!\rangle \vdash e_2 : [u/t](t\to\rho)}{\Gamma \vdash \langle e_1 \leftarrow\!\!\circ\, n{=}e_2\rangle : \sigma}\,(over) \qquad \Gamma \vdash \sigma \preceq \tau}{\Gamma \vdash \langle e_1 \leftarrow\!\!\circ\, n{=}e_2\rangle : \tau}\,(\preceq).$$

First observe that in all these derivations $\Gamma, u \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n : \rho_{\{\overline{m}\}}\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \vdash e_2 : [u/t](t\to\rho)$ occurs as a premise (in the first derivation, choose $\overline{m{:}\alpha} \equiv \overline{q{:}\eta}, \overline{p{:}\gamma}$). Then note that all the statements of clause $(a)$ occur as judgments and side-conditions in the first derivation, and all the statements of clause $(b)$ occur as judgments and side-conditions in the second derivation. For the third derivation, $\Gamma \vdash e_1 : \tau$ is derivable from $\Gamma \vdash e_1 : \sigma$ and $\Gamma \vdash \sigma \preceq \tau$. All the remaining conditions in clause $(c)$ occur as judgments in this derivation.

(7). It suffices to observe that a derivation for $e \hookleftarrow n$ ending with rule $(\preceq)$ has the following shape:

$$\dfrac{\dfrac{\Gamma \vdash e : \mathtt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{m}\}}\rangle\!\rangle\circ\langle\!\langle \overline{m{:}\alpha}\rangle\!\rangle \qquad \Gamma \vdash \sigma \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\rho_{\{\overline{m}\}}\rangle\!\rangle\circ\langle\!\langle\rangle\!\rangle}{\Gamma \vdash e \hookleftarrow n : [\sigma/t](t\to\rho)}\,(search) \qquad \Gamma \vdash [\sigma/t](t\to\rho) \preceq \tau}{\Gamma \vdash e \hookleftarrow n : \tau}\,(\preceq).$$

□

## 5.3. Subject Reduction

A final lemma is needed in the proof of Subject Reduction, to show that the typing of an object determines the typings of its method bodies.

**Lemma 5.5.** If $\Gamma \vdash \langle e_1 \leftarrow\!\!\circ\, n{=}e_2\rangle : \mathtt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{m}\}}\rangle\!\rangle\circ\langle\!\langle \overline{m{:}\alpha}\rangle\!\rangle$ is derivable, then so is the judgment $\Gamma, u \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\rho_{\{\overline{m}\}}\rangle\!\rangle\circ\langle\!\langle\rangle\!\rangle \vdash e_2 : [u/t](t\to\rho)$.

**Proof:**

If $\Gamma \vdash \langle e_1 \leftarrow\!\circ\, n{=}e_2 \rangle : \texttt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\alpha} \rangle\!\rangle$ is derivable, then, by Lemma 5.4(6), $\Gamma, u \preceq \texttt{class}\, t.\langle\!\langle \overline{m'{:}\alpha'}, n{:}\rho'_{\{\overline{m'}\}} \rangle\!\rangle \circ \langle\!\langle \rangle\!\rangle \vdash e_2 : [u/t](t{\to}\rho')$ also is derivable for some $u, \rho', \overline{m'}, \overline{\alpha'}$, and the conditions in clauses $(a)$, $(b)$ or $(c)$ are satisfied. We next consider these three cases.

In case $(a)$, we get $\overline{m'{:}\alpha'} \equiv \overline{q{:}\eta}, \overline{p{:}\gamma}$, and $\overline{q{:}\eta} \in R \circ C$, and

$$\Gamma \vdash \texttt{class}\, t.\langle\!\langle R \mid n{:}\rho'_{\{\overline{q},\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle C \mid \overline{p{:}\gamma} \rangle\!\rangle \preceq \texttt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\alpha} \rangle\!\rangle$$

is derivable for some $R, C, \overline{q}, \overline{p}, \overline{\eta}, \overline{\gamma}$. Now we have $\rho \equiv \rho', \overline{m} \equiv \overline{m'}, \overline{\alpha} \equiv \overline{\alpha'}$ by Lemma 5.3(2).

In case $(b)$, we have $\overline{m'{:}\alpha'} \in R \circ C$, and

$$\Gamma \vdash \texttt{class}\, t.\langle\!\langle R \mid n{:}\rho'_{\{\overline{m'}\}} \rangle\!\rangle \circ \langle\!\langle C \rangle\!\rangle \preceq \texttt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\alpha} \rangle\!\rangle$$

for some $R, C$. Then we have $\rho \equiv \rho', \overline{m} \equiv \overline{m'}, \overline{\alpha} \equiv \overline{\alpha'}$ by Lemma 5.3(2).

In case $(c)$ we have that $\Gamma \vdash \sigma \preceq \texttt{class}\, t.\langle\!\langle n{:}\rho'_{\{\overline{m'}\}} \rangle\!\rangle \circ \langle\!\langle \overline{m'{:}\alpha'} \rangle\!\rangle$ and $\Gamma \vdash \sigma \preceq \tau$ are derivable for $\tau \equiv \texttt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\alpha} \rangle\!\rangle$. Then we get $\rho_{\{\overline{m}\}} \equiv \rho'_{\{\overline{m'}\}}$ by Lemma 5.3(3), which implies $\rho \equiv \rho'$ and $\overline{m} \equiv \overline{m'}$. So Lemma 5.3(3) applies again to conclude $\overline{\alpha} \equiv \overline{\alpha'}$. □

**Theorem 5.1. (Subject Reduction)** If $\Gamma \vdash e_1 : \tau$ is derivable and $e_1 \overset{eval}{\longrightarrow\!\!\!\!\!\!\rightarrow} e_2$, then $\Gamma \vdash e_2 : \tau$ is also derivable.

**Proof:**

We prove the result by cases on the top-level reduction $\longrightarrow$. Then, the result follows for $\overset{eval}{\longrightarrow\!\!\!\!\!\!\rightarrow}$. In fact, by induction on the size of the contraction context $C[\ ]$, we can easily check that if $\Gamma \vdash C[e] : \tau$ is derived from $\Gamma' \vdash e : \sigma$, then $\Gamma \vdash C[e'] : \tau$ is derivable for any $e'$ such that $\Gamma' \vdash e' : \sigma$ is derivable. Finally, the theorem follows by transitivity.

$(\beta)$. In this case the redex is of the form $(\lambda x.e_1)e_2$ and its typing judgment is $\Gamma \vdash (\lambda x.e_1)e_2 : \tau$. Since this judgment is derivable, by Lemma 5.4(3), we know that $\Gamma \vdash \lambda x.e_1 : \tau_1 \to \tau_2$, $\Gamma \vdash e_2 : \tau_1$ and $\Gamma \vdash \tau_2 \preceq \tau$ are all derivable for some $\tau_1$, $\tau_2$. Now, the proof that $\Gamma \vdash [e_2/x]e_1 : \tau$ is derivable requires the following substitution property on term-variables

$$\textit{if}\ \ \Gamma_1, x : \sigma, \Gamma_2 \vdash e_1 : \tau \ \ \textit{and}\ \ \Gamma_1 \vdash e_2 : \sigma \ \ \textit{then}\ \ \Gamma_1, \Gamma_2 \vdash [e_2/x]e_1 : \tau,$$

that can easily be proved by induction on derivations. To be able to apply this substitution property, we next show that also $\Gamma, x : \tau_1 \vdash e_1 : \tau$ is derivable. By Lemma 5.4(2), if $\Gamma \vdash \lambda x.e_1 : \tau_1 \to \tau_2$ is derivable, then $\Gamma, x : \sigma_1 \vdash e_1 : \sigma_2$ and $\Gamma \vdash \sigma_1 \to \sigma_2 \preceq \tau_1 \to \tau_2$ are both derivable for some $\sigma_1, \sigma_2$. By Lemma 5.3(1) $\Gamma \vdash \sigma_1{\to}\sigma_2 \preceq \tau_1 \to \tau_2$ implies $\Gamma \vdash \tau_1 \preceq \sigma_1$ and $\Gamma \vdash \sigma_2 \preceq \tau_2$. From $\Gamma, x : \sigma_1 \vdash e_1 : \sigma_2$ and $\Gamma \vdash \tau_1 \preceq \sigma_1$, by Lemma 5.1(2), it follows that $\Gamma, x : \tau_1 \vdash e_1 : \sigma_2$ is derivable. From this judgment and from $\Gamma \vdash \sigma_2 \preceq \tau$, the desired judgment is derived again by $(\preceq)$.

