Inferring types for functional methods (where method calls come for free)

Luigi Liquori, Arnaud Spiwack

Inferring types for functional methods
(where method calls come for free)

Luigi Liquori[1] Arnaud Spiwack[2]

[1] INRIA Sophia-Antipolis, France
Luigi.Liquori@inria.fr
[2] ENS Cachan, France
Arnaud.Spiwack@dptinfo.ens-cachan.fr

Abstract. This paper introduces a functional calculus, called OhML that features objects and message sending via functional application. A sound (message-not-found preventing) first-order type system featuring width subtyping is presented. The paper presents also a sound and complete type inference algorithm that calculates a principal constrained type or fails.

1 Introduction

There are a lot of similarities between function applications and method calls. For example, if you have a type `int`, you would use a function `plus` : `int` → `int` → `int` to perform addition; on the other hand, if you have a class `int` it will feature a method `plus` : `int` → `int` to perform addition. In the first case, you'll get a prefix addition (`plus 3 5`), in the second case, an infix addition (`3.plus 5`). Moreover in the first case `plus` is a first class expression, and in the second one it is not.

Though those two points of view are really different as far as operational semantics is concerned, they are quite similar for the user's understanding. This paper gives a way to erase most of the syntactic differences between method calls and function applications.

Therefore, the main contributions of this proposal are the following:

1. method names are first-class citizens, *i.e.* they can be passed as function arguments.
2. method call is reduced to a simple function application.

This paper uses a functional calculus with objects (no classes) and simple types (no polymorphism) *à la* Curry. However, the method-lookup algorithm could be easily improved with an object-based inheritance *à la* Abadi-Cardelli [AC96], or *à la* Fisher-Honsell-Mitchell [FHM94]. The type system and the type inference algorithm can be improved in a conservative way with polymorphic functions and methods, customizing the classical W algorithm of Damas-Milner [DM82].

Summarizing, this study may be found useful in exploring other ways to cohabit objects, functions, first-class methods and method-labels, and width subtyping using an alternative type theory to the row-polymorphism of Mitchell-Wand-Remy, currently used, *e.g.* in Caml.

Road map. Section 2 introduces the syntax and the static and dynamic semantics of OhML, Section 3 presents the main meta-theoretics results, while Section 4 show some interesting examples. Section 5 presents the type inference algorithm. Section 6 show the future work and conclude.

2 OhML Calculus

2.1 Syntax

The syntax of our calculus, that we call OhML, for its "zen" nature, is presented in Figure 1.

OF	::=	O is obj x –> $\overline{\text{M}}$ $\overline{\text{TA}}$ | F is fun x –> e	Objects and Functors
OA	::=	O | OA with {m@n} | OA minus {m}	Object Alterations
M	::=	m = e	Methods
id	::=	c | m | x	Constants, Methods−labels, Variables
e	::=	id | F | O | e e	Expressions

Fig. 1. Syntax

Expressions. Intuitively:

- c, 1, 2, 3, ... are mere constants.
- m, n, p, q, ... are method names. In the calculus, method names are first-class citizen, you can use them as any other expressions.
- x, y, this, that, ... are variables; we denote by "_" any fresh variable that not occurs in the rest of the expression.
- fun x –> e is the λ-function.
- obj x –> o is the constructor for object, it is an object with the method defined in the list o and with self referred as x.
- e e is the application, as usual.

We also let the following aliases for fix point operator, and let- and letrec-expressions.

$$\texttt{fix} \triangleq \texttt{obj this} \text{ –> } \texttt{data rec} = \texttt{fun f} \text{ –> } \texttt{fun x} \text{ –> } \texttt{f ((rec this) f) x}$$

$$\texttt{let x} = \texttt{e}_1 \texttt{ in e}_2 \triangleq \texttt{foo (obj \_} \text{ –> } \texttt{data foo} = \texttt{fun x} \text{ –> } \texttt{e}_2\texttt{) e}_1$$

$$\texttt{let rec f} = \texttt{fun x} \text{ –> } \texttt{e}_1 \texttt{ in e}_2 \triangleq \texttt{let f} = \texttt{(rec fix) (fun f} \text{ –> } \texttt{fun x} \text{ –> } \texttt{e}_1\texttt{) in e}_2$$

Values. Values are defined as follows:

$$\texttt{v} ::= \texttt{c} \mid \texttt{m} \mid \texttt{fun x} \text{ –> } \texttt{e} \mid \texttt{obj x} \text{ –> } \texttt{o} \mid \texttt{error}$$

The special constant error is a value introduced by the message-not-found exception.

$$(\texttt{fun x} \text{ –> } \texttt{e})\ \texttt{v} \mapsto \texttt{e}[\texttt{v}/\texttt{x}] \qquad (\beta_\texttt{v})$$

$$\texttt{m}\ (\texttt{obj x} \text{ –> } \texttt{o}) \mapsto \mathit{mbody}(\texttt{o}\,;\texttt{m})\ [(\texttt{obj x} \text{ –> } \texttt{o})/\texttt{x}] \qquad (\text{Call}_\texttt{v})$$

Fig. 2. One-step

2.2 Small Step Reduction Semantics

We start with the (call-by-value) one-step reduction (Figure 2). Intuitively:

- ($\beta_\texttt{v}$) is the usual β-reduction.
- ($\text{Call}_\texttt{v}$) is the method call. A usual way to write it would be (obj x –> o).m for the left part, the right part would remain the same (the lookup algorithm *mbody* is defined in Figure 3). Here we just swap the method and object and drop the dot. This little change is the key point of this paper.

We denote by $\rightarrow$ the contextual closure induced by these rules. Its reflexive and transitive closure is denoted by $\twoheadrightarrow$. The symmetric and transitive closure of $\twoheadrightarrow$ is denoted by $=$.

$$\frac{}{mbody(\epsilon\,;\mathtt{m}) = \mathtt{error}}\ (\text{Empty})$$

$$\frac{}{mbody(\mathtt{o\ data\ m} = \mathtt{e}\,;\mathtt{m}) = \mathtt{e}}\ (\text{Top})$$

$$\frac{\mathtt{m} \neq \mathtt{n}}{mbody(\mathtt{o\ data\ n} = \mathtt{e}\,;\mathtt{m}) = mbody(\mathtt{o}\,;\mathtt{m})}\ (\text{Next})$$

Fig. 3. Method Lookup Rules

Main rules

$$\frac{}{\mathtt{v} \Downarrow \mathtt{v}}\ (\text{Value}) \qquad \frac{\mathtt{e}_1 \Downarrow \mathtt{fun\ x} \dashrightarrow \mathtt{e}_3 \quad \mathtt{e}_2 \Downarrow \mathtt{v}_1 \quad \mathtt{e}_3[\mathtt{v}_1/\mathtt{x}] \Downarrow \mathtt{v}_2}{\mathtt{e}_1\ \mathtt{e}_2 \Downarrow \mathtt{v}_2}\ (\text{Beta}_\mathtt{v})$$

$$\frac{\mathtt{e}_1 \Downarrow \overbrace{\mathtt{m}}^{\text{label}} \quad \mathtt{e}_2 \Downarrow \overbrace{\mathtt{obj\ x} \dashrightarrow \mathtt{o}}^{\text{receiver}} \qquad mbody(\mathtt{o}\,;\mathtt{m})\overbrace{[\mathtt{obj\ x} \dashrightarrow \mathtt{o}/\mathtt{x}]}^{\text{self application}} \Downarrow \mathtt{v}}{\mathtt{e}_1\ \mathtt{e}_2 \Downarrow \mathtt{v}}\ (\text{Object}_\mathtt{v})$$

Error propagation rules

$$\frac{\mathtt{e}_1 \Downarrow \mathtt{error}}{\mathtt{e}_1\ \mathtt{e}_2 \Downarrow \mathtt{error}}\ (\text{Error}_1) \qquad \frac{\mathtt{e}_1 \Downarrow \mathtt{fun/obj\ x} \dashrightarrow \mathtt{e}_3 \quad \mathtt{e}_2 \Downarrow \mathtt{error}}{\mathtt{e}_1\ \mathtt{e}_2 \Downarrow \mathtt{error}}\ (\text{Error}_2)$$

Derivable rules

$$\frac{\mathtt{e}_1 \Downarrow \mathtt{v}_1 \quad \mathtt{e}_2[\mathtt{v}_1/\mathtt{x}] \Downarrow \mathtt{v}_2}{\mathtt{let\ x} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 \Downarrow \mathtt{v}_2}\ (\text{Let}_\mathtt{v}) \qquad \frac{\mathtt{e}_1 \Downarrow \mathtt{v}_1 \equiv \mathtt{fun\ f} \dashrightarrow \mathtt{e}_3 \quad \mathtt{e}_2 \Downarrow \mathtt{v}_2 \quad \mathtt{v}_1\ (\mathtt{fix}\ \mathtt{v}_1)\ \mathtt{v}_2 \Downarrow \mathtt{v}_3}{\mathtt{fix}\ \mathtt{e}_1\ \mathtt{e}_2 \Downarrow \mathtt{v}_2}\ (\text{Fix}_\mathtt{v})$$

Fig. 4. Big Step Operational Semantics (We let $\mathtt{v} \not\equiv \mathtt{error}$)

Lookup algorithm. The lookup algorithm is defined in Figure 3. Intuitively:

- (Empty) If we didn't find the method `m` after reading the whole method list, we're "shouting" an `error`.
- (Top) If we find the method `m` then we have reach a goal, and we just need to return the body of the method.
- (Next) If this is not the method `m`, we are going to check elsewhere.

