Supporting Explicit Disambiguation of Multi-methods

Eric Amiel[1] and Eric Dujardin[2]

[1] NatSoft, Air Center,
1214 Genève, Suisse
E-mail:amiel@rodin.inria.fr
[2] INRIA, Projet RODIN, BP 105,
78153 Le Chesnay Cedex, France
E-mail:eric.dujardin@inria.fr

Abstract. Multiple inheritance and multiple dispatching are two sources of ambiguities in object-oriented languages. Solving ambiguities can be performed automatically, using techniques such as totally ordering the supertypes of each type or taking the order of the methods' arguments into account. Such implicit disambiguation has the drawback of being difficult to understand by the programmer and hiding programming errors. Conversely, solving ambiguities can be left up to the explicit intervention of the programmer. The most common explicit disambiguation technique in existing systems consists in defining new methods for ambiguous invocations. However, finding ambiguities and adding as few methods as possible is a difficult task, especially in multi-method systems. In this paper, we propose a tool to help the programmer solve multi-methods ambiguities. We show that there always exists a unique minimal set of method redefinitions that explicitly disambiguate a set of multi-methods. The tool offers two modes: batch, directly yielding the disambiguation set, or interactive, one signature of the disambiguation set at a time, allowing the programmer to either accept the method redefinition or to solve the ambiguity in any other way, which restarts the disambiguation algorithm. In both modes, for each method that is to be added, the programmer is given the set of methods that caused the ambiguity as an explanation.

Keywords: method dispatch, multi-methods, multiple inheritance, ambiguities.

1 Introduction

Ambiguities are a well-known problem of any classification system supporting multiple inheritance. They plague semantic networks in artificial intelligence as well as class hierarchies in many object-oriented languages. Multiple inheritance ambiguities occur when it is impossible to decide from which superclass to inherit a property, i.e. an instance variable or a method. Consider a class hierarchy where class *TeachingAssistant (TA)* is a subtype[3] of two superclasses *Student* and *Employee* and has one subclass *ForeignTA*. Assume that both

---

[3] In this paper, we indifferently use *subtyping* and *inheritance*, *class* and *type*, although we are primarily interested in the *typing* aspects.

Student and *Employee* define a method *vacation*[4] which computes the number of days of vacation. Then, any invocation of *vacation* on a *TA* or *ForeignTA* is ambiguous, as it is impossible to know which method must be called, *Student*'s or *Employee*'s *vacation*.

Multi-methods add another kind of ambiguity. Indeed, run-time method selection looks for the method whose arguments most closely match those of the invocation. Ambiguities may arise if two methods most closely match different subsets of an invocation's arguments. Consider the above class hierarchy with two multi-methods $m_1(TA, Student)$ and $m_2(Student, TA)$ and invocation $m(aTA, aTA)$. With respect to the first argument, m_1 is a closer match than m_2, while the reverse holds for the second argument. Thus, the invocation $m(aTA, aTA)$ is ambiguous, as are all invocations whose arguments are of class *TA* or *ForeignTA*.

There are two ways to eliminate ambiguities: *implicit* and *explicit* disambiguation. Implicit disambiguation consists in *automatically* solving ambiguities *in the place* of the programmer. For example, CLOS defines a total order on all the superclasses of each class to eliminate multiple inheritance ambiguities [BDG+88]. In the above example, if *Student* precedes *Employee* in *TA*'s definition, then invoking *vacation* on a *TA* results in the invocation of the *vacation* method of *Student*. Implicit disambiguation of multi-methods ambiguities is based on taking the order of the arguments into account: in this way, $m_1(TA, Student)$ is a closer match than $m_2(Student, TA)$, because its first argument is more specific than m_2's.

Explicit disambiguation, used in languages like C++ [ES92], Eiffel [Mey92] and Cecil [Cha93], consists in requiring *the programmer* to solve ambiguities. One way of achieving this consists in *redefining* the method for ambiguous invocations. For example, if the programmer redefines *vacation* for *TA*, invoking *vacation* on a *TA* is no longer ambiguous. Note that this redefinition also solves ambiguities for *ForeignTA*. In the same way, defining a method $m_3(TA, TA)$ solves the multiple dispatching ambiguity between m_1 and m_2 for all invocations with arguments of class *TA* or *ForeignTA*.

Implicit disambiguation is increasingly being criticized for mainly two reasons: first, the way it solves ambiguities can be difficult to understand and counter-intuitive in some cases. This is particularly obvious for multiple dispatching ambiguities where the order of the arguments is taken into account. Second, ambiguities can actually reveal programming errors, which implicit disambiguation hides.

On the other hand, explicit disambiguation imposes some burden on the programmer who faces two problems: first, (s)he must find which methods are ambiguous and with respect to which class(es) of argument(s). Second, (s)he must determine which methods must be added. Indeed, if carefully chosen, very few method redefinitions can solve all ambiguities at the same time. However, adding a method to solve an ambiguity may sometimes result

---

[4] For the rest of the paper we consider ambiguous methods as they capture the case of instance variables through encapsulation.

in the creation of a new ambiguity. Unfortunately, to our knowledge, no system assists the programmer in the task of explicit disambiguation. Such help is especially needed for multi-method systems, notably because multi-methods are more complex to master than mono-methods and suffer from two kinds of ambiguities, increasing the potential number of ambiguities.

In this paper, we address this need by proposing a tool to help the programmer solve multi-methods ambiguities. We show that there always exists a unique minimal set of method redefinitions that explicitly disambiguate a set of multi-methods. The tool offers two modes: batch, directly yielding the disambiguation set, or interactive, one signature of the disambiguation set at a time, allowing the programmer to either accept the method redefinition or to solve the ambiguity in any other way, which restarts the disambiguation algorithm. In both modes, for each method that is to be added, the programmer is given the set of methods that caused the ambiguity as an explanation. In our example, the tool outputs *vacation(TA)*[5] as the method that must be added and { *vacation(Student), vacation(Employee)* } as the explanation.

The paper is organized as follows. Section 2 surveys previous work on ambiguities. Section 3 defines the problem we address and gives an overview of our solution. Section 4 presents our disambiguation algorithm. Section 5 deals with implementation issues, notably optimization and complexity. We conclude with future work in section 6.

2   Background on Disambiguation

2.1   Basic Definitions

In traditional object-oriented systems, methods have a single specially designated argument – called the *receiver* or *target* – whose run-time type is used to select the method to execute at run-time. Such methods are called *mono-methods*. *Multi-methods*, first introduced in CommonLoops [BKK+86] and CLOS [BDG+88], generalize mono-methods by considering that all arguments are targets. Multi-methods are now a key feature of several systems such as Kea [MHH91], Cecil [Cha92], Polyglot [DCL+93], and Dylan [App95]. Following [ADL91], we denote *subtyping* by $\preceq$. Given two types T_1 and T_2, if $T_1 \preceq T_2$, we say that T_1 is a subtype of T_2 and T_2 is a supertype of T_1.

A generic function is defined by its name and its arity (in Smalltalk parlance, a generic function is called a *selector*). To each generic function m of arity n corresponds a set of methods $m_k(T_k^1, \ldots, T_k^n) \rightarrow R_k$, where T_k^i is the type of the ith formal argument, and where R_k is the type of the result. We call the list of argument types $(T_k^1, \ldots, T_k^n)$ of method m_k the *signature* of m_k[6]. An invocation of a generic function m is denoted $m(T^1, ..., T^n)$,

---

[5] For the rest of the paper, we use the functional notation as we consider multi-method systems.