$(\Leftarrow)$. If the redex is of the form $e \Leftarrow n$, then its typing judgment is $\Gamma \vdash e \Leftarrow n : \tau$ and, by Lemma 5.4(5), it follows that all of the following judgments are derivable for some $\sigma, \rho, \overline{m}, \overline{\alpha}$:

1. $\Gamma \vdash e : \sigma$,
2. $\Gamma \vdash \sigma \preceq \texttt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$,
3. $\Gamma \vdash [\sigma/t]\rho \preceq \tau$.

It is also easy to verify that

4. $\Gamma \vdash \texttt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \preceq \texttt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\alpha} \rangle\!\rangle$

is derivable by repeated applications of ($\preceq$ *shift*). Then, the typing for the reduct may be derived as follows. First construct the following derivation:

$$\dfrac{\dfrac{\dfrac{1 \quad \dfrac{2 \quad 4}{\Gamma \vdash \sigma \preceq \texttt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\alpha} \rangle\!\rangle}\,(\preceq trans)}{\Gamma \vdash e : \texttt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\alpha} \rangle\!\rangle}\,(\preceq) \quad 2}{\Gamma \vdash e \hookleftarrow n : [\sigma/t](t \to \rho)}\,(search) \quad 1}{\Gamma \vdash (e \hookleftarrow n)e : [\sigma/t]\rho}\,(exp\ appl).$$

Now apply ($\preceq$) to the root of this derivation and to 3.

($\hookleftarrow$ *succ*). In this case the typing of the redex is $\Gamma \vdash \langle e_1 \leftarrow\!\circ\, n{=}e_2 \rangle \hookleftarrow n : \tau$ and, by Lemma 5.4(7), all of the following judgments are derivable for some $\sigma, \rho, \overline{m}, \overline{\alpha}$:

1. $\Gamma \vdash \langle e_1 \leftarrow\!\circ\, n{=}e_2 \rangle : \texttt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\alpha} \rangle\!\rangle$,
2. $\Gamma \vdash \sigma \preceq \texttt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$,
3. $\Gamma \vdash [\sigma/t](t \to \rho) \preceq \tau$.

From 1, by Lemma 5.5, it follows that

4. $\Gamma, u \preceq \texttt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \vdash e_2 : [u/t](t \to \rho)$

also is derivable. From 2 and 4, by Lemma 5.2 (choosing $\Gamma_1 \equiv \Gamma$ and $\Gamma_2 \equiv \varepsilon$, the hypothesis $Var(\sigma) \cap Dom(\Gamma_2)$ is trivially satisfied), it then follows that $\Gamma \vdash e_2 : [\sigma/t](t \to \rho)$ is derivable. A further application of ($\preceq$) to this last judgment and to 3 yields the expected judgment $\Gamma \vdash e_2 : \tau$.

($\hookleftarrow$ *next*). In this case the typing of the redex is $\Gamma \vdash \langle e_1 \leftarrow\!\circ\, m{=}e_2 \rangle \hookleftarrow n : \tau$, and by Lemma 5.4(7) all of the following judgments are derivable for some $\sigma, \rho, \overline{p}, \overline{\alpha}$:

1. $\Gamma \vdash \langle e_1 \leftarrow\!\circ\, m{=}e_2 \rangle : \texttt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle \overline{p{:}\alpha} \rangle\!\rangle$,
2. $\Gamma \vdash \sigma \preceq \texttt{class}\, t.\langle\!\langle \overline{p{:}\alpha}, n{:}\rho_{\{\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$,
3. $\Gamma \vdash [\sigma/t](t \to \rho) \preceq \tau$.

From 1, by Lemma 5.4(6), the conditions in clause ($a$), ($b$) or ($c$) are satisfied. We next consider these three cases, showing that $\Gamma \vdash e_1 : \texttt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle \overline{p{:}\alpha} \rangle\!\rangle$ is derivable in all the cases: case ($c$) is immediate, while the remaining cases are given below.

Case ($a$). Both $\Gamma \vdash e_1 : \texttt{class}\, t.R \circ C$, and $\Gamma \vdash \texttt{class}\, t.\langle\!\langle R \mid m{:}\mu_{\{\overline{q},\overline{r}\}} \rangle\!\rangle \circ \langle\!\langle C \mid \overline{r{:}\gamma} \rangle\!\rangle \preceq \texttt{class}\, t.\langle\!\langle n : \rho_{\{\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle \overline{p{:}\alpha} \rangle\!\rangle$ are derivable for some $R, C, \overline{q}, \overline{r}, \overline{\mu}, \overline{\gamma}$. Since $n \not\equiv m$, we have $n{:}\rho_{\{\overline{p}\}} \in R$ by Lemma 5.3(2). This implies $\overline{p{:}\alpha} \in R \circ C$, since $\overline{p} \in \mathcal{L}(R)$ and by definition of well-formed types $\mathcal{L}(R) \subseteq \mathcal{M}(R) \cup \mathcal{M}(C)$. Thus we have that $\Gamma \vdash \texttt{class}\, t.R \circ C \preceq$

$\mathtt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{p}\}}\rangle\!\rangle \circ \langle\!\langle \overline{p{:}\alpha}\rangle\!\rangle$ is derivable, and from this judgment we may derive $\Gamma \vdash e_1 : \mathtt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{p}\}}\rangle\!\rangle \circ \langle\!\langle \overline{p{:}\alpha}\rangle\!\rangle$ by $(\preceq)$.

Case $(b)$. Both $\Gamma \vdash e_1 : \mathtt{class}\, t.R \circ \langle\!\langle C \mid m{:}\mu_{\{\overline{q}\}}\rangle\!\rangle$, and $\Gamma \vdash \mathtt{class}\, t.\langle\!\langle R \mid m{:}\mu_{\{\overline{q}\}}\rangle\!\rangle \circ C \preceq \mathtt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{p}\}}\rangle\!\rangle \circ \langle\!\langle \overline{p{:}\alpha}\rangle\!\rangle$ are derivable for some $R, C, \overline{q}, \overline{\mu}$. Since $n \not\equiv m$, $\Gamma \vdash \mathtt{class}\, t.R \circ \langle\!\langle C \mid m{:}\mu_{\{\overline{q}\}}\rangle\!\rangle \preceq \mathtt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{p}\}}\rangle\!\rangle \circ \langle\!\langle \overline{p{:}\alpha}\rangle\!\rangle$ is derivable, and from this judgment $\Gamma \vdash e_1 : \mathtt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{p}\}}\rangle\!\rangle \circ \langle\!\langle \overline{p{:}\alpha}\rangle\!\rangle$ is derivable by $(\preceq)$.

The typing for the reduct may then be derived from $\Gamma \vdash e_1 : \mathtt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{p}\}}\rangle\!\rangle \circ \langle\!\langle \overline{p{:}\alpha}\rangle\!\rangle$ as follows:

$$\dfrac{\dfrac{\Gamma \vdash e_1 : \mathtt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{p}\}}\rangle\!\rangle \circ \langle\!\langle \overline{p{:}\alpha}\rangle\!\rangle \qquad 2}{\Gamma \vdash e_1 \hookleftarrow n : [\sigma/t](t \to \rho)}\,(search) \qquad 3}{\Gamma \vdash e_1 \hookleftarrow n : \tau}\,(\preceq).$$

□

## 5.4. Absence of Stuck States

The reduction rules for the operational semantics given in Table 2 (Section 2) readily suggest how an interpreter for the untyped calculus can be defined. 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 $\lambda$-abstraction; $(iii)$ when searching an $m$ ($m \not\equiv n$) 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 $\lambda$-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 the 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 5.2. (Absence of stuck states)** Let $e$ be a closed expression such that $\varepsilon \vdash e : \tau$ is derivable for some type $\tau$. Then:

1. if $e \equiv e_1\, e_2$ and $e_1 \Downarrow val$, then $val \equiv \lambda x.e'$ for some $x$ and $e'$;
2. if $e \equiv e_1 \Leftarrow n$ and $e_1 \Downarrow val$, then $val \equiv obj$ for some object expression $obj$;
3. if $e \equiv \langle e_1 \leftarrow\!\circ\, m{=}e_2\rangle \hookleftarrow n \downarrow e_3$ and $e_1 \Downarrow val$, then $val \equiv obj$ for some object expression $obj$.

**Proof:**
The proof uses the subject reduction property for $\Downarrow$ and $\downarrow$. 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 5.1, since $e \xrightarrow{eval} val$ when $e \Downarrow val$ (see Proposition 2.1). A corresponding property holds for $\downarrow$. We next give a proof of the three cases.

