On Typed Calculi with a Merge Operator

Hideki Tsuiki

Department of Computer Science, Kyoto-Sangyo University
Kamigamo Motoyama, Kita-ku, Kyoto 606, Japan
tsuiki@cs.kyoto-su.ac.jp

Abstract. Extensions of the simply typed lambda calculus with a merge operator are defined and investigated. A merge of functions denotes a function which can be applied to more than one type and computes the result by merging the results of the component functions. We propose first two calculi in which the inconsistency between the return values of the component functions is solved using a non-symmetric rule, and show that they do not have good proof-theoretic properties: one does not have the Strong Normalization Property, and the other does not have the Transitivity of Coercions Property. Then, using the type system, we restrict the two calculi in a way that terms which may cause inconsistency are excluded, and show that the two calculi become equivalent and have all the desired properties.
A merge of functions in object oriented programming corresponds to writing a method in a subclass by modifying those in superclasses, and the above restriction corresponds to only allowing adding some behaviors in the modification. This result conforms with the programming methodology in which one should write a subclass method so that it does not override the behavior of superclass methods.

1 Introduction

Calculi with overloading and subtyping have recently been designed with the aim of providing a foundation for typed object-oriented languages ([CGL94], [Tsu94]). In the previous work [Tsu94], the author has shown that overloading, when used with subtyping, causes some undesired properties. That is, we can define two sets of reduction rules on the same syntax and they have different proof-theoretic properties: $\lambda\&C$ does not have the Strong Normalization Property, and $\lambda\&C^*$ does not have the Transitivity of Coercions Property.

In this paper, we propose another way of defining an overloaded function, which enables us to express the behavior on a subtype as a modification of those on supertypes. And then, we restrict the calculi so that an overloaded function behaves uniformly on each type of argument, and show that the restricted calculi have all the desired properties. Since our functions are not defined by overloading, we use the word *generic function* (or simply a function) instead of an overloaded function for a function applicable to more than one type of arguments and computes the result using different algorithms.

Our calculi are the explicit (Church) version of the simply typed lambda calculus extended with a merge operator ($\otimes$). A merge of functions denotes a generic function which computes the result by selecting all the component functions applicable to the argument and merging the results. This is in contrast to the $\lambda\&$-calculus ([CGL94]) and its variants in [Tsu94] in which only one function component is selected.

We first propose two calculi $\lambda_{\otimes}$ and $\lambda^{*}_{\otimes}$ in which the inconsistency between the return values of the component functions is solved using a non-symmetric rule, and show that they have the same undesired properties as $\lambda\&C$ and $\lambda\&C^{*}$have, respectively. Then, using the type system, we restrict the two calculi in a way that terms which may cause inconsistency are excluded, and define $\lambda^{p}_{\otimes}$ and $\lambda^{p*}_{\otimes}$, and show that $\lambda^{p}_{\otimes}$ and $\lambda^{p*}_{\otimes}$ become equivalent and have all the desired properties. Moreover, in $\lambda^{p}_{\otimes}$ and $\lambda^{p*}_{\otimes}$, a generic function works uniformly in that coercion and function application commutes. We will call such a generic function *coherent*.

In object oriented programming, the notion of merge of functions corresponds to writing a method in a subclass by modifying those in superclasses. That is, by adding some behaviors which are peculiar to the subclass, and by overriding some part of the superclass results. In conventional object oriented languages, it is done by calling a "super" method explicitly and then modifying the result. $\lambda_{\otimes}$ and $\lambda^{*}_{\otimes}$ can be considered as proposing this kind of method definition in a declarative way.

Then the restricted systems $\lambda^{p}_{\otimes}$ and $\lambda^{p*}_{\otimes}$ corresponds to allowing only addition of some behaviors in a subclass method definition. It is interesting to note that the fact that $\lambda^{p}_{\otimes}$ and $\lambda^{p*}_{\otimes}$ have good properties conforms with the programming methodology in object oriented programming that one should write a subclass method so that it does not override the behavior of superclass methods.

In the next section, we introduce the notion of a generic function. In Section 3, we explain informally the notion of merge of functions. In Section 4, we define $\lambda_{\otimes}$ and $\lambda^{*}_{\otimes}$, and in Section 5, we show that they do not have all the good properties of typed calculi. Then, in Section 6, we restrict them to define $\lambda^{p}_{\otimes}$ and $\lambda^{p*}_{\otimes}$and study their properties. And in Section 7, we study the relation to object oriented programming.

2 A Generic Function over a Subtype Structure

We consider a type structure with a subtype relation. Instead of considering subtyping as a set inclusion, we consider types as disjoint sets and consider $T \leq S$ as showing the existence of an inherent coercion function from T to S. This is because the type of a term is used in reduction and hence needs to be defined uniquely and preserved by reduction.

Next we consider how a generic function behaves on such a hierarchical type structure. Since a generic function is applicable to more than one types, we can view a generic function as a collection $e_t : t \longrightarrow F(t)$ of monomorphic functions which satisfy the following:

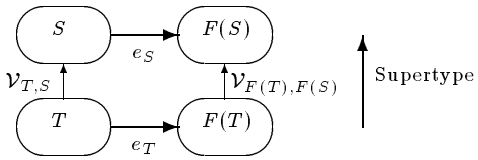
- the domains are downward closed; that is, if e_S is defined and $T \leq S$ then e_T is defined,
- F is a partial monotonic function; that is, if $T \leq S$ then $F(T) \leq F(S)$.