Again, inheritance is out of the scope of this paper, but it would not be difficult to add an object-based inheritance.

2.3 Big Step Natural Operational Semantics

We continue with a (call-by-value) big-step operational semantics on closed terms in Figure 2.2. The big-step semantics is self-explaining and immediately suggests how to build an interpreter for our calculus.

2.4 Type System

Type syntax. The syntax of types and contexts are defined in Figure 5. In a nutshell:

$$\tau ::= \mathtt{b} \mid \alpha \mid \langle \overline{\mathtt{m}}{:}\overline{\tau} \rangle \mid \tau \to \tau \qquad \text{Types}$$
$$\Gamma ::= \epsilon \mid \Gamma, \mathtt{x}{:}\tau \mid \Gamma, \mathtt{c}{:}\mathtt{b} \qquad \text{Contexts}$$

Fig. 5. Type Syntax

$$\frac{\mathtt{c}{:}\mathtt{b} \in \Gamma}{\Gamma{\cdot}\Delta \vdash \mathtt{c} : \mathtt{b}}\ (\text{Const}) \qquad \frac{}{\Gamma{\cdot}\Delta \vdash \mathtt{m} : \langle \mathtt{m}{:}\tau \rangle \to \tau}\ (\text{Meth})$$

$$\frac{\mathtt{x}{:}\tau \in \Gamma}{\Gamma{\cdot}\Delta \vdash \mathtt{x} : \tau}\ (\text{Var}) \qquad \frac{\Gamma{\cdot}\Delta \vdash \mathtt{e} : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma{\cdot}\Delta \vdash \mathtt{e} : \tau_2}\ (\text{Sub})$$

$$\frac{\Gamma, \mathtt{x}{:}\tau_1{\cdot}\Delta \vdash \mathtt{e} : \tau_2}{\Gamma{\cdot}\Delta \vdash \mathtt{fun\ x\ {-}{>}\ e} : \tau_1 \to \tau_2}\ (\text{Fun}) \qquad \frac{\Gamma{\cdot}\Delta \vdash \mathtt{e}_1 : \tau_1 \to \tau_2 \quad \Gamma{\cdot}\Delta \vdash \mathtt{e}_2 : \tau_1}{\Gamma{\cdot}\Delta \vdash \mathtt{e}_1\,\mathtt{e}_2 : \tau_2}\ (\text{Appl})$$

$$\frac{\begin{array}{ll}\mathtt{o} \triangleq \mathtt{data\ m}_1 = \mathtt{e}_1\ \ldots\ \mathtt{data\ m}_n = \mathtt{e}_n & \forall i, j = 1 \ldots n.\ \mathtt{m}_i \neq \mathtt{m}_j \\ \tau \triangleq \langle \mathtt{m}_1{:}\tau_1 \ldots \mathtt{m}_n{:}\tau_n \rangle \qquad \Gamma, \mathtt{x}{:}\tau, \Delta \vdash \mathtt{e}_i : \tau_i & \forall i = 1 \ldots n\end{array}}{\Gamma{\cdot}\Delta \vdash \mathtt{obj\ x\ {-}{>}\ o} : \tau}\ (\text{Obj})$$

Admissible rules

$$\frac{\tau \equiv \tau_1 \to \tau_2}{\Gamma{\cdot}\Delta \vdash \mathtt{fix} : (\tau \to \tau) \to \tau}\ (\text{Fix}) \qquad \frac{\Gamma{\cdot}\Delta \vdash \mathtt{e}_1 : \tau_1 \quad \Gamma, \mathtt{x}{:}\tau_1{\cdot}\Delta \vdash \mathtt{e}_2 : \tau_2}{\Gamma{\cdot}\Delta \vdash \mathtt{let\ x} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 : \tau_2}\ (\text{Let})$$

Subtyping rules

$$\frac{}{\Delta \vdash \tau <: \tau}\ (\text{Refl}) \qquad \frac{\Delta \vdash \tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\Delta \vdash \tau_1 <: \tau_3}\ (\text{Trans}) \qquad \frac{\Delta \vdash \tau_3 <: \tau_1 \quad \Delta \vdash \tau_2 <: \tau_4}{\Delta \vdash \tau_1 \to \tau_2 <: \tau_3 \to \tau_4}\ (\text{Arr})$$

$$\frac{}{\Delta, C \vdash C}\ (\text{Taut}) \qquad \frac{h \geq k \quad \Delta \vdash \tau_i = \tau_i' \quad (i \leq k)}{\Delta \vdash \langle \mathtt{m}_1{:}\tau_1 \ldots \mathtt{m}_h{:}\tau_h \rangle <: \langle \mathtt{m}_1{:}\tau_1' \ldots \mathtt{m}_k{:}\tau_k' \rangle}\ (\text{Width})$$

$$\frac{\Delta \vdash \langle \mathtt{m}{:}\tau_1 \ldots \rangle <: \langle \mathtt{m}{:}\tau_2 \ldots \rangle}{\Delta \vdash \tau_1 <: \tau_2}\ (\text{IWidth}_1) \qquad \frac{\Delta \vdash \langle \mathtt{m}{:}\tau_1 \ldots \rangle <: \langle \mathtt{m}{:}\tau_2 \ldots \rangle}{\Delta \vdash \tau_2 <: \tau_1}\ (\text{IWidth}_2)$$

$$\frac{\Delta \vdash \tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}{\Delta \vdash \tau_1' <: \tau_1}\ (\text{IArr}_1) \qquad \frac{\Delta \vdash \tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}{\Delta \vdash \tau_2' <: \tau_2}\ (\text{IArr}_2)$$

Fig. 6. Typing and Subtyping Rules

- $\mathtt{b}$ are just the basic types, while $\alpha, \beta, \ldots$ are type variables.
- $\langle \overline{\mathtt{m}}{:}\overline{\tau} \rangle$ is a set of (differents) method names with types. More formally it is a row of the form $\langle \mathtt{m}_1{:}\tau_1\ \ldots \mathtt{m}_n{:}\tau_n \rangle$, for some $n \geq 0$.
- contexts records the types of constants and of the free-variables.

Typing rules. The typing and subtyping rules are introduced in Figure 6. A brief explanation follows for the (sub)typing rules: We introduce a second context Δ which contains constraints of the form

- $\mathtt{b}_1 <: \mathtt{b}_2$

- $\alpha <: \tau$ where τ is either an object, a variable or a constant type. Basically anything but arrows but in a more general context, anything where subtyping is not defined recursively.
- $\tau <: \alpha$ where τ is either an object, a variable or a constant type.

The reason for the restriction on the constraints in Δ is that the other kind of constraints can be broken down into smaller ones, or they are trivially unsolvable. However, allowing any constraint to appear in Δ does not seem to be much of a complication, and the metatheory seems to be mostly independent from this choice.