[6] For our purposes, we do not include the return type in the signature.

where $(T^1, \ldots, T^n)$ is the signature of the invocation, and the T^i's represent the types of the expressions passed as arguments. Finally, we call *MSA method* for *Most Specific Applicable* method, the method selected at run-time for some invocation.

2.2 Method Ordering and Ambiguity

The basis of method specificity is a precedence relationship called *argument subtype precedence* in [ADL91]: a method m_i is more specific than a method m_j, noted $m_i \prec m_j$, if all the arguments of m_i are subtypes of the arguments of m_j. However, in the presence of multiple inheritance or multiple dispatching, argument subtype precedence may be unable to totally order applicable methods for some invocations, yielding several *conflicting* MSA methods. Such invocations are then *ambiguous.*

$vacation_1(Student)$ *Student*   *Employee* $m_1(Student,TA)$
$vacation_2(Employee)$ TA $m_2(TA,Student)$

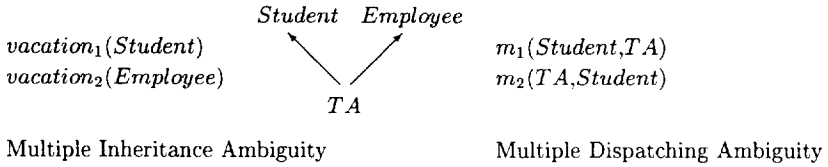Multiple Inheritance Ambiguity   Multiple Dispatching Ambiguity

Fig. 1.: Ambiguities

Example 1. Consider the type hierarchy and methods of Figure 1. Argument subtype precedence can order neither $vacation_2$ w.r.t. $vacation_1$ (multiple inheritance ambiguity), nor m_2 w.r.t. m_1 (multiple dispatching ambiguity). Thus, invocations $vacation(aTA, aTA)$ and $m(aTA, aTA)$ are ambiguous.

2.3 Disambiguation Techniques

As noted in [Cha92], "the key distinguishing characteristic of method lookup (...) is how exactly ambiguities are resolved". Techniques to solve ambiguities can be classified in two categories: *implicit* and *explicit* disambiguation.

Implicit Disambiguation Implicit disambiguation consists in augmenting the power of argument subtype precedence to automatically resolve ambiguities. To solve multiple inheritance ambiguities, the subtype relationship, $\preceq$, is complemented by a precedence relationship, α, that strictly orders all the supertypes of a type. The supertypes ordering is generally local to each type (local supertypes precedence) as in Loops [SB86], CommonLoops [BKK+86], Flavors [Moo86], Orion [BKKK87] and CLOS [BDG+88]. To solve multiple dispatching ambiguities, the formal arguments of the rival methods are examined in some given order, e.g. left-to-right, and the comparison stops as soon as an argument of one method strictly precedes the corresponding argument of the other method. [ADL91] extensively covers the different ways to augment argument subtype precedence to avoid ambiguities.

Example 2. Assume local supertype precedence establishes that *Student α Employee* and left-to-right argument examination is chosen. Then, the ambiguities of Figure 1 are resolved and $vacation_1$ has precedence over $vacation_2$ and m_2 has precedence over m_1.

Note that dispatch based on a local supertypes precedence ordering of methods may select a method in an counter-intuitive way. Indeed, CLOS, Loops and Dylan do not support *monotonicity* [DHHM92]. Monotonicity captures the intuitive property that, if a method is applicable to, but is not the most specific applicable method for some signature, it cannot be the MSA of a more specific signature. To address the anomalies created by local supertypes precedence, [DHHM94] proposes a *monotonous* supertypes linearization algorithm. However, in some inheritance hierarchies there are classes for which no monotonous linearization exist.

C++ [ES92] implicitly solves some multiple inheritance ambiguities by using the static type of the receiver object: the inheritance path between the corresponding class, and the class of the receiver at run-time, takes precedence.

Example 3. Consider the example in Figure 2 and a C++ invocation b->$m()$, where b is a variable of type $B*$. If b points to an instance of class E at run-time, then m_2 takes precedence. However, invocation e->$m()$, where e is of type $E*$, is still ambiguous.

$$A$$
$$B$$
$$C$$
$$D \quad F$$
$$E$$
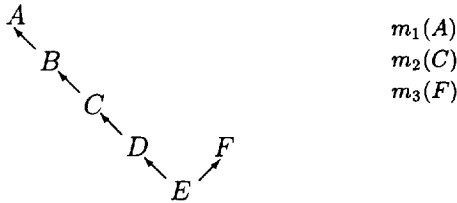
$$m_1(A)$$
$$m_2(C)$$
$$m_3(F)$$

Fig. 2.: Example Type Hierarchy and Methods

This disambiguation scheme also appears in [CG90] with the *points of view*, in Fibonacci [ABGO93] with its *roles*, in O_2 [$O_2$92] and in Self 2.0 with its *sender path tiebraker rule* [CUCH91]. However, these techniques cannot resolve all ambiguities (e.g. the invocation e->$m()$ above), and they go against the need to "ensure that the same function is called for an object independently of the expression used to access the object", as stated in [ES92].

As argued in [LR89], [Sny86b], [DH89] and [Cha92], implicitly solving ambiguities raises several serious problems. First, ambiguities may be the result of programming errors. Implicit disambiguation prevents the detection of such errors. Second, it makes programs hard to understand, maintain and evolve. This is particularly obvious for multiple dispatching ambiguities where the order of the arguments is taken into account. Third, they cannot resolve all ambiguities, except the algorithms of Loops and CLOS, which trade this for counter-intuitive selections. Finally, there are ambiguities that implicit disambiguation cannot resolve according to the programmer's wish, because it is not fine enough.

Example 4. Consider the type hierarchy and methods of Figure 3. Assume the programmer would like $vacation_1(Student)$ to be the MSA method for ambiguous invocation $vacation$ (aTA) and $taxes_2(Employee)$ to be the MSA method for ambiguous invocation $taxes(aTA)$. Such disambiguation cannot be automatically performed by ordering supertypes *Student* and *Employee*.

$$Student \quad Employee$$

$vacation_1(Student)$ ⟍ ⟋ $taxes_1(Student)$
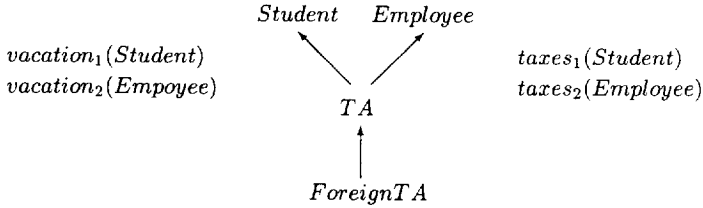$vacation_2(Empoyee)$ $\quad TA \quad$ $taxes_2(Employee)$

$$ForeignTA$$

Fig. 3.: Example Type Hierarchy and Methods

Explicit Disambiguation The second way of solving ambiguities is *explicit disambiguation*. In this approach, the programmer him/herself solves ambiguities at the level of either *invocations* or *methods*.

Explicit Disambiguation at the Invocation Level

In C++, multiple inheritance ambiguities can be resolved on a per invocation basis and in two different ways. First, the programmer can explicitly force a particular method to be the MSA method for some invocation by prefixing the invocation by the name of a class followed by the *scoping* operator "::". The MSA method for the invocation is then *statically* determined to be the MSA method for that class, bypassing late binding.

Example 5. Consider the type hierarchy of Figure 2. The C++ invocation $e->m()$, where e is of type $E*$, is ambiguous. The programmer can resolve the ambiguity by writing $e->B::m()$. This statically binds the invocation to the method applicable to class B, namely the method defined in class A.