Note that we did not mention the contents of each component function and they may have completely different behavior. The calculi $\lambda_{\otimes}$ and $\lambda^{*}_{\otimes}$ have this notion of a generic function.

Now we say that a generic function is coherent if it preserves the coercible relation:

- if $T \leq S$ and $x : T$ is coercible to $y : S$, then $e_T\ x$ is coercible to $e_S\ y$.

That is, the following diagram commutes:



Here, $\mathcal{V}_{T,S}$ is the coercion function from T to S. This means that the result of e applied to a subtype value x is the result of e applied to "x considered as a supertype element" with some modifications which do not change the functionality as an element of $F(S)$. In the calculi $\lambda^{p}_{\otimes}$ and $\lambda^{p*}_{\otimes}$, all the generic functions are coherent.

Note that this is a relaxization of the characterization of a parametric polymorphic function that it is a function which preserves all the relations between all the types ([Rey83], [Has91]).

A coherent generic function is not a new idea; operators which satisfy this condition have been studied for a decade. For example, consider the implicit coercion from int to $real$ in an Algol like language, and the + operator defined both as $int \times int \longrightarrow int$ and $real \times real \longrightarrow real$. This condition means that the integer addition and the real addition produce the same result when integer values are considered as reals. This is a natural condition for the two operators to be assigned the same symbol, and [Rey81] gave a semantics to such an operator, and Forsythe ([Rey88]) is a language based on this. However, there have been no works on allowing a programmer to define a coherent generic function. In this paper, we study calculi which propose a way of defining such a function.

3 The Notion of Merge of Functions

In this section, we explain informally how a generic function is composed using the merge operator. Before that, we explain the notion of merge using record examples. As we will see, records can be encoded using generic functions and the examples hold in our calculi through this encoding. A merge of records, or a join or a concatenation of records, can be defined naturally, and has been studied elaborately in the area of database programming language and foundation of object oriented computing ([OB88], [Wan89], [HP91], [Rem92]). A simple example of a merge is $\{x = 100, y = 50\} \otimes \{color = red\} = \{x = 100, y = 50, color = red\}$. When both records have a same label, merge is defined recursively. For example, $\{name = \{first = Alan\}\}$ and $\{name = \{last = Johnson\}, occupation = teacher,\ age = 30\}$ are two records describing a person, and their merge is $\{name = \{first = Alan,\ last = Johnson\},\ occupation = teacher,\ age = 30\}$. In order to express the type of this merge, we introduce an abbreviation $S \bowtie T$ on types which is called the merge type or the greatest lower bound type of S and T. For the first example, $\{x : int,\ y : int\} \bowtie \{color : string\}$ is defined to be $\{x : int,\ y : int,\ color : string\}$.

We must be careful about this merge operation because not all the expressions are mergeable. There are two cases. One is the merge of values of incompatible types. Examples of such a merge are $\{x = 100, y = 50\} \otimes \{x = \{z = 150\}\}$ and $\{x = 100, y = 50\} \otimes \{x = true, color = red\}$. Since we cannot give any practical meaning to such a merge, we consider them as programming errors, and exclude them using the type system. We introduce, in $\lambda_{\otimes}$ and $\lambda^{*}_{\otimes}$, the type compatibility relation $S \uparrow T$ which means that S and T are mergeable, or equivalently, that S and T have a common subtype. Thus we use the following typing rule:

$$\text{(T-MERGE)} \qquad \frac{e : S \quad e' : S' \quad S \uparrow S'}{e \otimes e' : S \bowtie S'}$$

The other case is the merge of two values of a same basic type. An example of such a merge is $\{x = 100, y = 50\} \otimes \{x = 150, color = red\}$, which conflict on the x-component. There may be two standpoints to this merge. One is to put higher priority to the left hand side, and reduce it to $\{x = 150, y = 50, color = red\}$. $\lambda_{\otimes}$ and $\lambda^{*}_{\otimes}$ adopt this reduction rule. Another is to exclude such expressions from the system. It can be done by introducing a stronger compatibility relation $S \Uparrow S'$, and replace $S \uparrow S'$ with $S \Uparrow S'$ in (T-MERGE). $\lambda^{p}_{\otimes}$ and $\lambda^{p*}_{\otimes}$ adopt this.

We extend this notion of merge to generic functions. First of all, we consider a usual function from S to S' as a generic function applicable to subtypes of S through the coercion function. We write $[S \longrightarrow S']$ for the type of such a generic function. Therefore, a lambda expression $\lambda x^{S}.e$ is a coherent generic function of type $[S \longrightarrow S']$.

Next, the merge $e_1 \otimes e_2$ of generic functions e_1 and e_2 is a generic function. As the type of a generic function, we use a list of the form $[S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$. The meaning of this type is different in $\lambda_{\otimes}$ and $\lambda^{*}_{\otimes}$ and are given in Section 5. We define $S_1 \bowtie S_2$ as the list composed by appending S_1 and S_2, and adopt the (T-MERGE) typing rule to generic functions.

