Object Calculi with Dynamic Messages

Michele Bugliesi*
Dip. di Informatica
Università "Ca' Foscari" di Venezia
Via Torino 155, I-30173 Mestre (VE), Italy
e-mail: michele@dsi.unive.it

Silvia Crafa
Dip. di Matematica
Università di Padova
Via Belzoni 7, I-35131 Padova, Italy
e-mail: crafas@math.unipd.it

Abstract

Dynamic messages, as proposed in [Nis98], are first-class expressions that may occur as messages within programs: being first-class, they may dynamically be bound to program variables, and evaluated to "ordinary" messages during the computation. We present an extension of Abadi and Cardelli's typed calculus $\mathbf{FOb}_{<:}$ [AC96] and a type system that give provision for dynamic messages. The new type system retains the flexibility and expressive power of the original system of [Nis98] while relying on a new class of types, called *message types*, to capture the desired typing of dynamic messages. We prove type soundness and the existence of minimum types for the new system. We also study the formal relationships between object types and message types, and discuss how message types can be encoded in terms of object types.

1 Introduction

In most-object oriented languages messages are formed around a method label – a constant that identifies the method to be invoked – and a list of arguments for the invocation. *Dynamic messages*, as proposed in [Nis98], provide additional flexibility by allowing methods to be invoked by means of expressions that reduce (i.e. evaluate) to "ordinary" messages during the computation. Dynamic messages are thus first-class expressions that may occur as messages within programs: being first-class, they may dynamically be bound to program variables, and hence allow powerful forms of higher-order programming where code (functions or methods) may be written that abstracts over the methods actually invoked by a message.

Dynamic messages have been studied in the context of statically typed formal calculi by S. Nishimura in [Nis98]. In that paper the author presents an untyped object calculus with dynamic messages, a sound type system, and a principal type inference algorithm. While Nishimura's type system gives important and elegant foundations for dynamic messages, the typing discipline that results from his system seems to depart significantly from the foundational models of objects and object types found in the previous literature [AC96, Bru94, BSvG95, PT94, Mic90, FHM94, FM95, BCP97, CHC90]. In his second-order system, no type is provided that directly represents the structure of objects and messages. Instead, the type information is expressed at the kind level: an object type has the form $t :: \kappa$, where t is a type variable that indexes the kind κ which, in turn, encodes all the information relative to the object's structure. Given this completely flat type structure, the system does not seem to lend itself to direct or smooth integration with any of the foundational models listed above.

The presupposed difficulty discussed above represents the main motivation of the work we report in this paper. We present an extension of the Abadi and Cardelli's typed calculus $\mathbf{FOb}_{<:}$ [AC96] and a type system that give provision for dynamic messages. Unlike in the original proposal of [Nis98], the treatment of dynamic messages in our type system is first-order, thus allowing a natural integration of the new feature with the original system for $\mathbf{FOb}_{<:}$.

The new type system retains the flexibility and expressive power of the original system of [Nis98] while relying on a new class of types – that we call *message-types* – to capture the desired typing of dynamic messages. Message types are, in certain respects, similar to the *variant-record* [CW85] or *sum* types found in traditional programming languages: as we show in the paper (see Section 6) a formal counterpart of Ghelli' encoding of sum types into record types [Ghe90] within $F_{<:}$ can be given to show that message types can be represented in terms of object types.

*This paper was initiated and partly developed when the author was at the Department of Mathematics at the University of Padova.

The extension of the type system with message types scales smoothly from the first-order system to the system with Self Types and structural rules from [AC96]. We give a proof of type soundness and of the existence of minimum types for the most powerful of these systems.

We organize the rest of the paper as follows. In Section 2 we review the main concepts and constructs of the $\mathbf{FOb}_{<:}$ calculus of [AC96]. In Section 3, we describe the extension of this calculus with dynamic messages, introducing the new class of message-types, and defining the operational semantics for the extended calculus. In Section 4 we describe the typing and subtyping rules for the new calculus, and illustrate their use with a few examples. In Section 5 we give the proofs of type soundness and of existence of minimum types. In Section 6 we then study a formal relationship between object types and message types discussing an encoding of the latter in terms of the former. We conclude in Section 7 with some final remarks. Two separate appendices collect the typing and subtyping rules of $\mathbf{FOb}_{<:}$ and of the the new type system.

2 Review of Object Calculi

In this section, we give an informal review of the concepts and constructs of Abadi and Cardelli's ς-calculus [AC96]. The core calculus is based a minimal set of construct and primitives: object formation, method invocation and method update. Additional features, including the functional constructs of λ-abstraction and application may then be included, encoding them in terms of the object-related forms, or taking them as primitives.

2.1 The Untyped ς-calculus.

An untyped object $[l_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$ is a collection of method labels l_i, with each label associated with a corresponding method body: labels of an object are assumed to be distinct and their relative order is immaterial. The object containing a given method is called the method's host object. Each method body $\varsigma(x_i)b_i$ is an abstraction of x_i, the *self* parameter, which represents the host object: sibling methods may be invoked within the body using the self parameter.

The interpretation of *self* as an abstraction of the host object is enforced by the *self-substitution* semantics of method invocation. Methods are invoked by sending corresponding messages to their host objects: a message-sent has the syntax $a.l$ where a is the recipient object and l, the message, a method label requesting an invocation of the corresponding method from a. If $a \equiv [\ldots, l_j = \varsigma(x_j)b_j, \ldots]$, the invocation $a.l_j$ reduces to $b_j[a/x_j]$, the expression that arises from substituting a (the host object) for every free occurrence of x_j in the body of the method.

The final construct of the ς-calculus is method override, a primitive that allows the behavior of an object to be modified by replacing one of the object's methods. A method override is an expression of the form $a.l_j := \varsigma(y)b$, which replaces the method associated with l_j in a with the new body $\varsigma(y)b$. The semantics is functional: an override produces a modified copy of the object to which it is applied.

2.2 Object Types and Self Types

In its simplest, first-order, form, an object type is a type expression of the form $[l_i : B_i \ ^{i \in 1..n}]$, denoting the collection of objects that have methods $l_1, \ldots, l_n$ which, when invoked, return values of types, respectively, $B_1, \ldots, B_n$.

To allow satisfactory typings for objects whose methods return or modify *self*, the syntax of object types is extended to include a construct for recursion. In this extended syntax, an object type is an expression of the form $Obj(X)[l_i : B_i\{X\} \ ^{i \in 1..n}]$, where Obj is a binder, X is a type variable, and the notation $B\{X\}$ indicates that X may occur free in B. The binder Obj scopes over the B_i's, and the bound variable X may occur free within the scope of the binder, with every free occurrence referring to the object type itself. Obj-types are a form of recursively-defined types, even though Obj should not to be understood as a standard fixed-point operator: as we shall discuss shortly, Obj-types obey typing and subtyping rules that are different from the corresponding rules for recursive types[1].

