プログラミング-言語・基礎・実践 14－5
（1993．10．29）

パラメトリックな多相とアドホックな多相の共生する計算系

鈴木大介

東京大学理学部情報科学科
〒113 文京区本郷 7-3-1
dai@is.s.u-tokyo.ac.jp

ポリモルフィズム（多相）はプログラミングの柔軟性と再利用性を向上させるのに必要な道具の一つである。Strachey は 多相を parametric なものと ad-hoc なものに区分した。これらはプログラム言語において広く用いられているが、実際には両方の多相を備えた言語はほとんど存在しないので、プログラムの再生産性には制限が生じる。我々は parametric 多相と ad-hoc 多相を備えたオブジェクト指向言語のための新しい枠組である $F^{m^*}_{\leq}$ — F に subtyping, bounded qantification, coercion, merge の機能を追加したもの — を提案する。そして、この計算系が、簡約による合流性、型保存性、強正規性などの性質を満たすことを示す。最後に、この計算系の実際のオブジェクト指向言語へのいくつかの応用例を与える。

A Calculus Integrating Parametric and Ad-hoc Polymorphism

Daisuke Suzuki

Department of Information Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, JAPAN
dai@is.s.u-tokyo.ac.jp

Polymorphism is one of the important tools for increasing programming flexibility and re-usability of programs. Strachey distinguished between two kinds of polymorphism: parametric and ad-hoc. The importance of both kinds of polymorphism is widely recognized, but existing programming languages usually support only one kind of polymorphism, and consequently we find restriction in re-using programs. In this paper, we propose a new framework named $F^{m^*}_{\leq}$ (an extension of F with subtyping, bounded quantification, coercion, and merge) for object-oriented programming languages with parametric and ad-hoc polymorphism, which enjoys the properties of Church-Rosser, Subject Reduction, and Strong Normalization. Finally we show some examples of applications of this calculus to object-oriented languages.

1 Introduction

1.1 Background — Parametric vs. Ad-Hoc

Polymorphism is one of the important tools for increasing programming efficiency and reusability of programs. Polymorphism has classification: "parametric v.s ad hoc" (see [11]). A *parametric polymorphic function* is a function which is defined uniformly over types. That is, though a parametric polymorphic function behaves as a collection of monomorphic functions, they have the *shared* procedure written as a single lambda expression. System **F**[7] enables us to treat parametric polymorphic functions as first-order objects. From the point of view of object-oriented languages which support the notion of subtyping, $\mathbf{F}_{\leq}$[3, 4, 6] is presented by Ghelli. $\mathbf{F}_{\leq}$ is a conservative extension of **F**, which allows us to specify bounds on $\forall$-function types. In addition, there are many studies about characterizing a parametric polymorphic function in model theory and category theory. However, in **F** (and $\mathbf{F}_{\leq}$), a polymorphic function whose behavior on different types is unrelated cannot be written.

Such a polymorphic function is called an *ad-hoc polymorphic function*. An example of such a function is a function which is written by type case. However, ad-hoc polymorphism has not received so much attention as parametric polymorphism. In [5], with the definition of the $\lambda\&$-calculus, Castagna, Ghelli and Longo started a theoretical analysis of ad-hoc polymorphism. In $\lambda\&$, an overloaded function is represented by a list of ordinary functions. When an overloaded function is applied to an argument, it selects only one function from the component functions according to the type of the argument, and returns its result. Therefore, in $\lambda\&$, an ad-hoc polymorphic function whose behavior on different argument types is entirely unrelated is easily written as a first-class object. In [13, 14], Tsuiki proposed a calculus λ_m, which is a simply typed λ-calculus enriched with subtyping, coercion and merge operator. Merge operator is provided by generalizing from record merge operator of extension of ML type system with record[9, 15]. In λ_m, the notion of merge to functions is introduced and corresponds to a message in object-oriented languages. It is good-natured in the sense that the behavior on a subtype *inherits* those on its supertypes. A merge of functions activates all of appropriate functions and merges their effects. Therefore, component functions work together to calculate a result; and thus the behavior of a function on each type is closely related. However, in $\lambda\&$ or λ_m, since they are conservative extensions of simply typed λ-calculus, no parametric polymorphic function cannot be written.

From these reasons, we want to introduce a new calculus $\mathbf{F}^{m^*}_{\leq}$, which integrates parametric and ad-hoc polymorphism.

1.2 Integration of Two Kinds of Polymorphism

Castagna introduced $\mathbf{F}^{\&}_{\leq}$[2], which integrates parametric polymorphism and "explicit" ad-hoc polymorphism. In $\mathbf{F}^{\&}_{\leq}$, a function which performs a dispatch on a type passed as an argument would be written as: (X means type variables)

$$\Lambda X \leq T_1.exp_1 \& \Lambda X \leq T_2.exp_2 \& \cdots \Lambda X \leq T_n.exp_n$$

When a type V is applied this function to, $\Lambda X \leq T_i.exp_i$ which satisfies $T_i = \min\{T_k \mid k = 1, \ldots, n, V \leq T_k\}$[1] is selected and returns the value $exp_i\{X := V\}$. However, in $\mathbf{F}^{\&}_{\leq}$, types permitted to be applied the sorts of functions to are restricted to base types[2], and when writing an "explicit" ad-hoc polymorphic function whose behavior on types is closely related, we cannot write the "common" procedure for describing the related behavior because of the same reason as in the case of $\lambda\&$.