Type casting, i.e. type coercion, is the second way of explicitly resolving ambiguities at the invocation level in C++. Contrary to the scoping operator, type casting preserves late binding.

Example 6. Consider again the type hierarchy of Figure 2. The programmer can resolve the ambiguous invocation $e->m()$ by writing $((B*)e)->m()$, making use of the implicit disambiguation rule described above. This forces e to be considered as referring to the B part of an E object. Late binding is preserved: the method that actually gets executed is the one defined in class C.

Explicit disambiguation at the invocation level provides the finest control over ambiguities. However, it imposes a heavy burden on the programmer who must disambiguate every

ambiguous invocation. Moreover, the scoping operator suspends late binding. This can be dangerous when the type hierarchy or the methods evolve:

Example 7. Consider again the type hierarchy of Figure 2. If the programmer disambiguates invocation $e->m()$ by writing $e->C :: m()$ and then, a new method is defined in class D, then the disambiguation must be rewritten $e->D :: m()$.

Explicit Disambiguation at the Method Level

A programmer can perform explicit disambiguation of methods by either *selecting* or *adding* methods. Method selection consists in explicitly declaring which of the conflicting methods takes precedence *for all invocations.* It is supported in Traits [CBLL82], Trellis [SCB+86] and CommonObject [Sny86a]. Eiffel [Mey92] performs method selection by either *renaming* all conflicting methods but one and using a "select" statement or *undefining* all conflicting methods but one. O₂ [O₂92] automatically performs renaming of conflicting methods by prefixing them with the name of the class.

The second way of performing explicit disambiguation at the method level consists in *adding* new methods so that argument subtype precedence is sufficient to totally order applicable methods for any invocation. The augmented set of methods then satisfies a condition, described in [LR89], and called *regularity* in [MGS89] and Zelig [DS92], and *consistency* in Cecil [Cha93]. This disambiguation policy is used in Extended Smalltalk [BI82], Zelig [DS92], Self 3.0 [ABC+93], Laure [CP93] and Cecil [Cha93].

Example 8. Consider again the type hierarchy and methods of Figure 1. To eliminate ambiguities, it is enough to define two new methods: $vacation_3(TA)$ and $m_3(TA, TA)$.

The new methods may perform specific code written by the programer or deduced from the code of the conflicting methods as in Laure [CP93]. These new methods may also just serve the purpose of resolving an ambiguity, by explicitly calling another method of the same generic function using a scoping operator like Cecil's "@@" or C++'s "::", or a special construct like "*call-method*" in CommonObject.

Example 9. Cecil [Cha92] has the *resend* construct to explicitly call another method of the same generic function. Given the type hierarchy and methods of Figure 1, $vacation_3(TA)$ can resolve the ambiguity in favor of $vacation_1(Student)$ as follows:

```
method vacation(c1@@TA) { resend(c1@@Student) }
```

As shown in Example 7, it is dangerous to suspend late binding like that. It would not be suspended if the semantics of the declaration above was to invoke the MSA method of the invocation $vacation(Student)$.

Methods selection declarations must be taken into account in the late binding mechanism, making it more complex. Explicit disambiguation by addition of methods encompasses the functionality of explicit disambiguation by selection without making it necessary to incorporate the selection declarations in the late binding mechanism.

2.4 Conclusion

From some recent language updates, it appears that language designers increasingly favor explicit disambiguation, because of the problems associated with implicit disambiguation. For example, Self 3.0 [Se393] has abandoned *prioritized inheritance*, a kind of local super-types precedence, together with the sender path tiebraker implicit disambiguation rule. The priority mechanism is described as being "of limited use, and had the potential for obscure errors". Cecil does not include implicit disambiguation either. In Dylan, the linear ordering of supertypes is similar to CLOS's, but Dylan does not assume any order on the multi-method's arguments, leaving room for multiple dispatching ambiguities and requiring explicit disambiguation.

Besides, even in languages with implicit disambiguation, solving ambiguities explicitly is useful in cases where implicit disambiguation leads to violate monotonicity.

3 Problem Statement and Overview of the Solution

The problem with explicit disambiguation is the burden it imposes on the programmer who faces two problems: first, (s)he must find which methods are ambiguous for which signature(s). Second, (s)he must determine which methods must be added to solve the ambiguities. An obvious solution is to define a method for each and every ambiguous signature. However, this results in the creation of a potentially huge number of disambiguating methods, whereas carefully choosing for which signatures to redefine methods can solve several or even all ambiguities at the same time. Consider the type hierarchy and methods of Figure 4. Signatures $(D,G), (D,I), (K,G), (K,I)$ are ambiguous because of methods $m_1(A,G)$ and $m_2(B,F)$. However, defining a method $m_4(D,G)$ is enough to solve these four ambiguities.
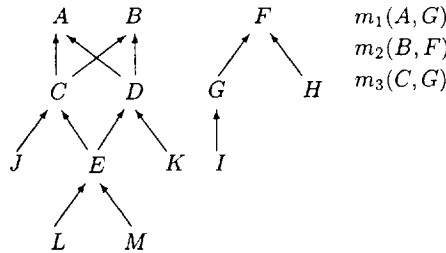

$$m_1(A,G)$$
$$m_2(B,F)$$
$$m_3(C,G)$$

Fig. 4.: Reference Type Hierarchy and Methods

Finding the minimal set of disambiguating methods is further complicated by the fact that adding a method to solve some ambiguity can actually result in the creation of a new ambiguity. Indeed, in the type hierarchy of Figure 4, invocation $m(E,G)$ is initially not ambiguous, with $m_3(C,G)$ as its most specific applicable method. However, the addition of method $m_4(D,G)$ to solve the $m(D,G)$ ambiguity makes $m(E,G)$ ambiguous, as m_3 now conflicts with m_4.

Unfortunately, to our knowledge, no system assists the programmer in the task of explicit disambiguation by method addition. Such help is especially needed for multi-methods systems, notably because multi-methods are more complex to master than mono-methods and suffer from two kinds of ambiguities, increasing the potential number of ambiguities.

To help explicitly disambiguate multi-method systems, we propose to provide the programmer with a disambiguation tool that can be integrated into the interpreter, compiler or programming environment. This tool takes as input the signatures of a generic function's methods. In batch mode, its output is the minimal set of signatures of the methods that must be added in order to eliminate ambiguities. The programmer must then provide bodies to the methods of the disambiguation set, be they just resend clauses. In interactive mode, it yields one signature of the disambiguation set at a time, in a specific order: the programmer can either accept to solve the ambiguity by the addition of a method with that signature, or stop the disambiguation algorithm and solve the ambiguity in another way (e.g. by deleting methods or changing the type hierarchy). The algorithm is then restarted from the beginning. The interactive mode gives the programmer finer control over the disambiguation process, as (s)he is not forced to solve all ambiguities through method addition. In both modes, for each new method's signature, the tool presents the set of methods that created the ambiguity as an explanation to the programmer. In this paper, we focus on the batch mode for space reason.

Our results apply to all languages in which the precedence ordering of multi-methods conforms to argument subtype precedence and monotonicity. This includes Cecil and Dylan, as well as CLOS and SQL3 provided the latter two slightly change their precedence order to adopt a monotonic order, as proposed in [DHHM94][7]. Our results also apply to languages with mono-methods, such as C++, Self or Laure.

Our disambiguation algorithm is based on two results: (i) the minimal disambiguation set is unique and (ii) it is included in a set of signatures called the *pole signatures*. The algorithm is composed of two steps. The first step consists in computing the pole signatures, using the signatures of the initial set of methods as follows. Multiple inheritance ambiguities are explicitly solved for each argument position, i.e. the set of types appearing at a given position is augmented with the minimal set of types needed to eliminate multiple inheritance ambiguities. This process yields a set of *pole types* or *poles* for each argument position. The set of pole signatures is the Cartesian product of the sets of poles at each argument position.