The meaning of this merge is given by the reduction rule. When $e_1 \otimes e_2$ is applied to an argument f of type T, the components applicable to the argument are selected, applied to the argument, and then their results are merged. Therefore, $(e_1 \otimes e_2)\ f$ is reduced to $e_1\ f$ if e_1 is applicable, to $e_2\ f$ if e_2 is applicable, and to $(e_1\ f) \otimes (e_2\ f)$ if both are applicable. This is explained in Figure 1 for the simple case that e_1 and e_2 have the type $[S_1 \longrightarrow S'_1]$ and $[S_2 \longrightarrow S'_2]$, respectively.

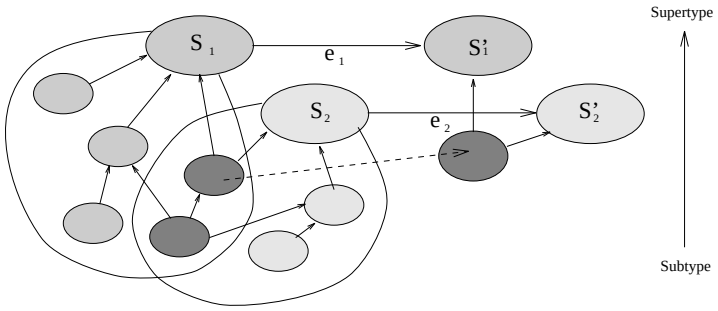


Fig. 1. A Merge of Functions

For this reduction rule to be well defined, we need to ensure that the result type is well formed. Note that two components $S_1 \longrightarrow S'_1$ and $S_2 \longrightarrow S'_2$ are both activated to an argument when S_1 and S_2 have a common subtype, and in which case, we need to ensure that S'_1 and S'_2 have the least upper bound. We put this compatibility condition on the formation of $[S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$.

4 $\lambda_{\otimes}$ and $\lambda^{*}_{\otimes}$: Calculi with a merge operator

We give the formal systems of $\lambda_{\otimes}$ and $\lambda^{*}_{\otimes}$. Suppose that a (possibly infinite) set of basic types (ranged over by the meta variable N) is given. Suppose also

that it includes two types *int* and *bool* to give concrete examples in the following sections.

We first define pretypes (ranged over by S and T) and function pre-components (ranged over by P) as follows:

$$S ::= N \mid [P_1, \ldots, P_n],$$
$$P ::= S \longrightarrow S' \ .$$

We call $[P_1, \ldots, P_n]$ a function (pre)type, which is ranged over by F and G.

We define the following compatibility relation $\uparrow$ on pretypes and function pre-components.

$$\text{(m-REFL)}\ \frac{}{N \uparrow N} \qquad \text{(m-FUN)}\ \frac{(\text{not } S_1 \uparrow S_2) \text{ or } (S_1 \uparrow S_2 \text{ and } S_1' \uparrow S_2')}{S_1 \longrightarrow S_1' \uparrow S_2 \longrightarrow S_2'}$$

$$\text{(m-FLIST)}\ \frac{P_i \uparrow P_j' \ \ (i = 1, \ldots, n,\ j = 1, \ldots, m)}{[P_1, \ldots, P_n] \uparrow [P_1', \ldots, P_m']}$$

Note that $\uparrow$ is well defined though a negation of $\uparrow$ appears in the precondition of (m-FUN); we can define a degree $Deg(S)$ of a pretype S as follows:

$$Deg(N) = 1$$
$$Deg(S \longrightarrow S') = max(Deg(S), Deg(S')) + 1$$
$$Deg([P_1, \ldots, P_n]) = max_{i=1,\ldots,n}(Deg(P_i)) + 1 \ .$$

We define $Deg(S \uparrow S') = max(Deg(S), Deg(S'))$. Then the preconditions of $S \uparrow S'$ have smaller degrees than $S \uparrow S'$ has. Thus, with this stratification, this definition is well defined.

Note that the precondition of (m-FUN) means that if S_1 and S_2 are compatible, then S_1' and S_2' are compatible. This is the condition we gave informally in the previous section if we identify compatibility and the existence of a least upper bound (See Lemma 1 below).

Now we impose the compatibility condition on $[P_1, \ldots, P_n]$ and define types:

- The pretype N is a type.
- if S and S' are types, then $S \longrightarrow S'$ is a function component.
- If $P_1, \ldots, P_n$ are function components and are pairwise compatible (i.e. $P_i \uparrow P_j$ for $i, j = 1, \ldots, n$), then $[P_1, \ldots, P_n]$ is a type.

When $S_1 \uparrow S_2$, we define $S_1 \bowtie S_2$, the merge of S_1 and S_2, as an abbreviation for a type expression as follows:

$$N \bowtie N = N$$
$$[P_1, \ldots, P_m] \bowtie [P_{m+1}, \ldots, P_n] = [P_1, \ldots, P_n].$$

The subtype relation of our calculi is defined as follows:

$$\text{(I-REFL)}\ \frac{}{S \leq S} \qquad \text{(I-FLIST1)}\ \frac{}{[P_1, \ldots, P_n] \leq [P_i]}$$

$$\text{(I-TRAN)}\ \frac{S \leq S' \quad S' \leq S''}{S \leq S''} \qquad \text{(I-FLIST2)}\ \frac{S_1 \uparrow S_2}{[S_1 \longrightarrow S_1', S_2 \longrightarrow S_2'] \leq [S_1 \bowtie S_2 \longrightarrow S_1' \bowtie S_2']}$$

$$\text{(I-FUN)}\ \frac{S \leq T \quad T' \leq S'}{[T \longrightarrow T'] \leq [S \longrightarrow S']} \qquad \text{(I-FLIST3)}\ \frac{[P_1, \ldots, P_n] \leq [P_i'] \ \ (i = 1, \ldots, m)}{[P_1, \ldots, P_n] \leq [P_1', \ldots, P_m']}$$