1. If $\varepsilon \vdash e_1\, e_2 : \tau$, then by Lemma 5.4(3), it follows that $\varepsilon \vdash e_1 : \tau_1 \to \tau_2$ is derivable for some $\tau_1$ and $\tau_2$ ($\tau_2 \preceq \tau$). Then also $\varepsilon \vdash val : \tau_1 \to \tau_2$ is derivable. That $val$ is of the form $\lambda x.e$ follows from observing that all the other possibilities can be rejected: the *obj* forms can be rejected by Lemmas 5.4(6) and 5.3(2); the $\langle\rangle$ form can be rejected by Lemma 5.4(4).
2. If $\varepsilon \vdash e_1 \Leftarrow n : \tau$, is derivable, then by Lemma 5.4(5), it follows that both $\varepsilon \vdash e_1 : \sigma$ and $\varepsilon \vdash \sigma \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$ are derivable for some $\sigma$, $\overline{m}$, $\overline{\alpha}$ and $\rho$. We get, by Lemma 5.3(3), $\sigma \equiv \mathtt{class}\, t.\langle\!\langle R \mid \overline{m{:}\alpha}, n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ C$ for some $R$ and $C$. Then $\varepsilon \vdash val : \mathtt{class}\, t.\langle\!\langle R \mid \overline{m{:}\alpha}, n{:}\rho_{\{\overline{m}\}} \rangle\!\rangle \circ C$ is derivable, and that $val \equiv obj$ follows since by the Generation Lemma no other values (i.e. the empty object and $\lambda$-abstractions) have this type for any $R$ and $C$.
3. If $\varepsilon \vdash \langle e_1 \hookleftarrow\!\circ\, m{=}e_2 \rangle \hookleftarrow n : \tau$, is derivable, then by Lemma 5.4(7), it follows that $\varepsilon \vdash \langle e_1 \hookleftarrow\!\circ\, m{=}e_2 \rangle : \mathtt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle \overline{p{:}\gamma} \rangle\!\rangle$ is derivable for some $\rho, \overline{p}, \overline{\gamma}$. By Lemma 5.4(6), it follows that $\varepsilon \vdash e_1 : \mathtt{class}\, t.R \circ C$ is derivable for some $R \circ C \not\equiv \langle\!\langle\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$, from which the claim follows as in the previous case. To see that $R \circ C \not\equiv \langle\!\langle\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$, consider the three clauses of Lemma 5.4(6). Clause ($a$) implies that $\varepsilon \vdash \mathtt{class}\, t.\langle\!\langle R \mid m{:}\alpha \rangle\!\rangle \circ \langle\!\langle C \mid \overline{q{:}\eta} \rangle\!\rangle \preceq \mathtt{class}\, t.\langle\!\langle n{:}\rho_{\{\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle \overline{p{:}\gamma} \rangle\!\rangle$, which, by Lemma 5.3(2), implies that $n{:}\rho_{\{\overline{p}\}} \in R$, being $m \not\equiv n$. Clause ($b$) implies that $n{:}\rho_{\{\overline{p}\}} \in C$. Finally, clause ($c$) implies that $R \circ C \equiv \langle\!\langle n{:}\rho_{\{\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle \overline{p{:}\gamma} \rangle\!\rangle$ □

## 6. Label Inference

We conclude the description of the type system with a discussion on labels and labeled types.

Labeled-types are clearly central to the soundness of the type system. In particular, the proof of subject reduction shows where labels are needed for constructing typing derivations: looking at the (*search*) case, one sees that the label associated to the type of the method being searched, in the search redex, provides precisely the information that is needed to single out the dependencies of this method, so that the same typing may be derived for the reduct.

One objection against using such types could be that they are somehow *ad hoc*, and hence difficult to explain and characterize semantically. In this section we offer a counter-argument, and show that labels can indeed be seen simply as "internal" devices needed to ensure type soundness (in fact, subject reduction) while they are not relevant to the semantics of types. We do this by outlining an algorithm for inferring labels from a "label-free" derivation.

The inference of labels is described as a constructive process that transforms any given "label-free" derivation $\Xi$ into a labeled derivation, whenever this is possible by only adding labels. Say that a type $\tau$ is a *label-free* type if $\tau$ conforms with the syntax of types defined in Section 3, and every labeled type occurring in $\tau$ is labeled with the distinguished symbol

• (the marker • is useful to distinguish labeled types with empty labels from "label-free" types). A *label-free* derivation is a derivation that is carried out in the system described in Section 3 using only label-free types, i.e. completely disregarding labels.

### 6.1. Inference as Rewriting

Assume that we are given a label-free derivation $\Xi$: the inference process is accomplished by an iterative rewriting of $\Xi$ that computes and propagates labels at the types occurring at each of the proof rules of $\Xi$. Labels are computed at the (*send*), (*search*), (*ext*), (*comp*) and (*over*) nodes of $\Xi$, and they are propagated to all nodes. Label computation and propagation are both described by means of a term rewriting system for the typing or subtyping proof rules of the type system.

Besides computing labels, the rewriting rules also verify that labels are propagated consistently. The consistency checks are needed because different occurrences of the same type in a rule may have different labels at intermediate steps of the rewriting. There are, in fact, few cases to consider: let $\Delta_1$ and $\Delta_2$ ($\Delta_1 \not\equiv \Delta_2$) be two labels for two occurrences of the same type $\tau$ in a given rule. If neither of the $\Delta_i$'s is $\bullet$, then the labeling is inconsistent and both the occurrences of $\tau$ are replaced by $\bot$ to signal a failure; if instead $\Delta_1 \not\equiv \bullet$ (respectively, $\Delta_2 \not\equiv \bullet$), then both occurrences of $\tau$ are labeled with $\Delta_1$ (resp. $\Delta_2$).

Given that labels are computed once at each type (in the way we just described), it is easy, (although time-consuming) to verify that the order of application of the rewriting does not affect the outcome. Also, the rewriting process is clearly terminating, since each label-free derivation has a finite number of types to be labeled.

In particular, the rewriting of a label-free derivation $\Xi$ terminates either because some of the types is rewritten to $\bot$, or because none of the types in the derivation is affected by further rewriting. In the first case, $\Xi$ is not sound because it fails to satisfy the invariance constraint of labels that is instead enforced in the type system of Section 3 to ensure type soundness.

In the second case, the resulting derivation may be turned into a labeled derivation by replacing all the residual occurrences of the marker • by by the empty label. It then remains to check whether all labeled types are well-formed according to their definition (Table 3), in particular whether for every occurrence of the ($\preceq$ *hide*) rule the labeled types computed by the inference process satisfy the well-formedness conditions on `class` types (cf. Table 3). If so, the labeled derivation is sound, otherwise it is not, and no (sound) labeled derivation may be inferred from $\Xi$ simply by adding labels.

### 6.2. Rewriting Rules

The definition of rewriting is based on operators that propagate labels on types, contexts and judgments. There are three such operators: $\diamond$ propagates labels between types with identical structure (i.e. types that are equal up to labels); $\lhd$ and $\rhd$ propagate labels between different types. These operators are all defined in terms of the operator $\sqcup$, given in Table B.1, that "unifies" two labels according to the idea we described earlier in this section. As we anticipated, the application of $\sqcup$ to two different labels is undefined.

Label propagation between two equal types, $\tau'$ and $\tau''$, is defined as expected, by induction on the (identical) structure of the two types; $\tau' \diamond \tau''$ returns the labeled type that results from the application of $\sqcup$ to the labels occurring at corresponding positions in $\tau'$ and $\tau''$.
Moreover, we need to propagate labels on the types occurring in the contexts and right-hand sides of the judgments. Since this propagation is performed under the hypothesis that these types differ only for their labels (because we start from a derivation, even though label-free), we use the operator $\diamond$ on types and we extend it to contexts and right-hand sides of judgments (Table B.2 gives all the definitions concerning $\diamond$).

The need for the two operators $\triangleleft$ and $\triangleright$ arises in the rewriting of the typing rules that involve subtyping judgments. Consider, for instance, the case of a subtyping judgment of the form:

$$\Gamma \vdash \texttt{class}\, t.R_1 \circ C_1 \preceq \texttt{class}\, t.R_2 \circ C_2.$$

For this judgment to be labeled consistently, it must be guaranteed that for every pair $m{:}\alpha$ occurring in both $R_1 \circ C_1$ and $R_2 \circ C_2$, both the occurrences of $\alpha$ be given the same labeling. Given that $R_1 \circ C_1$ and $R_2 \circ C_2$ may have different structure, this is accomplished in two steps, forming the judgment:

$$\Gamma \vdash \texttt{class}\, t.(R_1 \circ C_1 \triangleleft R_2 \circ C_2) \preceq \texttt{class}\, t.(R_1 \circ C_1 \triangleright R_2 \circ C_2).$$

Intuitively, $R_1 \circ C_1 \triangleleft R_2 \circ C_2$ computes the correct labeling for every type in $R_1 \circ C_1$, while $R_1 \circ C_1 \triangleright R_2 \circ C_2$ returns the desired labeling for $R_2 \circ C_2$. As an example, we have:

$$\langle\!\langle m{:}int_{\{m,n\}}, n{:}bool_{\bullet} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \triangleleft \langle\!\langle m{:}int_{\bullet} \rangle\!\rangle \circ \langle\!\langle n{:}bool_{\{m,n\}} \rangle\!\rangle = \langle\!\langle m{:}int_{\{m,n\}}, n{:}bool_{\{m,n\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$$

and

$$\langle\!\langle m{:}int_{\{m,n\}}, n{:}bool_{\bullet} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \triangleright \langle\!\langle m{:}int_{\bullet} \rangle\!\rangle \circ \langle\!\langle n{:}bool_{\{m,n\}} \rangle\!\rangle = \langle\!\langle m{:}int_{\{m,n\}} \rangle\!\rangle \circ \langle\!\langle n{:}bool_{\{m,n\}} \rangle\!\rangle.$$

The definition of $\triangleleft$ and $\triangleright$ is extended to types, labeled types and rows in Table B.3 following this idea: note, in particular, that the application of these operators over function-types is so defined as to comply with the contravariant rule of the arrow-type constructor.