To solve those problems, we propose another calculus integrating (explicit) parametric polymorphism and (implicit and explicit) ad-hoc polymorphism, by defining $\mathbf{F}^{m^*}_{\leq}$.

1.3 Merge Operator

Merge is the notion given by generalizing from record merge operator. In λ_m, the notion of merge to functions is introduced. A merge of functions activates several functions and merges their effects. Therefore, component functions work together to calculate a result; and thus the behavior of a function on each type is closely related. In other words,, an ad-hoc function whose

[1] min is defined over subtyping relation.

[2] I think this restriction will not be required.

behavior on each type is closely related can be written as less procedures in λ_m than in $\lambda\&$, because, by virtue of merge of functions, we need only to write a single "shared" procedure which describes the closely related behavior on each type in λ_m, while, in $\lambda\&$, for each type, we must write its own procedure instead.

Thus, it can be said that merge of functions bridge the gap between parametric polymorphism and ad-hoc polymorphism but it does not completely succeed yet in the sense that no complete parametric polymorphic function can be written. In this paper, by proposing $\mathbf{F}^{m*}_{\leq}$, we try extending the notion of merge to second-order lambda calculus to remove entirely the gap.

1.4 Outline

The rest of this paper consists of two parts.

In the first part (Section 2,3,4), we define a new calculus $\mathbf{F}^{m*}_{\leq}$ which integrates parametric and ad-hoc polymorphism. $\mathbf{F}^{m*}_{\leq}$ is an extension of **F** enriched with subtyping, coercion, bounded polymorphism, and merge operator. In order to dispatch appropriate functions depending on the type of the arguments, $\mathbf{F}^{m*}_{\leq}$ uses type information at runtime in contrast to second-order languages. Therefore, the type of each expression must be uniquely determined and preserved by reduction (which is not impossible with the subsumption rule).

In the second part (Section 5), we mainly discuss the relation between $\mathbf{F}^{m*}_{\leq}$ and object-oriented programming language. In $\mathbf{F}^{m*}_{\leq}$, a function which performs a dispatch on a type passed as an argument would written as:

```
Fun(X:Type) = X<T1 => exp_1
              ....
              X<Tn => exp_n
```

This function can be written in $\mathbf{F}^{\&}_{\leq}$, but there are a lot of differences: If this function is applied to a type S, the function executes all exp_i such that $S \leq T_i$ and merges all the effects into one result. In $\mathbf{F}^{m*}_{\leq}$, this function (to say the truth, $\forall$-function term) is denoted by:

$$(\Lambda X \leq T_1.exp_1 \wedge \ldots \Lambda X \leq T_n.exp_n)$$

and its type is $\forall X\{T_1.S_1, \ldots, T_n.S_n\}$ (where $exp_i{:}S_i$). However this type is a rough approximation yet and therefore we need some restrictions on the S_i's and T_i's, which is discussed in section 2. That description can be done in $\mathbf{F}^{\&}_{\leq}$, but, as in the discussion about ad-hoc polymorphism, the behavior of a polymorphic function whose argument is a subtype and that of the function whose argument is a supertype are closely related in λ_m, we need write only one "shared" procedure for describing the "related" behavior because of the merge operator, while, in $\mathbf{F}^{\&}_{\leq}$, for each type, its own procedure need be described. So, we believe that $\mathbf{F}^{m*}_{\leq}$ is better than $\mathbf{F}^{\&}_{\leq}$ for re-using existing programs to write new programs.

2 Type

In this section we describe the type system of $\mathbf{F}^{m*}_{\leq}$. In our paper we assume that there is no subtype relation between base types.

We use a following notation when no specific notation is declared.

Notation	Metavariables
Type Variables	$X, Y, \ldots$
Type Constants	$A, B, \ldots$
Type	$C, S, T, U, \ldots$
Term Variables	$x, y, \ldots$
Term Constants	$a, b, \ldots$
Term	$e, f, \ldots$

2.1 Raw Types

Let I, J be finite (countable) sets. *Raw types* are defined as follows:

S	::=	X	type variables
	|	A	basic type constants
	|	Top	
	|	$[S_i \to T_i]_{i \in I}$	($\to$-)function type
	|	$\forall X.\{S_i.T_i\}_{i \in I}$ $(X \notin FV(S_i))$	$\forall$-function type

When I is $\{1, \ldots, n\}$, $[S_i \to T_i]_{i \in I}$ denotes a multiset of function type $[S_1 \to T_1, \ldots, S_n \to T_n]$, and $\forall X.\{S_i.T_i\}_{i \in I}$ denotes a multiset of $\forall$-function type $\forall X.\{S_1.T_1, \ldots, S_n.T_n\}$.

2.2 Type Environments

In $\mathbf{F}^{m^*}_{\leq}$ (as in $\mathbf{F}_{\leq}$), all type variables have upper bounds, and, therefore, a *(type) environment* is defined as a list of pairs of free type variable and its "upper bound".

$$(\emptyset \text{ env}) \quad \emptyset \text{ env}$$

$$(\leq \text{ env}) \quad \frac{\Gamma \text{ env} \quad \Gamma \vdash S \text{ type} \quad X \notin \Gamma}{\Gamma, X \leq S \text{ env}}$$

2.3 Judgements

We have four kinds of type judgement: for type good-formation ($\Gamma \vdash S$ type), for the subtyping relation ($\Gamma \vdash S \leq T$), for the "merge"-ability relation ($\Gamma \vdash S \uparrow T$) and for the type assignment ($\Gamma \vdash e : S$). We also call the first three kinds of judgements *type judgements*. All kinds of *type judgements* are mutually defined.