Note that a basic type is only comparable with itself. For the intuitive meaning of $\leq$, see Proposition 9 below. The relation $\leq$ is a preorder on type expressions. When $S \leq T$, we call S is a subtype of T. When $S \leq T$ and $T \leq S$, we say that S and T are equivalent, and write $S \simeq T$. We can prove the followings:

Lemma 1. *1)* $S_1 \uparrow S_2$ *iff* S_1 *and* S_2 *have a lower bound.*
2) $S_1 \bowtie S_2$ *is the greatest lower bound of* S_1 *and* S_2 *with this preorder.*

Definition 2. Let F be $[S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$.
1) We say that F is applicable to T if at least one of the S_i $(i = 1, \ldots, n)$ satisfy $T \leq S_i$.
2) When F is applicable to T, define $cod(F,T)$ as $S'_{\phi(1)} \bowtie \ldots \bowtie S'_{\phi(l)}$, where $\phi(i)$ satisfies $T \leq S_{\phi(i)}$.
3) When F is applicable to T, define $dom(F,T)$ as $S_{\phi(1)} \bowtie \ldots \bowtie S_{\phi(l)}$, where $\phi(i)$ satisfies $T \leq S_{\phi(i)}$.

As we shall define, a function $e : F$ is applicable to $f : T$ iff F is applicable to T, and $e\ f$ has the type $cod(F,T)$. Let $\mathcal{T}$ be the poset of all types modulo equivalence plus the top element ω[1]. Then we can consider F as defining a partial monotonic function $\mathcal{G}(F)$ from $\mathcal{T}$ to $\mathcal{T}$ which maps $[T]$ to $[cod(F,T)]$ when T is applicable to F, and to ω when it is not. Here, $[T]$ is the equivalence class T belongs to. Our order relation is natural in that it is sound and complete with respect to this interpretation:

Proposition 3. *For two function types* F *and* G, $F \leq G$ *iff* $\mathcal{G}(F)$ *is smaller than* $\mathcal{G}(G)$ *by the pointwise order on the function space from* $\mathcal{T}$ *to itself.*

Since $\leq$ is used in the reduction, we need the decidability of this order relation, which is not trivial because there exists the rule (I-TRANS) which is not structural.

Theorem 4. $T \leq S$ *is decidable.*

Definition 5. For a function type F, $expand(F)$ is the set of all function components of the form $dom(F,T) \longrightarrow cod(F,T)$ for T a type.

Note that $expand(F)$ is a finite set. The decision procedure checks for function types whether all the elements of $expand(S)$ are lower bounded by an element of $expand(T)$. See [Tsu92] for the detail.

Note that our typing system can be considered as a variant of intersection type system, in which intersection is limited to function types and the set of types is consistent ([CC90], [Mon89]).

Now, we move on to expressions. The pseudo expressions of our calculi are defined as follows:

$$e ::= c^S \mid x^S \mid \lambda x^S.\ e \mid e\ f \mid e|_S \mid e_1 \otimes e_2$$

Here, c^S means a constant of type S and x^S means a variable of type S. In order to give concrete examples, we assume that $1, 2, 3 \ldots$ are constants of type int, and $true$ and $false$ are constants of type $bool$. $e|_S$ means the coercion of e to S.

The typing rule of $\lambda_\otimes$ and $\lambda^*_\otimes$ is defined as follows:

[1] Intuitively, ω means type error.

$$\text{(T-CONST)}\ \frac{}{c^S : S} \qquad \text{(T-APP)}\ \frac{e : F \quad f : S \quad F \text{ is applicable to } S}{e\ f : cod(F, S)}$$

$$\text{(T-VAR)}\ \frac{}{x^S : S} \qquad \text{(T-COE)}\ \frac{e : S \quad S \leq S'}{e|_{S'} : S'}$$

$$\text{(T-FUN)}\ \frac{e : S'}{\lambda x^S.\ e : [S \longrightarrow S']} \qquad \text{(T-MERGE)}\ \frac{e : S \quad e' : S' \quad S \uparrow S'}{e \otimes e' : S \bowtie S'}$$

Recall that $S \bowtie S'$ is an abbreviation for a type expression. Therefore, many types have expressions of the form $e \otimes e'$. For example, $1 \otimes 1$ and $1 \otimes 2$ are expressions of type $int \bowtie int = int$, and $\lambda f^{[int \longrightarrow int, bool \longrightarrow bool]}.(\lambda x^{int}.f\ f\ x \otimes \lambda x^{bool}.f\ f\ x)$ is an expression of type $[[int \longrightarrow int, bool \longrightarrow bool] \longrightarrow [int \longrightarrow int, bool \longrightarrow bool]]$. Note also that (T-APP) includes the simple case

$$\text{(T-APP')}\ \frac{e : [S \longrightarrow S'] \quad f : T \quad T \leq S}{e\ f : S'}.$$

$\lambda_\otimes$ and $\lambda^*_\otimes$ only differs in their reduction rules. $\lambda_\otimes$ has the following (E-APP), (E-APPL), (E-CONST), (E-COE) and (E-APPC). On the other hand, $\lambda^*_\otimes$ has the following (E-APP), (E-APPL), (E-CONST), (E-COE) and (E-APPC*).