It is worth noting that the fact that $\sigma \triangleleft \tau$ and $\sigma \triangleright \tau$ are both defined does not imply that $\sigma \triangleleft \tau \preceq \sigma \triangleright \tau$ is satisfied, since the operators $\triangleleft$ and $\triangleright$ provide for label propagation over subtyping judgments without, however, checking the associated well-formedness conditions on `class` types. It is also easy to verify that if $\sigma$ and $\tau$ differ only for their labels, then $\sigma \triangleleft \tau \equiv \sigma \triangleright \tau$. As such, the definition of the $\diamond$ operator may, in fact, be given indirectly in terms of $\triangleleft$ (equivalently $\triangleright$): simply define $\tau' \diamond \tau''$ as $\tau' \triangleleft \tau''$, proviso that $\tau'$ and $\tau''$ have the same structure.

The rewriting of the typing rules (system $\mathcal{R}$) is given in Appendix B. If some of the partial operators which occur in the right-hand side of a rule are undefined, we get a failure. In this case it does not really matter how we rewrite the typing rule, proviso that at least one $\perp$ occurs in the resulting derivation.

We use the notational convention that types, contexts,... etc. which differ only for their labels are denoted by the same letter with different superscripts.

For example, in the rewriting rule for ($ext$) (Table B.7), we need to unify $\Delta$ and $\Delta'$ with $\{\overline{m},\overline{p}\}$, and we need to unify the labels which occur in $\tau$, $\tau'$ and $\tau''$. This is accomplished in the definition of $\tau'''_{\Delta''}$. Notice that this implies either $\Delta'' \equiv \{\overline{m},\overline{p}\}$ or failure (in this case $\tau'''_{\Delta''} \equiv \bot$). Moreover, we need to propagate labels between $R \circ C$ and $R' \circ C'$, and at the same time between the types of the $\overline{m}$ methods of $R' \circ C'$ and $\overline{\alpha}$, $\overline{\alpha}'$. The definition of $R'' \circ C''$ accounts for both these aspects, and $\overline{\alpha}''$ denotes the resulting types of the $\overline{m}$ methods. As an example, let $\mathtt{e} \stackrel{\triangle}{=} \lambda\mathtt{self}.\lambda\mathtt{dx}.\langle \mathtt{self} \leftarrow\!\circ\ \mathtt{x} = \lambda\mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx}\rangle$. By applying this rule we have that

$$\frac{\begin{array}{l}\varepsilon \vdash \langle\rangle : \mathtt{class}\, t.\langle\!\langle\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \\ u \preceq \mathtt{class}\, t.\langle\!\langle \mathtt{mv}{:}(int \to t)_{\bullet}, \mathtt{x}{:}int_{\bullet}\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \vdash \mathtt{e} : u \to int \to u\end{array}}{\varepsilon \vdash \langle \mathtt{mv} = \mathtt{e}\rangle : \mathtt{class}\, t.\langle\!\langle \mathtt{mv}{:}(int \to t)_{\bullet}\rangle\!\rangle \circ \langle\!\langle \mathtt{x}{:}int_{\bullet}\rangle\!\rangle}\ (ext)$$

rewrites to

$$\frac{\begin{array}{l}\varepsilon \vdash \langle\rangle : \mathtt{class}\, t.\langle\!\langle\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \\ u \preceq \mathtt{class}\, t.\langle\!\langle \mathtt{mv}{:}(int \to t)_{\{\mathtt{x}\}}, \mathtt{x}{:}int_{\bullet}\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \vdash \mathtt{e} : u \to int \to u\end{array}}{\varepsilon \vdash \langle \mathtt{mv} = \mathtt{e}\rangle : \mathtt{class}\, t.\langle\!\langle \mathtt{mv}{:}(int \to t)_{\{\mathtt{x}\}}\rangle\!\rangle \circ \langle\!\langle \mathtt{x}{:}int_{\bullet}\rangle\!\rangle}\ (ext)$$

Let $u$ be a type variable, $x$ a (term) variable, $\tau$ and $\tau'$ types, and $\Gamma$ a context: with $[u \preceq \tau \diamond \tau'/u \preceq \tau]\Gamma$ and $[x : \tau \diamond \tau'/x : \tau]\Gamma$ we denote the substitution in $\Gamma$ of, respectively, the subtyping declaration $u \preceq \tau$ by $u \preceq \tau \diamond \tau'$ and the substitution of the variable declaration $x : \tau$ by $x : \tau \diamond \tau'$.

We conclude this discussion by formally stating our results and sketching their proofs.

**Proposition 6.1.** The rewriting system $\mathcal{R}$ is Church-Rosser and terminating.

The Church-Rosser property can be proved as usual by showing the "diamond property" for critical pairs [22]. Critical pairs arise here since a typing judgment can occur both as conclusion of one rule and as premise of another rule. Termination is obvious since only a finite number of labels can be generated.

We say that the application of the rewriting $\mathcal{R}$ to a label free derivation is *successful* if and only if after replacing the residual occurrences of the marker $\bullet$ by the empty label we get a derivation that:

(a) contains only well-formed labeled types, and
(b) does not contain any occurrence of $\bot$.

**Proposition 6.2. (Soundness)** If $\Xi$ is the output of a successful application of $\mathcal{R}$ to a label-free derivation, then $\Xi$ is a well-formed derivation.

Soundness easily follows from the fact that the result of each application of a rewrite rule in $\mathcal{R}$ is a well-formed rule, whenever it satisfies conditions (a) and (b) above.

**Proposition 6.3. (Completeness)** If $\Xi$ is a label-free derivation obtained from a well-formed derivation $\Xi'$ by replacing every label by the marker $\bullet$, then $\Xi'$ will be the output of the application of $\mathcal{R}$ to $\Xi$.

This follows from the fact that the rewriting of rules (*send*), (*search*), (*ext*), (*comp*) and (*over*) determine in a unique way some labels and all other rewriting rules simply propagate them.

The notion of completeness we just stated is limited, since we can build two different label-free derivations with the same conclusion, such that the rewriting of the first is successful while the rewriting of the second is unsuccessful. This means that a "no" answer does not prove that a correct derivation does not exist, it only proves that a particular derivation is not correct w.r.t. the labeling. So, one could design a more refined rewriting, which also modifies the underlining label-free derivation, correcting it. Clearly, however, this would substantially increase the overall cost of the rewriting.

## 7. Conclusion

We have presented an extension of the *Lambda Calculus of Objects* [16] with a new type system that gives provision both for rapid prototyping, by allowing the typing of partially specified objects in the style of [6], and for a relation of width subtyping, in the style of [7].

The main technical tool of the system is represented by labeled types, that are central both to the rendering of method polymorphism based on (implicit) bounded quantification, and to the soundness of the type system. On the other side, as shown in Section 6, types and type derivations in the system may be understood independently of labels, as labels may be inferred automatically from label-free types and derivations, and then checked for soundness. Thus, in principle, labels can be made transparent from "outside" class-types (hence, to the user of the system) and only seen as "internal" devices needed to ensure type soundness.

A system that exhibits features comparable to ours is *Baby Modula–3* [1] which, however, we generalize in two respects:

(i) we allow object extensions and subsumptions in any order, while in [1] all the extensions must be done before any subsumption;
(ii) our completions may be extended as a result of a method addition, while in [1] completions are fixed ahead of time, prior to any addition.

A feature of [1] that, instead, we do not provide (even though we could) is the distinction between fields and methods, that allows one to isolate the state of an object from the operations on the state.