2.3 Typed Syntax and Typing Rules

Typed object calculi attach to every object a type describing the object's structure. A typed object is an expression of the form $obj(X = A)[l_i = \varsigma(x_i : X)b_i \ ^{i \in 1..n}]$, where A

[1] Obj-types are called *SelfTypes* by Abadi and Cardelli. As they show in [AC96], SelfTypes can be encoded within the second-order ς-calculus by a combination of recursion and bounded existential types. Here, however, we take them as primitive, and define ad hoc rules, as in [AC96] (Chap. 16).

is the object type $Obj(X)[l_i : B_i\{X\}^{\ i\in 1..n}]$. The *self* parameters have the same type, X, in all the methods of the object, and this type is required to be equal to the type A of object itself: this guarantees that the self-substitution involved in the reduction of a method invocation is well typed. Also note that object types exhibit only the result types B_i of their component methods, disregarding the type of the self variables: no information is missing, however, because the type of self is just the type A associated with the host object. The typing rule for objects, given below, should help explain and motivate the format of the construct for object formation.

$$\text{(Val Obj)}\qquad A \equiv Obj(X)[l_i : B_i\{X\}^{\ i\in 1..n}]$$
$$\frac{\Gamma, x_i : A \vdash b_i\{A\} : B_i\{A\} \qquad \forall i \in 1..n}{\Gamma \vdash obj(X = A)[l_i = \varsigma(x_i : X)b_i\{X\}^{\ i\in 1..n}]\ : A}$$

The syntax of method update is also affected in the change from untyped to typed syntax. A typed method update is written $a.l_j := (Y <: A)\varsigma(x : Y)b$: the type variable Y, used as the type of *self* in the method body provided in the update represents the (possibly unknown) type A of the object a being updated. The need for the type variable Y is best explained looking at the corresponding typing rule:

$$\text{(Val Update)}\qquad A' \equiv Obj(X)[l_i : B_i\{X\}^{\ i\in 1...n}]$$
$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A <: A' \quad \Gamma, Y <: A, x : Y \vdash\ b : B_j\{Y\}}{\Gamma \vdash a.l_j := (Y <: A)\varsigma(x : Y)b : A}$$

An override for a method l_j on the object a requires a to have a type containing the method name l_j being replaced by the override. This is insured by the judgement $\Gamma \vdash A <: A'$ in the premises of the rule. The remaining two judgements in the premise insist, respectively, that the override preserve the type of the object being updated, and that the new body provided for l_j have the same type B_j as the original body. Note, however, that the typing of the new body is expressed in terms of the variable Y rather than in terms of A: this is required for soundness, since the type A may happen to be a proper super-type of the "true" type of a, obtained by a number of subsumption steps. Also note that A may either be an *Obj* type, or else an unknown type (i.e., a type variable) occurring (bounded) in the context Γ. Rules like (Val Update) are sometimes referred to as *structural rules* [AC96], and their use is critical for an adequate rendering of SelfTypes.

2.4 Subtyping

The subtyping rules for objects resemble the corresponding subtyping rules for (recursive) records in functional calculi. In the first-order case, the subtyping relation between two object types A and B states that A is a subtype of B, written $A <: B$, if "B has fewer methods than A". Unlike record types, however, object types are invariant in their components: the subtyping relationship $[l_i : B_i^{\ i\in 1..n+m}] <: [l_i : B_i'^{\ i\in 1..n}]$ requires B_i and B_i' ($i \in 1..n$) to be the same type. Invariance for components of object types is required for soundness, essentially because all components are both readable and writable. Subtyping over SelfTypes is defined as a generalization of the subtyping relation we just described, with the additional constraint that the component types be covariant in the bound variable (see Appendix A). As discussed in [AC96] the additional covariance constraint, is needed for sound uses of method invocation in the presence of object subsumption.

2.5 $\mathbf{FOb}_{<:}$: Objects and Functions

As we anticipated, extended calculi may be defined, that include additional features and constructs in the core ς-calculus. In the rest of the paper, we will consider one such calculus, called $\mathbf{FOb}_{<:}$, that extends the typed object calculus with SelfTypes and structural rules we just illustrated, with typed λ-abstraction and function application. The complete set of typing rules for $\mathbf{FOb}_{<:}$ is reported in Appendix A.

3 Dynamic Messages in $\mathbf{FOb}_{<:}$

$\mathbf{FOb}^{\mathbf{dyn}}_{<:}$ is a proper extension of $\mathbf{FOb}_{<:}$, the typed calculus we illustrated in the previous section, with dynamic messages.

3.1 Syntax

Types and expressions are defined in Table 1 The syntax (as well as the intended semantics) of variables, abstractions, applications and object formation are exactly as in $\mathbf{FOb}_{<:}$. Instead, the syntax of messages, message-sends and method updates generalize the corresponding constructs from $\mathbf{FOb}_{<:}$ by treating messages as first-order values. The generalization arises from allowing arbitrary

a, b, m	$::=$	x	variable
		$\lambda(x : A)b$	abstraction
		$b(a)$	application
		$obj(X = A)[l_i = \varsigma(x_i : X)b_i\{x_i, X\}\ ^{i \in 1..n}]$	object formation
		$a.m := (Y <: A)\varsigma(x : Y)b$	method update
		$a \Leftarrow m$	message send
		$l\langle b_1, ..., b_k\rangle$	message ($k \geq 0$)
A, B	$::=$	X	type variable
		Top	the biggest type
		$A \rightarrow B$	function type
		$Obj(X)[l_i : B_i\{X\}\ ^{i \in 1..n}]$	object type ($n \geq 0$, l_i distinct, B_i covariant in X)
		$\langle\!\langle l_i\langle B_{i,1}, ..., B_{i,k}\rangle^{i \in 1..n \quad k \in 1..m}\rangle\!\rangle$	message type ($n \geq 0$, l_i distinct)

Table 1: Types and Expressions

expressions to occur in the message position of a message-send and in the label position of a method update: messages are thus computed values in $\mathbf{FOb}^{\mathbf{dyn}}_{<:}$ rather than constant labels as in $\mathbf{FOb}_{<:}$. The expected values of a and m in the expression $a \Leftarrow m$ are, respectively, an object containing a, say, l method, and a message $l\langle b_1...b_k\rangle$, invoking the method labeled by l in a with arguments $b_1 \ldots b_k$. As we shall prove, the evaluation of $a \Leftarrow m$ is guaranteed to follow the expected pattern whenever the expression is well typed. Similarly, the expected value of m in the expression $a.m := (Y <: A)\varsigma(x : Y)b$ is a constant label that refers to a method in a, and whenever this expression is well typed, the semantics of the update is just as in $\mathbf{FOb}_{<:}$.

