Supporting Explicit Disambiguation of Multi-Methods

Eric Amiel, Eric Dujardin

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Supporting Explicit Disambiguation of Multi-Methods

Eric Amiel, Eric Dujardin

N ° 2590

Juin 1995

PROGRAMME 1

Architectures parallèles,

bases de données,

réseaux et systèmes distribués

1995

ISSN 0249-6399

Supporting Explicit Disambiguation of Multi-Methods

Eric Amiel*, Eric Dujardin*

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Rodin

Rapport de recherche n2590 — Juin 1995 — 23 pages

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 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 report, we show that there always exists a unique minimal set of method redefinitions to explicitly disambiguate a set of multi-methods. We propose an algorithm to compute the minimal disambiguation set, together with *explanations*: for each method that is to be added, the programmer is given the set of methods that caused the ambiguity.

Key-words: method dispatch, multi-methods, multiple inheritance, ambiguities,topological sort

(Résumé : tsvp)

*amiel@rodin.inria.fr, Eric.Dujardin@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Un outil pour supporter la désambiguïsation explicite des multi-méthodes

Résumé : L'héritage multiple et la sélection multiple représentent deux sources d'ambiguïté dans les langages orientés-objets. La résolution des ambiguïtés peut se faire automatiquement, en utilisant des techniques basées sur un ordre total entre les supertypes, et l'ordre des arguments des multi-méthodes. Ces désambiguïsations implicites ont l'inconvénient d'être difficile à comprendre pour le programmeur, et de cacher des erreurs de programmation. À l'inverse, la résolution des ambiguïtés peut être laissée à la charge du programmeur. La technique de désambiguïsation la plus courante est de définir de nouvelles méthodes pour les invocations ambigües. Cependant, il est difficile de trouver les ambiguïtés et d'ajouter un nombre minimal de méthodes, notamment dans les systèmes supportant les multi-méthodes. Dans cet article, nous montrons qu'il y a toujours un unique plus petit ensemble de redéfinitions de méthodes permettant de désambiguïser explicitement un ensemble de multi-méthodes. Nous proposons un algorithme pour calculer le plus petit ensemble de désambiguïsation, ainsi que des *explications* : à chaque méthode à définir est associé l'ensemble des méthodes qui ont causé l'ambiguïté.

Mots-clé : sélection des méthodes, multi-méthodes, héritage multiple, ambiguïtés, tri topologique.

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[1] of two superclasses *Student* and *Employee* and has one subclass *ForeignTA*. Assume that both *Student* and *Employee* define a method *vacation*[2] 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 [DHHM92]. 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

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

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

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 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 report, we address this need by showing that there always exists a unique minimal set of method redefinitions to explicitly disambiguate a set of multi-methods. We propose an algorithm which computes the minimal disambiguation set and provides *explanations*: for each method that is to be added, the programmer is given the set of methods that caused the ambiguity. In our example, the algorithm outputs *vacation(TA)*[3] as the method that must be added and $\{$*vacation(Student)*, *vacation(Employee)*$\}$ as the explanation.

The report 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 most specific applicable method to execute. 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 [App94]. 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

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

of methods $m_k(T_k^1, \ldots, T_k^n) \rightarrow R_k$, where T_k^i is the type of the i^{th} 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[4]. An invocation of a generic function m is denoted $m(T^1, ..., T^n)$, 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, the method selected at run-time for some invocation is called the *Most Specific Applicable* (*MSA*) method.

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*.

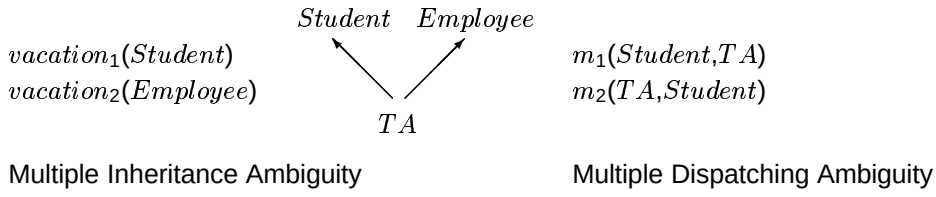

Figure 1: Ambiguities

Example 2.1 : 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.

2.3.1 Implicit Disambiguation

Implicit disambiguation consists in augmenting the power of argument subtype precedence to automatically resolve ambiguities. To solve multiple inheritance ambiguities, the subtype

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

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 2.2 : Assume local supertype precedence establishes that $Student\ \alpha\ Employee$ 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 unintuitive way. Indeed, CLOS, Loops and Dylan do not support *monotonicity* [DHHM94]. Monotonicity captures the intuitive property that, if a method is not the MSA 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.