(E-APP) $(\lambda x^S.\ e)\ f \ \triangleright\ e[x^S := f|_S]$

(E-APPL) $(e_1 \otimes e_2)\ f \ \triangleright$
$$\left\{\begin{array}{l} e_1\ f \quad (F_1 \text{ is applicable to } S) \\ e_2\ f \quad (F_2 \text{ is applicable to } S) \\ e_1\ f \otimes e_2\ f \quad (\text{both } F_1 \text{and } F_2 \text{ are applicable to } S) \end{array}\right\}$$
$(e_1 : F_1, e_2 : F_2 \text{ and } f : S)$

(E-CONST) $n \otimes m \ \triangleright\ m$ (n and m have the same basic type.)

(E-COE) $e|_N \ \triangleright\ e$ (N is a basic type)

(E-APPC) $(e|_F)\ f \ \triangleright\ (e\ f)|_{cod(F,S)}$ (S is the type of f)

(E-APPC*) $(e|_F)\ f \ \triangleright\ (e\ f|_{dom(F,S)})|_{cod(F,S)}$ (S is the type of f)

As we have noted, we need that the type of a term is defined uniquely and is preserved by reductions so that only one branch is selected in the (E-APPL) rule. For that purpose, we insert a coercion on the argument in (E-APP), the β reduction. Intuitively, $e|_S$ is e, but considered as an element of type S. The meaning of $e|_S$ is given through the reduction rules (E-COE), (E-APPC), and (E-APPC*).

(E-APPL) is the rule for an application of a merged function. The typing rule (T-APP) ensures that at least one of the components are applicable. Note that the type of an expression is used at runtime. It is in contrast to the simply typed lambda calculus and its extensions which ignore type information at runtime. It corresponds to what a practical object oriented language like CLOS is doing; find an appropriate method using the class of the argument.

(E-APPC) and (E-APPC*) are the only difference between $\lambda_\otimes$ and $\lambda^*_\otimes$. They correspond to two possible meanings of a function type. In $\lambda_\otimes$, we consider that $F = [S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$ is a set of generic functions which, when applied to a value of type T, returns a value of type $cod(F, T)$. And we consider $e|_F$ as e, but the type of the return value is adjusted to $cod(F, T)$. Thus, when $e|_F$ is applied to f, we apply e to f and coerce the result to $cod(F, T)$.

In contrast, in $\lambda^*_\otimes$, we consider that $G = [S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$ is a type of functions whose behavior is determined by that on the types of the

	Unicity of Type	Subject Reduction	Church Rosser	Strong Normalization	Transitivity of Coercions	Congruence of Equivalent Types
$\lambda_{\otimes}$	○	○	○	×	○	○
$\lambda^*_{\otimes}$	○	○	○	○	×	×

Fig. 2. Properties of Calculi.

form $\{dom(G,T)|T \in \mathcal{T}\}$. Or, in other words, it is a product of monomorphic functions of types $expand(G)$ modulo equivalence. A coercion $e|_F$ is considered as re-constructing a set of monomorphic functions of types $expand(F)$, where the component defined on $dom(F,T)$ is the behavior of e on $dom(F,T)$.

These informal explanations of the meanings of a function in $\lambda_{\otimes}$ and $\lambda^*_{\otimes}$ are justified by the syntactic and semantic properties shown in [Tsu92].

5 Properties of $\lambda_{\otimes}$ and $\lambda^*_{\otimes}$

With this slight difference, the two calculi present different behaviors, and present different syntactic properties, as is listed in Figure 2. Here, Transitivity of Coercions means that $e|_T|_S$ is congruent with $e|_S$ when $T \leq S$, and Congruence of Equivalent Types means that $e|_T$ is congruent with e when $S \simeq T$ and $e : S$, with the congruence relation defined as follows:

Definition 6 [Mil77]. Two closed terms e and f are congruent ($e \approx f$) if in any context $C[]$ of a basic type, $C[e]$ reduces to a normal form c iff $C[f]$ reduces to c.

Note that $e \approx f$ means that e and f are extensionally equal. The results in Fig. 2 are similar to those of $\lambda\&C$ and $\lambda\&C^*$: calculi with overloading studied in [Tsu94]. Therefore, we omit the detailed proofs. In the rest of this section, we give the counterexamples, which show that they do not have good properties.

A non-normalizing term in $\lambda_{\otimes}$ is given as follows. First, note that $(\lambda x^S.x\ x)$ is well-typed for $S = [i \longrightarrow i,\ [i \longrightarrow i] \longrightarrow i]$, since S is a subtype of $[i \longrightarrow i]$ and S is applicable to $[i \longrightarrow i]$. Here, we abbreviate i for int. Now, consider the type

$$T = \begin{bmatrix} i \longrightarrow i, \\ [i \longrightarrow i] \longrightarrow i \\ S \longrightarrow i, \end{bmatrix}.$$

