Typechecking and Modules for Multi-Methods

Craig Chambers and Gary T. Leavens

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, Washington 98195 USA

Technical Report 95-08-05
August 1995

Typechecking and Modules for Multi-Methods

Craig Chambers
Department of Computer Science and Engineering
309 Sieg Hall, FR-35
University of Washington
Seattle, Washington 98195
(206) 685-2094; fax: (206) 543-2969
chambers@cs.washington.edu

Gary T. Leavens
Department of Computer Science
229 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040
(515) 294-1580
leavens@cs.iastate.edu

UW CS&E Technical Report 95-08-05

ISU CS Technical Report #95-19

Abstract

Two major obstacles hindering the wider acceptance of multi-methods are concerns over the lack of encapsulation and modularity and the absence of static typechecking in existing multi-method-based languages. This paper* addresses both of these problems. We present a polynomial-time static typechecking algorithm that checks the conformance, completeness, and consistency of a group of method implementations with respect to declared message signatures. This algorithm improves on previous algorithms by handling separate type and inheritance hierarchies, abstract classes, and graph-based method lookup semantics. We also present a module system that enables independently-developed code to be fully encapsulated and statically typechecked on a per-module basis. To guarantee that potential conflicts between independently-developed modules have been resolved, a simple well-formedness condition on the modules comprising a program is checked at link-time. The typechecking algorithm and module system are applicable to a range of multi-method-based languages, but the paper uses the Cecil language as a concrete example of how they can be applied.

1 Introduction

Multiple dispatching of multi-methods as found in CLOS [Bobrow *et al.* 88, Steele 90, Paepcke 93] and Cecil [Chambers 92, Chambers 93] is a more general form of message passing (dynamic binding) than traditional single dispatching of receiver-based methods as found in Smalltalk [Goldberg & Robson 83] and C++ [Stroustrup 91] or static overloading of functions as found in C++, Ada [Ada 83, Barnes 91], and Haskell [Hudak *et al.* 92]. With multiple dispatching, method lookup can depend on the run-time types or classes of any subset of the arguments to a message, not just the run-time class of the single receiver argument as in singly-dispatched systems nor just the arguments' compile-time types as in systems with static overloading.

To illustrate, consider the following matrix multiplication implementations, written in a close approximation to Cecil syntax:†

```
-- matrix is the abstract superclass of all matrix implementations
abstract type matrix;
  method fetch(m:matrix, row:int, col:int):num {
    abstract } -- this method must be provided by concrete descendants
  method +(m1:matrix, m2:matrix):matrix {
    ... } -- add matrices, invoking implementation-specific fetch functions to fetch elements
```


* An earlier version of this paper appeared in the proceedings of *OOPSLA '94*.

† For simplicity, in this paper we ignore issues relating to parameterized types. Hence the matrix is a matrix of numbers rather than being parameterized by the element type as it really is in Cecil.

```
  method *(m1:matrix, m2:matrix):matrix {
    ... } -- multiply matrices, invoking implementation-specific fetch functions to fetch elements

concrete type dense_matrix isa matrix;
  method fetch(m@dense_matrix, row:int, col:int):num {
    ... } -- the implementation of fetching for a dense matrix
  method +(m1@dense_matrix, m2@dense_matrix):matrix {
    ... } -- an optimized implementation of + for two dense matrices

concrete type sparse_matrix isa matrix;
  method fetch(m@sparse_matrix, row:int, col:int):num {
    ... } -- the implementation of fetching for a sparse matrix

let a, b: matrix := ...;
print(a + b*b); -- invoke most specific + and * functions, depending on run-time classes of a and b
```

Some of the formals in the above methods are declared using the form `name@specializer`. Such a formal is called a *specialized formal* and is subject to dynamic dispatching. A method is only applicable to actual argument objects that descend from the formal's *argument specializer class* named after the `@` symbol. Moreover, argument specializers determine the overriding relationships among methods: methods with more specific argument specializers override methods with less specific argument specializers.

Unspecialized formals are treated as being specialized on a distinguished `any` class that is an ancestor of all other classes; an unspecialized formal applies to all actual argument objects and is less specific than any specialized formal. An unspecialized formal may still be declared to be of a particular type, using the notation `name:type`. Such a type declaration specifies the *interface* required of actual arguments but places no constraints on their *implementation*. Static type checking must guarantee that these interface requirements are satisfied.

In the matrix arithmetic example, the method `*` is unspecialized and hence acts like a normal function. The methods named `fetch` are specialized on their first argument and so emulate singly-dispatched receiver-based methods. The first + method does not specialize on any arguments, and so it acts like a default method, while the second + method is specialized on multiple arguments. The ability of each method individually to specialize on any subset of its arguments integrates unspecialized, singly-dispatched, and multiply-dispatched methods in a uniform framework, facilitating the definition of algebraic data types with binary operations and other kinds of operations where knowledge of or access to the representations of several arguments is needed.