2.3.1 Types Good-Formation ($\Gamma \vdash S$ type)

We define the judgement of type "good-formation".

$$\text{(Top Form)} \quad \frac{\Gamma \text{ env}}{\Gamma \vdash \text{Top type}}$$

$$\text{(X Form)} \quad \frac{\Gamma \vdash S \text{ type} \quad X \text{ does not occur in } \Gamma}{\Gamma, X \leq S \vdash X \text{ type}}$$

$$\text{($\rightarrow$ Form)} \quad \frac{\Gamma \vdash S_i \text{ type } (i \in I) \quad \Gamma \vdash T_i \text{ type } (i \in I) \quad \left\{ \begin{array}{l} \text{For } i,j \in I. \text{ such that } \Gamma \vdash S_i \uparrow S_j \text{ holds} \\ \Gamma \vdash T_i \uparrow T_j \end{array} \right.}{\Gamma \vdash [S_i \rightarrow T_i]_{i \in I} \text{ type}}$$

$$\text{($\forall$ Form)} \quad \frac{\Gamma \vdash S_i \text{ type } (i \in I) \quad \Gamma, X \leq S_i \vdash T_i \text{ type } (i \in I) \quad \left\{ \begin{array}{l} \text{For } i,j \in I. \text{ such that } \Gamma \vdash S_i \uparrow S_j \text{ holds} \\ \Gamma, X \leq S_i \wedge S_j \vdash T_i \uparrow T_j \end{array} \right.}{\Gamma \vdash \forall X.\{S_i.T_i\}_{i \in I} \text{ type}}$$

2.3.2 Subtyping Relation ($\Gamma \vdash S \leq T$)

The definitions of subtype relation is listed as below:

$$(\text{Refl} \leq) \quad \frac{\Gamma \vdash S \text{ type}}{\Gamma \vdash S \leq S}$$

$$(\text{Trans} \leq) \quad \frac{\Gamma \vdash S \leq T \quad \Gamma \vdash T \leq U}{\Gamma \vdash S \leq U}$$

$$(\text{Taut} \leq) \quad \frac{\Gamma, X \leq S, \Gamma' \text{ env}}{\Gamma, X \leq S, \Gamma' \vdash X \leq S}$$

$$(\text{Top} \leq) \quad \frac{\Gamma \vdash S \text{ type}}{\Gamma \vdash S \leq \text{Top}}$$

The definitions of subtyping relation between function types and $\forall$-function types are slightly complicated and postponed.

2.3.3 "Merge"-ability Relation ($\Gamma \vdash S \wedge T$)

Before defining subtype relation between function types and $\forall$-function types, we need to define the notion of merge and "merge"-ability relation.

Merge of two type S and T is written as $S \wedge T$.

We can define $S \wedge T$ as an abbreviation for a type expression as follows:

$$\begin{array}{l} X \wedge X \\ = X \end{array}$$

$$\begin{array}{ll} X \wedge S & (\Gamma \vdash X \leq S \text{ holds}) \\ = X & \end{array}$$

$$\begin{array}{l} S \wedge \text{Top} \;=\; \text{Top} \wedge S \\ = S \end{array}$$

$$\begin{array}{l} [S_1 \rightarrow T_1, \ldots, S_n \rightarrow T_n] \wedge [S'_1 \rightarrow T'_1, \ldots, S'_m \rightarrow T'_m] \\ = [S_1 \rightarrow T_1, \ldots, S_n \rightarrow T_n, S'_1 \rightarrow T'_1, \ldots, S'_m \rightarrow T'_m] \end{array}$$

$$\begin{array}{l} \forall X.\{S_1.T_1, \ldots, S_n.T_n\} \wedge \forall X.\{S'_1.T'_1, \ldots, S'_m.T'_m\} \\ = \forall X.\{S_1.T_1, \ldots, S_n.T_n, S'_1.T'_1, \ldots, S'_m.T'_m\} \end{array}$$

It should be noted that, under an environment Γ, $S \wedge T$ can be defined only when "merge"-ability relation $S \uparrow T$ holds.

$$\frac{X \in \Gamma}{\Gamma \vdash X \uparrow X} \quad (\wedge \text{ ReflVar})$$

$$\frac{\Gamma \vdash X \leq A}{\Gamma \vdash X \uparrow A} \qquad \frac{\Gamma \vdash X \leq A}{\Gamma \vdash A \uparrow X} \quad (\wedge \text{ SubVar})$$

$$\Gamma \vdash C \uparrow C \quad (C \text{ is a base type.}) \quad (\wedge \text{ ReflConst})$$

$$\frac{\left\{\begin{array}{l}\text{For } i \in I, j \in J \text{ such that } \Gamma \vdash S_i \uparrow S'_j \text{ holds,} \\ \Gamma \vdash T_i \uparrow T'_j\end{array}\right.}{\Gamma \vdash [S_i \to T_i]_{i \in I} \uparrow [S'_j \to T'_j]_{j \in J}} \quad (\wedge\text{-}\to)$$

$$\frac{\left\{\begin{array}{l}\text{For } i \in I, j \in J \text{ such that } \Gamma \vdash S_i \uparrow S'_j \text{ holds,} \\ \Gamma, X \leq S_i \wedge S'_j \vdash T_i \uparrow T'_j\end{array}\right.}{\Gamma \vdash \forall X.\{S_i.T_i\}_{i \in I} \uparrow \forall X.\{S'_j.T'_j\}_{j \in J}} \quad (\wedge\text{-}\{\})$$

Lemma 1 *1.* $\Gamma \vdash S \uparrow S$

2. $\Gamma \vdash S \uparrow T$ *implies* $\Gamma \vdash T \uparrow S$

3. $\Gamma \vdash S \uparrow T$ *is decidable.*

Note that $\uparrow$ is symmetric and reflexive, but not a transitive relation.