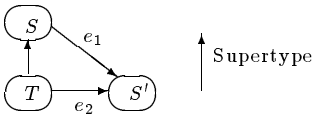
Let $e = (\lambda x^i.x) \otimes (\lambda x^{i \longrightarrow i}.1) \otimes (\lambda x^S.x\ x)$. The term e has the type T. Consider the term $e\ e$. It is type correct and has a type i. It reduces infinitely as follows:

$$\begin{aligned} e\ e &= ((\lambda x^i.x) \otimes (\lambda x^{i \longrightarrow i}.1) \otimes (\lambda x^S.x\ x))\ e \\ &\triangleright (\lambda x^{i \longrightarrow i}.1)\ e \otimes (\lambda x^S.x\ x)\ e \\ &\triangleright^+ (e|_S\ e|_S) \\ &\triangleright (e\ e|_S)|_i \qquad (*) \\ &\triangleright ((\lambda x^{i \longrightarrow i}.1)\ e|_S \otimes (\lambda x^S.x\ x)\ e|_S)|_i \\ &\triangleright^+ e|_S|_S\ e|_S|_S \\ &\ldots. \end{aligned}$$

Here, $\triangleright^+$ means one or more reduction steps. On the other hand, the same term e reduces to 1 in $\lambda^*_\otimes$; the result of $(*)$ step is $(e\ e|_S|_{[i\longrightarrow i]})|_i$ in $\lambda^*_\otimes$, which makes the branch selected in the next reduction different. Note that the same kind of non-normalizing term can be constructed in any type. And we can also define a term $y^S : [[S \longrightarrow S] \longrightarrow S]$ for all S, for which $y\ f$ reduces to $f\ (y\ f)$.

Before giving the next example, we explain how records are encoded in this calculus. Let $L_1, \ldots, L_n, \ldots$ be an infinite set of basic types, and $l_1, \ldots, l_n, \ldots$ be an infinite set of constants with $l_i : L_i$. Then a record type $\{l_1 : T_1, \ldots, l_n : T_n\}$ can now be expressed as $[L_1 \longrightarrow T_1, \ldots, L_n \longrightarrow T_n]$, a record value $\{l_1 = e_1, \ldots, l_n = e_n\}$ can be expressed as $\lambda x^{L_1}.e_1 \otimes \ldots \otimes \lambda x^{L_n}.e_n$, and a record selection $e.l$ is expressed as an application $e\ l$. With this encoding, we have $\{l_1 = e_1, \ldots, l_n = e_n\}.l_i \triangleright^+ e_i$ ([CGL94]).

Using records, we give an example that the Transitivity of Coercions Property does not hold in $\lambda^*_\otimes$. Let $S = \{l_1 : int\}$, $T = \{l_1 : int, l_2 : int\}$, $S' = int$, $e_1 = \lambda x^S.x.l_1$, and $e_2 = \lambda x^T.x.l_2$. Note that $T \leq S$.



Let $f = \{l_1 = 1, l_2 = 2\}$. Then $(e_1 \otimes e_2)|_{[S\longrightarrow S']}|_{[T\longrightarrow S']}\ f$ reduces to $(e_1 \otimes e_2\ f|_T|_S)|_{S'}|_{S'}$ and then to $(e_1\ f|_T|_S)|_{S'}|_{S'}$, and then to 1. On the other hand, $(e_1 \otimes e_2)|_{[T\longrightarrow S']}\ f$ reduces to $(e_1 \otimes e_2\ f|_T)|_{S'}$ and then to $1 \otimes 2$, and then to 2. This means that $(e_1 \otimes e_2)|_{[S\longrightarrow S']}|_{[T\longrightarrow S']} \not\approx (e_1 \otimes e_2)|_{[T\longrightarrow S']}$, and therefore, Transitivity of Coercions do not hold in $\lambda^*_\otimes$. Note that in $\lambda_\otimes$, both expressions reduce to 2.

In the same way, it is easy to show that though $[S \longrightarrow S', T \longrightarrow S']$ is equivalent to $[S \longrightarrow S']$, $e_1 \otimes e_2$ and $(e_1 \otimes e_2)|_{[S\longrightarrow S']}$ are not congruent. Thus, the Congruence of Equivalent Types Property do not hold in $\lambda^*_\otimes$.

6 $\lambda^p_\otimes$ and $\lambda^{p*}_\otimes$: Calculi with Coherent Generic Functions

As we have noted, the unpleasant properties of $\lambda_\otimes$ and $\lambda^*_\otimes$ also appear in $\lambda\&C$ and $\lambda\&C^*$ which are calculi of overloading. This suggests that these properties are caused by "overriding", not by "pure merging". In this section, we define and study restrictions of $\lambda_\otimes$ and $\lambda^*_\otimes$ which corresponds to "pure merging". This is done by excluding an expression like $1 \otimes 2$ using the type system. For this purpose, we introduce a stronger compatibility relation $S_1 \Uparrow S_2$ as follows:

$$\text{(M-FUN)}\quad \frac{(\text{not } S_1 \uparrow S_2) \text{ or } (S_1 \uparrow S_2 \text{ and } S_1' \Uparrow S_2')}{S_1 \longrightarrow S_1' \Uparrow S_2 \longrightarrow S_2'},$$

$$\text{(M-FLIST)}\quad \frac{P_i \Uparrow P_j'(i = 1, \ldots, n,\ j = 1, \ldots, m)}{[P_1, \ldots, P_n] \Uparrow [P_1', \ldots, P_m']}.$$

And modify the (T-MERGE) rule so that it uses this new compatibility relation:

$$\text{(T-MERGE')}\quad \frac{e : S \quad e' : S' \quad S \Uparrow S'}{e \otimes e' : S \bowtie S'}.$$

The modifications of $\lambda_{\otimes}$ and $\lambda^{*}_{\otimes}$are called $\lambda^{p}_{\otimes}$ and $\lambda^{p*}_{\otimes}$, respectively. Note that $\uparrow$ is used in defining $\Uparrow$. Recall that $S_1 \uparrow S_2$ iff a common subtype of S_1 and S_2 exists, and the precondition of (M-FUN) means if S_1 and S_2 have a common subtype, then S_1 and S_2 are mergeable.