- $\Delta \vdash \tau_1 = \tau_2$ means $\Delta \vdash \tau_1 <: \tau_2$ and $\Delta \vdash \tau_2 <: \tau_2$
- (Const, Var) Nothing surprising here, these are usual rules for constants and variables.
- (Meth) This is the key rule: since methods are to be used as functions, they have to be typed as functions. They get here their natural type: if you give a method `m` an object with a method `m` of type τ then it returns an expression of type τ. Note that the type requires object with only `m` as a method, however subtyping allows you to use an object with any other methods.
- (Fun, Appl) These are usual rules.
- (Obj) The type of an object is a row, where each method is given the type of its body in a enriched context that takes into account the type of the object itself (represented by `x`).
- (Sub) The subsumption rule means that if `e` has a type τ, then `e` has any supertype of τ too. *e.g.*: $\vdash$ `obj x -> data m = fun x->x data n = fun x -> fun y -> x y` : $\langle$`m`:$\alpha \to \alpha\rangle$ because of (Sub) rule.
- For the admissible rules:
  - (Fix) Monomorphic type for the fix point combinator. This is usual, yet confusing. It basically means that you can only do a fix point of a term shaped as `fun f -> fun x -> e`.
  - (Let) A usual monomorphic typing rule for `let`.
- (Refl, Trans, Arr) These are the common rules when working with subtyping. (Refl) and (Trans) are the rule that enforce $<:$ to be a preorder. (Arr) is a structural rule which describes how the super-terms are behaving in regards of $<:$.
- (Taut) This rules allows the constraints in the context to be understood by the typing system. It's the only rule which "reads" Δ, basically assuming that what Δ says is true.
- (Width) This is the base case of the subtyping order. An object can "forget" some of its methods (*i.e.* the methods he remember must keep the same type though). To understand why depth subtyping is not possible see [AC96]. Note that since there is no inheritance, the present system could have accepted some *covariant* depth subtyping; since we want the system to be ready for upgrade with inheritance, we actually drop this feature.
- (IWidth_i) these two rules invert the property of the width subtyping. The ... in it mean "any other methods with any type" They don't need to be the same in both objects.
  Usually inverting the rules can, of course, be done by an induction on the derivation of the subtyping judgement. Unfortunately, with the addition of constraints, we can't always do it without these rules. For example

  $$\Delta = \langle \texttt{m}{:}\alpha\rangle <: \gamma, \gamma <: \langle \texttt{m}{:}\beta\rangle$$

  Then, we can derive $\Delta \vdash \langle \texttt{m}{:}\alpha\rangle <: \langle \texttt{m}{:}\beta\rangle$ but it's not true that $\Delta \vdash \alpha = \beta$. Which is unsound, even though Δ is a perfectly reasonable constraint context.
- (IArr_i) these two rules invert the property of the arrow subtyping. The idea is much the same as in (IWidth). An example of a Δ which would require it would be:

  $$\Delta = \langle \texttt{m}{:}\alpha_1 \to \alpha_2\rangle <: \gamma, \gamma <: \langle \texttt{m}{:}\beta_1 \to \beta_2\rangle$$

  We have $\Delta \vdash \alpha_1 \to \alpha_2 <: \beta_1 \to \beta_2$ thanks to the (IWidth_1) rule. But not $\Delta \vdash \beta_1 <: \alpha_1$ if we don't have the (IArr_1) rule.

3 Metatheory

This section introduces the metatheory of our calculus. Soundness and completeness of the big-step semantic *w.r.t.* the small-step one basically ensure that both semantics do essentially the same thing.

Theorem 1 (Soundness and Completeness of $\rightarrow$ *w.r.t.* $\Downarrow$).

1. *(Soundness) If* $\mathtt{e}_1 \Downarrow \mathtt{v}$ *and* $\mathtt{v} \neq \mathtt{error}$*, then* $\mathtt{e} \rightarrow \mathtt{v}$.
2. *(Completeness) If* $\mathtt{e}_1 \rightarrow \mathtt{v}$*, then the big step converges, i.e. there exists* $\mathtt{v}_0$ *such that* $\mathtt{e} \Downarrow \mathtt{v}_0$.

Proof. 1. (Soundness) Straightforward induction on the derivation of $\mathtt{e}_1 \Downarrow \mathtt{v}$.
2. (Completeness) This part can be proved via Melliès's standardization lemma [Mel96].

Definition 1.
A constraint context Δ is said flat if all the constraint it contains are of the form: $\mathtt{b}_1 <: \mathtt{b}_2$

Definition 2.
A constraint context Δ is said consistent if:

- $\Delta = \Delta_1, \Delta_2$ *and* Δ_1 *is flat*
- *there exists a substitution σ such that, for all* $C \in \Delta_2$ $\Delta_1 \vdash C\sigma$

Lemma 1 (Substitution in Subtyping).
Let Δ be a consistent constraint context, and σ a substitution such that for all $C \in \Delta$, $\Delta' \vdash C\sigma$ for some constraint context Δ'. If $\Delta \vdash \tau_1 <: \tau_2$ then $\Delta' \vdash \tau_1\sigma <: \tau_2\sigma$.

Proof. Straightforward induction on the derivation of $\Delta \vdash \tau_1 <: \tau_2$.

Lemma 2 (Subtyping Without Inversion).
We call a derivation without inversion of $\Delta \vdash \tau_1 <: \tau_2$ any such derivation which does not use any of the rules (IWidth_i) *or* (IArr_i)*. Here is a list of proprieties of such derivation. Let Δ_1 be a flat constraint context.*

1. *If $\Delta_1 \vdash \tau_1 \rightarrow \tau_2 <: \tau'$ is derivable without inversion, then $\tau' \equiv \tau_1' \rightarrow \tau_2'$ and $\Delta_1 \vdash \tau_1' <: \tau_1$ and $\Delta_1 \vdash \tau_2 <: \tau_2'$ are derivable without inversion.*
2. *If $\Delta_1 \vdash \langle \mathtt{m}_1{:}\tau_1 \dots \mathtt{m}_h{:}\tau_h \rangle <: \tau'$ is derivable without inversion, then $\tau' \equiv \langle \mathtt{m}_1{:}\tau_1' \dots \mathtt{m}_k{:}\tau_k' \rangle$ with $h \geq k$ and $\Delta_1 \vdash \tau_i <: \tau_i'$ and $\Delta_1 \vdash \tau_i' <: \tau_i$ are derivable without inversion for each $1 \leq i \leq h$.*
3. *If $\Delta_1 \vdash \tau <: \tau'$ is derivable, then it is also derivable without inversion.*

Proof.

1. Straightforward induction on the derivation without inversion of $\Delta_1 \vdash \tau_1 \rightarrow \tau_2 <: \tau'$.
2. Straightforward induction on the derivation without inversion of $\Delta_1 \vdash \langle \mathtt{m}_1{:}\tau_1 \dots \mathtt{m}_h{:}\tau_h \rangle <: \tau'$ (remember that Δ_1 is flat, thus does not contain any constraint involving such types).
3. By induction on the derivation of $\Delta_1 \vdash \tau <: \tau'$, all cases are straightforward, except the inversion ones which are resolved simply by applying the previous two results.

This is a small useful lemma for the proof of the following.

Lemma 3 (Subtyping).
Let Δ be a consistent context.

1. *If $\Delta \vdash \langle \mathtt{m}_1{:}\tau_1 \dots \mathtt{m}_h{:}\tau_h \rangle <: \langle \dots \rangle$ where $\langle \dots \rangle$ is any object type, then $\langle \dots \rangle \equiv \langle \mathtt{m}_1{:}\tau_1' \dots \mathtt{m}_k{:}\tau_k' \rangle$ with $h \geq k$ with $\Delta \vdash \tau_i = \tau_i'$ for all $1 \leq i \leq k$.*

Proof. 1. Since Δ is consistent, there exist a σ such that $\Delta = \Delta_1, \Delta_2$ with Δ_1 flat, and for all $C \in \Delta_2$ $\Delta_1 \vdash C\sigma$. It is obvious then, that for all $C \in \Delta$, $\Delta_1 \vdash C\sigma$.
Then we have, applying Lemma 1 (Substitution in Subtyping), that $\Delta_1 \vdash \langle \mathtt{m}_1{:}\tau_1 \dots \mathtt{m}_h{:}\tau_h \rangle\sigma <: \langle \dots \rangle\sigma$.
Hence $\langle \dots \rangle\sigma \equiv \langle \mathtt{m}_1{:}\tau_1'' \dots \mathtt{m}_k{:}\tau_k'' \rangle$ for some $k \leq h$ (by Lemma 2, Subtyping Without Inversion).
It obviously means that $\langle \dots \rangle \equiv \langle \mathtt{m}_1{:}\tau_1' \dots \mathtt{m}_k{:}\tau_k' \rangle$. Then we conclude because (IWidth_i) ensure us that $\Delta \vdash \tau_i = \tau_i'$ for all $1 \leq i \leq k$.

This lemma states that you can specialize (*i.e.* replace a type by a subtype) any type in the environment, and still be able to derive the same types.

Lemma 4 (Specialization).
If $\Gamma, \mathtt{x}{:}\tau_1 \cdot \Delta \vdash \mathtt{e} : \tau$ *and* $\Delta \vdash \tau_2 <: \tau_1$ *then* $\Gamma, \mathtt{x}{:}\tau_2 \cdot \Delta \vdash \mathtt{e} : \tau$.

Proof. By induction on the derivation of $\Gamma, \mathtt{x}{:}\tau_1 \cdot \Delta \vdash \mathtt{e} : \tau$:

- (Var) Either $\mathtt{e} \equiv \mathtt{y} \not\equiv \mathtt{x}$ and this is straightforward. Either $\mathtt{e} \equiv \mathtt{x}$ and $\tau \equiv \tau_1$, then:

$$\dfrac{\dfrac{}{\Gamma, \mathtt{x}{:}\tau_2 \cdot \Delta \vdash \mathtt{x} : \tau_2}\,(\text{Var}) \qquad \Delta \vdash \tau_2 <: \tau_1}{\Gamma, \mathtt{x}{:}\tau_2 \cdot \Delta \vdash \mathtt{x} : \tau_1}\,(\text{Sub})$$

- The other cases are straightforward.