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 2.3 : 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. □

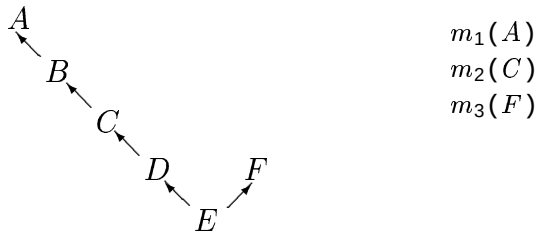

Figure 2: Example Type Hierarchy and Methods

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 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. Finally, there are ambiguities that implicit disambiguation cannot resolve according to the programmer's wish, because it is not fine enough.

Example 2.4 : 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 $Employee$. □

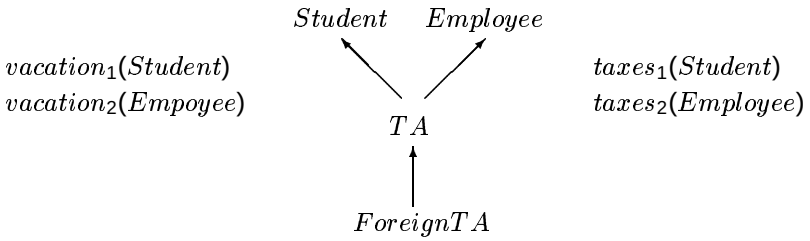

Figure 3: Example Type Hierarchy and Methods

2.3.2 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 2.5 : 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 2.6 : 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 2.7 : 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. O_2 [$O_2$92] 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 Zelig [DS92] and *consistency* in Cecil [Cha93]. This disambiguation policy is used in Extended Smalltalk [BI82], Zelig [DS92], Self 3.0 [ABC+93], and Cecil [Cha93].

Example 2.8 : 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 or 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 2.9 : 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) }
```
□

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 supertypes 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. Dylan borrows CLOS's linear ordering of supertypes, but does not assume any order on the multi-method's arguments, leaving room for multiple dispatching ambiguities and requiring explicit disambiguation.

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.

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

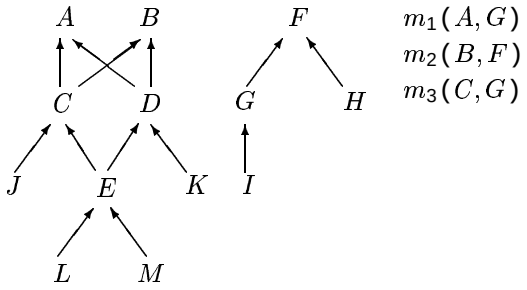

Figure 4: Reference Type Hierarchy and Methods

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. Its output is the minimal set of signatures of the methods that must be added in order to eliminate ambiguities. Moreover, for each new method's signature, the tool outputs the set of methods that created the ambiguity as an explanation to the programmer.

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 3.1 : 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 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 3.2 : 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 new method $m_4(D, G)$ is thus added 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, a new method $m_5(E, G)$ is added 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 report, 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:

$$\begin{array}{ll} & \exists k \in \{1, \ldots, p\} \text{ s.t. } T\text{=}T_k^i \\ \text{or} & |min_{\preceq} \{T' \in \Theta \mid is\_pole_m^i(T') \text{ and } T' \succ T\}| > 1 \end{array}$$

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 4.1 : 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.

We assume the existence of a total order on $Poles_m$, 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].

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:

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

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

Assuming that the pole signatures have been computed, we define a sequence of signatures $(s_k)_{k>p}$ in the following way:

- s_{p+1}=$max_{\leq}\{s \in Poles_m \mid \quad |conflicting(s, \{s_1, \ldots, s_p\})| > 1\}$.
- $\forall k > p$, s_{k+1}=$max_{\leq}\{s \in Poles_m \mid s_k > s$ 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 Δ^S_{min}=$\{s_k \mid p < k\}$ *is finite, and is the minimal disambiguation set of* S.

Proof: see Appendix A.

Example 4.2 : Consider again the types and methods in Figure 4. The original signature set is S=$\{(A,G), (B,F), (C,G)\}$, and Δ^S_{min}=$\{(D,G), (E,G)\}$. □

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 argumetn 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^S_{min} 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.

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 4.3 : 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

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

Step 1: Computation of the Ordered Pole Signatures
S ← signatures(M) ;                    /* method signatures */
P ← OrderedPoleSignatures(S) ;

Step 2: Computation of Δ^S_min with explanations
Δ ← ∅ ;                                /* disambiguation signatures */
result ← ∅ ;
for s in P do
    CONF ← conflicting( s, S ∪ Δ ) ;
    if |CONF| > 1 then                 /* s is ambiguous in S ∪ Δ */
        insert s into Δ ;
        add ( s, CONF ) to result ;
return(result) ;
```

Figure 5: Disambiguation Algorithm

added to Δ and $((E,G),\{(C,G),(D,G)\})$ to `result`, which is then returned as the minimal disambiguation set with associated explanations. □

Note that the algorithm is also applicable to languages that use implicit disambiguation to solve multiple inheritance ambiguities, but leave multiple dispatching ambiguities to explicit disambiguation. The only requirement is that the method ordering be monotonous. This unfortunately rules out Dylan [App94].