Example 10. Given the methods of Figure 4, the set of poles on the first argument position is the union of $\{A, B, C\}$ and $\{D, E\}$. $\{A, B, C\}$ is the set of types appearing in the first argument position, and $\{D, E\}$ is the minimal disambiguation set. The poles of the second argument position are only the types appearing in the second argument position, $\{F, G\}$, as this

---

[7] Such a change was recently adopted for Dylan

set is not ambiguous. The pole signatures are then $\{(A, F), (B, F), (C, F), (D, F), (E, F), (A, G), (B, G), (C, G), (D, G), (E, G)\}$.

The second step of the disambiguation algorithm is the following: for each pole signature, the algorithm computes the MSA method of the corresponding invocation. If there are more than one MSA method, then the invocation is ambiguous and a method with that signature must be added to the initial set of methods with the signatures of the conflicting MSA methods for explanation. In order to minimize the number of methods to add and to detect ambiguities created by the addition of a method, the algorithm processes the pole signatures in a total order that is compatible with argument subtype precedence, from the most general to the most specific signatures.

Example 11. The pole signatures given above are already ordered in a way that is compatible with argument subtype precedence from the most general to the most specific signature. The first pole signature for which there is more than one MSA is (D, G) with $m_1(A, G)$ and $m_2(B, F)$ as conflicting MSA methods. A first method $m_4(D, G)$ is thus added to the disambiguation set with $m_1(A, G)$ and $m_2(B, F)$ as explanation. The next and last signature, (E, G), also has more than one MSA method, namely $m_3(C, G)$ and the newly added $m_4(D, G)$. Hence, method $m_5(E, G)$ is added to the disambiguation set with $m_3(C, G)$ and $m_4(D, G)$ as explanation and the algorithm ends.

Notice that the algorithm tests for ambiguity a number of signatures that is much smaller than the total number of well-typed invocations. In the example of Figure 4, there are 34 different well-typed invocations and the algorithm only needs to test 7 signatures (the signatures of the three methods m_1, m_2, and m_3 can be skipped as they are obviously not ambiguous).

4 Disambiguation Algorithm

Before presenting the disambiguation algorithm, we first give some definitions and the theoretical result on which the algorithm is based.

4.1 Definitions

For the rest of this paper, we call Θ the set of existing types, and we consider a generic function m of arity n, whose methods are $m_1, \ldots, m_p$. We also consider a set $S = \{s_1, \ldots, s_p\}$ such that for all k, s_k is the signature of m_k, i.e. $s_k = (T_k^1, \ldots, T_k^n)$.

We first review some notations and results introduced in [AGS94]. Here is the formal definition of a *pole* type that is used to solve multiple inheritance ambiguities at each argument position:

I-Pole. A type $T \in \Theta$ is an i-pole of generic function m, $i \in \{1, \ldots, n\}$, denoted $is\_pole_m^i(T)$, iff:

$$\exists k \in \{1, \ldots, p\} \text{ s.t. } T = T_k^i$$

$$\text{or} \quad |min_{\preceq} \{T' \in \Theta \mid is\_pole_m^i(T') \text{ and } T' \succ T\}| > 1$$

The first part of the conjunction corresponds to the types appearing at the i-th argument position (*primary poles*), while the second part defines which types must be added to solve multiple inheritance ambiguities (*secondary poles*). The set of i-poles of m is denoted $Pole_m^i = \{T \mid is\_pole_m^i(T)\}$. The set of *pole signatures* is denoted $Poles_m = \prod_{i=1}^{n} Pole_m^i$.

Example 12. Going back to Figure 4, D is a 1-pole because of the ambiguity created by the 1-poles A and B. In the same way, E is a 1-pole because of C and D.

Signature Specificity. A signature $s = (T^1, \ldots, T^n) \in \Theta^n$ is more specific than a signature $s' = (T'^1, \ldots, T'^n) \in \Theta^n$, noted $s \preceq s'$, iff for all i in $\{1, \ldots, n\}$, $T^i \preceq T'^i$.

By analogy with methods, if $s \preceq s'$, s' is said to be applicable to s.

Method Precedence Order. We note by α_s the precedence order of method signatures with respect to signature s: $s_1 \alpha_s s_2$ means that s_1 and s_2 are the signatures of two methods applicable to s (i.e., $s \preceq s_1$ and $s \preceq s_2$), and s_1 is more specific than s_2 for s. This order is used to determine the MSA method. We assume that α conforms to argument subtype precedence, and that it is monotonic:

$$s_1 \preceq s_2 \text{ and } s \preceq s_1, \ s \preceq s_2 \ \Rightarrow \ s_1 \ \alpha_s \ s_2 \quad \text{(argument subtype precedence)}$$
$$s_1 \ \alpha_s \ s_2 \text{ and } s' \prec s \ \Rightarrow \ s_1 \ \alpha_{s'} \ s_2 \quad \text{(monotonicity)}$$

Conflicting Signatures. Given a set of signatures $S' \subseteq \Theta^n$, the set of conflicting signatures of S' w.r.t. a signature $s \in \Theta$, noted $conflicting(s, S')$, is defined as follows:

$$conflicting(s, S') = min_{\alpha_s}\{s' \in S' \mid s' \succeq s\}$$

If S' is a set of method signatures and s the signature of an invocation, then $conflicting(s, S')$ represents the signatures of the most specific applicable methods for the invocation. This is a generalization of the notion of MSA method that takes ambiguities into account. It is used both to test a signature for ambiguity and to determine the origin of the ambiguity as an explanation.

Ambiguity of a Signature Set, Ambiguous Signature. A set of signatures $S' \subseteq \Theta^n$ is ambiguous iff there exists a signature $s \in \Theta$ such that $|conflicting(s, S')| > 1$. s is then said to be an ambiguous signature w.r.t. S'.

Disambiguation Set. Given a set of signatures $S' \subseteq \Theta^n$, $\Delta \subseteq \Theta^n$ is a disambiguation set of S' iff $S' \cup \Delta$ is not ambiguous.

4.2 Main Theorem

We assume the existence of a total order on the pole signatures $Poles_m$, denoted by $\leq$, and compatible with argument subtype precedence, i.e. $\forall s, s' \in Poles_m, (s \preceq s' \Rightarrow s \leq s')$. Such an order always exists and can be found using a topological sort [Knu73].

We then define a sequence of signatures $(s_k)_{k>p}$ in the following way:

- $s_{p+1} = max_\leq \{s \in Poles_m \mid |conflicting(s, \{s_1, \ldots, s_p\})| > 1\}$.
- $\forall k > p, s_{k+1} = max_\leq \{s \in Poles_m \mid s_k > s \text{ and } |conflicting(s, \{s_1, \ldots, s_k\})| > 1\}$.

As each signature s_k is found by applying $max_\leq$, building $(s_k)_{k>p}$ from this definition is achieved by going over $Poles_m$ in the order of $\leq$, starting from the most generic signatures. Note that testing the ambiguity of a signature at stage $k+1$ is done using *all* preceding signatures, not just the first p ones.

Theorem 1. $\Delta_{min}^S = \{s_k \mid p < k\}$ *is finite, and is the minimal disambiguation set of S.*

Proof: see Appendix.

Example 13. Consider again the types and methods in Figure 4. The original signature set is $S = \{(A, G), (B, F), (C, G)\}$, and $\Delta_{min}^S = \{(D, G), (E, G)\}$.