Unfortunately, the potential increased expressiveness of multi-methods is hampered by several drawbacks that limit their wider acceptance:

- The programming style often associated with multi-methods, based on generic functions, is viewed by many as contrary to the object-centered programming style employed in singly-dispatched object-oriented languages. This problem was addressed in an earlier paper that described a programming methodology, language design, and programming environment for multi-methods that attempts to preserve much of the flavor of object-centered programming [Chambers 92].
- The semantics of multi-method lookup is considered to be very complicated. This problem also was addressed in the earlier paper, where a simple lookup semantics was presented which was based on deriving the partial ordering over methods from the partial ordering over their specializers. This semantics considers ambiguously-defined multi-methods to be a programming error, unlike the CLOS semantics which attempts to resolve such ambiguities automatically.
- Multi-methods might be slower to select and invoke than singly-dispatched methods. Work is progressing on this front, however [Chen & Turau 94, Amiel *et al.* 94], and we expect that the run-time

performance difference between singly- and multiply-dispatched systems to become negligible in the near future.

- Multi-methods are seen to prevent object encapsulation. One approach to solving this problem was presented in the earlier paper, but that approach did not allow privileged access to be restricted to a single, well-defined area of program text.
- The few static type systems that have been designed for multi-method-based languages have dealt with a fairly restrictive language model. Recent multi-method languages contain features such as abstract classes, mixed specialized and unspecialized formals, partially-ordered multi-method definitions, and independent inheritance and subtyping graphs, and these features cannot be handled by previously proposed static type systems for multi-method-based languages.
- With multi-methods, independently-developed libraries cannot be typechecked completely separately, but instead must be typechecked at link-time. Similarly, code written in one library might interact unintentionally with code written in another independently-developed library, leading to message lookup errors that did not exist when the libraries were separate.

In this paper we address the last three points above:

- We present a type system for languages with multi-methods and separate subtyping and code inheritance, with *signatures* specifying the set of legal messages (and required implementations) in a program.
- We describe a polynomial-time type checking algorithm that guarantees statically the absence of message lookup errors for a much more general and realistic class of languages than does previous work.
- We describe a module mechanism that allows privileged access to be textually restricted, enables parts of a program to be typechecked independently, and eases integration of independently-developed code.

The next section of this paper reviews related work. Section 3 describes the language model and type system that our algorithm supports and shows how the Cecil language fits into this model. Section 4 then specifies the typechecking problem, details our algorithm, argues for its correctness, and analyzes its complexity. Section 5 introduces our module mechanism and discusses its impact on the typechecking algorithm. Section 6 offers our conclusions.

2 Related Work

2.1 Type Checking

Agrawal, DeMichiel, and Lindsay present a polynomial-time algorithm for typechecking Polyglot, a CLOS-like database type system [Agrawal *et al.* 91]. Their algorithm divides the typechecking problem into two components: checking that the collection of multi-methods comprising a generic function is *consistent*, and checking that calls of generic functions are type-correct. Our algorithm makes a similar division between client-side checking and implementation-side checking, mediated by a set of legal signatures. However, their algorithm depends on a number of restrictive assumptions about the language they typecheck:

- The multi-methods within a generic function must be *totally ordered* in terms of specificity. Graph-based method lookup semantics found in most object-oriented languages with multiple inheritance [Snyder 86], where the method overriding relationship only forms a partial order, cannot be handled. Our algorithm supports such partially ordered method hierarchies while still detecting whether any ambiguously-defined messages are sent.
- All classes in Polyglot are assumed to be *concrete* and fully implemented; all of the multi-methods in a generic function are complete implementations. This assumption is needed because their algorithm declares a call site legal exactly when there is a method implementation that applies to the static types of the formals. Our algorithm is more flexible because it allows a call to be declared legal as long as all

concrete implementations of the arguments' static types provide an implementation for the method. This allows the use of abstract classes defining interfaces whose implementation is deferred to concrete subclasses, as with the abstract `matrix` class and the `fetch` function earlier.