This is the most important lemma in order to prove Subject Reduction theorem.

Lemma 5 (Generation).

1. *If* $\Gamma \cdot \Delta \vdash \mathtt{x} : \tau$ *then there exists* τ_0 *such that* $\mathtt{x}{:}\tau_0 \in \Gamma$ *and* $\Delta \vdash \tau_0 <: \tau$.
2. *If* $\Gamma \cdot \Delta \vdash \mathtt{m} : \tau$ *then there exists* τ_0 *such that* $\Gamma \cdot \Delta \vdash \mathtt{m} : \langle \mathtt{m}{:}\tau_0 \rangle \to \tau_0$ *and* $\Delta \vdash \langle \mathtt{m}{:}\tau_0 \rangle \to \tau_0 <: \tau$
3. *If* $\Gamma \cdot \Delta \vdash \mathtt{fun\ x} \mathrel{-\!\!>} \mathtt{e} : \tau$ *then there exists* τ_1, τ_2 *such that* $\Delta \vdash \tau_1 \to \tau_2 <: \tau$ *and* $\Gamma, \mathtt{x}{:}\tau_1 \cdot \Delta \vdash \mathtt{e} : \tau_2$. *Moreover, if* $\tau \equiv \tau_1' \to \tau_2'$ *then* $\Gamma, \mathtt{x}{:}\tau_1' \cdot \Delta \vdash \mathtt{e} : \tau_2'$.
4. *If* $\Gamma \cdot \Delta \vdash \mathtt{obj\ x} \mathrel{-\!\!>} \mathtt{o} : \tau$ *then there exists* τ_i *with* $i = 0 \ldots n$ *such that* $\tau_0 \equiv \langle \mathtt{m}_1{:}\tau_1 \ldots \mathtt{m}_n{:}\tau_n \rangle$ *and* $\Gamma, \mathtt{x}{:}\tau_0 \cdot \Delta \vdash \mathit{mbody}(\mathtt{o}\,; \mathtt{m}_i) : \tau_i$ *for all* $i = 1 \ldots n$ *and* $\Gamma \cdot \Delta \vdash \mathtt{obj\ x} \mathrel{-\!\!>} \mathtt{o} : \tau_0$ *and* $\Delta \vdash \tau_0 <: \tau$
5. *If* $\Gamma \cdot \Delta \vdash \mathtt{e}_1\ \mathtt{e}_2 : \tau$ *then there exists* τ_1 *such that* $\Gamma \cdot \Delta \vdash \mathtt{e}_1 : \tau_1 \to \tau$ *and* $\Gamma \cdot \Delta \vdash \mathtt{e}_2 : \tau_1$.

Proof. 1. Straightforward induction on the derivation of $\Gamma \cdot \Delta \vdash \mathtt{x} : \tau$ (there are only two cases: (Var) and (Sub)).
2. Straightforward induction on the derivation of $\Gamma \cdot \Delta \vdash \mathtt{m} : \tau$: (again, there are only two cases (Meth) and (Sub)).
3. Induction on the derivation of $\Gamma \cdot \Delta \vdash \mathtt{fun\ x} \mathrel{-\!\!>} \mathtt{e} : \tau$.
   (Fun) Obvious.
   (Sub) By induction hypothesis (let's note the previous type τ' with $\Delta \vdash \tau' <: \tau$) we have that there exists τ_1 and τ_2 such that $\Delta \vdash \tau_1 \to \tau_2 <: \tau'$ and $\Gamma, \mathtt{x}{:}\tau_1 \cdot \Delta \vdash \mathtt{e} : \tau_2$.
   Then, by transitivity, $\Delta \vdash \tau_1 \to \tau_2 <: \tau$ moreover, if $\tau \equiv \tau_1' \to \tau_2'$ we have that $\Delta \vdash \tau_1' <: \tau_1$, by Lemma 4 (Specialization), we have that $\Gamma, \mathtt{x}{:}\tau_1' \cdot \Delta \vdash \mathtt{e} : \tau_2$. We also have that $\Delta \vdash \tau_2 <: \tau_2'$, by the subsumption rule, we have the result.
4. Straightforward induction on the derivation of $\Gamma \cdot \Delta \vdash \mathtt{obj\ x} \mathrel{-\!\!>} \mathtt{o} : \tau$.
5. Straightforward induction on the derivation of $\Gamma \cdot \Delta \vdash \mathtt{e}_1\ \mathtt{e}_2 : \tau$.

This is the usual Substitution Lemma, it is crucial to prove Subject Reduction.

Lemma 6 (Substitution).
If $\Gamma, \mathtt{x}{:}\tau_1 \cdot \Delta \vdash \mathtt{e}_1 : \tau$ *and* $\Gamma \cdot \Delta \vdash \mathtt{e}_2{:}\tau_1$ *then* $\Gamma \cdot \Delta \vdash \mathtt{e}_1[\mathtt{e}_2/\mathtt{x}] : \tau$.

Proof. Straightforward induction on the derivation of $\Gamma, \mathtt{x}{:}\tau_1 \vdash \mathtt{e}_1 : \tau$.

Theorem 2 (Subject Reduction).
If Δ *is consistent,* $\Gamma \cdot \Delta \vdash \mathtt{e}_1 : \tau$, *and* $\mathtt{e}_1 \to \mathtt{e}_2$, *then* $\Gamma \cdot \Delta \vdash \mathtt{e}_2 : \tau$

Proof. ($\beta_{\mathtt{v}}$) Easy with Lemma 10 (Generation) and Lemma 6 (Substitution).

(Call_v) Because of Lemma 10, we have that $\Gamma{\cdot}\Delta \vdash \mathtt{m} : \tau_1 \to \tau$ and $\Gamma{\cdot}\Delta \vdash \mathtt{obj\ self\ -> o} : \tau_1$.
Also there exists τ_0 such that $\Gamma{\cdot}\Delta \vdash \mathtt{m} : \langle \mathtt{m}{:}\tau_0 \rangle \to \tau_0$ and $\Delta \vdash \langle \mathtt{m}{:}\tau_0 \rangle \to \tau_0 <: \tau_1 \to \tau$. And τ_i' with $i = 0 \dots n$ such that $\tau_0' \equiv \langle \mathtt{m}_1{:}\tau_1' \dots \mathtt{m}_n{:}\tau_n' \rangle$ and $\Gamma, \mathtt{self}{:}\tau_0' {\cdot} \Delta \vdash \mathit{mbody}(\mathtt{o}\,; \mathtt{m}_i) : \tau_i'$ for all $i = 1 \dots n$ and $\Gamma{\cdot}\Delta \vdash \mathtt{obj\ self\ -> o} : \tau_0'$ and $\Delta \vdash \tau_0' <: \tau_1$.
Then, we can derive :

$$\dfrac{\Delta \vdash \tau_0' <: \tau_1 \qquad \dfrac{\Delta \vdash \langle \mathtt{m}{:}\tau_0 \rangle \to \tau_0 <: \tau_1 \to \tau}{\Delta \vdash \tau_1 <: \langle \mathtt{m}{:}\tau_0 \rangle}\,(\text{IArr}_1)}{\Delta \vdash \tau_0' <: \langle \mathtt{m}{:}\tau_0 \rangle}\,(\text{Trans})$$

Now, we apply Lemma 3 (Subtyping), and we get that one of the $\mathtt{m}_i$ is actually $\mathtt{m}$ and that $\Delta \vdash \tau_i = \tau_0$. Hence the result (using Lemma 6 (Substitution), and the fact that $\tau_0 <: \tau$).

Lemma 7 (Progress).
Let Δ be a consistent constraint context.

1. *If $\Gamma{\cdot}\Delta \vdash \mathtt{e} : \tau$, then the evaluation of $\mathtt{e}$ cannot produce $\mathtt{error}$, i.e. $\mathtt{e} \not\to \mathsf{C}[\mathtt{error}]$ for every context $\mathsf{C}[\cdot]$.*
2. *If $\cdot\Delta \vdash \mathtt{e}_1 : \tau$ then, either $\mathtt{e}_1$ is a value, or there exists $\mathtt{e}_2$ such that $\mathtt{e}_1 \longrightarrow \mathtt{e}_2$.*

Proof. 1. Direct corollary for Theorem 2 (Subject Reduction), since $\mathtt{error}$ has no type.
2. Straightforward proof by induction on the typing derivation. We also need usual extension of Lemma 3 (Subtyping) to ensure that arrows types and object types are inhabited only by function and objects respectively (or looping expression. Remember that we are in an empty type context, meaning that e_1 is closed.