A few additional remarks are in order on the relationships with the system of [17]. As we noted, the two systems are incomparable: on one side, we allow extensions, overrides and subtypings on the same object in any order, while [17] forbids extending or overriding objects for which one already used subtyping. On the other, method encapsulation is not accounted for in our system and it is instead provided in [17]. To this regard, we note that the solution proposed in [17] could be accommodated just as well in our system. As in [17], we would need to distinguish the types of prototypes from the types of objects, so as to allow altering the structure of the former with method additions and overrides while instead preventing such operations to be applied to the latter. Methods of a complete prototype (i.e. a prototype whose completion is empty) could then be "sealed" (hence encapsulated) within the object corresponding to the prototype exactly as in the system of [17].

A few other studies on object-based languages have recently been proposed as elaborations of the Lambda Calculus of Objects and related calculi:

· [4] presents a type system for the Lambda Calculus of Objects based on *matching*;
· [24] describes a refined subtyping for extensible and incomplete objects;
· [15] gives provision for prototyping in a statically typed, imperative, class-based language;
· [20] adds object extension and width subtyping to the system of [3];
· [21] presents a (decidable) "fully" typed version of the calculus of [16].

The system of [4] and the one of this paper share the same idea of using bounded type variables to capture polymorphic method-types. The key difference is that [4] uses a simplified notion of *matching* [8, 2] (without subsumption) and *match*-bound variables, whereas here we use subtyping and *subtype*-bound variables. In [4], it is shown that match-bounded type variables and row-variables have the same expressive power, more precisely that the systems of [4] and [16] derive the same judgments from the empty basis. This result, instead, does not hold for our system, as there is a fundamental trade-off between our use of subtyping that allows the derivation of judgments that are not derivable in [16], and the reliance of subtyping, on labeled types, that prevents us from deriving some judgments which are instead valid in [16].

The calculus considered in [24] is an extension of the Abadi and Cardelli calculus of [3]. While there are similarities with our proposal – notably, the use of subtyping for dealing with object extension – the two type systems have some fundamental differences.

On one side, the system of [24] gives provision for subtyping in depth for extensible objects, thus improving on the solution of [17] based on the distinction between `pro`- and `obj`- types. This flexibility would not be possible in our system. On the other hand, our system appears superior in the treatment of binary methods, that are left for future investigation in [24], and are instead dealt with for free by our subtyping rules. Also, our system allows methods to be invoked on incomplete objects, while this is forbidden in [24]. Finally, while in [24] object types are interpreted as total functions from method labels to types, in our system we rely on the more conventional (and seemingly more flexible) interpretation of object types as partial functions.

The approach to prototyping developed in [15] is largely based on the ideas of [5] that we have illustrated here in further detail. Besides the different settings – imperative versus functional, class-based versus object-based – the system of [15] and ours have other fundamental differences. First, the system of [15] does not allow subtyping and, as such, it is more liberal than ours in dealing with incompleteness (incompleteness is allowed at a global level in [15], as the body of a method may contain references to non-implemented methods in other classes). Secondly, and more importantly, the system of [15] relies on a notion of "weak" soundness whereby a program is accepted as type correct even though it may cause *message-not-understood* errors, due to incompleteness.

**Acknowledgments.** Suggestions by Adriana Compagnoni and Sophia Drossopoulou are gratefully acknowledged: they were very helpful in improving the technical presentation of the paper.

The present version of this paper has been deeply influenced by comments and remarks of two anonymous referees. In particular, Section 5 and 6 were almost completely rewritten following their advices. Therefore the authors feel strongly indebted to the referees.

The final version of this paper was written while three of the authors were on leave from their departments.

Viviana Bono was visiting the Computer Science Department at Stanford. She would like to thank her host John C. Mitchell and her Ph.D. colleagues for the ideal environment they provided.

Mariangiola Dezani-Ciancaglini was visiting the Tokyo Institute of Technology. She would like to thank her host Masako Takahashi-Horai and the whole Department of Mathematical and Computing Sciences for the ideal working conditions they provided.

Luigi Liquori held a temporary position at CSELT, Centro Studi e Laboratori Telecomunicazioni, Torino. He would like to thank the members of his group for their helpful discussions.

## References

[1] M. Abadi. Baby Modula–3 and a Theory of Objects. *Journal of Functional Programming*, 4(2):249–283, 1994.

[2] M. Abadi and L. Cardelli. On Subtyping and Matching. In *ECOOP'95*, *LNCS* 952, 145–167, Springer–Verlag, 1995.

[3] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer–Verlag, 1996.

[4] V. Bono and M. Bugliesi. Matching Constraints for the Lambda Calculus of Objects. In *TLCA'97*, *LNCS* 1210, 46-62, Springer-Verlag, 1997.

[5] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping Constraints for Incomplete Objects. In *CAAP'97*, *LNCS* 1214, 465-477, Springer-Verlag, 1997.

[6] V. Bono, M. Bugliesi, and L. Liquori. A Lambda Calculus of Incomplete Objects. In *MFCS'96*, *LNCS* 1113, 218–229, Springer-Verlag, 1996.

[7] V. Bono and L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In *CSL'94*, *LNCS* 933, 16–30, Springer-Verlag, 1995.

[8] K.B. Bruce. A Paradigmatic Object–Oriented Programming Language: Design, Static Typing and Semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

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

[10] L. Cardelli. A Language with Distributed Scope. *Computing Survey*, 8(1):27–59, 1995.
[11] L. Cardelli and J.C. Mitchell. Operations on Records. *Mathematical Structures in Computer Sciences*, 1(1):3–48, 1991.
[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 *POPL'90*, 125–135, ACM Press, 1990.
[14] W.R. Cook. *A Denotational Semantics of Inheritance*. Ph.D. Thesis, Brown University, 1989.
[15] M.J. Dickinson. *Typed Object-Oriented Prototyping*. M.SC. Thesis, Imperial College, 1997.
[16] 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.
[17] K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *FCT'95*, *LNCS* 965, 42–61, Springer-Verlag, 1995.
[18] A. Goldberg and D. Robson. *Smalltalk-80, The Language and its Implementation*. Addison Wesley, 1983.
[19] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *J.ACM*, 40(1):143–184, 1993.
[20] L. Liquori. An Extended Theory of Primitive Objects: First Order System. In *ECOOP'97*, *LNCS* 1241, 146–169, Springer-Verlag, 1997.
[21] L. Liquori and B. Castagna. A Typed Lambda Calculus of Objects. In *Asian'96*, *LNCS* 1179, 129–141, Springer-Verlag, 1996.
[22] D. Knuth and P. Bendix, Simple Word Problems in Universal Algebras. In *Computational Problems in Universal Algebras*, 263–297, Pergamon Press, 1970.
[23] J. McKinna and R. Pollack. Pure Type Systems Formalized. In *TLCA'93*, *LNCS* 664, 289–305, Springer-Verlag, 1993.
[24] D. Rémy. From Classes to Objects via Subtyping. In *ESOP'98*, *LNCS* 1381, 200–220, Springer-Verlag, 1998.
[25] D. Ungar and R. B. Smith. Self: the Power of Simplicity. In *OOPSLA'87*, 227–241, ACM Press, 1987.
[26] M. Wand. Complete Type Inference for Simple Objects. In *LICS'87*, 37–44, IEEE Press, 1987.

# A. The Type System

## A.1. Typing Rules for Contexts

$$\frac{}{\varepsilon \vdash *}\ (start) \qquad \frac{\Gamma \vdash * \quad x \notin \Gamma}{\Gamma,\, x{:}\tau \vdash *}\ (exp\ var) \qquad \frac{\Gamma \vdash * \quad u \notin \Gamma \quad u \notin R \circ C}{\Gamma, u \preceq \mathtt{class}\, t.R \circ C \vdash *}\ (type\ var)$$

## A.2. Subtyping Rules

$$\frac{\Gamma \vdash *}{\Gamma \vdash \tau \preceq \tau}\ (\preceq refl) \qquad \frac{\Gamma \vdash * \quad u \preceq \tau \in \Gamma}{\Gamma \vdash u \preceq \tau}\ (\preceq proj)$$

$$\frac{\Gamma \vdash \sigma \preceq \tau \quad \Gamma \vdash \tau \preceq \rho}{\Gamma \vdash \sigma \preceq \rho}\ (\preceq trans) \qquad \frac{\Gamma \vdash \sigma' \preceq \sigma \quad \Gamma \vdash \tau \preceq \tau'}{\Gamma \vdash \sigma \to \tau \preceq \sigma' \to \tau'}\ (\preceq arrow)$$

$$\frac{\Gamma \vdash *}{\Gamma \vdash \mathtt{class}\, t.\langle\!\langle R \mid m{:}\alpha \rangle\!\rangle \circ C \preceq \mathtt{class}\, t.R \circ \langle\!\langle C \mid m{:}\alpha \rangle\!\rangle}\ (\preceq \mathit{shift})$$

$$\frac{\Gamma \vdash *}{\Gamma \vdash \mathtt{class}\, t.R \circ \langle\!\langle C \mid \overline{p{:}\gamma} \rangle\!\rangle \preceq \mathtt{class}\, t.R \circ C}\ (\preceq \mathit{hide})$$