4.3 Main Algorithm

As explained in Section 3, the disambiguation algorithm takes place in two steps: first, the poles of every argument position are computed to yield the pole signatures in an order compatible with argument subtype precedence, then the minimal disambiguation set is computed by iterating over the set of pole signatures. The algorithm in Figure 5 invokes a subroutine that builds the ordered list of pole signatures, and then performs the second step of this process. We describe the ordering of pole signatures in the next section, and pole computation is isomorphic to the second step of the disambiguation algorithm. Indeed, computing the poles amounts to determining the minimal disambiguation set of the types appearing as arguments: the types of the hierarchy are iterated over in an order compatible with argument subtype precedence; for each type, the most specific applicable poles are computed. If there are more than one most specific applicable pole, then the type becomes a secondary pole. Note that, as for the second step of the main algorithm, the order in which types are considered guarantees the *minimality* of the disambiguation set.

From the definition of (s_k), it is straightforward to build an algorithm that produces Δ_{min}^S by going over $Poles_m$ in the order of $\leq$ and adding ambiguous signatures to the original set of signatures to test following signatures. Moreover, the set of conflicting signatures is associated with each ambiguous signatures as an explanation.

Disambiguation algorithm
input: a set of methods M, a boolean *interactive*
output: a list *result* of 2-tuples (disambiguation signature,conflicting signatures)

Step 1: Computation of the Ordered Pole Signatures
$S \leftarrow$ signatures(M) ;   // method signatures
$P \leftarrow$ OrderedPoleSignatures(S) ;

Step 2: Computation of Δ_{min}^{S} with explanations
$\Delta \leftarrow \emptyset$;   // disambiguation signatures
$result \leftarrow \emptyset$;

for s in P do
   $CONF \leftarrow conflicting(s, S \cup \Delta)$;
   if $|CONF| > 1$ then   // s is ambiguous in $S \cup \Delta$
     if *interactive* then
       if *user-refuse*$(s, CONF)$ then // ask the user if (s)he wants to stop
         exit   // stop execution
     insert s into Δ ;
     add $(s, CONF)$ to *result* ;

return($result$) ;

Fig. 5.: Disambiguation Algorithm

The algorithm assumes the existence of two subroutines: OrderedPoleSignatures(S) returns the list of pole signatures in an order that is compatible with argument subtype precedence and $conflicting(s, S)$ returns the signatures in S that conflict as most specific applicable to s.

Example 14. Back to Figure 4, let us assume that *OrderedPoleSignatures*(S) returns $((A, F), (B, F), (C, F), (D, F), (E, F), (A, G), (B, G), (C, G), (D, G), (E, G))$. First, (D, G) is found to be ambiguous as it has (A, G) and (B, F) as conflicting signatures. (D, G) is thus added to Δ and $((D, G), \{(A, G), (B, F)\})$ to *result*. Then, (E, G) is found to be ambiguous, with (C, G) and (D, G) as conflicting signatures. (E, G) is thus added to Δ and $((E, G), \{(C, G), (D, G)\})$ to *result*, which is then returned as the minimal disambiguation set with associated explanations.

Finally, in testing pole signatures for ambiguity, the disambiguation algorithm can also fill the dispatch table of the generic function, presented in [AGS94]. Indeed, the dispatch table stores the MSA method of all pole signatures: if *conflicting* yields a singleton set, then the single element is the signature of the MSA method.

5   Implementation And Complexity

This section describes the ordering of poles in *OrderedPoleSignature*, and the computation of conflicting signatures in *conflicting*.

5.1   Ordering the Pole Signatures

Ordering the pole signatures in an order that is compatible with argument subtype prece-
dence comes down to turning a partially ordered set into a linear list. A classical algorithm
is given in [Knu73]. The basic idea is to pick as first element one that has no predecessor,
remove this element from the original set to append it to the originally empty list, and start
over until no elements are left. In the case of pole signatures, it is necessary to scan the set
of pole signatures to find that a given signature has no predecessor. Hence, ordering the pole
signatures has a complexity of $O(|Poles_m|^2)$.

However, it is possible to obtain a complexity of $O(|Poles_m|)$ if the poles of each argu-
ment position, $Pole_m^i$, are themselves sorted in an order compatible with argument subtype
precedence. Indeed, it is easy to show that it suffices to produce the signatures in the lexi-
cographic ordering generated by the total orders on the poles[8].

Example 15. The table in Figure 6 represents the pole signatures of the methods and types
of Figure 4. The order on 1-poles (resp. 2-poles) in lines (resp. columns) is compatible with
argument subtype precedence. A total order of $Poles_m$ is a path through this table. Such a
path is compatible with argument subtype precedence, if it traverses each signature s before
the signatures on the right and below s. The path given by a lexicographic ordering, as
shown in Figure 6 satisfies the condition. For example, the signatures that are more specific
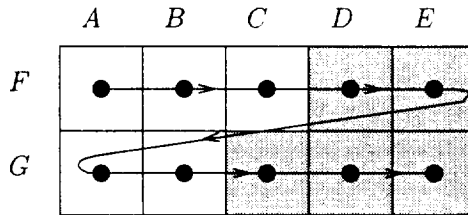than (C, F) are all included in the grayed area.

Fig. 6.: Order of Pole Signatures

5.2   Computing the Conflicting Signatures

The computation of the conflicting signatures consists in finding the most specific applicable
signatures. In the case of a totally ordered set, there is a single smallest element, and the cost
to find it is linear in the number of elements of the set, if the comparison of two elements is
done in a constant time. Unfortunately, signatures are only partially ordered, increasing the
complexity to the square of the number of signatures to compare. As $S \cup \Delta_{min}^S$ is a superset

---

[8] This ordering of poles is not expensive as poles are produced in this order, from an ordering of
types that is done once for all pole computations.

of the set of applicable signatures, the worst-case complexity of *conflicting* is $O(|S \cup \Delta|^2)$ signature comparisons[9]. However, this complexity can be lowered when there is no ambiguity, i.e. there is a single most specific applicable signature.

The basic idea of our optimization is to store $S \cup \Delta$ in the total order of $\leq$. To compute $conflicting(s, S \cup \Delta)$, the signatures in $S \cup \Delta$ are examined in the order $\leq$ starting from the smallest, i.e. most specific signatures. If there is a single MSA s_k, then it is the first signature s_k applicable to s. Moreover, a single loop over the ordered $S \cup \Delta$ is enough to prove it, bringing the complexity down to $O(|S \cup \Delta|)$. Indeed, if there is a single MSA method s_k, we have:

$$\forall s_{k'} \in S \cup \Delta, \ s_{k'} \succeq s \Rightarrow s_{k'} \succeq s_k \Rightarrow s_{k'} \geq s_k.$$

On the other hand, if the iteration finds another applicable signature that is not more generic than s_k, then s is ambiguous and the complexity is $O(|S \cup \Delta|^2)$. Hence the complexity of disambiguating m is $O(|S \cup \Delta|^2 \times |\Delta| + |S \cup \Delta| \times (|Poles_m - \Delta|))$ comparisons of signatures.

Example 16. Assume a new method $m_4(C, I)$ is added to the schema of Figure 4, so that I is now a 2-pole. Assume that (D, G) and (E, G) have already been found to be ambiguous and added to Δ. Signature (D, I) must now be tested for ambiguity. Using the lexicographic order represented in Figure 6 to sort $S \cup \Delta$ yields $((C, I), (E, G), (D, G), (C, G), (A, G), (B, F))$. The first signature applicable to (D, I) is (D, G) and no following signatures is more generic than it. Hence (D, G) is not ambiguous. On the other hand, when (E, I) is tested for ambiguity, (C, I), then (E, G) are found to be applicable to (E, I), and $(E, G) \not\succ (C, I)$. Hence (E, I) is ambiguous $(conflicting((E, I)) = \{(E, G), (C, I)\})$.