4 Principal Typing

We will now be interested in the property of principal typing. The goal of this section is two present an algorithm to infer a principal constrainted type.

Definition 3 (Principal Constrainted Type). *A principal constrainted type of $\mathtt{e}$ under $\Gamma \cdot \Delta$ is a pair (Δ', τ). Such that:*

- *Δ' is realisable by Δ meaning that there exists a substitution σ such that for all $\mathsf{C} \in \Delta'$, $\Delta \vdash \mathsf{C}\sigma$.*
- *$\Gamma \cdot \Delta, \Delta' \vdash \mathtt{e} : \tau$.*
- *For all τ' such that $\Gamma \cdot \Delta \vdash \mathtt{e} : \tau'$ there exists a substitution σ with $\tau' = \tau\sigma$ and for all $\mathsf{C} \in \Delta'$, $\Delta \vdash \Delta'\sigma$.*

*All substitution σ refered in this definition are supposed to leave Δ invariant (*i.e. *$\Delta\sigma = \Delta$)*

Note that by Lemma 1, the two last points are converse to each other. The first point means that objects with no type have no principal constrainted type either.

In the following we will be interested mainly in inference where Δ is empty. The extension to non-empty Δ is costless in term of decidability as long as the subtyping relation entailed by Δ is decidable. An usual example of non-empty context would be a flat context, with such a context, complexity either should not be an issue.

We can build an inference algorithm in the following fashion:

$$i(\Gamma; \Delta; \mathtt{x}) = (\tau <: \alpha; \alpha) \qquad \text{where } \mathtt{x}{:}\tau \in \Gamma$$
$$i(\Gamma; \Delta; \mathtt{e}_1\ \mathtt{e}_2) = (\Delta', \Delta'', \tau' <: \tau'' \mathtt{->} \alpha; \alpha) \qquad \text{where } i(\Gamma; \Delta; \mathtt{e}_1) = (\Delta'; \tau') \wedge i(\Gamma; \Delta; \mathtt{e}_2) = (\Delta''; \tau'')$$
$$i(\Gamma; \Delta; \mathtt{fun\ x\ -> e}) = (\Delta, \alpha \to \tau <: \beta; \beta) \qquad \text{where } i(\Gamma, \mathtt{x}{:}\alpha; \Delta; \mathtt{e}) = (\Delta'; \tau)$$
$$i(\Gamma; \Delta; \mathtt{m}) = (\langle \mathtt{m}{:}\gamma \rangle \to \gamma <: \alpha; \alpha)$$
$$i(\Gamma; \Delta; \mathtt{obj\ x\ -> data}\ \overline{\mathtt{m}}\ = \overline{\mathtt{e}}) = (\overline{\Delta'}, \overline{\alpha} = \overline{\tau}, \overline{\alpha} <: \beta; \beta) \qquad \text{where } i(\Gamma, \mathtt{x}{:}\langle \overline{\mathtt{m}}{:}\overline{\alpha} \rangle; \Delta; \overline{\mathtt{e}}) = (\overline{\Delta'}; \overline{\tau})$$

Where all the variables introduced are fresh.

Note that this algorithm only has variable types as output, which ensures that the constraint set it engenders is a constraint context.

It is not difficult to see that $i(\Gamma; \Delta; \mathtt{e})$ verifies both two last points of the definition of principal constrainted type. Now of course, since this algorithm always terminates with a solution, and there are indeed untyped terms, we need to discriminate which are principal constrainted types for `e` and which are actual untyped terms. This reduces, when Δ is empty to the satisfiability of constraints which is known to be decidable in polynomial time for this kind of subtyping relation [Pot98].

This is achieved roughly by saturating the constraint set with all possible derivable contraints and represent all the constraints in a graph.

We now know that we can find a principal typing for any typable expression in OhML or answering that no type is available when none is. Still, this type is presented as a cluster of constraints which are to be applied to a type variable. This is barely readable, but there are ways to make it clearer. One can hope benefiting from the constraint solving algorithm to get a smaller constraint set, also when a variable α appears in a type τ only in covariant positions, and if α has an greatest lower bound derivable from the constraint context, then, it is safe to substitute α with its greatest lower bound in τ. Respectively if α appears only in contravariant positions and have a least upper bound.

5 Examples

This section shows some type assignments for some interesting programs. The purpose of this section is to drive the reader to the next, more formal section, with a clear intuition of the inference algorithm.

5.1 Heating up

Those examples show some simple inferences in suitable constraint contexts.

$\emptyset \cdot \gamma <: \alpha \vdash \mathtt{fun\ x->(n\ x)(m\ x)} : \langle \mathtt{n}{:}\alpha \mathtt{->} \beta, \mathtt{m}{:}\gamma \rangle \mathtt{->} \beta$

$\emptyset \cdot \alpha <: \langle \mathtt{m}{:}\gamma \rangle \vdash \mathtt{fun\ x->fun\ y->x\ y; m\ y} : (\alpha \mathtt{->} \beta) \mathtt{->} \alpha \mathtt{->} \gamma$

$\emptyset \cdot \langle \mathtt{m}{:}\gamma \mathtt{->} \gamma \rangle <: \alpha \vdash \mathtt{fun\ x->fun\ y->x\ y; x\ (obj\ \_->data\ m = fun\ x->x)} : (\alpha \mathtt{->} \beta) \mathtt{->} \alpha \mathtt{->} \beta$

$\emptyset \cdot \delta <: \alpha, \delta <: \beta \vdash \mathtt{fun\ x->fun\ y->x\ (y\ x)(y\ x)} : (\alpha \mathtt{->} \beta \mathtt{->} \gamma) \mathtt{->} ((\alpha \mathtt{->} \beta \mathtt{->} \gamma) \mathtt{->} \delta) \mathtt{->} \gamma$

$\emptyset \cdot \alpha <: \langle \mathtt{n}{:}\beta \rangle \vdash \mathtt{fun\ x->\ (n\ (m\ x))} : \langle \mathtt{m}{:}\alpha \rangle \mathtt{->} \beta$

5.2 Find the Principal Type: a Manual Run

The following example shows how the algorithm is able to generate all constraints and finally find the principal type (it it exist).

Take the following function:

$$\mathtt{fun\ x->fun\ y->x\ (y\ x)}$$

1. First of all we have $\Gamma \triangleq \mathtt{x}{:}\alpha, \mathtt{y}{:}\beta$;
2. Then we try to infer some type for `x (yx)` and we fold all the constraints, *i.e.* $\beta <: \alpha \mathtt{->} \delta$ and $\alpha <: \delta \mathtt{->} \gamma$;
3. Then, we simplify the constraints, *i.e.* $\beta_1 \mathtt{->} \beta_2 <: \alpha \mathtt{->} \delta$ and $\alpha_1 \mathtt{->} \alpha_2 <: \delta \mathtt{->} \gamma$, *i.e.*

$$\begin{array}{l} \delta <: \alpha_1 \\ \alpha_2 <: \gamma \\ \alpha <: \beta_1 \\ \beta_2 <: \delta \end{array}$$

4. By transitivity, on could again simplify: we add $\alpha_1 \texttt{->} \alpha_2 <: \beta_1$;
5. By constraint rewriting, we simplifiy again: we substitute $\alpha_1 \texttt{->} \alpha_2 <: \beta_1$ for $\beta_{11} <: \alpha_1$, et $\alpha_2 <: \beta_{12}$;
6. We can build a Δ, by taking all the created constraints (of course trivial equality constraint are dropped, *e.g.*
$$\Delta \triangleq \beta_2 <: \delta, \delta <: \alpha_1, \alpha_2 <: \gamma, \beta_{11} <: \alpha_1, \alpha_2 <: \beta_{12}$$
7. Now, we get $\Gamma \triangleq \texttt{x}{:}\alpha_1 \texttt{->} \alpha_2, \texttt{y}{:}(\beta_{11} \texttt{->} \beta_{12}) \texttt{->} \beta_2$;
8. We fix the typing derivation as follows:
$$\emptyset \cdot \Delta \vdash \texttt{fun x->fun y->x (y x)} : (\alpha_1 \texttt{->} \alpha_2) \texttt{->} ((\beta_{11} \texttt{->} \beta_{12}) \texttt{->} \beta_2) \texttt{->} \gamma$$
9. We minimize Δ by replacing all type variables in covariant position with their lower bound, and al contravariant position with their upper bound, *i.e.*
$$\Delta_{min} \triangleq \emptyset$$
10. Finally, the inferred judgment will be:
$$\emptyset \cdot \emptyset \vdash \texttt{fun x->fun y->x (y x)} : (\alpha_1 \texttt{->} \alpha_2) \texttt{->} ((\alpha_1 \texttt{->} \alpha_2) \texttt{->} \alpha_1) \texttt{->} \alpha_1$$

6 Adding Inheritance

We will now show how inheritance can be featured in OhML. It is well known that inheritance and subtyping do not mix well (see for instance [Liq97]). In a first attempt we will implement inheritance in a "dynamic update" style, meaning that we can override methods in objects to create new ones, but not to add new method names.