- Inheritance and subtyping are synonymous in Polyglot. While many common object-oriented languages link code inheritance with subtyping, many researchers have noted that conceptually the two relations are different (e.g., [Snyder 86, Cook *et al.* 90, Leavens & Weihl 90]). Subclassing refers to an implementation strategy, where a subclass inherits some of its implementation from its superclasses. Subtyping, on the other hand, refers to a relationship between the interfaces of two types. If a class *conforms* to a type, then all of its (direct) instances support the interface specified by the type. Moreover, if one type is a *subtype* of another, then all classes that conform to the subtype also conform to the supertype. This definition allows instances of the subtype (i.e., direct instances of classes that conform to the subtype) to be substituted wherever instances of the supertype are expected, since the instances of the subtype also support the interface of the supertype. Subclassing and subtyping need not be tied together: a class can inherit code from some other class without being required to be a subtype, and the type of a class can be a subtype of the type of some other class without forcing the subclass to also inherit code from the other class. Keeping inheritance and subtyping separate allows for more flexible and extensible organizations of code, and some more recent languages including Cecil, POOL [America 87, America & van der Linden 90], and Strongtalk [Bracha & Griswold 93] do in fact separate the two relations. Our algorithm allows the type partial order to be specified independently of the code inheritance graph, and the set of legal messages (described by *signatures*) can be defined independently of the set of multi-method implementations.
- In Polyglot, all arguments are dispatched. Methods are ordered using the declared types of all their formals in Polyglot. Our algorithm allows any subset of a method's formals to be specialized, with the unspecialized formals receiving normal type declarations that must be guaranteed statically. As a result, our algorithm includes the standard contravariant method typechecking rules of singly-dispatched languages as a special case.

Kea is a higher-order polymorphic functional language supporting multi-methods [Mugridge *et al.* 91]. Like Polyglot, code inheritance and subtyping in Kea are unified. Kea's type checking includes the notion that a collection of multi-methods must be *exhaustive* and *unambiguous*, and these notions appear in our type system as well. The semantics of typechecking in Kea is specified formally, but an efficient typechecking algorithm is not presented. As with Polyglot, our contribution in the area of typechecking relative to Kea is that we typecheck several important language features not found in Kea, including mutable state, separate subtyping and inheritance graphs, abstract classes, and mixed specialized and unspecialized arguments. Moreover, we present a typechecking algorithm, argue for its correctness, and analyze its complexity.

Other researchers have developed more theoretical accounts of multi-method-based languages [Rouaix 90, Leavens & Weihl 90, Ghelli 91, Castagna *et al.* 92, Pierce & Turner 92, Bruce *et al.* 95]. These papers are more concerned with specifying the semantics of multi-methods and with defining type systems than with algorithms for typechecking. As a result, they ignore many of the language features specifically addressed by our work. Most other work on type systems for object-oriented programming (e.g., [Cardelli & Wegner 85, Cardelli & Mitchell 89, Bruce *et al.* 93, Palsberg & Schwartzbach 94]) only deals with singly-dispatched languages.

2.2 Module Systems

The only module system for a multi-method-based language of which we are aware is the Common Lisp package system [Steele 90]. This system provides name space management, allowing symbols to be clustered into packages and allowing some symbols to be private to a package. In Common Lisp, encapsulation is only advisory, and users may always circumvent the encapsulation of a symbol `s` in a package `p` by writing `p::s`. Common Lisp does not include static type checking. In contrast, encapsulation can be enforced in our module system and our module system cooperates with our static typechecking algorithm.

A few other object-oriented languages include some form of separate module system, including Modular Smalltalk [Wirfs-Brock & Wilkerson 88], Modula-3 [Nelson 91], and Oberon-2 [Mössenböck & Wirth 91]. In Modular Smalltalk, modules provide name space management for class names, and a separate mechanism provides access control for the methods of a class. Our module design is closer to the Common Lisp, Modula-3, and Oberon-2 approach, with a single construct, the module, providing all name space management and access control.

Many object-oriented languages enable access to the operations on classes to be controlled. C++ classes, for example, have three levels of access control: one level for clients (`public`), one for subclasses (`protected`), and one restricted to the class and its explicitly named friends (`private`). Because of C++'s `friend` mechanism, one can write software that has privileged access to more than one type of data while still textually limiting private access. Our module design also supports these degrees of visibility. Trellis supports these notions except for friends [Schaffert *et al.* 86], and Eiffel supports public and protected levels of visibility [Meyer 88, Meyer 92].

Canning, Cook, Hill, and Olthoff define a notion of interfaces for languages like Smalltalk [Canning *et al.* 89]. Their notation distinguishes types from classes, as do we, and they are concerned with type checking against such interfaces. They also have an interesting notion of interface inheritance. However, they do not consider multi-methods or encapsulation issues.

More sophisticated module systems than ours are found in the functional language Standard ML [Milner *et al.* 90, Paulson 91] and in the equational specification language OBJ2 [Goguen 84]. SML's modules are first-class and can be parameterized. OBJ2's theories are like SML's signatures (the interfaces to SML modules), but allow for behavioral specifications as well as type information. Both SML and OBJ2 have ways of importing modules that allow for sophisticated kinds of renaming. We omit such sophisticated features to keep our proposal simple and to focus on support for multi-methods.