Since $int \not\Uparrow int$, $1 \otimes 2$ and $1 \otimes 1$ are not well-typed expressions. Since the redex of (E-CONST) is no longer well-typed, we can discard this rule. We need to show that they are well-defined.

Proposition 7. *$\lambda^{p}_{\otimes}$ and $\lambda^{p*}_{\otimes}$ are well-defined; that is, when e is a well-typed expression and $e \triangleright e'$, then e' is also a well-typed expression.*

$\lambda^{p}_{\otimes}$ and $\lambda^{p*}_{\otimes}$ have good properties. First, they inherit the good properties of $\lambda_{\otimes}$ and $\lambda^{*}_{\otimes}$. From the definition of the congruence relation, $e_1 \approx e_2$ in $\lambda_{\otimes}$ implies $e_1 \approx e_2$ in $\lambda^{p}_{\otimes}$. Therefore, all the properties of $\lambda_{\otimes}$ about congruence also hold in $\lambda^{p}_{\otimes}$. Thus, $\lambda^{p}_{\otimes}$ has the Transitivity of Coercions and Congruence of Equivalent Types properties. As for $\lambda^{p}_{\otimes}$, since we did not modify the reduction rule, Strong Normalization of $\lambda^{p*}_{\otimes}$ follows directly from that of $\lambda^{*}_{\otimes}$.

Next, we show that they have good properties.

Theorem 8. *A function in $\lambda^{p}_{\otimes}$ is a coherent generic function. That is, when $e : F$ is a closed term, $(e\ f|_S) \approx (e\ f)|_{S'}$ for $S' = cod(F, S)$.*

This is proved by induction on the structure of e. As a corollary to this theorem, the following holds.

Corollary 9. *In $\lambda^{p}_{\otimes}$, $e|_F\ f \approx (e\ f|_{dom(F,S)})|_{cod(F,S)}$ when $f : S$.*

This means that $e|_F\ f$ and $(e\ f|_{dom(F,S)})|_{cod(F,S)}$ are extensionally equal in $\lambda^{p}_{\otimes}$, and shows that the congruence relation in $\lambda^{p}_{\otimes}$ is unchanged even if we add the reduction rule (E-APPC*). Using this, we can relate the Normalization property of $\lambda^{p}_{\otimes}$ and $\lambda^{p*}_{\otimes}$.

Theorem 10. *$\lambda^{p}_{\otimes}$ is weak normalizing on basic types. That is, if e has a basic type, there is a reduction which reduces e to a normal form.*

Proof. Since $\lambda^{p*}_{\otimes}$ is strong normalizing, e reduces to a normal form c in $\lambda^{p*}_{\otimes}$. Then from Corollary 9, $e \approx c$ in $\lambda^{p}_{\otimes}$. This means, by definition, that there is a reduction from e to c.

Corollary 11. *When e has a basic type, $e \triangleright^{+} c$ in $\lambda^{p*}_{\otimes}$ iff $e \triangleright^{+} c$ in $\lambda^{p}_{\otimes}$.*

Corollary 12. *For all terms e and f, $e \approx f$ in $\lambda^{p}_{\otimes}$ iff $e \approx f$ in $\lambda^{p*}_{\otimes}$.*

This corollary says that $\lambda^{p}_{\otimes}$ and $\lambda^{p*}_{\otimes}$ have the same extensional term model. In this sense, we can say that $\lambda^{p}_{\otimes}$ and $\lambda^{p*}_{\otimes}$ are equivalent.

Note that though $\lambda^{p}_{\otimes}$ is weak normalizing on basic types, it is not strong normalizing, nor is it weak normalizing on function types. For example, let

$$S = \left[\begin{array}{c} i \longrightarrow \{l_1 : i\}, \\ [i \longrightarrow \{l_1 : i\}] \longrightarrow \{l_3 : \{l_2 : i\}\} \end{array}\right]$$

and consider a term $e = (\lambda x^{i}.\{l_1 = x\}) \otimes (\lambda x^{[i \longrightarrow \{l_1:i\}]}.\{l_3 = \{l_2 = 1\}\}) \otimes (\lambda x^{S}.(x\ x).l_3)$. Then $e\ e$ has the type $\{l_3 : \{l_2 : i\}, l_2 : i\}$. The term $e\ e$ reduces infinitely, and do not have a normal form. However, since $(e\ e).l_2$ and $(e\ e).l_3.l_2$ have a basic type i, Theorem 10 says that they have a Normal Form. Actually, they are reduced to 1. This implies that if we prohibit the reductions inside λ and on both sides of $\otimes$, then the Weak Normalization Property holds on function types.

Through the equivalence of $\lambda^{p}_{\otimes}$ and $\lambda^{p*}_{\otimes}$, $\lambda^{p*}_{\otimes}$ has all the good properties of $\lambda^{p}_{\otimes}$ with respect to congruence such as Transitivity of Coercions, Congruence of Equivalent Types, and Coherence of generic functions. Recall that these properties do not hold in $\lambda^{*}_{\otimes}$.