This method of inheritance can appear weaker but it has been shown [AC94] that it allows encoding of usual class-based inheritance. Being able to add new method names to an object can still be an improvement to our system, but in the presence of an expression based and functional calculus, our first attempts have been disrupted, showing an unexpected number of difficulties.

This extension however, is rather straightforward, and can easily be understood from the inheritance-free calculus.

Let's add an inheritance operator to the syntax of objects:

$$\texttt{o} ::= \epsilon \mid \texttt{compose e} \mid \texttt{o data m} = \texttt{e}$$

We have now two types of objects. The basic ones allowing defining new signatures, and the composed ones inheriting from another object. We will enforce the new methods of composed objects to have the same name as some method of the inherited object with the type system.

But first, we need to update the semantics of objects. We will first give the new definition of values:

$$\texttt{v} ::= \nu \mid \texttt{fun x -> e} \mid \texttt{obj x -> } \omega \mid \texttt{error}$$
$$\omega ::= \epsilon \mid \texttt{compose obj x -> } \omega \mid \omega \texttt{ data m} = \texttt{e}$$

The idea behind this new value definition, is that we'll need to reduce the composed object to ensure that the lookup algorithm makes any sense at all.

Let us give the new rule of the lookup for composition (for simplicity purpose, we suppose the objects from which it is inherited to have been α-renamed to have the same self name):

$$\frac{\mathit{mbody}(\omega\,;\texttt{m}) = e}{\mathit{mbody}(\texttt{compose obj x -> } \omega\,;\texttt{m}) = e}\ (\text{Inherit})$$

And the updated rule for (Call_v)-reduction:

$$\texttt{m (obj x -> } \omega) \mapsto \mathit{mbody}(\omega\,;\texttt{m})\ [(\texttt{obj x -> } \omega)/\texttt{x}] \qquad (\text{Call}_\text{v})$$

This is a simple tweaking dealing with the need of reduction inside composition.

The type syntax stays the same and all prior typing and subtyping rules still holds. We only need an extra rule for composed object to type. This rule will allow us to ensure that the created object has no extra method names:

$$\frac{\begin{array}{ll}\mathtt{o} \triangleq \mathtt{compose}\ \mathtt{e}_0\ \mathtt{data}\ \mathtt{m}_1 = \mathtt{e}_1\ \ldots\ \mathtt{data}\ \mathtt{m}_n = \mathtt{e}_n & \forall i,j = 1\ldots n.\ \mathtt{m}_i \neq \mathtt{m}_j \\ \Gamma \cdot \Delta \vdash \mathtt{e}_0 : \tau \qquad \tau \triangleq \langle \mathtt{m}_1{:}\tau_1 \ldots \mathtt{m}_n{:}\tau_n \rangle \qquad \Gamma, \mathtt{x}{:}\tau, \Delta \vdash \mathtt{e}_i : \tau_i & \forall i = 1\ldots n\end{array}}{\Gamma \cdot \Delta \vdash \mathtt{obj\ x\ -\!\!>\ o} : \tau}\ (\text{Obj})$$

Note by the way that by this definition, we consider the self name unbound in the compose expression.

The metatheory of this new calculus is a straightforward extension of the earlier one, the intuition behind this extension behind rather clear. We won't state it here but the same results hold.

7 Adding polymorphism

The system has been designed until now without polymorhpic let. But adding it is not an difficult issue. We just have to take care of the fact that we have constraints. The idea is simple, it comes clear from the syntax of type schemes:

$$\mathtt{s} ::= \forall \overline{\alpha}.\Delta.\tau$$

A type scheme has to contain all the constraint needed over its variables to ensure safe instanciation. For instance consider:

$$\mathtt{e} = \mathtt{fun\ x\ -\!\!>\ obj\ \_\ -\!\!>\ compose\ x\ data\ m} = \mathtt{e}_1\ \mathtt{data\ n} = \mathtt{e}_2$$

If $\cdot \vdash \mathtt{e}_1 : \tau_1$ and $\cdot \vdash \mathtt{e}_2 : \tau_2$, then the type scheme

$$\forall \alpha.\alpha <: \langle \mathtt{m}{:}\tau_1, \mathtt{n}{:}\tau_2 \rangle.\alpha \to \alpha$$

Is a scheme which entails all the types of $\mathtt{e}$ provided τ_1 and τ_2 being unique types.

To carry the intuition into the type system, let us define a generation and instanciation operator:

$$\mathsf{Gen}(\Gamma; \Delta, \Delta'; \tau) = (\Delta; \forall \overline{\alpha}.\Delta'.\tau) \qquad \text{if } \overline{\alpha} \cap \mathrm{FTV}(\Gamma) = \emptyset \wedge \overline{\alpha} \cap \mathrm{FTV}(\Delta) = \emptyset \wedge \forall \mathtt{C} \in \Delta'.\ \overline{\alpha} \cap \mathrm{FTV}(\mathtt{C}) \neq \emptyset$$

$$\mathsf{Inst}(\Delta; \forall \overline{\alpha}.\Delta'.\tau) = \{\tau\sigma \mid \forall \beta \notin \overline{\alpha}.\ \beta\sigma = \beta \wedge \forall \mathtt{C} \in \Delta'.\ \Delta \vdash \mathtt{C}\sigma\}$$

These two operator are a constrainted extension of those in more usual ML-like type systems. They give the needed typing rules straightforwardly. We need to change the variable rule:

$$\frac{\tau \in \mathsf{Inst}(\Delta; \mathtt{s})}{\Gamma, x{:}\mathtt{s} \cdot \Delta \vdash \mathtt{x}{:}\tau}\ (\text{Var})$$

$$\frac{\Gamma \cdot \Delta' \vdash \mathtt{e}_1 : \tau_1 \quad (\Delta; \mathtt{s}) \triangleq \mathsf{Gen}(\Gamma; \Delta'; \tau_1) \quad \Gamma, \mathtt{x}{:}\mathtt{s} \cdot \Delta \vdash \mathtt{e}_2 : \tau_2}{\Gamma \cdot \Delta \vdash \mathtt{let\ x} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 : \tau_2}\ (\text{Let})$$

The metatheory extends naturally to polymorphism.

8 Type inference algorithm

Type inference in presence of subtyping is a delicate problem: in the last ten years a lot of researchers have studied this topic on various forms and techniques (for a good survey we refers to the Pottier's Ph.D. thesis [Pot98], or its Habilitation thesis [Pot04], or the course note of Leroy-Pottier [LP]). The Mitchell's paper [Mit91] on *Type Inference With Simple Subtypes* has shown that adding subtyping with subsumption rule in a simple type system complicates the inference algorithm. This means that solving the equation system is a *non trivial problem* especially in presence of function-types (that behave contravariantly in the function argument) and in presence of non atomic types, like *e.g.* object-types. Mitchell's paper designed a type inference algorithm, called $\mathcal{G}$. In our OhML calculus, the presence of objects and object-types complicate the inference. The algorithm $\mathcal{A}lg$ presented in this paper is inspired and an extension of Mitchell's one.

Prerequisites. The following definition sets up the semantic structure of *constraint* set used by an inference algorithm.

Definition 4 (The constraint-set $\mathbb{C}$).

1. *A constraint-set of subtyping equations is inductively defined as follows:*

$$\mathbb{C} ::= \emptyset \mid \mathbb{C} \cup \{\tau <: \tau\}$$

2. *A subtyping statement $\tau_1 <: \tau_2$ is provable in $\mathbb{C}$, if $\tau_1 <: \tau_2$ can be proved using the subtyping in $\mathbb{C}$ plus the rules of Figure 6. ($\tau_1 <: \tau_2$ could be also be written as $\mathbb{C} \vdash \tau_1 <: \tau_2$.)*
3. *We write $\mathbb{C}_1 \vdash \mathbb{C}_2$ if $\mathbb{C}_1 \vdash \tau_1 <: \tau_2$ for all $\tau_1 <: \tau_2 \in \mathbb{C}_2$.*
4. *(Transitivity) If $\mathbb{C}_1 \vdash \mathbb{C}_2$, and $\mathbb{C}_2 \vdash \mathbb{C}_3$, then $\mathbb{C}_1 \vdash \mathbb{C}_3$.*

The algorithm. In what follows we present a type inference algorithm, called $\mathcal{A}lg$, conceived as an extension of the Mitchell's $\mathcal{G}$ algorithm; that is, given an untyped term, $\mathcal{A}lg$ produces a triple $(\mathbb{C}, \Gamma, \tau)$, such that $\Gamma \vdash \mathtt{e} : \tau$ (in a given $\mathbb{C}$), or fails. The set $\mathbb{C}$, usually called *constraint-set*, contains the subtyping assertions that must be satisfied in order the type judgment $\Gamma \vdash \mathtt{e} : \tau$ being derivable. It is worthy noticing that the presentation of the type system of Figure 6 omits the set $\mathbb{C}$ in the premises of the judgment, *i.e.* $\Gamma \vdash \mathtt{e} : \tau$ should be intended as $\mathbb{C} \cdot \Gamma \vdash \mathtt{e} : \tau$.