6   Future Work

Three issues are worth future investigation: impossible signatures, mixing method addition with method selection and incremental disambiguation.

Impossible signatures We call *impossible signatures*, signatures that can never occur as invocation signatures of a given generic function. They may be signatures containing an abstract class, that is, a class that cannot have instances, or they may be forbidden signatures as described in [ABDS94]. These signatures are user-defined and their use as invocation signatures are type errors.

In case an impossible signature appears in a disambiguation set, defining the body of a method with this signature may have no sense. Hence for some impossible signatures, the

---

[9] This comparison is done in constant time, if n is constant and the technique of [Cas93] is used to keep the transitive closure of the subtyping relationship. The latter computation is done once for all generic functions.

programmer may want to define methods for the signatures that are directly more specific, and not impossible. Our algorithm should be extended to ignore those signatures and rather examine the signatures that are directly more specific.

Mixing Method Addition and Method Selection Another interesting research direction consists in studying how disambiguation by method selection can be mixed with disambiguation by method addition. Indeed, whenever the disambiguation algorithm finds the first ambiguous signature s, it could pause and ask the programmer to choose between two alternatives: add a new method with signature s or add a method selection clause, saying "select method m_i for signature s", where m_i is one of the conflicting methods. This is especially relevant in the case where the new method only serves as a forwarder to one of the conflicting methods. The algorithm would then look for the next ambiguous signature. Interestingly, choosing method selection instead of method addition is not indifferent: adding a method selection clause can actually solve more ambiguities and lead to a smaller set of ambiguous signatures *depending on which conflicting method is chosen.*
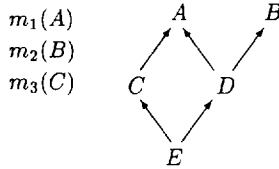
$$m_1(A)$$
$$m_2(B)$$
$$m_3(C)$$

Fig. 7.: Conflicting Method Selection

Example 17. Consider the schema in Figure 7. In the case of disambiguation by method addition, two disambiguation methods $m(D)$ and $m(E)$ must be added. The conflicting signatures for (E) are (C) and (D). Assume that instead, a selection clause is added when finding ambiguous signature (D). If the user selects $m_1(A)$ for (D), $m_2(B)$ cannot be selected for $m(E)$ because of the monotonicity property [DHHM92]. The applicable methods for $m(E)$ are then $m_1(A)$ and $m_3(C)$, and the latter being more specific than the former, $m(E)$ is not ambiguous as in the case of method addition. Note that (E) is ambiguous if the programmer selects $m_2(B)$ for (D).

Incremental Disambiguation As the type hierarchy and the methods may evolve, especially during application development, it is interesting to investigate if it is possible to compute the minimal disambiguation set based on the evolution operations performed on the type hierarchy and methods.

7 Conclusion

In this paper, we addressed the problem of supporting programmers in the task of explicitly disambiguating multi-methods by method addition. This process involves finding a set of

disambiguating methods as small as possible. We proved that there always exists a single minimal disambiguation set, and proposed an algorithm to compute it. This algorithm is efficient in that it avoids testing all possible invocations for ambiguity, examining instead a much smaller set of signatures, the *pole signatures*. Moreover, this algorithm associates to each disambiguating method the set of conflicting signatures that caused the ambiguity. This provides explanations and allows implementation of a disambiguating method as a forwarder to one of the conflicting methods.

Future work involves taking impossible signatures into account. Moreover, the algorithm can be extended to allow explicit disambiguation by mixing method addition with method selection. Finally, we are interested in studying the relationship between the minimal disambiguation set and evolution operations on the type hierarchy and methods.

8   Appendix : Proof of Theorem 1

We first introduce the following definition:

Well-typed signatures. $Well\text{-}Typed(m) = \{s \in \Theta^n \mid \exists k \in \{1, \ldots, p\} \text{ s.t. } s \preceq s_k\}$

To prove Theorem 1, we start from a slightly different definition of the sequence (s_k), as follows:

- $s_{p+1} = max_\leq\{s \in Well\text{-}Typed_m \mid |conflicting(s, \{s_1, \ldots, s_p\})| > 1\}$
- $\forall k > p, s_{k+1} = max_\leq\{s \in Well\text{-}Typed_m \mid s_k > s \text{ and } |conflicting(s, \{s_1, \ldots, s_k\})| > 1\}$

In this definition, we take $\{s_k \mid p < k\}$ in $Well\text{-}Typed_m$ instead of $Poles_m$. We use this definition to prove that (s_k) is finite and that it is the smallest disambiguation set. Then we prove that it is included in $Poles_m$, which shows that both definitions of (s_k) are equivalent.

A   Finiteness

To prove that $\{s_k \mid k \in \mathbb{N}\}$ is finite, it suffices to remark that it is included in Θ^n, which is finite. For the rest of the paper, q the index of the last element of the sequence (s_k).

B   Lemma

This lemma basically expresses that the way Δ^S_{min} is built "does not leave ambiguous signatures behind":

Lemma 2. $\forall k \in \{p + 1, \ldots, q\}, \nexists s \in Well\text{-}Typed_m \text{ s.t. } s \text{ is ambiguous w.r.t. } \{s_1, \ldots, s_k\}$ *and* $s \geq s_k$.

Proof: Let $k > p$ and $s \in Well\text{-}Typed_m$. We assume that s is ambiguous w.r.t. $\{s_1, \ldots, s_k\}$ and $s \geq s_k$. In the following, we prove that $s \in \{s_1, \ldots, s_k\}$, to conclude that the assumption is false.

We have $|conflicting(s, \{s_1, \ldots, s_k\})| > 1$, i.e. $|min_{\alpha_s}\{s_j \mid j \leq k \text{ and } s_j \succeq s\}| > 1$. Obviously the members of this minimum are strictly more generic than s, hence $|min_{\alpha_s}\{s_j \mid j \leq k$ and $s_j \succ s\}| > 1$. Let us consider $\{s_j \mid p < j \leq k$ and $s_j > s\}$. Two cases may occur:

- $\{s_j \mid p < j \leq k$ and $s_j > s\} = \emptyset$, which implies that $s \geq s_{p+1}$.

  We show that s is ambiguous w.r.t. S. $\leq$ being compatible with $\preceq$, $\nexists j$ s.t. $p < j \leq k$ and $s_j \succ s$. Hence the members of $conflicting(s, \{s_1, \ldots, s_k\})$ are in S, i.e. s is ambiguous w.r.t. S.

  From $s \geq s_{p+1}$ and the construction of s_{p+1}, it follows that $s = s_{p+1}$.

- $\{s_j \mid p < j \leq k$ and $s_j > s\} \neq \emptyset$: let $k' = max_\leq\{j \mid p < j \leq k$ and $s_j > s\}$. $k' \neq q$ since $s_{k'} > s \geq s_k \geq s_q$. Thus $s_{k'+1}$ exists, and $s_{k'} > s \geq s_{k'+1}$.

  We show that s is ambiguous w.r.t. $\{s_1, \ldots, s_{k'}\}$. $\leq$ being compatible with $\preceq$, $\nexists j$ s.t. $k' < j$ and $s_j \succ s$. Hence the members of $conflicting(s, \{s_1, \ldots, s_k\})$ are in $\{s_1, \ldots, s_{k'}\}$, i.e. s is ambiguous w.r.t. $\{s_1, \ldots, s_{k'}\}$.

  By construction of $s_{k'+1}$, it follows that $s = s_{k'+1}$.