The syntax of types parallels the syntax of expressions discussed above. Again, the first four productions are exactly as in $\mathbf{FOb}_{<:}$. The object type $Obj(X)[l_i : B_i\{X\}\ ^{i \in 1..n}]$ is the type of the objects containing (at least) the l_i methods ($i \in 1..n$). When invoked, each of the l_i's returns a value of type B_i, with every free occurrence of X substituted by the object type itself. As discussed in Section 2, the B_i's must be covariant in X: as in [AC96], we use covariance as a syntactic condition for well formedness of object types.

A new class of types describes the structure of messages. A message type $\langle\!\langle l_i\langle B_{i,1}, ..., B_{i,k}\rangle^{i \in 1..n \quad k \in 1..m}\rangle\!\rangle$ is the type assigned to the messages that may invoke one of the methods labeled l_i ($i \in 1..n$) with k arguments of type $B_{i,1}, ..., B_{i,k}$. Message types are similar to the *variant-record* or *sum* types found in traditional programming languages: the difference is that a message type does not specify the return types of its labels, as the return types depend on the objects where (the methods associated to) the labels reside. To ease the notation, in the following we omit writing $k \in 1..m$ in the syntax of message types, and take k as a dummy variable that may be different for any i.

3.2 Operational Semantics

Following the standard practice, the operational semantics of $\mathbf{FOb}^{\mathbf{dyn}}_{<:}$ is lazy: it does not work under λ-abstractions, and similarly it defers reducing under primitives of object and message 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 one of the following:

$$\begin{array}{rl} v ::= & \lambda(x : A)b \\ & obj(X = A)[l_i = \varsigma(x_i : X)b_i\ ^{i \in 1..n}] \\ & l\langle a_1, \ldots, a_k\rangle \quad k \geq 0 \end{array}$$

The operational semantics is defined by cases below. The evaluation of function application is standard. Evaluation of overrides is defined by a simple generalization of the corresponding rule of $\mathbf{FOb}_{<:}$. Message-sends also are evaluated, essentially, as in $\mathbf{FOb}_{<:}$, by self-substitution: the differences are that (i) reducing an invocation may require prior evaluation of the message, and (ii) (some of) the arguments of the call may be passed along directly with the message.

(Red Val)

$$\frac{}{v \rightsquigarrow v}$$

(Red App)

$$\frac{b \rightsquigarrow \lambda(x:A)e\{x\} \quad e\{a\} \rightsquigarrow v}{b(a) \rightsquigarrow v}$$

(Red Update)

$$\frac{a \rightsquigarrow obj(X = A')[l_i = \varsigma(x_i : X)b_i^{i \in 1..n}] \qquad m \rightsquigarrow l_j \qquad j \in 1..n}{a.m := (Y <: A)\varsigma(x : Y)b\{Y\} \rightsquigarrow obj(X = A')[l_j = \varsigma(x : X)b\{X\}, l_i = \varsigma(x_i : X)b_i\{X\}^{i \in 1..n-j}]}$$

(Red Send)

$$\frac{\begin{array}{l} v' \equiv obj(X = A')[l_i = \varsigma(x_i : X)b_i\{X, x_i\}^{i \in 1..n}] \\ a \rightsquigarrow v' \qquad m \rightsquigarrow l_j\langle a_1, \ldots, a_k\rangle \qquad (..(b_j\{A', v'\})a_1..)a_k \rightsquigarrow v \qquad j \in 1..n \end{array}}{a \Leftarrow m \rightsquigarrow v}$$

Table 2: Reduction Rules

4 Typing and Subtyping

For the most part, the typing and subtyping rules of $\mathbf{FOb}^{\mathbf{dyn}}_{<:}$ are as in the type system of $\mathbf{FOb}_{<:}$. The new rules, for messages, message-sends and updates, are illustrated below.

The rule to type a message is routine.

$$\frac{\Gamma \vdash a_i : B_i \qquad i \in 1...k}{\Gamma \vdash l\langle a_1...a_k\rangle : \langle\!\langle l\langle B_1, \ldots, B_k\rangle\rangle\!\rangle}$$

The type of a message contains the label mentioned in the message, with input types corresponding to the types of the arguments passed along with the message.

The rule for message-sends generalizes the corresponding rule (Val Select) from [AC96].

$$\frac{\begin{array}{c} A' \equiv Obj(X)[l_i : B_{i,1}\{X\} \cdots \to \cdots B_{i,k}\{X\} \to C\{X\}^{i \in 1...n}] \\ \Gamma \vdash a{:}A \quad \Gamma \vdash A <: A' \quad \Gamma \vdash m : \langle\!\langle l_i\langle B_{i,1}\{A\} \ldots B_{i,k}\{A\}\rangle^{i \in 1..n}\rangle\!\rangle \end{array}}{\Gamma \vdash a \Leftarrow m : C\{A\}}$$

As in [AC96] we use a structural rule allowing the type A of the recipient object a to be a (possibly unknown) subtype of the type A': this type must be an object type containing (possibly a superset of) the labels l_i's listed in the type of the message that is sent to the recipient, with corresponding types $B_{i,1}, \ldots, B_{i,k}$. The return type, C, is assumed to be the same for all of the l_i's: this assumption is required to uniquely determine the type of the invocation, which is obtained by a type substitution (mimicking the self-substitution used in the operational semantics) that replaces the true type A of a for the free occurrences of X.

A similar generalization of the typing rule for overrides is used to defined the typing of a method update in $\mathbf{FOb}^{\mathbf{dyn}}_{<:}$.

$$\frac{\begin{array}{l} A' \equiv Obj(X)[l_i : B\{X\}\ ^{i \in 1..n}] \\ \Gamma \vdash a : A \qquad \Gamma \vdash A <: A' \\ \Gamma \vdash m : \langle\!\langle l_i\ ^{i \in 1..n}\rangle\!\rangle \qquad \Gamma, Y <: A, x : Y \vdash\ b : B\{Y\} \end{array}}{\Gamma \vdash a.m := (Y <: A)\varsigma(x : Y)b : A}$$

The rule requires the type A' to list the set of labels l_i's occurring in type of m. The judgement $\Gamma \vdash A <: A'$ insures then that all such labels are also contained in the type A of the object being updated. As in the typing of message sends, the labels occurring in A' are required to have the same type to ensure a sound typing of the update.