Definition 5 (The Algorithm $\mathcal{A}lg$).
The algorithm $\mathcal{A}lg$ is given via an inference system presented in Figure 10; it uses the classical algorithm mgu*, that is the unification algorithm between first-order terms of [Rob65] (hence omitted).*

A brief explication of the algorithm follows:

- ($\mathtt{r}$AConst), ($\mathtt{r}$AVar) Those two rules are axioms that gives the most general context satisfying the subtyping constraints for constants and variables.
- ($\mathtt{r}$AMeth) This rule is an axiom for method inference; it just set a suitable constraint-set.
- ($\mathtt{r}$AAbs$_1$), ($\mathtt{r}$AAbs$_2$) Those two rules deal with abstraction; the main difference is that the in former rule the variable $\mathtt{x}$ occurs in $\mathtt{e}$, while in the latter not. Suitable constraint-sets and type-contexts are built.
- ($\mathtt{r}$AAppl) This rule (directly inherited from $\mathcal{G}$), deals with function/method application/invocation: two constraint-sets $\mathbb{C}_1$, and $\mathbb{C}_2$ are built together with two contexts Γ_1, and Γ_2, and two inferred types τ_1, and τ_2 for $\mathtt{e}_1$ and $\mathtt{e}_2$, respectively. An hygiene condition on the free type-variable of τ_1 and τ_2 is required. If there exists a substitution θ, most general unifier between the common variables in contexts Γ_1 and Γ_2, and unifying the domain of the arrow-type of $\mathtt{e}_1$ with the inferred type of $\mathtt{e}_2$, then a new merged constraint-set, a new merged type context, and an inferred type is given.

All α type-variables are considered fresh.They can be indexed.

$$\frac{}{\mathcal{Alg}(\mathtt{c}) = (\{\mathtt{b}_1 <: \mathtt{b}_2\}\,;\mathtt{c}{:}\mathtt{b}_1\,;\mathtt{b}_2)}\,(\mathtt{rAConst}) \qquad \frac{}{\mathcal{Alg}(\mathtt{x}) = (\{\alpha_1 <: \alpha_2\}\,;\mathtt{x}{:}\alpha_1\,;\alpha_2)}\,(\mathtt{rAVar})$$

$$\frac{}{\mathcal{Alg}(\mathtt{m}) = (\{\langle \mathtt{m}{:}\alpha_1\rangle \to \alpha_1 <: \alpha_2\}\,;\emptyset\,;\alpha_2)}\,(\mathtt{rAMeth})$$

$$\frac{\mathcal{Alg}(\mathtt{e}) = (\mathbb{C}\,;\Gamma\,;\tau_1) \quad \exists \tau_2.\ \mathtt{x}{:}\tau_2 \in \Gamma}{\mathcal{Alg}(\mathtt{fun\ x\ {-}{>}\ e}) = (\mathbb{C} \cup \{\tau_2 \to \tau_1 <: \alpha_1 \to \alpha_2\}\,;\Gamma \setminus \{\mathtt{x}\}\,;\alpha_1 \to \alpha_2)}\,(\mathtt{rAAbs_1})$$

$$\frac{\mathcal{Alg}(\mathtt{e}) = (\mathbb{C}\,;\Gamma\,;\tau_1) \quad \nexists \tau_2.\ \mathtt{x}{:}\tau_2 \in \Gamma}{\mathcal{Alg}(\mathtt{fun\ x\ {-}{>}\ e}) = (\mathbb{C} \cup \{\alpha_3 \to \tau_1 <: \alpha_1 \to \alpha_2\}\,;\Gamma\,;\alpha_1 \to \alpha_2)}\,(\mathtt{rAAbs_2})$$

$$\frac{\begin{array}{l}\mathcal{Alg}(\mathtt{e}_1) = (\mathbb{C}_1\,;\Gamma_1\,;\tau_1) \qquad \mathcal{Alg}(\mathtt{e}_2) = (\mathbb{C}_2\,;\Gamma_2\,;\tau_2) \qquad \mathsf{Ftv}(\tau_1) \cap \mathsf{Ftv}(\tau_2) = \emptyset \\ \exists\theta = \mathsf{mgu}(\{\ \Gamma_1(\mathtt{x}) \stackrel{?}{=} \Gamma_2(\mathtt{x}) \mid \forall \mathtt{x}.\ \mathtt{x} \in \mathsf{Dom}(\Gamma_1) \cap \mathsf{Dom}(\Gamma_2)\} \cup \{\tau_1 \stackrel{?}{=} \tau_2 \to \alpha_1\})\end{array}}{\mathcal{Alg}(\mathtt{e}_1\ \mathtt{e}_2) = (\mathbb{C}_1\theta \cup \mathbb{C}_2\theta \cup \{\alpha_1\theta <: \alpha_2\}\,;\Gamma_1\theta, \Gamma_2\theta\,;\alpha_2)}\,(\mathtt{rAAppl})$$

$$\frac{\begin{array}{l}\mathtt{o} \triangleq \mathtt{data\ m}_1 = \mathtt{e}_1\ \ldots\ \mathtt{data\ m}_n = \mathtt{e}_n \qquad \bigcap_{i=1\ldots n} \mathsf{Ftv}(\tau_i' \to \tau_i'') = \emptyset \\ \mathcal{Alg}(\mathtt{fun\ x\ {-}{>}\ e}_i) = (\mathbb{C}_i\,;\Gamma_i\,;\tau_i' \to \tau_i'') \qquad \forall i = 1\ldots n \\ \exists\theta = \mathsf{mgu}\left(\begin{array}{l}\{\Gamma_i(\mathtt{y}) \stackrel{?}{=} \Gamma_j(\mathtt{y}) \mid \forall \mathtt{y}.\ \mathtt{y} \in \mathsf{Dom}(\Gamma_i) \bigcap_{i,j=1\ldots n} \mathsf{Dom}(\Gamma_j)\} \\ \bigcup_{i=1\ldots n}\{\langle \mathtt{m}_1{:}\tau_1'' \ldots \mathtt{m}_n{:}\tau_n''\rangle \stackrel{?}{=} \tau_i'\}\end{array}\right)\end{array}}{\mathcal{Alg}(\mathtt{obj\ x\ {-}{>}\ o}) = (\bigcup_{i=1\ldots n} \mathbb{C}_i\theta \bigcup_{i=1\ldots n}\{\tau_i'\theta <: \alpha_1\}\,;\bigcup_{i=1\ldots n} \Gamma_i\theta \setminus \{\mathtt{x}\}\,;\alpha_1)}\,(\mathtt{rAObj})$$

Fig. 7. The Type Inference Algorithm

– (rAObj) This rule deals with objects. Given an object with n methods, first we calculate n-triples of *(constraint-set;type-context;inferred-type)*. An hygiene condition on the free type-variables of all the inferred arrow-types is required. If there exists a substitution θ, most general unifier between all the common variables in contexts Γ_i, with $i = 1\ldots n$, and unifying all the arrow-types of $\mathtt{fun\ x}{-}{>}\ \mathtt{e}_i$, with $i = 1\ldots n$, where $\mathtt{x}$ is the type of the full object (*i.e.* the type of `this`), then a new merged constraint-set, a new merged type context, and an inferred type is given.