From these definitions, function type and $\forall$-function type can be constructed as merged ordinary function types and as merged ordinary $\forall$-function types, respectively.

2.3.4 Subtyping Relation (continued)

After notions of merge and "merge"-ability are introduced, the subtype relation about function type and $\forall$-function type is defined as follows:

Definition 1 *Under an environment* Γ, *let* F *be* $[S_i \to T_i]_{i \in I}$ *and* I' *be* $\{i \in I \mid C \leq S_i\}$. *Define* $AT(F,C)$, $AP(F,C)$, *and* $M(F)$ *as follows:*

$$\begin{array}{lcl} AT(F,C) & = & \wedge_{i \in I'} S_i \\ AP(F,C) & = & \wedge_{i \in I'} T_i \\ M(F) & = & \{[A \to B] \mid A = \wedge_{i \in J} S_i, \\ & & B = AP(F,A) \\ & & \emptyset \neq J \subseteq I \\ & & (\text{For all } i,j \in J, \Gamma \vdash S_i \uparrow S_j.)\} \end{array}$$

The subtyping rule between function types ($\to - \leq$) is defined in figure 1.

Definition 2 *Under an environment* Γ, *let* F *be* $\forall X.[S_i, T_i]_{i \in I}$ *and* I' *be* $\{i \in I' \mid C \leq S_i\}$. *Define* $AT2(F,C)$, $AP2'(F,C)$, $AP2(F,C)$, *and* $M2(F)$ *as follows:*

$$\begin{array}{lcl} AT2(F,C) & = & \wedge_{i \in I'} S_i \\ AP2'(F,C) & = & \wedge_{i \in I'} T_i \\ AP2(F,C) & = & AP2'(F,C)[X \leftarrow C] \\ & = & \wedge_{i \in J} T_i[X \leftarrow C] \\ M2(F) & = & \{(\wedge_{i \in I'} S_i) \mid \emptyset \neq J \subseteq I \\ & & (\text{For all } i,i' \in J \text{ ,} \Gamma \vdash S_i \uparrow S_{i'} .)\} \end{array}$$

The subtyping rule between $\forall$-function types ($\forall - \leq$) is defined in Figure 1.

We write $\Gamma \vdash S \equiv T$ when both of $\Gamma \vdash S \leq T$ and $\Gamma \vdash T \leq S$ hold.

We call $S_1, \ldots, S_n$ (*pairwise*) *compatible* under an environment Γ when $\Gamma \vdash S_i \uparrow S_j$ holds for any $i, j = 1, \ldots, n$

From these definitions, some properties as listed below are satisfied.

Lemma 2 *1. When* $\Gamma \vdash S \uparrow T$, $\Gamma \vdash S \wedge T$ type.

2. $S \wedge S = S$

3. $\Gamma \vdash (S \wedge T) \uparrow S$ *and* $\Gamma \vdash (S \wedge T) \uparrow T$

4. When $(S \wedge S')$ *and* T *are compatible,* $S \uparrow T$, $S' \uparrow T$, *and* $(S \wedge S') \wedge T = S \wedge (S' \wedge T)$

5. When $S_1, \ldots, S_n$ *and* T *are compatible,* $\wedge_{i=1,\cdots,n} S_i$ *and* T *are compatible*

6. If $\Gamma \vdash S \leq T$ *then* $\Gamma \vdash S$ type *and* $\Gamma \vdash T$ type

7. $\Gamma \vdash S \leq T$ *and* $\Gamma \vdash S \uparrow S'$ *implies* $\Gamma \vdash S' \uparrow T$. *Especially,* $\Gamma \vdash S \leq T$ *implies* $\Gamma \vdash S \uparrow T$.

8. $\Gamma \vdash S \leq T$, $\Gamma \vdash S' \leq T'$, *and* $\Gamma \vdash S \uparrow S'$ *implies* $\Gamma \vdash T \uparrow T'$ *and* $\Gamma \vdash S \wedge S' \leq T \wedge T'$. *Especially,* $\Gamma \vdash S \leq T$ *and* $\Gamma \vdash S \leq T'$ *implies* $\Gamma \vdash T \uparrow T'$ *and* $\Gamma \vdash S \leq T \wedge T'$.