A final rule defines the subtype relation over message types. As for variant records [CW85], supertype of any given message type may be obtained by (i) extending the set of component labels, and (ii) by taking supertypes at the components.

$$\frac{\Gamma \vdash B'_{i,j} <: B_{i,j}}{\Gamma \vdash \langle\!\langle l_i\langle B'_{i,1} \ldots B'_{i,k}\rangle^{i \in 1...n}\rangle\!\rangle <: \langle\!\langle l_i\langle B_{i,1} \ldots B_{i,k}\rangle^{i \in 1...n+m}\rangle\!\rangle}$$

Note that message types are covariant in their components, unlike object types (cf. Appendix A), that are instead invariant. Covariance is natural for message types as messages are read-only values.

4.1 Examples of Typing

We conclude this section with a few examples that illustrate the behavior of the type system, and the use of dynamic messages in programming.

As a first example, consider the object o defined as follows:

$$\begin{aligned} o \stackrel{\triangle}{=}\ & \mathit{obj}(X = A) \\ & [\mathbf{x} = \varsigma(s : X)0, \\ & \ \mathbf{add} = \varsigma(s : X)\lambda(i : \mathit{int})(s \Leftarrow \mathbf{x}) + i, \\ & \ \mathbf{set} = \varsigma(s_1 : X)\lambda(i : \mathit{int})s_1.\mathbf{x} := (Y <: A)\varsigma(s_2 : Y)i] \end{aligned}$$

The object has three methods: a integer field $\mathbf{x}$, an **add** method that returns the sum of $\mathbf{x}$ and the value of the argument passed for the parameter i, and a **set** method that sets the $\mathbf{x}$ field. A routine check verifies that for $A \equiv \mathit{Obj}(X)[\mathbf{x} : \mathit{int}, \mathbf{add} : \mathit{int} \to \mathit{int}, \mathbf{set} : \mathit{int} \to X]$, the judgement $\vdash o : A$ is derivable in the type system. Given this typing, we may now write code fragments that abstract over messages to o.

$$\begin{array}{llll} a : \mathit{int} & \stackrel{\triangle}{=} & \lambda(m : \langle\!\langle \mathbf{x}, \mathbf{add}\langle \mathit{int}\rangle \rangle\!\rangle)\, o \Leftarrow m & \textbf{OK} \\ a : \mathit{int} \to A & \stackrel{\triangle}{=} & \lambda(m : \langle\!\langle \mathbf{set} \rangle\!\rangle)\, o \Leftarrow m & \textbf{OK} \\ a : \mathit{any\ type} & \stackrel{\triangle}{=} & \lambda(m : \langle\!\langle \mathbf{set}, \mathbf{x} \rangle\!\rangle)\, o \Leftarrow m & \textbf{Error} \end{array}$$

The last expression does not typecheck, as the return type of the **set** and $\mathbf{x}$ methods of o are different and hence the type of the invocation may not be determined uniquely.

The next example illustrates the use of dynamic messages in higher-order object-oriented programming.

$$\begin{aligned} \mathbf{w} : \textsc{Win} \stackrel{\triangle}{=}\ & \mathit{obj}(X = \textsc{Win}) \\ & [\mathbf{wact}_i = \ldots^{i \in 1..n}, \\ & \ \mathbf{wev} = \varsigma(x : X)\lambda(m : \textsc{Wmsg})(x \Leftarrow m) \Leftarrow \mathbf{wev}] \end{aligned}$$

The object $\mathbf{w}$ implements a window object, with the $\mathbf{wact}_i$ methods associated with the different window actions, and a dispatcher method **wev** that behaves like a server for the possible window events. The dispatcher is implemented as an infinite loop that first invokes the method corresponding to the message passed as argument, and then invokes itself recursively. To allow the recursive invocation, the $\mathbf{wact}_i$ methods should be coded so as to return *self* after performing the action. It is again a routine check to verify that the definition of $\mathbf{w}$ typechecks if we define the types Win and Wmsg as follows:

$$\begin{aligned} \textsc{WinMsg} &\equiv \langle\!\langle \mathbf{wact}_i\langle B_{i,1}, \ldots, B_{i,k}\rangle^{i \in 1..n} \rangle\!\rangle \\ \textsc{Win} &\equiv \mathit{Obj}(X)[\mathbf{wact}_i : B_{i,1} \to \ldots \to B_{i,k} \to X^{i \in 1..n}, \\ &\qquad \mathbf{wev} : \textsc{Wmsg} \to \textsc{Unit}] \end{aligned}$$

A final example shows a simple but efficient use of the generalized update construct. Consider an object that collects a set of fields containing values to be read or written. In a realistic implementation, such fields would be protected from direct access, enriching the interface of the object with **get** and **set** methods for each field. In $\mathbf{FOb}^{\mathbf{dyn}}_{<:}$, an alternative solution exists that relies on the generalized update construct: two methods that abstract over the field name may be used to *get* or *set* all of the fields (as long as the fields have all the same type). The format of the two methods is described below.

$$\begin{aligned} o : A \stackrel{\triangle}{=}\ & \mathit{obj}(X = A) \\ & [\mathbf{x_1} = .., \ \mathbf{x_2} = .., \ \cdots \ \mathbf{x_n} = .., \\ & \ \mathbf{set} = \varsigma(\mathit{self} : X)\lambda(m : M)\lambda(\mathit{val} : B)\ \mathit{self}.m := \mathit{val}, \\ & \ \mathbf{get} = \varsigma(\mathit{self} : X)\lambda(m : M)\ \ \mathit{self} \Leftarrow m\ ] \end{aligned}$$

It is again routine to verify that the definition of o typechecks if we define the types A an M as follows:

$$\begin{aligned} A &\equiv \mathit{Obj}(X)[\mathbf{x_1} : B,\ \mathbf{x_2} : B,\ \cdots\ \mathbf{x_n} : B, \\ &\qquad \mathbf{set} : M \to B \to X,\ \ \mathbf{get} : M \to B\ ] \\ M &\equiv \langle\!\langle \mathbf{x_i}\ ^{i \in 1..n} \rangle\!\rangle \end{aligned}$$

5 Properties of the Type System

5.1 Type Soundness

The soundness proof follows the standard pattern: we prove that types are preserved by evaluation, then we show that the evaluation of well-typed closed expressions does not get stuck, and finally we derive the soundness result as a corollary.