Hence $s \in \{s_1, \ldots, s_k\}$, which implies that s in not ambiguous w.r.t. this set, in contradiction with the assumption. This concludes the proof of Lemma 2.

C $\quad \Delta^S_{min}$ is a Disambiguation Set

To prove that Δ^S_{min} is a disambiguation set, let us assume the existence of $s \in Well\text{-}Typed_m$ ambiguous w.r.t. $S \cup \Delta^S_{min}$. We show that $s \leq s_q$, and then we apply Lemma 2 to conclude that the assumption is false.

If $s_q > s$ we have $\{s' \in Well\text{-}Typed_m \mid s_q > s'$ and $|conflicting(s', \{s_1, \ldots, s_q\})| > 1\} \neq \emptyset$, i.e. s_{q+1} exists, in contradiction with the construction of q. Thus $s \geq s_q$.

As a consequence, s is ambiguous w.r.t. $\{s_1, \ldots, s_q\}$ and $s \geq s_q$, in contradiction with Lemma 2. Hence $S \cup \Delta^S_{min}$ is not ambiguous, and Δ^S_{min} is a disambiguation set.

D $\quad \Delta^S_{min}$ is the Smallest Disambiguation Set

We prove that Δ^S_{min} is included in every disambiguation set.

Let $\{s^d_{p+1}, \ldots, s^d_r\}$ be a disambiguation set. For convenience, we note that for all k, $k \leq p$, $s^d_k = s_k$. Thus, $\{s^d_1, \ldots, s^d_r\}$ is not ambiguous. We prove by induction on k, that $\forall k \in \{p+1, \ldots, q\}$, $\{s_1, \ldots, s_k\} \subset \{s^d_1, \ldots, s^d_r\}$. This induction hypothesis obviously holds for $k = 1, \ldots, p$ by construction of $(s^d_k)_{k \in \{1, \ldots, r\}}$.

Assuming it is true for some k such that $p \leq k < q$, we take j such that $\{s_j^d\} = conflicting(s_{k+1}, \{s_1^d, \ldots, s_r^d\})$. s_j^d is unique because $\{s_1^d, \ldots, s_r^d\}$ is not ambiguous. Let us show that $s_j^d = s_{k+1}$. We first show that $s_j^d \neq s_{k+1} \Rightarrow s_j^d$ is ambiguous w.r.t. $\{s_1, \ldots, s_{k+1}\}$, then we apply Lemma 2.

Let $A = \{s_i \mid i \leq k \text{ and } s_i \succeq s_{k+1}\}$. We have $min_{\alpha_{s_{k+1}}} A = conflicting(s_{k+1}, \{s_j \mid j \leq k\})$. From the induction hypothesis, $A \subset \{s_1^d, \ldots, s_r^d\}$, hence the elements of this set are more generic than s_j^d. This means that $A \subset \{s_i \mid i \leq k \text{ and } s_i \succeq s_j^d\}$. Reciprocally, $s_i \succeq s_j^d \Rightarrow s_i \succeq s_{k+1}$, thus $A = \{s_i \mid i \leq k \text{ and } s_i \succeq s_j^d\}$.

Assuming s_j^d were not ambiguous w.r.t. $\{s_1, \ldots, s_k\}$, let $\{s_a\} = conflicting(s_j^d, \{s_1, \ldots, s_k\}) = min_{\alpha_{s_j^d}} A$. As α is monotonous, for s_{k+1} also s_a would be more specific than the other elements of A, i.e. we would have $\{s_a\} = conflicting(s_{k+1}, \{s_1, \ldots, s_k\})$, which is impossible by construction of s_{k+1}. Hence s_j^d is ambiguous w.r.t. $\{s_1, \ldots, s_k\}$.

Let us assume that $s_j^d \succ s_{k+1}$: s_{k+1} is not applicable to s_j^d, hence s_j^d is ambiguous w.r.t. $\{s_1, \ldots, s_{k+1}\}$. As $s_j^d \geq s_{k+1}$, we apply Lemma 2 to show that this assumption is false.

Finally $s_j^d \succeq s_{k+1}$ and $s_j^d \not\succ s_{k+1}$ implies $s_j^d = s_{k+1}$, thus $\{s_1, \ldots, s_{k+1}\} \subset \{s_1^d, \ldots, s_r^d\}$, which concludes our proof.

E Δ_{min}^S is Made of Poles

We first recall some results mainly introduced in [AGS94].

Influence of a Pole. For all i in $\{1, \ldots, n\}$, and all T_π in $Pole_m^i$,

$$Influence_m^i(T_\pi) = \{T \in \Theta \mid T \preceq T_\pi \text{ and } \forall T_\pi' \in Pole_m^i, T \not\preceq T_\pi' \text{ or } T_\pi \preceq T_\pi'\}.$$

Ith Dynamic Argument. $Dynamic_m^i = \{T \in \Theta \mid \exists k \in \{1, \ldots, p\} \text{ s.t. } T \preceq T_k^i\}$

Note that $\forall (T^1, \ldots, T^n) \in Well\text{-}Typed_m, \forall i \in \{1, \ldots, n\}, T^i \in Dynamic_m^i$.

Theorem 3. Let $\{T_\pi^1, \ldots, T_\pi^l\}$ be $Pole_m^i$. Then $\{Influence_m^i(T_\pi^1), \ldots, Influence_m^i(T_\pi^l)\}$ is a partition of $Dynamic_m^i$.

The latter theorem allows to define the following functions:

Pole of a Type, Poles of a Signature. For all i in $\{1, \ldots, n\}$, all T in $Dynamic_m^i$, and all $(T^1, \ldots, T^n)$ in $Well\text{-}Typed_m$, we define:

$$pole_m^i(T) = T_\pi, \text{ s.t. } T_\pi \in Pole_m^i \text{ and } T \in influence_m^i(T_\pi)$$
$$pole_m((T^1, \ldots, T^n)) = (pole_m^1(T^1), \ldots, pole_m^n(T^n))$$

We also introduce the following lemma:

Lemma 4. $\forall s \in Well\text{-}Typed_m, \forall s' \in Poles_m, s' \succeq s \Rightarrow s' \succeq pole_m(s)$.

Proof: Let $s = (U^1, \ldots, U^n) \in Well\text{-}Typed_m$, $s_\pi = pole_m(s) = (U^1_\pi, \ldots, U^n_\pi)$, and $s' \in Poles_m$, $s' = (T^1, \ldots, T^n)$. Let us show that $\forall i \in \{1, \ldots, n\}$, $T^i \succeq pole^i_m(U_i)$.

Let $i \in \{1, \ldots, n\}$. We have $T^i \succeq U^i$ because $s' \succeq s$.

From the definition of $Influence^i_m$, as $U^i \in Influence^i_m(U^i_\pi)$, $T^i \in Pole^i_m$, and $U^i \preceq T^i$, we have $U_\pi \preceq T^i$. Thus $s_\pi \preceq s'$, which concludes the proof of Lemma 4.

To show that $\Delta^S_{min} \subset Poles_m$, we prove by induction that $\forall k \geq p, \{s_1, \ldots, s_k\} \subset Poles_m$. This is true for $k = p$ from the definitions of an i-pole and of $Poles_m$, because $s_1, \ldots, s_p$ are the signatures of $m_1, \ldots, m_p$.

Assuming the induction hypothesis holds for some k such that $p \leq k < q$, we show that $s_{k+1} \in Poles_m$, by proving that $s_{k+1} = pole_m(s_{k+1})$.