To conclude, $\lambda^{p*}_{\otimes}$ have the following properties:

Theorem 13. *$\lambda^{p*}_{\otimes}$ have the following properties:*
1) Unicity of Types,
2) Subject Reduction,
3) Church Rosser Property,
4) Strong Normalization,
5) Transitivity of Coercions,
6) Congruence of Equivalent types,
7) All the Generic Functions are Coherent.

It seems that the properties 5), 6), and 7), which are related to congruence relation, cannot be proved easy without referring to the calculus $\lambda^{p}_{\otimes}$. We have used SN of $\lambda^{p*}_{\otimes}$ in this proof, and it seems that this is essential in this proof.

7 Relation to Object Oriented Programming

One of the motivations of the study of calculi with overloading is to give foundation to object oriented programming by interpreting a message send as a call to an overloaded function. ([CGL94] and [Cas93]).

It is recommended that, when programming in an object oriented language, one should write a method in a subclass by modifying those with the same name in superclasses, and one should not change the fundamental behavior of the superclass methods in the modification. For example, when "Bus" is a subclass of "Car", it is misleading to write the method "move" for the class "Bus" as moving a bus backward, while "move" for the class "Car" moves a car forward. In conventional object oriented languages, one can write programs in this style by calling a "super" method explicitly in a subclass method definition and then modifying the result. However, this is only a programming methodology, and one cannot judge whether a program is written in this style or not without following the execution. Moreover, it encounters difficulties when multiple inheritance exists.

Our notion of merge of functions provide this kind of method definition in a declarative way. That is, by writing only the difference from the superclass methods as the function component on a subclass, and by introducing the convention that a subclass function component is placed on the left of a superclass function component, a subclass component works as adding some behaviors which are peculiar to the subclass, and overriding some part of the results of superclass methods. Recall the way merge works on records in Section 3.

The restrictions on $\lambda^{p}_{\otimes}$ and $\lambda^{p*}_{\otimes}$ correspond to allowing only addition of some behaviors in a subclass method definition. The properties of the calculi we have

proposed are connected to the higher order treatment of generic functions and, in this sense, we cannot link them directly to the properties of object oriented languages. However, it is interesting to note that the fact that $\lambda^{p}_{\otimes}$ and $\lambda^{p*}_{\otimes}$ have good properties conforms with the programming methodology in which one should write a subclass method so that it does not override the behavior of superclass methods.

One of the characteristics of object oriented languages is late binding. Since our calculi do not allow late binding, they do not reflect "all" the features of object oriented languages; they only aimed at defining a subtype method as a modification of supertype methods. The overloading calculus $\lambda\&$-calculus has the facility to write late binding by allowing the type of a term to change in reduction. However, we cannot add late binding to our calculi in this way because compatibility is not preserved by reduction if we allow this. One solution to this is to restrict type inheritance to record extensions, and extend the mergeability constraints on records in [HP91] to function merges. This is left as a future work.

Acknowledgement The author thanks Benjamin Pierce, Masami Hagiya, Daisuke Suzuki, and Sinya Nishizaki for stimulating discussions and comments.

References

[Cas93] G. Castagna. A meta-language for typed object-oriented languages. In *13th Conference on the Foundations of Software Technology and Theoretical Computer Science*, number 761 in LNCS, Bombay, India, 1993.

[CC90] F. Cardone and M. Coppo. Two extensions of curry's type inference system. In *Logic and Computer Science*, pp. 19–66. Academic Press, 1990.

[CGL94] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, Vol. to appear,, 1994.

[Has91] R. Hasegawa. Parametricity of extensionally collapsed term models of polymorphism and their categorical properties. In *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pp. 495–512, 1991.

[HP91] R. Harper and B. Pierce. A record calculus based on symmetric concatenation. In *Proc. The Eighteenth Annual Symp. on Principles of Programming Languages*, pp. 131–142, 1991.

[Mil77] R. Milner. Fully abstract models of typed λ-calculus. *Theoret. Comput. Sci*, Vol. 4, pp. 1–22, 1977.

[Mon89] Gennaro Monteleone. Generalized conjunctive types. In *Proc. The Sixteenth Annual Symposium on Principles of Programming Languages*, pp. 242–249. ACM, 1989.

[OB88] A. Ohori and P. Buneman. Type inference in a database programming language. In *Proc. ACM Conf. on LISP and Functional Programming*, pp. 174–183, 1988.

[Rem92] Didier Remy. Typing record concatenation for free. In *Proc. The Nineteenth Annual Symposium on Principles of Programming Languages*, pp. 166–176. ACM, 1992.

[Rey81] J. C. Reynolds. Using category theory to design implicit conversions and generic operators. In *Lecture Notes in Computer Science, 94*. Springer-Verlag, 1981.

[Rey83] J. C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pp. 513–523. North-Holland, 1983.

[Rey88] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.

[Tsu92] Hideki Tsuiki. *A Record Calculus with a Merge Operator*. PhD thesis, Keio University, 1992.

[Tsu94] Hideki Tsuiki. A normalizing calculus with overloading and subtyping. In *2'nd Int. Symp. on Theoretical Aspects of Computer Software, LNCS 789*, pp. 273–295, 1994.

[Wan89] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Proc. Fourth IEEE Annual Symp. on Logic in Computer Science*, pp. 92–97, 1989.