Lemma 5.1.1 (Substitution)

1. *If* $E, x{:}D, E' \vdash \jmath\{x\}$ *and* $E \vdash d{:}D$ *then* $E, E' \vdash \jmath\{d\}$
2. *If* $E, X <: D, E'\{X\} \vdash \jmath\{X\}$ *and* $E \vdash D' <: D$ *then* $E, E'\{D'\} \vdash \jmath\{D'\}$

Theorem 5.1.2 (Subject Reduction) *Let c be a closed term and v a value s.t. $c \rightsquigarrow v$. If $\vdash c : C$ then $\vdash v : C$.*

Proof. By induction on the derivation of $c \rightsquigarrow v$. The case **(Red Val)** is trivial, since $c \equiv v$, while the case **(Red App)** is standard. The remaining cases are worked out below.

(Red Update). Assume $\vdash a.m := (Y <: A)\varsigma(x : Y)b\{Y\} : C$. An inspection of the typing rules shows that this judgement must have come from a derivation of the following form:

$$\cfrac{\begin{array}{ll}\vdash a : A & \vdash A <: D \\ \vdash m : \langle\!\langle l_i\ ^{i\in 1..n'} \rangle\!\rangle & Y <: A, x : Y \vdash b\{Y\} : B\{Y\}\end{array}}{\begin{array}{c}\vdash a.m := (Y <: A)\varsigma(x : Y)b\{Y\} : A \\ \vdots\ (\textit{subsumption steps}) \\ \vdash a.m := (Y <: A)\varsigma(x : Y)b\{Y\} : C\end{array}}$$

where $D \equiv Obj(X)[l_i : B\{X\}^{i\in 1..n'}]$.

By inductive hypothesis, since $m \rightsquigarrow l_j$, we have $\vdash l_j : \langle\!\langle l_i\ ^{i\in 1..n'} \rangle\!\rangle$ and this must have come form the judgement $\vdash l_j : \langle\!\langle l_j \rangle\!\rangle$ by some subsumption steps that imply $j \in 1..n'$.

Let be $v \equiv obj(X = A')[l_i = \varsigma(x_i : X)b_i^{i\in 1..n}]$ where $A' \equiv Obj(X)[l_i : B'_i\{X\}\ ^{i\in 1..n}]$, from $\vdash a : A$ and $a \rightsquigarrow v$, by induction hypothesis we have $\vdash v : A$. Then A must be an object type, and the judgement $\vdash v : A$ must have come from a derivation of the form:

$$\cfrac{x_i : A' \vdash b_i\{A'\} : B'_i\{A'\} \quad i \in 1..n}{\begin{array}{c}\vdash v : A' \\ \vdots\ (\textit{subsumption steps}) \\ \vdash v : A\end{array}}$$

Now, from $\vdash A' <: A$ and $\vdash A <: D$ we have $n \geq n'$, $B_i\{X\} \equiv B'_i\{X\}$ and $B_i \equiv B$ for $i \in 1..n'$, so $B_j \equiv B$. Now from $Y <: A, x : Y \vdash b\{Y\} : B_j\{Y\}$, by Substitution Lemma (2), we have $x : A' \vdash b\{A'\} : B_j\{A'\}$ and collecting all leaves we may conclude with an application of (Val Object) followed by two subsumption steps.

(Red Send). Assume $\vdash a \Leftarrow m : C$. This judgement must have come from:

$$\cfrac{\begin{array}{ll}\vdash a : A & \vdash A <: D \\ \multicolumn{2}{l}{\vdash m : \langle\!\langle l_i \langle B_{i,1}\{A\}, \ldots, B_{i,k}\{A\}\rangle^{i\in 1..n} \rangle\!\rangle}\end{array}}{\begin{array}{c}\vdash a \Leftarrow m : C'\{A\} \\ \vdots\ (\textit{subsumption steps}) \\ \vdash a \Leftarrow m : C\end{array}}$$

where $D \equiv Obj(X)[l_i : B_{i,1}\{X\} \to \cdots \to B_{i,k}\{X\} \to C'\{X\}^{i\in 1..n}, \ldots]$.

From $\vdash m : \langle\!\langle l_i \langle B_{i,1}\{A\}, \ldots, B_{i,k}\{A\}\rangle^{i\in 1..n} \rangle\!\rangle$ and from $m \rightsquigarrow l_j\langle a_1 ... a_k\rangle$, by induction hypothesis, we have $\vdash l_j\langle a_1, \ldots, a_k\rangle : \langle\!\langle l_i \langle B_{i,1}\{A\}, \ldots, B_{i,k}\{A\}\rangle^{i\in 1..n} \rangle\!\rangle$ and the last judgement must have been defined as follows:

$$\cfrac{\begin{array}{c}\vdash a_i : B''_{j,i} \quad j \in 1..n \quad \forall i \in 1..k \\ \vdots\ (\textit{subsumption steps}) \\ \vdash a_i : B'_{j,i} \quad j \in 1..n \quad \forall i \in 1..k\end{array}}{\begin{array}{c}\vdash l_j\langle a_1, \ldots, a_k\rangle : \langle\!\langle l_j \langle B'_{j,1}, \ldots, B'_{j,k}\rangle \rangle\!\rangle \\ \vdots\ (\textit{subsumption steps}) \\ \vdash l_j\langle a_1, \ldots, a_k\rangle : \langle\!\langle l_i \langle B_{i,1}\{A\}, \ldots, B_{i,k}\{A\}\rangle\ ^{i\in 1..n} \rangle\!\rangle\end{array}}$$

From the lower subsumption steps it follows that $j \in 1..n$ and that $\Gamma \vdash B'_{j,i} <: B_{j,i}\{A\}$, while the upper steps of subsumption imply that $\Gamma \vdash B''_{j,i} <: B'_{j,i}$. Now, from $\vdash a : A$ and from $a \rightsquigarrow v' \equiv obj(X = A')[l_i = \varsigma(x_i : X)b_i\{X, x_i\}^{i\in 1..m}]$ we have $\vdash v' : A$. The last judgement must have come from a derivation of the form:

$$\cfrac{x_i : A' \vdash b_i\{A', x_i\} : \widehat{B}_i\{A'\} \quad i \in 1..m}{\begin{array}{c}\vdash obj(X = A')[l_i = \varsigma(x_i : X)b_i\{X, x_i\}^{i\in 1..m}] : A' \\ \vdots\ (\textit{subsumption steps}) \\ \vdash v' : A\end{array}}$$

From $\vdash A' <: A$ and $\vdash A <: D$, it follows that $A' \equiv Obj(X)[l_i : \widehat{B}_i\{X\}^{i\in 1..m}, \ldots]$ where $m \geq n$, and $\widehat{B}_i\{A'\} \equiv B_{i,1}\{A'\} \to \cdots \to B_{i,k}\{A'\} \to C'\{A'\}$ for $i \in 1..n$.