### A.3. Typing Rules for Terms

$$\frac{\Gamma \vdash * \quad x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ (proj) \qquad \frac{\Gamma,\, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ (exp\ abs)$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}\ (exp\ appl) \qquad \frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash \sigma \preceq \tau}{\Gamma \vdash e : \tau}\ (\preceq)$$

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

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash \sigma \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\tau_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle}{\Gamma \vdash e \Leftarrow n : [\sigma/t]\tau}\ (send)$$

$$\frac{\Gamma \vdash e : \mathtt{class}\, t.\langle\!\langle n{:}\tau_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\alpha} \rangle\!\rangle \quad \Gamma \vdash \sigma \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\tau_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle}{\Gamma \vdash e \hookleftarrow n : [\sigma/t](t \to \tau)}\ (search)$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \mathtt{class}\, t.R{\circ}C \quad \overline{m{:}\alpha} \in R \circ C \quad n, \overline{p} \notin \mathcal{M}(R) \cup \mathcal{M}(C) \\ \Gamma, u \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, \overline{p{:}\gamma}, n{:}\tau_{\{\overline{m},\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \vdash e_2 : [u/t](t \to \tau)\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!\circ\, n{=}e_2 \rangle : \mathtt{class}\, t.\langle\!\langle R \mid n{:}\tau_{\{\overline{m},\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle C \mid \overline{p{:}\gamma} \rangle\!\rangle}\ (ext)$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \mathtt{class}\, t.R{\circ}\langle\!\langle C \mid n{:}\tau_{\{\overline{m}\}} \rangle\!\rangle \quad \overline{m{:}\alpha} \in R \circ C \\ \Gamma, u \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\tau_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \vdash e_2 : [u/t](t \to \tau)\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!\circ\, n{=}e_2 \rangle : \mathtt{class}\, t.\langle\!\langle R \mid n{:}\tau_{\{\overline{m}\}} \rangle\!\rangle \circ C}\ (comp)$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \sigma \quad \Gamma \vdash \sigma \preceq \mathtt{class}\, t.\langle\!\langle n{:}\tau_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\alpha} \rangle\!\rangle \\ \Gamma, u \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\tau_{\{\overline{m}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \vdash e_2 : [u/t](t \to \tau)\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!\circ\, n{=}e_2 \rangle : \sigma}\ (over)$$

# B. Rewriting Rules for Label Inference

## B.1. The Operator $\sqcup$ on Extended Labels

- $\Delta \sqcup \bullet = \bullet \sqcup \Delta = \Delta \sqcup \Delta = \Delta$ for every $\Delta$, including $\bullet$;
- $\Delta_1 \sqcup \Delta_2 = \perp$ otherwise.

## B.2. The Operator $\diamond$

The operator $\diamond$ on types, labeled types, rows, contexts and right-hand sides of judgments is inductively defined as follows:

- $\tau_1 \diamond \tau_2 = \tau$ if $\tau_1 = \tau_2 = \tau$ are the same type variable;
- $(\sigma_1 \to \tau_1) \diamond (\sigma_2 \to \tau_2) = (\sigma_1 \diamond \sigma_2) \to (\tau_1 \diamond \tau_2)$ if $\sigma_1 \diamond \sigma_2 \neq \perp$ and $\tau_1 \diamond \tau_2 \neq \perp$;
- $\mathtt{class}\, t.R_1 \circ C_1 \diamond \mathtt{class}\, t.R_2 \circ C_2 = \mathtt{class}\, t.(R_1 \circ C_1 \diamond R_2 \circ C_2)$ if $R_1 \circ C_1 \diamond R_2 \circ C_2 \neq \perp$;
- $\tau_1 \diamond \tau_2 = \perp$ otherwise;
- $\sigma_{\Delta_1} \diamond \tau_{\Delta_2} = (\sigma \diamond \tau)_{\Delta_1 \sqcup \Delta_2}$ if $\sigma \diamond \tau \neq \perp$ and $\Delta_1 \sqcup \Delta_2 \neq \perp$;
- $\sigma_{\Delta_1} \diamond \tau_{\Delta_2} = \perp$ otherwise;
- $\langle\!\langle R_1 \mid m{:}\alpha_1 \rangle\!\rangle \diamond \langle\!\langle R_2 \mid m{:}\alpha_2 \rangle\!\rangle = \langle\!\langle R_1 \diamond R_2 \mid m{:}\alpha_1 \diamond \alpha_2 \rangle\!\rangle$ if $R_1 \diamond R_2 \neq \perp$ and $\alpha_1 \diamond \alpha_2 \neq \perp$;
- $\langle\!\langle R_1 \mid m{:}\alpha_1 \rangle\!\rangle \diamond \langle\!\langle R_2 \mid m{:}\alpha_2 \rangle\!\rangle = \perp$ otherwise;
- $\Gamma \diamond \Gamma' = \{x : \tau \diamond \tau' \mid x : \tau \in \Gamma, x : \tau' \in \Gamma'\} \cup \{u \preceq \tau \diamond \tau' \mid u \preceq \tau \in \Gamma, u \preceq \tau' \in \Gamma'\}$;
- $(e : \sigma) \diamond (e : \tau) = e : \sigma \diamond \tau$;
- $(\sigma \preceq \tau) \diamond (\sigma' \preceq \tau') = (\sigma \diamond \sigma') \preceq (\tau \diamond \tau')$.

### B.3. The Operators $\triangleleft$ and $\triangleright$.

The operators $\triangleleft$ and $\triangleright$ on types, labeled types, and pairs of rows are inductively defined as follows:

- $\tau_1 \bowtie \tau_2 = \tau$ if $\tau_1 = \tau_2 = \tau$ are the same type variable;
- $(\sigma_1 \to \tau_1) \bowtie (\sigma_2 \to \tau_2) = (\sigma_1 \diamondsuit \sigma_2) \to (\tau_1 \bowtie \tau_2)$ if $\sigma_1 \diamondsuit \sigma_2 \neq \bot$ and $\tau_1 \bowtie \tau_2 \neq \bot$;
- $\texttt{class}\, t.R_1 \circ C_1 \bowtie \texttt{class}\, t.R_2 \circ C_2 = \texttt{class}\, t.(R_1 \circ C_1 \bowtie R_2 \circ C_2)$
  if $R_1 \circ C_1 \bowtie R_2 \circ C_2 \neq \bot$;
- $\tau_1 \bowtie \tau_2 = \bot$ otherwise;
- $\sigma_{\Delta_1} \bowtie \tau_{\Delta_2} = (\sigma \bowtie \tau)_{\Delta_1 \sqcup \Delta_2}$ if $\sigma \bowtie \tau \neq \bot$ and $\Delta_1 \sqcup \Delta_2 \neq \bot$;
- $\sigma_{\Delta_1} \bowtie \tau_{\Delta_2} = \bot$ otherwise;
- $R_1 \circ C_1 \triangleright R_2 \circ C_2 = R \circ C$ where
  - $m : \alpha \in R$ iff $\alpha = \beta \triangleright \gamma \neq \bot$, and $m : \beta \in R_1$, $m : \gamma \in R_2$;
  - $m : \alpha \in C$ iff $\alpha = \beta \triangleright \gamma \neq \bot$, and $m : \beta \in R_1 \circ C_1$, $m : \gamma \in C_2$;
- $R_1 \circ C_1 \triangleleft R_2 \circ C_2 = R \circ C$ where
  - $m : \alpha \in R$ iff either $\alpha = \beta \triangleleft \gamma \neq \bot$, and $m : \beta \in R_1$, $m : \gamma \in R_2 \circ C_2$, or $m : \alpha \in R_1$ and $m \notin R_2 \circ C_2$;
  - $m : \alpha \in C$ iff either $\alpha = \beta \triangleleft \gamma \neq \bot$, and $m : \beta \in C_1$, $m : \gamma \in C_2$, or $m : \alpha \in C_1$ and $m \notin R_2 \circ C_2$;
- $R_1 \circ C_1 \bowtie R_2 \circ C_2 = \bot$ otherwise.

Here $\bowtie$ stands for $\triangleright$ or $\triangleleft$, while $\diamondsuit = \triangleleft$ if $\bowtie = \triangleright$, and $\diamondsuit = \triangleright$ if $\bowtie = \triangleleft$.