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

5.1 Ordering the Pole Signatures

Ordering the pole signatures in an order that is compatible with argument subtype precedence 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 argument position, $Pole^i_m$, 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 lexicographic ordering generated by the total orders on the poles.

Example 5.1 : 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. □

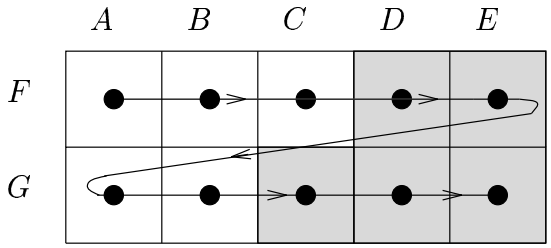

Figure 6: Order of Pole Signatures

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. Unfortunately, signatures are only partially ordered, increasing the complexity to the square of the number of signatures to compare. As $S \cup \Delta^S_{min}$ is a superset of the set of applicable signatures, the worst-case complexity of $conflicting$ is $O(|S \cup \Delta|^2)$. 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 a total order $\leq$. 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 $O(|S \cup \Delta|^2)$.

Example 5.2 : 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 $(E, G) \not> (C, I)$. Hence (E, I) is ambiguous ($conflicting((E, I)) = \{(E, G), (C, I)\}$). □

6 Future Work

Three issues are worth future investigation: detecting ambiguous invocations, mixing method addition with method selection and incremental disambiguation.

Detection of Ambiguous Invocations

In some languages like C++ [ES92] or Cecil [Cha93], ambiguities are checked on a per invocation basis, and not at the method level. The question asked is: does the program contain an actual (compile time) or potential (run-time) ambiguous invocation ? Disambiguating on a per invocation basis is a lengthy and costly process as it involves testing every run-time signature of every invocation for ambiguity, i.e. determining whether there is a single MSA method or multiple conflicting MSA methods for that signature. To speed up this process, pole signatures can be used and in particular, the dispatch table scheme described in [AGS94]. It consists in using tables to efficiently map the set of run-time signatures to the much smaller set of pole signatures. The MSA method of each pole signature can be quickly fetched by an array access to the dispatch table. If there are multiple conflicting methods for some pole signatures, this can be indicated in the dispatch table.

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*.

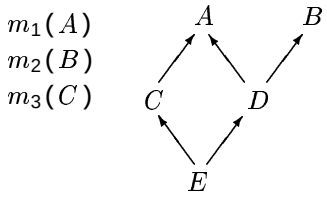

Figure 7: Conflicting Method Selection

Example 6.1 : 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 report, 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 extending the algorithm to allow explicit disambiguation by mixing method addition with method selection. Moreover, pole signatures can be used to optimize checking ambiguities on a per invocation basis. Finally, we are interested in studying the

relationship between the minimal disambiguation set and evolution operations on the type hierarchy and methods.

Acknowledgments: We would like to thank Eric Simon and Marie-Jo Bellosta for their insightful comments on earlier versions of this report.

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.

[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.

[App94] Apple Computer. *Dylan Interim Reference Manual*, June 1994. Available by ftp from ftp.cambridge.apple.com in /pub/dylan/dylan-manual.

[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. CommonLoops: 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.

[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.

[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.

[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.

8 Appendices

A 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 :

- $s_{p+1} = max_{\leq}\{s \in Well\text{-}Typed_m \mid |conflicting(s, \{s_1, \ldots, s_p\})| > 1\}$
- $\forall k > p$, $s_{k+1} = max_{\leq}\{s \in Well\text{-}Typed_m \mid s_k > s$ 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.1 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 report, q the index of the last element of the sequence (s_k).

A.2 Lemma

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

Lemma 8.1 $\forall k \in \{p+1, \ldots, q\}, \nexists s \in Well\text{-}Typed_m$ *s.t.* s *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. $|min_{\preceq}\{s_j \mid j \leq k \text{ and } s_j \succeq s\}| > 1$. Obviously the members of this minimum are strictly more generic than s, hence $|min_{\preceq}\{s_j \mid j \leq k$ 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 $s_{k'+1}$, 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 8.1.