Let s'_{k+1} be $pole_m(s_{k+1})$. We show that $s'_{k+1} \succ s_{k+1} \Rightarrow s'_{k+1}$ is ambiguous w.r.t. $\{s_1, \ldots, s_{k+1}\}$), then we apply Lemma 2.

Let $A = \{s_i \mid i \leq k$ and $s_i \succeq s_{k+1}\}$. We have $min_{\alpha_{s_{k+1}}} A = conflicting(s_{k+1}, \{s_j \mid j \leq k\})$. From the induction hypothesis, $A \subset Poles_m$, hence the elements of this set are more generic than s'_{k+1}. This means that $A \subset \{s_i \mid i \leq k$ and $s_i \succeq s'_{k+1}\}$. Reciprocally, $s_i \succeq s'_{k+1} \Rightarrow s_i \succeq s_{k+1}$, thus $A = \{s_i \mid i \leq k$ and $s_i \succeq s'_{k+1}\}$.

Assuming s'_{k+1} were not ambiguous w.r.t. $\{s_1, \ldots, s_k\}$, let $\{s_a\} = conflicting(s'_{k+1}, \{s_1, \ldots, s_k\}) = min_{\alpha_{s'_{k+1}}} A$. As α is monotonous, for s_{k+1} also s_a would be more specific than the other elements of A, i.e. we would have $\{s_a\} = conflicting(s_{k+1}, \{s_1, \ldots, s_k\})$, which is impossible by construction of s_{k+1}. Hence s'_{k+1} is ambiguous w.r.t. $\{s_1, \ldots, s_k\}$.

Let us assume that $s'_{k+1} \succ s_{k+1}$: s_{k+1} is not applicable to s'_{k+1}, hence s'_{k+1} is ambiguous w.r.t. $\{s_1, \ldots, s_{k+1}\}$. Moreover $s'_{k+1} \geq s_{k+1}$, hence we can apply Lemma 2 to show that this assumption is false.

From $s'_{k+1} \succeq s_{k+1}$ and $s'_{k+1} \nsucc s_{k+1}$, it follows that $s'_{k+1} = s_{k+1}$, thus $\{s_1, \ldots, s_{k+1}\} \subset Poles_m$. This concludes our proof.

References

[ABC+93] O. Agesen, L. Bak, C. Chambers, B.-W. Chang, U. Hölzle, J. Maloney, R. B. Smith, D. Ungar, and M. Wolcsko. *The Self 3.0 Programmer's Reference Manual.* Sun Microsystems and Stanford University, 1993. Available by ftp from self.stanford.edu as /pub/Self-3.0/manuals/progRef.ps.gz.

[ABDS94] E. Amiel, M.-J. Bellosta, E. Dujardin, and E. Simon. Supporting exceptions to schema consistency to ease schema evolution. In *Proc. Intl. Conf. on VLDB*, 1994.

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Proc. Intl. Conf. on Very Large Data Bases*, 1993.

[ADL91]   R. Agrawal, L. G. DeMichiel, and B. G. Lindsay. Static type checking of multi-methods. In *Proc. OOPSLA*, 1991.

[AGS94]   E. Amiel, O. Gruber, and E. Simon. Optimizing multi-methods dispatch using compressed dispatch tables. In *Proc. OOPSLA*, 1994.

[App95]   Apple Computer. *Dylan Reference Manual, draft* , September 1995. Available from the Dylan Home Page on the World-Wide Web at address http://www.cambridge.apple.com/dylan/dylan.html.

[BDG+88]   D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System specification. *SIGPLAN Notices*, 23, Sept. 1988.

[BI82]   A. H. Borning and D. H. H. Ingalls. Multiple inheritance in Smalltalk-80. In *Proc. AAAI*, 1982.

[BKK+86]   D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. Common-Loops: Merging Lisp and object-oriented programming. In *Proc. OOPSLA*, 1986.

[BKKK87]   J. Banerjee, W. Kim, H.J. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proc. ACM SIGMOD Intl. Conf. on Management Of Data*, 1987.

[Cas93]   Yves Caseau. Efficient handling of multiple inheritance hierarchies. In *Proc. OOPSLA*, 1993.

[CBLL82]   G. Curry, L. Baer, D. Lipkie, and B. Lee. Traits : An approach to multiple-inheritance subclassing. In *Proc. ACM SIGOA Conference on Office Automation Systems*, 1982.

[CG90]   B. Carré and J.-M. Geib. The point of view notion for multiple inheritance. In *Proc. ECOOP/OOPSLA*, 1990.

[Cha92]   C. Chambers. Object-oriented multi-methods in Cecil. In *Proc. ECOOP*, 1992.

[Cha93]   C. Chambers. The Cecil language, specification and rationale. Technical Report 93-03-05, Dept of Computer Science and Engineering, FR-35, University of Washington, March 1993.

[CP93]   Yves Caseau and Laurent Perron. Attaching second-order types to methods in an object-oriented language. In *Proc. ECOOP*, 1993.

[CUCH91]   C. Chambers, D. Ungar, B.-W. Chang, and U. Hoelzle. Parents are shared parts of objects : inheritance and encapsulation in SELF. *Lisp and Symbolic Computation*, 4(3), 1991.

[DCL+93]   L. G. DeMichiel, D. D. Chamberlin, B. G. Lindsay, R. Agrawal, and M. Arya. Polyglot: Extensions to relational databases for sharable types and functions in a multi-language environment. In *Proc. Intl. Conf. on Data Engineering*, 1993.

[DH89]   R. Ducournau and M. Habib. La multiplicité de l'héritage dans les langages à objets. *Technique et Science Informatiques*, January 1989.

[DHHM92]   R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *Proc. OOPSLA*, 1992.

[DHHM94]   R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Proposal for a monotonic multiple inheritance linearization. In *Proc. OOPSLA*, 1994.

[DS92]   S. Danforth and E. Simon. *The Next Generation of Information Systems - from Data to Knowledge*, chapter A Data and Operation Model for Advanced Database Systems. Springer Verlag, 1992.

[ES92]   M. A. Ellis and B. Stroustrup. *The annotated C++ reference manual*. Addison-Wesley, Reading, Mass., 1992.

[Knu73]   D. Knuth. *The Art of Computer Programming, Fundamental Algorithms, Second Edition*. Addison-Wesley, 1973.

[LR89]   C. Lecluse and P. Richard. Manipulation of structured values in object-oriented databases. In *Proc. Intl. Workshop on Database Programming Languages*, 1989.

[Mey92]   B. Meyer. *EIFFEL : The Language*. Prentice Hall Intl., 1992.

[MGS89]   José Meseguer, Joseph Goguen, and Gert Smolka. Order-sorted unification. *Journal of Symbolic Computation*, 8:383–413, 1989.

[MHH91]   W. B. Mugridge, J. Hamer, and J. G. Hosking. Multi-methods in a statically-typed programming language. In *Proc. ECOOP*, 1991.

[Moo86]   D. A. Moon. Object-oriented programming with Flavors. In *Proc. OOPSLA*, 1986.

[$O_2$92] O_2 Technology. *The O_2 User's Manual*, 1992.

[SB86]   M. Stefik and D. G. Bobrow. Object-oriented programming: Themes and variations. *The AI Magazine*, 6(4), 1986.

[SCB+86]   C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpot. An introduction to Trellis/Owl. In *Proc. OOPSLA*, 1986.

[Se393]   Self 3.0 - about this release. Available by ftp from self.stanford.edu as /pub/Self3.0/manuals/aboutThisRelease.ps.gz, 1993.

[Sny86a]   A. Snyder. CommonObjects : An overview. *Sigplan Notices*, 21(10), 1986.

[Sny86b]   A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proc. OOPSLA*, 1986.