9. $\Gamma \vdash S \leq T$ *is decidable*

3 Term

Let F be $[S_i \to T_i]_{i \in I}$

$$(\to \leq) \quad \frac{\begin{array}{l}\text{Under an environment } \Gamma, \text{ for any } [S \to T] \in M(F), \text{ there exists } AT(G,S) \\ \text{such that } \Gamma \vdash S \leq AT(G,S) \text{ and also } \Gamma \vdash AP(G, AT(G,S)) \leq T\end{array}}{\Gamma \vdash G \leq F}$$

Let F be $\forall X.[S_i, T_i]_{i \in I}$

$$(\forall \leq) \quad \frac{\begin{array}{l}\text{Under an environment } \Gamma, \text{ for all } S \in M2(F), \text{ there exists } AT2(G,S) \\ \text{such that } \Gamma, X \leq S \vdash AP2'(G, AT2(G,S))) \leq AP2'(F,S)\end{array}}{\Gamma \vdash G \leq F}$$

Figure 1: Subtype relation about function type

In this section, we describe the terms of $\mathbf{F}^{m}_{\leq}{}^{*}$. We start by the definition of *raw terms*, among which we distinguish the *terms*, i.e. those raw terms possessing types. Raw terms are divided into five classes: variables and constants, function terms (functions), $\forall$-function terms, coerced terms, and merged terms. Coercion is introduced to hold unicity of type and subject reduction. A merged term is represented by a list of ordinary function terms or $\forall$-function terms, i.e., A merged term is a merged function (term) or a merged $\forall$-function term. The $\wedge$'s are their connectives for describing merged terms.

3.1 Raw Terms

e	::=	x	variable
	|	c	constant
	|	top	only constant term of Top
	|	$\lambda x^S.e$	λ-abstraction
	|	ef	λ-application
	|	$\Lambda X \leq S.e$	Λ-abstraction
	|	$e\{S\}$	Λ-application
	|	$e\|_S$	coercion
	|	$\wedge \vec{e}$	merged term

3.2 Type Assignment ($\Gamma \vdash e : S$)

We use one meta notation: $e[x \leftarrow f], e[X \leftarrow S], T[X \leftarrow S]$ for substitutions. Terms are selected by the rules below; since term variables are indexed by their type, the rules do not needed assumptions of the term $(x : T)$:

$$\text{(Top)} \quad \Gamma \vdash \text{top} : \text{Top}$$

$$\text{(Var)} \quad \frac{\Gamma \vdash S \text{ type}}{\Gamma \vdash x^S : S}$$

$$\text{(Coerce)} \quad \frac{\Gamma \vdash e : T \quad \Gamma \vdash T \leq S}{\Gamma \vdash e|_S : S}$$

$$(\to\text{-I}) \quad \frac{\Gamma \vdash e : T}{\Gamma \vdash \lambda x^S.e : [S \to T]}$$

$$(\to\text{-E}) \quad \frac{\begin{array}{l}\Gamma \vdash e : F = [S_i \to T_i]_{i \in I} \\ f : S \qquad AT(F,S) \text{ exists}\end{array}}{\Gamma \vdash ef : AP(F,C)}$$

$$(\forall\text{-I}) \quad \frac{\Gamma, X \leq S \vdash e : T}{\Gamma \vdash \Lambda X \leq S.e : \forall X.\{S.T\}}$$

$$(\forall\text{-E}) \quad \frac{\begin{array}{l}\Gamma \vdash f : F = \forall X.\{S_i.T_i\}_{i \in I} \\ \Gamma \vdash S \text{ type} \qquad AT2(F,S) \text{ exists}\end{array}}{\Gamma \vdash f\{S\} : AP2(F,S)}$$

$$\text{(Merge)} \quad \frac{\text{For all } i = 1, \ldots, n, \qquad \Gamma \vdash e_i : S_i \text{ holds}}{\Gamma \vdash e_1 \wedge \ldots \wedge e_n : S_1 \wedge \ldots \wedge S_n}$$

$\lambda x^S.e$ is a function abstraction and ef is a func-

(App1)	$(\lambda x^S.e)f$	$\Rightarrow$	$e[x \leftarrow f\|_S]$	
(App2)	$(\Lambda X \leq S.e)\{T\}$	$\Rightarrow$	$e[X \leftarrow T]$	$(T \leq S)$
(Appc1)	$(e\|_F)f$	$\Rightarrow$	$(ef\|_{AT(F,S)})\|_{AP(F,S)}$	$(f : S)$
(Appc2)	$(e\|_F)\{S\}$	$\Rightarrow$	$(e\{S\})\|_{AP2(F,S)}$	
(AppL1)	$(\wedge_{i\in I} e_i)f$	$\Rightarrow$	$\wedge_{i\in I'}(e_i f)$	$(e_i : [S_i \Rightarrow T_i],\ f : C$ $I' = \{i \in I \mid C \leq S_i\})$
(AppL2)	$(\wedge_{i\in I} e_i)\{C\}$	$\Rightarrow$	$\wedge_{i\in I'}(e_i\{S\})$	$(e_i : \forall X.\{S_i.T_i\})$
(Mer1)	$e \wedge f$	$\Rightarrow$	f	e and f have the same type
(Mer2)	$e \wedge f$ (or $f \wedge e$)	$\Rightarrow$	f	e : Top
(Cor1)	$e\|_S$	$\Rightarrow$	e	$e : S$
(Cor2)	$e\|_{\mathrm{Top}}$	$\Rightarrow$	top	

Figure 2: Reduction rule

tion application. $\Lambda X \leq S.e$ is a $\forall$-function abstraction and $e\{S\}$ is a $\forall$-function application.

Theorem 3 *If* $\Gamma \vdash e : S$ *then* $\Gamma \vdash S$ type

Theorem 4 (Unicity of Type) *If* $\Gamma \vdash e : S$ *and* $\Gamma \vdash e : T$ *then* $S \equiv T$

4 Reduction System

The reduction rules in $\mathbf{F}^{m*}_{\leq}$ are listed in Figure 2:

(App1) is the rule of function application. From this rule, the coercion operator occur after reduction. (App2) is the β-reduction rule for $\forall$-function terms.