### B.4. Rewrite Rules for Contexts

$$\frac{\Gamma \vdash * \quad x \notin \Gamma}{\Gamma', x{:}\tau \vdash *}\ (exp\ var) \quad \leadsto \quad \frac{\Gamma \diamond \Gamma' \vdash * \quad x \notin \Gamma \diamond \Gamma'}{\Gamma \diamond \Gamma', x{:}\tau \vdash *}\ (exp\ var)$$

$$\frac{\Gamma \vdash * \quad u \notin \Gamma \quad u \notin \tau}{\Gamma', u \preceq \tau' \vdash *}\ (type\ var) \quad \leadsto \quad \frac{\begin{array}{l}\Gamma \diamond \Gamma' \vdash * \\ u \notin \Gamma \diamond \Gamma' \quad u \notin \tau \diamond \tau'\end{array}}{\Gamma \diamond \Gamma', u \preceq \tau \diamond \tau' \vdash *}\ (type\ var)$$

### B.5. Rewrite Rules for Subtyping

$$\frac{\Gamma \vdash *}{\Gamma' \vdash \tau \preceq \tau}\ (\preceq refl) \quad \leadsto \quad \frac{\Gamma \diamond \Gamma' \vdash *}{\Gamma \diamond \Gamma' \vdash \tau \preceq \tau}\ (\preceq refl)$$

$$\frac{\Gamma \vdash * \quad u \preceq \tau \in \Gamma}{\Gamma' \vdash u \preceq \tau'}\ (\preceq proj) \quad \leadsto \quad \frac{\Gamma'' \vdash * \quad u \preceq \tau'' \in \Gamma''}{\Gamma'' \vdash u \preceq \tau''}\ (\preceq proj)$$

where $\Gamma'' = ([u \preceq \tau \diamond \tau'/u \preceq \tau]\Gamma) \diamond \Gamma'$ and $u \preceq \tau'' \in \Gamma''$.

$$\frac{\Gamma \vdash \sigma \preceq \tau \quad \Gamma' \vdash \tau' \preceq \rho}{\Gamma'' \vdash \sigma' \preceq \rho'}\ (\preceq trans) \quad \leadsto \quad \frac{\Gamma''' \vdash \sigma'' \preceq \tau'' \quad \Gamma''' \vdash \tau'' \preceq \rho''}{\Gamma''' \vdash \sigma'' \preceq \rho''}\ (\preceq trans)$$

where $\Gamma''' = \Gamma \diamond \Gamma' \diamond \Gamma''$, $\sigma'' = (\sigma \lhd (\tau \diamond \tau')) \diamond (\sigma' \lhd (\rho \diamond \rho'))$,
$\tau'' = ((\sigma \diamond \sigma') \rhd \tau) \diamond (\tau' \lhd (\rho \diamond \rho'))$, $\rho'' = ((\sigma \diamond \sigma') \rhd \rho) \diamond ((\tau \diamond \tau') \rhd \rho')$.

$$\frac{\Gamma \vdash \sigma_1 \preceq \sigma_2 \quad \Gamma' \vdash \tau_1 \preceq \tau_2}{\Gamma'' \vdash \sigma'_2 \to \tau'_1 \preceq \sigma'_1 \to \tau'_2}\ (\preceq arrow) \quad \leadsto \quad \frac{\Gamma''' \vdash \sigma''_1 \preceq \sigma''_2 \quad \Gamma''' \vdash \tau''_1 \preceq \tau''_2}{\Gamma''' \vdash \sigma''_2 \to \tau''_1 \preceq \sigma''_1 \to \tau''_2}\ (\preceq arrow)$$

where $\Gamma''' = \Gamma \diamond \Gamma' \diamond \Gamma''$, $\sigma''_1 = (\sigma_1 \lhd (\sigma_2 \diamond \sigma'_2)) \diamond \sigma'_1$,
$\sigma''_2 = ((\sigma_1 \diamond \sigma'_1) \rhd \sigma_2) \diamond \sigma'_2$, $\tau''_1 = (\tau_1 \lhd (\tau_2 \diamond \tau'_2)) \diamond \tau'_1$, $\tau''_2 = ((\tau_1 \diamond \tau'_1) \rhd \tau_2) \diamond \tau'_2$.

$$\frac{\Gamma \vdash *}{\Gamma' \vdash \mathtt{class}\, t.\langle\!\langle R \mid m{:}\alpha \rangle\!\rangle \circ C \preceq \mathtt{class}\, t.R \circ \langle\!\langle C \mid m{:}\alpha \rangle\!\rangle}\ (\preceq shift)$$

$\leadsto$

$$\frac{\Gamma \diamond \Gamma' \vdash *}{\Gamma \diamond \Gamma' \vdash \mathtt{class}\, t.\langle\!\langle R \mid m{:}\alpha \rangle\!\rangle \circ C \preceq \mathtt{class}\, t.R \circ \langle\!\langle C \mid m{:}\alpha \rangle\!\rangle}\ (\preceq shift)$$

$$\frac{\Gamma \vdash *}{\Gamma' \vdash \mathtt{class}\, t.R \circ \langle\!\langle C \mid \overline{p{:}\gamma} \rangle\!\rangle \preceq \mathtt{class}\, t.R \circ C}\ (\preceq hide)$$

$\leadsto$

$$\frac{\Gamma \diamond \Gamma' \vdash *}{\Gamma \diamond \Gamma' \vdash \mathtt{class}\, t.R \circ \langle\!\langle C \mid \overline{p{:}\gamma} \rangle\!\rangle \preceq \mathtt{class}\, t.R \circ C}\ (\preceq hide)$$

### B.6. Rewrite Rules for Typing Terms: Part I

$$\frac{\Gamma \vdash * \quad x:\tau \in \Gamma}{\Gamma' \vdash x:\tau'}\ (proj) \quad \rightsquigarrow \quad \frac{\Gamma'' \vdash * \quad x:\tau'' \in \Gamma''}{\Gamma'' \vdash x:\tau''}\ (proj)$$

where $\Gamma'' = ([x:\tau \diamond \tau'/x:\tau]\Gamma) \diamond \Gamma'$ and $x:\tau'' \in \Gamma''$.

$$\frac{\Gamma,\, x{:}\tau_1 \vdash e:\tau_2}{\Gamma' \vdash \lambda x.e:\tau_1' \to \tau_2'}\ (exp\ abs) \quad \rightsquigarrow \quad \frac{\Gamma \diamond \Gamma',\, x{:}\tau_1 \diamond \tau_1' \vdash e:\tau_2 \diamond \tau_2'}{\Gamma \diamond \Gamma' \vdash \lambda x.e:\tau_1 \diamond \tau_1' \to \tau_2 \diamond \tau_2'}\ (exp\ abs)$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1:\tau_1 \to \tau_2\\ \Gamma' \vdash e_2:\tau_1'\end{array}}{\Gamma'' \vdash e_1 e_2:\tau_2'}\ (exp\ appl) \quad \rightsquigarrow \quad \frac{\begin{array}{l}\Gamma''' \vdash e_1:\tau_1 \diamond \tau_1' \to \tau_2 \diamond \tau_2'\\ \Gamma''' \vdash e_2:\tau_1 \diamond \tau_1'\end{array}}{\Gamma''' \vdash e_1 e_2:\tau_2 \diamond \tau_2'}\ (exp\ appl)$$

where $\Gamma''' = \Gamma \diamond \Gamma' \diamond \Gamma''$.

$$\frac{\Gamma \vdash e:\sigma \quad \Gamma' \vdash \sigma' \preceq \tau}{\Gamma'' \vdash e:\tau'}\ (\preceq) \quad \rightsquigarrow \quad \frac{\begin{array}{l}\Gamma''' \vdash e:(\sigma \diamond \sigma') \triangleleft (\tau \diamond \tau')\\ \Gamma''' \vdash (\sigma \diamond \sigma') \triangleleft (\tau \diamond \tau') \preceq (\sigma \diamond \sigma') \triangleright (\tau \diamond \tau')\end{array}}{\Gamma''' \vdash e:(\sigma \diamond \sigma') \triangleright (\tau \diamond \tau')}\ (\preceq)$$

where $\Gamma''' = \Gamma \diamond \Gamma' \diamond \Gamma''$.

$$\frac{\Gamma \vdash *}{\Gamma' \vdash \langle\rangle : \mathtt{class}\, t.\langle\!\langle\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle}\ (empty) \quad \rightsquigarrow \quad \frac{\Gamma \diamond \Gamma' \vdash *}{\Gamma \diamond \Gamma' \vdash \langle\rangle : \mathtt{class}\, t.\langle\!\langle\rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle}\ (empty)$$

$$\frac{\begin{array}{l}\Gamma \vdash e:\sigma\\ \Gamma' \vdash \sigma' \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\tau_\Delta \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle\end{array}}{\Gamma'' \vdash e \Leftarrow n : [\sigma''/t]\tau'}\ (send) \quad \rightsquigarrow \quad \frac{\Gamma''' \vdash e:\rho_1 \quad \Gamma''' \vdash \rho_1 \preceq \rho_2}{\Gamma''' \vdash e \Leftarrow n : [\rho_1/t]\rho}\ (send)$$

where $\Gamma''' = \Gamma \diamond \Gamma' \diamond \Gamma''$, $\tau''_{\Delta'} = \tau_\Delta \diamond \tau'_{\{\overline{m}\}}$,
$\rho_1 = (\sigma \diamond \sigma' \diamond \sigma'') \triangleleft \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\tau''_{\Delta'} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$,
$\rho_2 = (\sigma \diamond \sigma' \diamond \sigma'') \triangleright \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, n{:}\tau''_{\Delta'} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle = \mathtt{class}\, t.\langle\!\langle R \mid n{:}\rho_{\Delta''} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$.