Example 1 (A run of $\mathcal{Alg}$).
Consider the term $\mathtt{e} \triangleq \mathtt{fun\ y}{-}{>}\ \mathtt{obj\ x}\ {-}{>}\ \mathtt{data\ m} = \mathtt{y}$. Let $\Gamma \triangleq \mathtt{y}{:}\alpha_1$, and let:

$$\begin{array}{ll}\mathbb{C}_1 \triangleq \{\alpha_1 <: \alpha_2\} & \mathbb{C}_2 \triangleq \{\alpha_3 \to \alpha_2 <: \alpha_4 \to \alpha_5\} \\ \mathbb{C}_3 \triangleq \{\langle \mathtt{m}{:}\alpha_5\rangle <: \alpha_6\} & \mathbb{C}_4 \triangleq \{\alpha_1 \to \alpha_6 <: \alpha_7 \to \alpha_8\}\end{array}$$

$$\frac{\dfrac{\dfrac{}{\mathcal{Alg}(\mathtt{y}) = (\mathbb{C}_1\,;\Gamma\,;\alpha_2)}}{\begin{array}{l}\mathcal{Alg}(\mathtt{fun\ x\ {-}{>}y}) = (\mathbb{C}_1 \cup \mathbb{C}_2\,;\Gamma\,;\alpha_4 \to \alpha_5) \\ \exists\theta = [\langle \mathtt{m}{:}\alpha_5\rangle/\alpha_4].\ \mathsf{mgu}(\{\langle \mathtt{m}{:}\alpha_5\rangle \stackrel{?}{=} \alpha_4\})\end{array}}}{\mathcal{Alg}(\mathtt{obj\ x}\ {-}{>}\ \mathtt{data\ m} = \mathtt{y}) = (\mathbb{C}_1\theta \cup \mathbb{C}_2\theta \cup \mathbb{C}_3\theta\,;\Gamma\theta\,;\alpha_7 \to \alpha_8)}$$

$$\mathcal{Alg}(\mathtt{e}) = (\mathbb{C}_1\theta \cup \mathbb{C}_2\theta \cup \mathbb{C}_3\theta \cup \mathbb{C}_4\,;\emptyset\,;\alpha_7 \to \alpha_8)$$

given the expression e, its principal typing is found by finding a *minnum* set $\mathbb{C}_5$ such that $\mathbb{C}_1\theta \cup \mathbb{C}_2\theta \cup \mathbb{C}_3\theta \cup \mathbb{C}_4 \vdash \mathbb{C}_5$. This is usually done by using a term rewriting system that "propagates" from left-to-right all the minimum-type constraints until a single (singleton) constraint-set of the form $\{\tau_1 <: \tau_2\}$ is obtained. Constraints between arrow-types are decomposed as usual, *i.e.* $\{\tau_1 \to \tau_2 <: \tau_3 \to \tau_4\} \rightsquigarrow \{\tau_3 <: \tau_1\} \cup \{\tau_3 <: \tau_4\}$. Constraints between object-types are decomposed pointwise, *i.e.* $\{\langle \mathtt{m}_1{:}\tau_1 \dots \mathtt{m}_n{:}\tau_n\rangle <: \langle \mathtt{m}_1{:}\tau_1 \dots \mathtt{m}_n{:}\tau'_n\rangle\} \rightsquigarrow \bigcup_{i=1\dots n}\{\tau_i <: \tau'_i\}$. If this rewriting terminates, then, the type τ_1 is the principal type of e. Pragmatically.

$$
\begin{aligned}
\mathbb{C}_1\theta \cup \mathbb{C}_2\theta \cup \mathbb{C}_3\theta \cup \mathbb{C}_4 &\rightsquigarrow (\mathbb{C}_2\theta \cup \mathbb{C}_3\theta \cup \mathbb{C}_4)[\alpha_1/\alpha_2] \\
&\equiv \{\alpha_3 \to \alpha_1 <: \langle \mathtt{m}{:}\alpha_5\rangle \to \alpha_5\} \cup \mathbb{C}_3 \cup \mathbb{C}_4 \\
&\equiv \{\langle \mathtt{m}{:}\alpha_5\rangle <: \alpha_3\,, \alpha_1 <: \alpha_5\} \cup \mathbb{C}_3 \cup \mathbb{C}_4 \\
&\rightsquigarrow (\mathbb{C}_3 \cup \mathbb{C}_4)[\langle \mathtt{m}{:}\alpha_5\rangle/\alpha_3, \alpha_1/\alpha_5] \equiv \{\langle \mathtt{m}_1{:}\alpha_1\rangle <: \alpha_6\} \cup \mathbb{C}_4 \\
&\rightsquigarrow \mathbb{C}_4[\langle \mathtt{m}_1{:}\alpha_1\rangle/\alpha_6] \equiv \{\underbrace{\alpha_1 \to \langle \mathtt{m}_1{:}\alpha_1\rangle}_{\text{principal type}} <: \alpha_7 \to \alpha_8\} \\
&\equiv \mathbb{C}_5
\end{aligned}
$$

Then the principal type of e exists and it is $\alpha_1 \to \langle \mathtt{m}_1{:}\alpha_1\rangle$ for some α_1.

Lemma 8 (Soundness and Completeness of mgu).

(Soundness) *If $\mathsf{mgu}(\mathbb{C}) = \theta$, then θ is a solution for $\mathbb{C}$.*
(Completeness) *If ϑ is a solution for $\mathbb{C}$, then $\mathsf{mgu}(\mathbb{C}) = \theta$ such that $\theta \leq \vartheta$ (where $\theta \leq \vartheta \iff \exists\phi.\ \vartheta = \phi \circ \theta$).*

Theorem 3 (Soundness and Completeness of $\mathcal{A}lg$).

1. *(Soundness) If $\mathcal{A}lg(\mathsf{e}) = (\mathbb{C}\,;\Gamma\,;\tau)$, and $\mathbb{C} \rightsquigarrow\!\!\!\!\to \{\tau_1 <: \tau_2\}$, then $\Gamma \vdash \mathsf{e} : \tau_1$, and $\tau \preceq \tau_1$.*
2. *(Completeness) Suppose $\Gamma_2 \vdash \mathsf{e} : \tau_2$. Then $\mathcal{A}lg(\mathsf{e}) = (\mathbb{C}_1\,;\Gamma_1\,;\tau_1)$, and $\mathbb{C}_1 \rightsquigarrow\!\!\!\!\to \mathbb{C}_2 \equiv \{\tau_1 <: \tau_3\}$. Then $\Gamma_1 \preceq \Gamma_2$ and $\tau_1 \preceq \tau_2$. ($\preceq$ is the usual preorder).*

Proof. The proof from Mitchell paper seem to fit this adaptation well; it relies in propagate and simplify constraints; it is quite technical and it is currently being checked. A bit of care must be devoted in studying complexity of the algorithm.

9 Conclusions and future work

We introduced a functional calculus *à la* Curry that features uniformly message sending and function application in a simple and natural way. The type system is rather simple, just function-types, record-types and subtyping to deals with functional fix points (fix points are not really essential to the first-class method label encoding). The system is proved to prevents message-not-found run-time errors. We also adapt of the Mitchell's $\mathcal{G}$ type inference algorithm [Mit91]. There are a few propositions of such things in the literature, such as Bugliesi-Crafa [BC99], or Müller-Nishimura [MN00]. Besides using a rather richer language (featuring first-class functions), we the think that our proposition is extremely simple since it does not need extreme complication neither in the type system nor in the source language.

Among the possible further work we would like to mention those few items:

- study the complexity of $\mathcal{A}lg$.
- improve OhML with an object-based inheritance.
- add polymorphic-types and polymorphic type-inference.
- add references and reference-type.

Acknowledgments. The authors are sincerely grateful to François Pottier and Didier Remy, for the fruitful discussions on type inference with subtyping and for suggesting a further simplification of the core OhML calculus. We also warmly thank all the anonymous referees of a previous submission for their insight comments and sharp suggestions that helped us to improve the paper. This work is supported by the French grant CNRS *ACI Modulogic*.

References

[AC94] Martin Abadi and Luca Cardelli. A Theory of Primitive Objects: untyped and first order systems. In *Proc. of TACS*, volume 789 of *Lecture Notes in Computer Science*, pages 296–320. Springer-Verlag, 1994.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[BC99] M. Bugliesi and S. Crafa. Object calculi with dynamic messages. In *FOOL'6 Workshop on Foundation of Object Oriented Languages*, 1999.

[DM82] L. Damas and R. Milner. Principal Type-Schemes for Functional Programs. In *Proc. of POPL*, pages 207–212. The ACM Press, 1982.

[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.

[Liq97] L. Liquori. Bounded Polymorphism for Extensible Object. Technical Report CS-24-96, Computer Science Department, University of Turin, Italy, 1997. `ftp://lambda.di.unito.it/pub/liquori/bounded97.ps`.

[LP] X. Leroy and F. Pottier. Course DEA: Typage et programmation. `http://pauillac.inria.fr/~fpottier/mpri/dea-typage.ps.gz`.

[Mel96] P-A Melliès. *Description Abstraite des Systèmes de Réécriture*. PhD thesis, Université Paris 7, December 1996.

[Mit91] J. C. Mitchell. Type inference with simple subtypes. In *Journal of Functional Programming*, 1991.

[MN00] M. Müller and S. Nishimura. Type inference for first-class messages with feature constraints. *International Journal of Foundations of Computer Science*, 11(1):29–63, 2000.

[Pot98] F. Pottier. *Synthèse de types en présence de sous-typage: de la théorie à la pratique*. PhD thesis, Université Paris 7, July 1998.

[Pot04] F Pottier. *Types et Contraintes*. Mémoire d'habilitation à diriger des recherches, Université Paris 7, 2004.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–49, 1965.