(AppL1) ((AppL2)) is the rule of an application of a merged function (a merged $\forall$-function term). The application of a merged function (a merged $\forall$-function term) is the merge of the result of applicable functions ($\forall$-function terms).

(Appc1) and (Appc2) are the rules for applying a function defined by coercing another function. In (Appc1), first, an argument is essentially applied a coerced function to after it is coerced into the appropriate argument type, and then the result is coerced into the appropriate type. In (Appc2), the argument type is applied to, then the result is coerced similarly.

(Mer1) says that the merge of the two value reduces to "right" value[3]. It enables to implement, for example, record type with replacement function.

$\mathbf{F}^{m*}_{\leq}$ satisfies some good properties as follows: the proof of these theorems will appear in [12]

Theorem 5 (Subject Reduction) *The above reduction rule is well-defined. That is, when e is reduced to e′ and e has type S, then e′ also has type S.*

Theorem 6 (Church-Rosser) *When an expression e reduces to* e_1 *and* e_2*, there is an expression f such that* $e_1 \rightarrow^* f$ *and* $e_2 \rightarrow^* f$

Theorem 7 (Strong Normalization) *In* $\mathbf{F}^{m*}_{\leq}$*, all expressions are strongly normalizing.*

5 Object-Oriented Programming Languages

We discuss the relation between object-oriented languages and $\mathbf{F}^{m*}_{\leq}$ more deeply.

Firstly, we show that records can be implemented in $\mathbf{F}^{m*}_{\leq}$, and secondly we propose an example of an application of $\mathbf{F}^{m*}_{\leq}$ to object-oriented languages. Before discussing these topics, it should be noted that recursive types (*Int*, *List*, etc.) are easily defined because $\mathbf{F}^{m*}_{\leq}$ is an extension of $\mathbf{F}$.

[3] In λ_m [13], only the merge of the same constant can be reduced. This idea of selection of right value is abstracted from $\lambda\&$ [5] and $\mathbf{F}^{\&}_{\leq}$ [2]

Subtyping relation

$$\frac{\text{For all } i = 1, \ldots, k,\ \Gamma \vdash V_i \leq U_i}{\Gamma \vdash \langle\langle l_1{:}V_1\ ;\ \ldots\ ;\ l_k{:}V_k\ ;\ \ldots\ ;\ l_{k+j}{:}V_{k+j}\rangle\rangle \leq \langle\langle l_1{:}U_1\ ;\ \ldots\ ;\ l_k{:}U_k\rangle\rangle}$$

"Merge"-ability relation

$$\frac{\text{For all } i = 1, \ldots, k,\ \Gamma \vdash V_i \uparrow U_i}{\Gamma \vdash \langle\langle l_1{:}V_1\ ;\ \ldots\ ;\ l_k{:}V_k\ ;\ \ldots\ ;\ l_{k+j}{:}V_{k+j}\rangle\rangle \uparrow \langle\langle l_1{:}U_1\ ;\ \ldots\ ;\ l_k{:}U_k\ ;\ l_{k+j+1}{:}U_{k+j+1}, \ldots, l_{k+m}{:}U_{k+m}\rangle\rangle}$$

Type assignment rule

$$\textbf{(Record)}\quad \frac{\text{For all } i = 1, \ldots, n,\ \Gamma \vdash M_i{:}V_i}{\Gamma \vdash \langle l_1 = M_1, \ldots, l_n = M_n\rangle{:}\langle\langle l_1{:}V_1, \ldots, l_n{:}V_n\rangle\rangle}$$

$$\textbf{(Dot)}\quad \frac{\Gamma \vdash M{:}\langle\langle l_1{:}V_1, \ldots, l_n{:}V_n\rangle\rangle}{\Gamma \vdash M.l_i : V_i}$$

Reduction rule

$$\langle l_1 = M_1\ ;\ \ldots\ ;\ l_n = M_n\rangle \quad \Rightarrow \quad M_i$$

$$\begin{array}{l}\langle l_1 = M_1\ ;\ \ldots\ ;\ l_n = M_n\ ;\ l_{n+1} = M_{n+1}\ ;\ \ldots\ ;\ l_{n+j} = M_{n+j}\rangle \\ \wedge\langle l_1 = N_1\ ;\ \ldots\ ;\ l_n = N_n\ ;\ l_{n+j+1} = N_{n+j+1}\ ;\ \ldots\ ;\ l_{n+k} = M_{n+k}\rangle\end{array} \quad \Rightarrow \quad \begin{array}{l}\langle l_1 = M_1 \wedge N_1\ ;\ \ldots\ ;\ l_n = M_n \wedge N_n \\ \quad ;\ l_{n+1} = M_{n+1}\ ;\ \ldots\ ;\ l_{n+j} = M_{n+j} \\ \quad ;\ l_{n+j+1}{:}N_{n+j+1}\ \ldots\ ;\ l_{n+k} = M_{n+k}\rangle\end{array}$$

Figure 3: Rules of Record Type

5.1 Records

In various approaches to object-oriented programming, records play an important role. In particular, current functional treatments of object-oriented features formalize objects directly as records. Moreover, if records are not included in a calculus, the subtyping relation may be quite trivial. In our system, records can be encoded in two straightforward ways as in [2, 5], i.e., in terms of implicit ad-hoc polymorphism and in terms of explicit ad-hoc polymorphism.