Now from the j-th judgement $x_j : A' \vdash b_j\{A', x_j\} : \widehat{B}_i\{A'\}$ and from $\vdash v' : A'$, by substitution lemma (1), we have $\vdash b_j\{A', v'\} : \widehat{B}_j\{A'\}$. Since $\widehat{B}_j\{X\}$ is covariant in X by hypothesis, from $\vdash A' <: A$ we also have $\vdash \widehat{B}_j\{A'\} <: \widehat{B}_j\{A\}$, and hence $\vdash b_j\{A', v'\} : \widehat{B}_j\{A\} \equiv B_{j,1}\{A\} \to \ldots \to B_{j,k}\{A\} \to C'\{A\}$ (remember that $j \in 1..n$). Now we can construct the following derivation:

$$\cfrac{\begin{array}{l}\vdash b_j\{A', v'\} : B_{j,1}\{A\} \to \ldots \to B_{j,k}\{A\} \to C'\{A\} \\ \vdash a_i : B_{j,i}\{A\}\ i \in 1..k\end{array}}{\vdash (..(b_j\{A', v'\})a_1..)a_k : C'\{A\}}$$

where $\vdash a_i : B_{j,i}\{A\}$ $i \in 1..k$ are derived by $\vdash a_i : B''_{j,i}$ and $\vdash B''_{j,i} <: B_{j,i}\{A\}$, $i \in 1..k$. Now, from $(..(b_j\{A', v'\})a_1..)a_k \rightsquigarrow v$ and the last judgement by induction hypothesis we have $\vdash v : C'\{A\}$, from which $\vdash v : C$ by a final subsumption step. □

Theorem 5.1.3 (Absence of Stuck states) *Let e be a closed expression such that the judgement $\vdash e : C$ is derivable for some type C. Then:*

1. *if $e = e_1(e_2)$ and $e_1 \rightsquigarrow val$, then $val = \lambda(x : A)b$ for some x and b;*
2. *if $e = a.m := (Y <: C)\varsigma(x : Y)b$ and $a \rightsquigarrow val$, $m \rightsquigarrow val'$, then $val = obj$ for some object expression obj and $val' = l$ for some label l;*
3. *if $e = a \Leftarrow m$ and $a \rightsquigarrow val$, $m \rightsquigarrow val'$, then $val = obj$ for some object expression obj and $val' = msg$ for some message expression msg.*

Proof. Standard, using Subject Reduction (and a few Generation Lemmas, that we omit for brevity). □

5.2 Minimum Types

We conclude proving a minimum-type property for $\mathbf{FOb}^{\mathbf{dyn}}_{<:}$. In that direction, we first define a new system **Min** obtained from the original one by removing (Val Subsumption) and replacing (Val Appl), (Val Object), (Val Update), and (Val Send) with the new rules in Table 3.

The proof of existence of minimum types is then standard, and follows directly from the next three propositions, whose proof is by easy inductions on the derivations. Below, we write $\Gamma \vdash a : A$ and $\Gamma \vdash_{\text{MIN}} a : A$ to say that the judgement $\Gamma \vdash a : A$ is derivable, respectively, in the original system for $\mathbf{FOb}^{\mathbf{dyn}}_{<:}$ and in the system **Min**.

Lemma 5.2.1 (Soundness) *If $\Gamma \vdash_{\text{MIN}} a : A$, then also $\Gamma \vdash a : A$.*

Lemma 5.2.2 (Completeness) *If $\Gamma \vdash a : A$, then then $\Gamma \vdash_{\text{MIN}} a : A'$ for some A' such that the judgement $\Gamma \vdash A' <: A$ is derivable (in either systems).*

Lemma 5.2.3 (Uniqueness) *If $\Gamma \vdash_{\text{MIN}} a : A$ and $\Gamma \vdash_{\text{MIN}} a : A'$, then $A \equiv A'$.*

Theorem 5.2.4 (Minimum Types) *If $\Gamma \vdash a : A$ then there exists B such that $\Gamma \vdash a : B$ and, for any A', if $\Gamma \vdash a : A'$ then $\Gamma \vdash B <: A'$.*

□

6 Message Types vs Object Types

We conclude our analysis of the new type system showing that message types can be encoded into object types, in ways similar to how variant types have been encoded into record types by Ghelli in [Ghe90].

As in [Ghe90], our encoding relies on the use of (bounded) universal types: therefore, strictly speaking, the encoding cannot be given directly in the theory of $\mathbf{FOb}^{\mathbf{dyn}}_{<:}$. However, the extension of the type system with bounded universal types seems a relatively mild one, given that a construct for bounded universal quantification is already implicitly present in the type system, in the subtyping constraints used in the typing rule for method override.

We illustrate the encoding in a simple case, where we assume that object types are first-order, and that message types are formed around method labels, without argument types.

Assume that we are given the type M of a message m, and let M be the message type $\langle\!\langle l_i \ ^{i \in 1..n} \rangle\!\rangle$. The encoding relies on the idea that the message m may legally be used in conjunction with any object that contains (at least) the l_i labels, provided that the labels have the same return type.

More specifically, assume that m occurs in the expression $a \Leftarrow m$. This expression has a type, say B, provided that we may prove that a has a type Y such that $Y <: [l_i : B \ ^{i \in 1..n}]$. Abstracting over B and Y, we have the encoding of M, namely:

$$[\![\langle\!\langle l_i \ ^{i \in 1..n} \rangle\!\rangle]\!] \triangleq \forall Z \ \forall (Y <: [l_i : Z \ ^{i \in 1..n}]) \ Y \rightarrow Z$$

Given this definition, we may prove a simple result relating the encoding to the subtype relation.