A.3 Δ^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 8.1 to conclude that the assumption is false.

If $s_q > s$ we have $\{s' \in Well\text{-}Typed_m \mid s_q > s'$ and $|conflicting(s', \{s_1, \ldots, s_q\})| > 1\}$ / =$\emptyset$, 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 8.1. Hence $S \cup \Delta^S_{min}$ is not ambiguous, and Δ^S_{min} is a disambiguation set.

A.4 Δ^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 $k \geq p$, we take j such that $\{s^d_j\}$=$conflicting(s_{k+1}, \{s^d_1, \ldots, s^d_r\})$. s^d_j is unique because $\{s^d_1, \ldots, s^d_r\}$ is not ambiguous. Let us show that s^d_j=s_{k+1}. We first show that $s^d_j \not= s_{k+1} \Rightarrow s^d_j$ is ambiguous w.r.t. $\{s_1, \ldots, s_{k+1}\}$, then we apply Lemma 8.1.

By construction of s_{k+1}, $|min_{\preceq}\{s_j \mid j \leq k \text{ and } s_j \succeq s_{k+1}\}| > 1$. From the induction hypothesis, the elements of this minimum are in $\{s^d_1, \ldots, s^d_r\}$. By construction of s^d_j, they are strictly more generic than s^d_j (were one of them equal to s^d_j, it would be the single most specific signature). Hence s^d_j is also ambiguous w.r.t. $\{s_1, \ldots, s_k\}$.

Note that this last implication does not hold if the precedence ordering of methods is not monotonous. Argument subtype precedence is monotonous, because the ordering is $\preceq$, the same w.r.t. all signatures.

Let us assume that $s^d_j \succ s_{k+1}$: s_{k+1} is not applicable to s^d_j, hence s^d_j is ambiguous w.r.t. $\{s_1, \ldots, s_{k+1}\}$. As $s^d_j \geq s_{k+1}$, we apply Lemma 8.1 to show that this assumption is false.

Finally $s^d_j \succeq s_{k+1}$ and $s^d_j \not\succ s_{k+1}$ implies s^d_j=s_{k+1}, thus $\{s_1, \ldots, s_{k+1}\} \subset \{s^d_1, \ldots, s^d_r\}$, which concludes our proof.

A.5 Δ^S_{min} 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 $Pole^i_m$,

$$Influence^i_m(T_\pi) = \{T \in \Theta \mid T \preceq T_\pi \text{ and } \forall T'_\pi \in Pole^i_m, T \not\preceq T'_\pi \text{ or } T_\pi \preceq T'_\pi\}.$$

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

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

Theorem 2 *Let $\{T^1_\pi, \ldots, T^l_\pi\}$ be $Pole^i_m$. Then $\{Influence^i_m(T^1_\pi), \ldots, Influence^i_m(T^l_\pi)\}$ is a partition of $Dynamic^i_m$.*

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 $Dynamic^i_m$, and all $(T^1, \ldots, T^n)$ in $Well\text{-}Typed_m$, we define :

$$pole^i_m(T) = T_\pi, \text{ s.t. } T_\pi \in Pole^i_m \text{ and } T \in influence^i_m(T_\pi)$$
$$pole_m((T^1, \ldots, T^n)) = (pole^1_m(T^1), \ldots, pole^n_m(T^n))$$

We also introduce the following lemma :

Lemma 8.2 $\forall s \in Well\text{-}Typed_m, \forall s' \in Poles_m, s' \succeq s \Rightarrow s' \succeq pole_m(s)$.

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 $\forall i \in \{1,\ldots,n\},\ T^i \preceq pole^i_m(U_i)$.

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 8.2.

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 $k \geq p$, 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 8.1.

By construction of s_{k+1}, we have $|min_\preceq\{s_j \mid j \leq k \text{ and } s_j \succeq s_{k+1}\}| > 1$. By the induction hypothesis, these signatures of $conflicting(s_{k+1},\{s_1,\ldots,s_k\})$ are in $Poles_m$. From Lemma 8.2, they are strictly more generic than s'_{k+1} (were one of them equal to s'_{k+1}, it would be the single most specific signature). Hence s'_{k+1} is ambiguous w.r.t. $\{s_1,\ldots,s_k\}$. As above, this last implication does not hold if the precedence ordering of methods is not monotonous.

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 8.1 to show that this assumption is false.

From $s'_{k+1} \succeq s_{k+1}$ and $s'_{k+1} \not\succ s_{k+1}$, it follows that $s'_{k+1}=s_{k+1}$, thus $\{s_1,\ldots,s_{k+1}\} \subset Poles_m$. This concludes our proof.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399