(Using implicit ad-hoc polymorphism)

Let $L_1, L_2, \ldots$ be base types. Assume that they are *isolated* (i.e., for any type V, if $L_i \uparrow V$, then $V = L_i$ or $V = \text{Top}$) to each other, and introduce, for each L_i, a unique *constant* $l_i{:}L_i$. It is now possible to encode record types, record values, overwriting, and selection, as follows:

$$\begin{array}{l}\langle\langle l_1{:}V_1, \ldots, l_n{:}V_n\rangle\rangle \\ \equiv \quad [L_1 \to V_1, \ldots, L_n \to V_n] \quad \text{(Record Type)}\end{array}$$

$$\begin{array}{l}\langle l_1 = M_1\ ;\ \ldots\ ;\ l_n = M_n\rangle \\ \equiv \quad \lambda x^{L_1}.M_1 \wedge \ldots \wedge \lambda x^{L_n}.M_n \\ \qquad (x^{L_i} \notin FV(M_i)) \quad \text{(Record Term)}\end{array}$$

$$\begin{array}{l}\langle M \leftarrow l_i = N\rangle \\ \equiv \quad M \wedge \lambda x^{L_i}.N \quad \text{(Overwriting)}\end{array}$$

$$\begin{array}{l}M.l \\ \equiv \quad Ml \quad \text{(Selection)}\end{array}$$

(Using explicit ad-hoc polymorphism)

The way of implementing records using explicit ad-hoc polymorphism is quite similar to that using implicit ad-hoc polymorphism. Let $L_1, L_2, \ldots$ be base types. Assume that they are isolated to each other.

$\langle\langle l_1{:}V_1, \ldots, l_n{:}V_n \rangle\rangle$
$\equiv \quad \forall X.\{L_1.V_1, \ldots, L_n.V_n\}$ (Record Type)
$(X \notin \cup_{i=1,\ldots n} FV(T_i))$

$\langle l_1 = M_1 \ ; \ldots ; \ l_n = M_n \rangle$
$\equiv \quad \Lambda X \leq L_1.M_1 \wedge \ldots \wedge \Lambda X \leq L_n.M_n$
$(X \notin FV(M_i))$ (Record Term)

$\langle M \leftarrow l_i = N \rangle$
$\equiv \quad M \wedge \Lambda X \leq L_i.N$ (Overwriting)

$M.l_i$
$\equiv \quad ML_i$ (Selection)

Some Properties of Record Type

Since $L_1, \ldots, L_n$ are isolated, regardless of the implementation of records applied, we can easily derive the subtyping and "merge"-able rule, the type assignment rule, and the reduction rules for records from those of merged types and merged terms.

Note that two implementations of records have no compatibility to each other. Therefore, we should write programs taking care that two implementations of records do not exist simultaneously.

All the rules are listed in figure 3.

5.2 Applications of $F^{m*}_{\leq}$ to Object-Oriented Languages

We use the name *class* to type the objects of that class. Then, a *message* is (an identifier of) a merged function whose branches are the methods associated to that message. The methods to be executed are selected according to the type (the class-name) passed as an argument.

In $F^{\&}_{\leq}$[2], because all class names are basic types, subtype relation must be defined in the framework. However, in $F^{m*}_{\leq}$, these restrictions are not required, and when we write programs, we must describe less procedures in $F^{m*}_{\leq}$ than in $F^{\&}_{\leq}$.

We will provide an example, and, in this section, we assume that there is no type variable, i.e., the environment is $\emptyset$ (empty).

Let these types defined as follows using record types:

$$
\begin{array}{lcl}
\mathit{point1D} & = & \langle\langle x{:}\mathrm{Int} \rangle\rangle \\
\mathit{point2D} & = & \langle\langle x{:}\mathrm{Int} \ ; \ y{:}\mathrm{Int} \rangle\rangle \\
\mathit{point3D} & = & \langle\langle x{:}\mathrm{Int} \ ; \ y{:}\mathrm{Int} \ ; \ z{:}\mathrm{Int} \rangle\rangle \\
\mathit{color} & = & \langle\langle c{:}\mathrm{Str} \rangle\rangle \\
\mathit{point1D{+}color} & = & \langle\langle x{:}\mathrm{Int} \ ; \ c{:}\mathrm{Str} \rangle\rangle \\
\mathit{point2D{+}color} & = & \langle\langle x{:}\mathrm{Int} \ ; \ y{:}\mathrm{Int} \ ; \ c{:}\mathrm{Str} \rangle\rangle \\
\mathit{point3D{+}color} & = & \langle\langle x{:}\mathrm{Int} \ ; \ y{:}\mathrm{Int} \ ; \ z{:}\mathrm{Int} \\
 & & \qquad ; \ c{:}\mathrm{Str} \rangle\rangle
\end{array}
$$

From these definitions, for example, $\vdash \mathit{point3D} \leq \mathit{point2D}$, and $\vdash \mathit{point1D{+}color} \leq \mathit{point1D}$ hold. We can define a message *Norm* working on these types:

$$
\begin{array}{l}
\mathit{Norm} \quad \equiv \quad (\\
\quad \Lambda \mathit{Mytype} \leq \mathit{point1D}.\lambda \mathit{self}^{\mathit{Mytype}}.\sqrt{\mathit{self}.x^2} \\
\wedge \Lambda \mathit{Mytype} \leq \mathit{point2D}.\lambda \mathit{self}^{\mathit{Mytype}}. \\
\qquad\qquad \sqrt{\mathit{self}.x^2 + \mathit{self}.y^2} \\
\wedge \Lambda \mathit{Mytype} \leq \mathit{point3D}.\lambda \mathit{self}^{\mathit{Mytype}}. \\
\qquad\qquad \sqrt{\mathit{self}.x^2 + \mathit{self}.y^2 + \mathit{self}.z^2} \\
)
\end{array}
$$