Proposition 6.1 (Preservation of Subtyping) *Let $M \equiv \langle\!\langle l_i \ ^{i \in 1..n} \rangle\!\rangle$ and M' be two message types s.t. $\Gamma \vdash M <: M'$ is derivable. Then $\Gamma \vdash [\![M]\!] <: [\![M']\!]$ is derivable as well.*

Proof. We use the following, standard, subtyping rule for bounded universal quantifiers:

(Val Min Appl)

$$\frac{\Gamma \vdash b : A \to B \quad \Gamma \vdash a : A' \quad \Gamma \vdash A' <: A}{\Gamma \vdash b(a) : B}$$

(Val Min Obj)

$$\frac{\Gamma, x_i : A \vdash b_i\{A\} : B'_i\{A\} \quad \Gamma \vdash B'_i\{A\} <: B_i\{A\} \quad i \in I}{\Gamma \vdash obj(X = A)[l_i = \varsigma(x_i : X)b_i\{X\}^{i \in I}] \;:\; A \equiv Obj(X)[l_i : B_i\{X\}^{i \in I}]}$$

(Val Min Update)

$$\frac{\begin{array}{l}\Gamma \vdash A <: Obj(X)[l_i : B\{X\}^{i \in 1..n}] \\ \Gamma \vdash a : A' \\ \Gamma \vdash A' <: A \quad \Gamma \vdash m : \langle\!\langle l_i \;^{i \in 1..k} \rangle\!\rangle \;\; n \geq k \quad \Gamma, Y <: A, x : Y \vdash b : B'\{Y\} \quad \Gamma \vdash B'\{Y\} <: B\{Y\}\end{array}}{\Gamma \vdash a.m := (Y <: A)\varsigma(x : Y)b : A}$$

(Val Min Send)

$$\frac{\begin{array}{l}\Gamma \vdash a : A \\ \Gamma \vdash A <: A' \equiv Obj(X)[l_i : B_{i,1}\{X\} \to \cdots \to B_{i,k}\{X\} \to C\{X\}^{i \in 1...n}, \ldots] \\ \Gamma \vdash m : \langle\!\langle l_j \langle B'_{j,1} \ldots B'_{j,k} \rangle \;^{j \in 1..n} \rangle\!\rangle \\ \Gamma \vdash B'_{j,i} <: B_{j,i}\{A\} \quad j \in 1..n \quad i \in 1..k\end{array}}{\Gamma \vdash a \Leftarrow m : C\{A\}}$$

Table 3: Typing Rules for Miminum Types

(Sub All)

$$\frac{\Gamma \vdash A' <: A \quad \Gamma, X <: A' \vdash B <: B'}{\Gamma \vdash \forall(X <: A)B <: \forall(X <: A')B'}$$

From the assumption that $\Gamma \vdash M <: M'$, it must be the case that $M' \equiv \langle\!\langle l_i^{i \in 1...n'} \rangle\!\rangle$ for some $n' \geq n$. Now the proof follows immediately by an application of the (Sub All) rule, noting that $\Gamma, Z \vdash [l_i : Z^{i \in 1..n'}] <: [l_i : Z^{i \in 1..n}]$ is derivable. □

In the general case of message types with argument types and SelfTypes the encoding can be defined following the same idea we just illustrated. Below we give the case for just one argument: the case of multiple arguments is obtained by an immediate generalization.

$$[\![\langle\!\langle l_i \langle B_i\{A\} \rangle^{i \in 1..n} \rangle\!\rangle]\!] \triangleq$$
$$\forall Z \; \forall(Y <: Obj(X)[l_i : B_i\{X\} \to Z \;^{i \in 1..n}]) \; Y \to Z$$

There are, however, two problems with this encoding. The first has to do with subtyping, and arises from the asymmetry between the invariant subtyping of object types and the covariant subtyping of message types. As a consequence, the encoding may be shown to preserve subtypes, only if we assume a weaker (i.e. invariant) subtyping rule for message types (or, dually, a covariant rule for object types, which is however incompatible with method updates).

A further, more serious, problem is that the encoding does not seem really useful as a basis for defining an adequate encoding of terms in the presence of SelfType. To see the problem, consider the following simple case, of a message type with one label and no argument types. We have:

$$[\![\langle\!\langle l \rangle\!\rangle]\!] \triangleq \forall Z \; \forall(Y <: Obj(X)[l : Z\,]) \; Y \to Z$$

Now consider the object type $A \equiv Obj(X)[l : X]$. It would seem reasonable for A to be a legal substitution for the type variable Y: this is not the case, however, as A is not a subtype of $Obj(X)[l : Z]$ for any legal choice of a type Z, because this type may not contain free occurrences of X. A legal substitute for Y is, instead, the object type $Obj(X)[l : A]$, which however does not reflect the fact that the return type of l is the object type itself:

in other words, the expressive power of SelfTypes is lost in the encoding.

7 Conclusions

We have presented an extension of the Abadi and Cardelli's typed calculus $\mathbf{FOb}_{<:}$ and a type system that give provision for dynamic messages. The system is a proper extension of the original system for $\mathbf{FOb}_{<:}$ from which it also inherits the properties of type soundness and existence of minimum types. The novelty over previous work [Nis98] is the first-order treatment of dynamic messages: besides being technically simpler than the original one, the new system is amenable to a smooth integration with existing models of objects and object types found in the literature. The existence of minimum types also is important as it is potentially useful for developing type-checking and type inference algorithms.

We have also studied a formal relationship between object types and message types discussing an encoding of the latter in terms of the former. While the encoding appears relatively satisfactory in the case of first-order object types, in its present form it is still inadequate for the case of SelfTypes. Work towards a more satisfactory solution is under way at the time of writing.

References

[AC95] M. Abadi and L. Cardelli. An Imperative Objects Calculus. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *Proceedings of TAPSOFT'95: Theory and Practice of Software Development*, volume 915 of *LNCS*, pages 471–485. Springer–Verlag, May 1995.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

[BCP97] K. Bruce, L. Cardelli, and B. Pierce. Comparing Object Encodings. In *Proc. of TACS'97*, volume 1281 of *Lecture Notes in Computer Science*, pages 415–438. Springer-Verlag, 1997.

[Bru94] K.B. Bruce. A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantcs. *Journal of Functional Programming*, 1(4):127–206, 1994.

[BSvG95] K.B. Bruce, A. Shuett, and R. van Gent. PolyTOIL: a Type-Safe Polymorphic Object-Oriented Language. In *Proceedings of ECOOP'95: European Conference on Object-Oriented Programming*, volume 952 of *LNCS*. Springer–Verlag, August 1995.

[CHC90] W. Cook, W. Hill, and P. Canning. Inheritance is not Subtyping. In *Proc. of ACM Symp. POPL*, pages 125–135. ACM Press, 1990.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[FHM94] 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.

[FM95] K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proc. of FCT*, volume 965 of *Lecture Notes in Computer Science*, pages 42–61. Springer-Verlag, 1995.

[Ghe90] G. Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism.* PhD thesis, Dipartimento di Informatica, Università di Pisa, March 1990.

[Ghe98] G. Ghelli. E-mail communication. Nov. 1998.

[Mic90] J. C. Michell. Toward a Typed Foundation for Method Specialization and Inheritance. In *Proc. of ACM Symp. POPL*, pages 109–124. ACM Press, 1990.

[Nis98] Susum Nishimura. Static typing for dynamc messages. In *Proc. of POPL'98*. ACM Press, 1998.