$$\frac{\begin{array}{l}\Gamma \vdash e : \mathtt{class}\, t.\langle\!\langle n{:}\tau_\Delta \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\alpha} \rangle\!\rangle\\ \Gamma' \vdash \sigma \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha'}, n{:}\tau'_{\Delta'} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle\end{array}}{\Gamma'' \vdash e \hookleftarrow n : [\sigma'/t](t \to \tau'')}\ (search)$$

$\rightsquigarrow$

$$\frac{\begin{array}{l}\Gamma''' \vdash e : \mathtt{class}\, t.\langle\!\langle n{:}\tau'''_{\Delta''} \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\beta} \rangle\!\rangle\\ \Gamma''' \vdash \rho \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\beta}, n{:}\tau^{iv}_{\Delta'''} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle\end{array}}{\Gamma''' \vdash e \hookleftarrow n : [\rho/t](t \to \tau^{iv})}\ (search)$$

where $\Gamma''' = \Gamma \diamond \Gamma' \diamond \Gamma''$, $\tau'''_{\Delta''} = \tau_\Delta \diamond \tau'_{\Delta'} \diamond \tau''_{\{\overline{m}\}}$,
$\rho = (\sigma \diamond \sigma') \triangleleft \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha \diamond \alpha'}, n{:}\tau^{iv}_{\Delta'''} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$,
$\mathtt{class}\, t.\langle\!\langle \overline{m{:}\beta}, n{:}\tau'^{v}_{\Delta'''} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle = (\sigma \diamond \sigma') \triangleright \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha \diamond \alpha'}, n{:}\tau'''_{\Delta''} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$.

### B.7. Rewrite Rules for Typing Terms: Part II

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \mathtt{class}\, t.R \circ \langle\!\langle C \mid n{:}\tau_{\Delta} \rangle\!\rangle \qquad \overline{m{:}\alpha} \in R \circ C \\ \Gamma', u \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha'}, n{:}\tau'_{\Delta'} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \vdash e_2 : [u/t](t \to \tau'')\end{array}}{\Gamma'' \vdash \langle e_1 \leftarrow\!\circ\; n{=}e_2 \rangle : \mathtt{class}\, t.\langle\!\langle R' \mid n{:}\tau'''_{\Delta''} \rangle\!\rangle \circ C'} \quad (comp)$$

$\leadsto$

$$\frac{\begin{array}{l}\Gamma''' \vdash e_1 : \mathtt{class}\, t.R'' \circ \langle\!\langle C'' \mid n{:}\tau^{\iota v}_{\Delta'''} \rangle\!\rangle \qquad \overline{m{:}\alpha''} \in R'' \circ C'' \\ \Gamma''', u \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha''}, n{:}\tau^{\iota v}_{\Delta'''} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \vdash e_2 : [u/t](t \to \tau^{\iota v})\end{array}}{\Gamma''' \vdash \langle e_1 \leftarrow\!\circ\; n{=}e_2 \rangle : \mathtt{class}\, t.\langle\!\langle R'' \mid n{:}\tau^{\iota v}_{\Delta'''} \rangle\!\rangle \circ C''} \quad (comp)$$

where $\Gamma''' = \Gamma \diamond \Gamma' \diamond \Gamma''$, $\tau^{\iota v}_{\Delta'''} = \tau_{\Delta} \diamond \tau'_{\Delta'} \diamond \tau'''_{\Delta''} \diamond \tau''_{\{\overline{m}\}}$,
$R'' \circ C'' = ([\overline{m{:}\alpha \diamond \alpha'}/\overline{m{:}\alpha}]R \circ C) \diamond R' \circ C'$ and $\overline{m{:}\alpha''} \in R'' \circ C''$.

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \sigma \qquad \Gamma' \vdash \sigma' \preceq \mathtt{class}\, t.\langle\!\langle n{:}\tau_{\Delta} \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\alpha} \rangle\!\rangle \\ \Gamma'', u \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha'}, n{:}\tau'_{\Delta'} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \vdash e_2 : [u/t](t \to \tau'')\end{array}}{\Gamma''' \vdash \langle e_1 \leftarrow\!\circ\; n{=}e_2 \rangle : \sigma''} \quad (over)$$

$\leadsto$

$$\frac{\begin{array}{l}\Gamma^{\iota v} \vdash e_1 : \rho \qquad \Gamma^{\iota v} \vdash \rho \preceq \mathtt{class}\, t.\langle\!\langle n{:}\tau^{iv}_{\Delta'''} \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\beta} \rangle\!\rangle \\ \Gamma^{\iota v}, u \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\beta}, n{:}\tau^{iv}_{\Delta'''} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \vdash e_2 : [u/t](t \to \tau^{\prime v})\end{array}}{\Gamma^{\iota v} \vdash \langle e_1 \leftarrow\!\circ\; n{=}e_2 \rangle : \rho} \quad (over)$$

where $\Gamma^{\iota v} = \Gamma \diamond \Gamma' \diamond \Gamma'' \diamond \Gamma'''$, $\tau'''_{\Delta''} = \tau_{\Delta} \diamond \tau'_{\Delta'} \diamond \tau''_{\{\overline{m}\}}$,
$\rho = (\sigma \diamond \sigma') \triangleleft \mathtt{class}\, t.\langle\!\langle n{:}\tau'''_{\Delta''} \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\alpha \diamond \alpha'} \rangle\!\rangle$,
$\mathtt{class}\, t.\langle\!\langle n{:}\tau^{\prime v}_{\Delta'''} \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\beta} \rangle\!\rangle = (\sigma \diamond \sigma') \triangleright \mathtt{class}\, t.\langle\!\langle n{:}\tau'''_{\Delta''} \rangle\!\rangle \circ \langle\!\langle \overline{m{:}\alpha \diamond \alpha'} \rangle\!\rangle$.

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \mathtt{class}\, t.R \circ C \qquad \overline{m{:}\alpha} \in R \circ C \qquad n, \overline{p} \notin \mathcal{M}(R) \cup \mathcal{M}(C) \\ \Gamma', u \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha'}, \overline{p{:}\gamma}, n{:}\tau_{\Delta} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \vdash e_2 : [u/t](t \to \tau')\end{array}}{\Gamma'' \vdash \langle e_1 \leftarrow\!\circ\; n{=}e_2 \rangle : \mathtt{class}\, t.\langle\!\langle R' \mid n{:}\tau''_{\Delta'} \rangle\!\rangle \circ \langle\!\langle C' \mid \overline{p{:}\gamma'} \rangle\!\rangle} \quad (ext)$$

$\leadsto$

$$\frac{\begin{array}{l}\Gamma''' \vdash e_1 : \mathtt{class}\, t.R'' \circ C'' \qquad \overline{m{:}\alpha''} \in R'' \circ C'' \qquad n, \overline{p} \notin \mathcal{M}(R'') \cup \mathcal{M}(C'') \\ \Gamma''', u \preceq \mathtt{class}\, t.\langle\!\langle \overline{m{:}\alpha''}, \overline{p{:}\gamma \diamond \gamma'}, n{:}\tau'''_{\Delta''} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \vdash e_2 : [u/t](t \to \tau''')\end{array}}{\Gamma''' \vdash \langle e_1 \leftarrow\!\circ\; n{=}e_2 \rangle : \mathtt{class}\, t.\langle\!\langle R'' \mid n{:}\tau'''_{\Delta''} \rangle\!\rangle \circ \langle\!\langle C'' \mid \overline{p{:}\gamma \diamond \gamma'} \rangle\!\rangle} \quad (ext)$$

where $\Gamma''' = \Gamma \diamond \Gamma' \diamond \Gamma''$, $\tau'''_{\Delta''} = \tau_{\Delta} \diamond \tau'_{\{\overline{m},\overline{p}\}} \diamond \tau''_{\Delta'}$,
$R'' \circ C'' = ([\overline{m{:}\alpha \diamond \alpha'}/\overline{m{:}\alpha}]R \circ C) \diamond R' \circ C'$ and $\overline{m{:}\alpha''} \in R'' \circ C''$.