The type of this message is:

$$
\begin{array}{l}
\forall \mathit{Mytype}.\{\mathit{point1D}.[\mathit{Mytype} \rightarrow \mathit{Real}] \\
\qquad\qquad , \mathit{point2D}.[\mathit{Mytype} \rightarrow \mathit{Real}] \\
\qquad\qquad , \mathit{point3D}.[\mathit{Mytype} \rightarrow \mathit{Real}]\}
\end{array}
$$

We have used the variable *self* to denote the receiver of the message, and, following the notation of [1], the type variable *Mytype* to denote the type of the receiver.

Here is an another message that sets the internal state of an object to zero (or "white"):

$$
\begin{array}{l}
\mathit{Erase} \quad \equiv \quad (\\
\quad \Lambda \mathit{Mytype} \leq \mathrm{Top}.\lambda \mathit{self}^{\mathit{Mytype}}.\mathit{self} \\
\wedge \Lambda \mathit{Mytype} \leq \mathit{point1D}.\lambda \mathit{self}^{\mathit{Mytype}}.\langle x = 0 \rangle. \\
\wedge \Lambda \mathit{Mytype} \leq \mathit{point2D}.\lambda \mathit{self}^{\mathit{Mytype}}.\langle y = 0 \rangle \\
\wedge \Lambda \mathit{Mytype} \leq \mathit{point3D}.\lambda \mathit{self}^{\mathit{Mytype}}.\langle z = 0 \rangle \\
\wedge \Lambda \mathit{Mytype} \leq \mathit{color}.\lambda \mathit{self}^{\mathit{Mytype}}.\langle c = \text{“White”} \rangle \\
)
\end{array}
$$

In $F^{\&}_{\leq}$, we need 7 components corresponding to $\mathit{color}, \mathit{point1D}, \ldots, \mathit{point3D{+}color}$, but, in $F^{m*}_{\leq}$, we need only 5 components.

6 Conclusion

We proposed a new calculus $\mathbf{F}^{m\bullet}_{\leq}$ integrating parametric and ad-hoc polymorphism, and showed its fundamental properties of Church-Rosser, Subject Reduction and Strong Normalization. We also discussed the relation between $\mathbf{F}^{m\bullet}_{\leq}$ and object-oriented programming by providing some examples.

As future works, we try constructing a new calculus by restricting rules of $\mathbf{F}^{m\bullet}_{\leq}$ such that transitivity of coercion operator is always satisfied. We also plan to restrict rules of subtyping when coercion operator is replaced by subsumption in such a calculus so that some decidable type-checking algorithms may exist, as in $\mathbf{F}^{\&T}_{\leq}$[2] which is an alternative of $\mathbf{F}^{\&}_{\leq}$ and where a decidable type-checking algorithm exists.

References

[1] Bruce, K. B., A paradigmatic object-oriented programming language: Design, static typing and semantics. Technical Report CS-92-01, Williams College, 1992.

[2] Castagna, G.,$F^{\&}_{\leq}$: integrating parametric and "ad hoc" second order polymorphism, *Proc., the 4th International Workshop on Database Programming Languages*, 1993. Full version titled "Integration of parametric and "ad hoc" second order polymorphism in a calculus with subtyping" is submitted.

[3] Curien, P. L., and G. Ghelli, Coherence of subsumption, minimum typing and the type checking in $\mathbf{F}_{\leq}$, *Mathematical Structures in Computer Science*, vol. 2, 1992.

[4] Curien, P. L., and G. Ghelli, Confluence and decidability of $\beta\eta$top $\leq$ reduction in $\mathbf{F}_{\leq}$. *Information and Computation*, To appear.

[5] Castagna, G.,G. Ghelli, and G. Longo, A calculus for overloaded functions with subtyping,*Proc., ACM Conference on Lisp and Functional Programming*, pp. 182–192,1992. Full version will appear in *Information and Computation.*

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

[7] Girard, J. Y., *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*, Thèse d'Etat, Université Paris VII (1972).

[8] Girard, J. Y., Y. Lafont, and P. Taylor. *Proof and Types.* Cambridge University Press, 1989.

[9] Ohori, A., and P. Buneman, Static type inference for parametric classes, *ACM conference on Object Oriented Programming Systems,Languages, and Applications*, pp. 445–455, 1989.

[10] Pierce, B., Bounded quantification is undecidable, In *Proc., ACM Conference on Principles of Programming Languages*, 1992.

[11] Strachey, C., Fundamental concepts in programming languages, Lecture notes for International Summer School in Computer Programming, Copenhagen, 1967.

[12] Suzuki, D., $\mathbf{F}^{m\bullet}_{\leq}$ — Another Calculus Integrating Parametric and Ad-hoc Polymorphism, In preparation.

[13] Tsuiki, H., *A Record Calculus with a Merge Operator.* PhD Thesis, Keio University, 1992.

[14] Tsuiki, H., A record calculus with a merge operator (in Japanese). *Journal of Information Processing Society*, vol. 34, No. 5, pp. 954–962, 1993.

[15] Wand, M., Type inference for record concatenation and multiple inheritance, *Proc, Fourth IEEE Annal Symposium on Logic in Computer Science*, pp. 92–97, 1989.