[PT94] B. Pierce and D. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, 1994.

A The Type System for $\mathbf{FOb}_{<:}$

Judgements

$\Gamma \vdash \diamond$	well-formed environment judgement
$\Gamma \vdash A$	type judjement
$\Gamma \vdash A <: B$	subtyping judgement
$\Gamma \vdash a : A$	value typing judgement

Environments

(Env $\emptyset$)

$$\frac{}{\vdash \diamond}$$

(Env x)

$$\frac{\Gamma \vdash A \quad x \notin dom(\Gamma)}{\Gamma, x : A \vdash \diamond}$$

(Env $X <:$)

$$\frac{\Gamma \vdash A \quad X \notin dom(\Gamma)}{\Gamma, X <: A \vdash \diamond}$$

Subtyping

(Sub Refl)

$$\frac{\Gamma \vdash A}{\Gamma \vdash A <: A}$$

(Sub Trans)

$$\frac{\Gamma \vdash A <: B \quad \Gamma \vdash B <: C}{\Gamma \vdash A <: C}$$

(Val Subs)

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A <: B}{\Gamma \vdash a : B}$$

Top

(Type Top)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{Top}}$$

(Sub Top)

$$\frac{\Gamma \vdash A}{\Gamma \vdash A <: \mathit{Top}}$$

Variables

(Type $X <:$)

$$\frac{\Gamma', X <: A, \Gamma'' \vdash \diamond}{\Gamma', X <: A, \Gamma'' \vdash X}$$

(Sub X)

$$\frac{\Gamma', X <: A, \Gamma'' \vdash \diamond}{\Gamma', X <: A, \Gamma'' \vdash X <: A}$$

(Val x)

$$\frac{\Gamma', x : A, \Gamma'' \vdash \diamond}{\Gamma', x : A, \Gamma'' \vdash x : A}$$

Arrows

(Type $\rightarrow$)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \rightarrow B}$$

(Sub $\rightarrow$)

$$\frac{\Gamma \vdash A' <: A \quad \Gamma \vdash B <: B'}{\Gamma \vdash A \rightarrow B <: A' \rightarrow B'}$$

(Val Fun)

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A)b : A \rightarrow B}$$

(Val Appl)

$$\frac{\Gamma \vdash b : A \rightarrow B \quad \Gamma \vdash a : A}{\Gamma \vdash b(a) : B}$$

Objects

(Type Obj)

$$\frac{\Gamma, X <: Top \vdash B_i\{X\} \quad B_i \textit{ covariant in } X \ \forall i \in 1..n}{\Gamma \vdash Obj(X)[l_i : B_i\{X\}^{\ i \in 1..n}]}$$

(Val Obj) $\quad A \equiv Obj(X)[l_i : B_i\{X\}^{\ i \in 1..n}]$

$$\frac{\Gamma, x_i : A \vdash b_i\{A\} : B_i\{A\} \qquad \forall i \in 1..n}{\Gamma \vdash obj(X = A)[l_i = \varsigma(x_i : X)b_i\{X\}^{\ i \in 1..n}] \ : A}$$

(Sub Obj)

$$\frac{\Gamma \vdash Obj(X)[l_i : B_i\{X\}^{\ i \in 1..n+m}]}{\Gamma \vdash Obj(X)[l_i : B_i\{X\}^{\ i \in 1..n+m}] <: Obj(X)[l_i : B_i\{X\}^{\ i \in 1..n}]}$$

(Val Update) $\quad A' \equiv Obj(X)[l_i : B_i\{X\}^{\ i \in 1...n}]$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A <: A' \quad \Gamma, Y <: A, x : Y \vdash \ b : B_j\{Y\}}{\Gamma \vdash a.l_j := (Y <: A)\varsigma(x : Y)b \ : A}$$

(Val Select) $\quad A' \equiv Obj(X)[l_i : B_i\{X\}^{\ i \in 1..n}]$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A <: A' \quad \Gamma \vdash j \in 1..n}{\Gamma \vdash a.l_j : B_j\{A\}}$$

B Typing Rules for $\mathbf{FOb}^{\mathbf{dyn}}_{<:}$

In the Type System of $\mathbf{FOb}^{\mathbf{dyn}}_{<:}$ there is a modified rule (Val Update) and a new rule (Val Send) that substitutes the old (Val Select):

(Val Update) $\quad A' \equiv Obj(X)[l_i : B\{X\}^{\ i \in 1...n}]$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A <: A' \quad \Gamma \vdash m : \langle\!\langle l_i^{\ i \in 1..n} \rangle\!\rangle \quad \Gamma, Y <: A, x : Y \vdash \ b : B\{Y\}}{\Gamma \vdash a.m := (Y <: A)\varsigma(x : Y)b \ : A}$$

(Val Send)

$$\frac{\begin{array}{c} A' \equiv Obj(X)[l_i : B_{i,1}\{X\} \to \cdots \to B_{i,k}\{X\} \to C\{X\}^{i \in 1...n}, \ldots] \\ \Gamma \vdash a : A \quad \Gamma \vdash A <: A' \quad \Gamma \vdash m : \langle\!\langle l_j \langle B_{j,1}\{A\} \ldots B_{j,k}\{A\}\rangle^{j \in 1..n} \rangle\!\rangle \end{array}}{\Gamma \vdash a \Leftarrow m : C\{A\}}$$

There are also new rules for message types:

Messages

(Type Message)

$$\frac{\Gamma \vdash B_{i,j} \quad j \in 1 \ldots k \quad \forall i \in I}{\Gamma \vdash \langle\!\langle l_i \langle B_{i,1} \ldots B_{i,k}\rangle^{i \in I} \rangle\!\rangle}$$

(Sub Message)

$$\frac{\Gamma \vdash B'_{i,j} <: B_{i,j} \qquad \forall i \in 1 \ldots n \qquad \Gamma \vdash \langle\!\langle l_i \langle B_{i,1} \ldots B_{i,k}\rangle^{i \in 1 \ldots n+m} \rangle\!\rangle}{\Gamma \vdash \langle\!\langle l_i \langle B'_{i,1} \ldots B'_{i,k}\rangle^{i \in 1 \ldots n} \rangle\!\rangle <: \langle\!\langle l_i \langle B_{i,1} \ldots B_{i,k}\rangle^{i \in 1 \ldots n+m} \rangle\!\rangle}$$

(Val Message)

$$\frac{\Gamma \vdash e_i : B_i \qquad i \in 1...k}{\Gamma \vdash l\langle e_1 ... e_k \rangle : \langle\!\langle l\langle B_1, \ldots, B_k \rangle \rangle\!\rangle}$$