3 Programming Model and Type System

Our typechecking algorithm is designed for object-oriented languages that have a class inheritance graph, a potentially separate subtyping graph, a set of multi-method implementations specialized to classes, and a set of *message signatures* that define the message interface supported by types. The following subsections elaborate on these assumptions and show how the Cecil language's constructs meet these assumptions.

3.1 Classes and Inheritance

We assume that the program declares a finite set of classes, C. We assume that a subset of these classes, $C_{concrete} \subseteq C$, indicates which classes are concrete and instantiable. Classes in C but not $C_{concrete}$ are abstract and cannot be directly instantiated at run-time. Abstract classes can model pure virtual classes in C++ and deferred classes in Eiffel.

We assume also that the program defines a fixed binary relation direct-inherits on C modeling direct inheritance of implementation between classes. We then define the binary relation $\leq_{inh}$ on C as the reflexive, transitive closure of direct-inherits. We require that $\leq_{inh}$ impose a partial order on C, i.e., there cannot be cycles in the declared inheritance graph. Finally, we assume that there exists a class any $\in C$ that is the single greatest element of $\leq_{inh}$; this class is used as the specializer of formals with no explicit specializer. For languages such as C++ where no explicit root class of the inheritance hierarchy is required, an implicit root class can be created that is the superclass of all other (explicit) classes.

In Cecil, the class inheritance graph is derived from `representation` declarations and `inherits` clauses. For example, the Cecil declarations

```
abstract representation matrix_rep;
template representation dense_matrix_rep inherits matrix_rep;
```

are modeled with two classes named `matrix_rep` and `dense_matrix_rep`, with `dense_matrix_rep` inheriting from `matrix_rep`. The class `matrix_rep` is an abstract class, while `dense_matrix_rep` is a concrete class, since in Cecil a template representation acts like a pattern for run-time-created objects while an abstract representation cannot be instantiated at run-time. Cecil includes a predefined class `any` which is implicitly the ancestor of all other classes (corresponding to any in our formal model) and which is used as the specializer of otherwise unspecialized formals of multi-methods. Additionally, Cecil includes closure objects, first-class lexically-nested anonymous function objects that execute their bodies when sent the `eval` message. Each textual occurrence of a closure constructor expression is modeled as a distinct class.

3.2 Types and Subtyping

We assume that the program declares a finite set of types, T, and an associated reflexive, transitive binary relation $\leq_{sub}$ on T that models subtyping: $t_1 \leq_{sub} t_2$ iff t_1 is equal to t_2 or t_1 is a subtype of t_2. In our model, types and subtyping can be completely independent from classes and code inheritance, and the $\leq_{sub}$ ordering on types is independent of the $\leq_{inh}$ ordering on classes. To make some of our typechecking rules simpler to express, we assume that $\langle T, \leq_{sub} \rangle$ forms a lattice, i.e., for every pair of types t_1 and $t_2 \in T$ there exists a unique greatest lower bound (most specific supertype) $\mathsf{glb}(t_1, t_2) \in T$, and a unique least upper bound (most general subtype) $\mathsf{lub}(t_1, t_2) \in T$. Most object-oriented languages only require their subtyping graph to be a partial order, not a lattice, but it is simple to create implicit g.l.b. and l.u.b. types to transform any partial order into a lattice. Alternatively, the typechecking rules given below could be changed to compute sets of top lower bound types and bottom upper bound types, instead of such fictional g.l.b. and l.u.b. types.

In Cecil, types and subtyping are derived from `type` declarations and `subtypes` clauses. For example, the Cecil declarations

```
type matrix_type;
type dense_matrix_type subtypes matrix_type;
```

are modeled with two types named `matrix_type` and `dense_matrix_type` with `dense_matrix_type` subtyping from `matrix_type`. The $\leq_{sub}$ subtype relation includes the reflexive, transitive closure of these explicitly-declared direct subtype relationships. Cecil also includes the following types and type constructors:

- `void`, the return type of functions that return no useful result to their callers, which is implicitly the supertype of all other types (the top of the type lattice): $\forall\, t \in T.\ t \leq_{sub}$ `void`;
- `any`, which is implicitly the supertype of all non-`void` types: $\forall\, t \in T, t \neq$ `void`. $t \leq_{sub}$ `any`;
- `none`, the type of functions that do not return to their callers, which is implicitly a subtype of all other types (the bottom of the type lattice): $\forall\, t \in T.$ `none` $\leq_{sub} t$;
- $t_1 \mid t_2$, the least upper bound of two types;
- t_1 & t_2, the greatest lower bound of two types; and
- $\lambda(\vec{t}):t_{result}$, the types of closures,* which use standard contravariant rules for subtyping:

  $\lambda(\vec{t}):t_{result} \leq_{sub} \lambda(\vec{t}\,'):t_{result}' \Leftrightarrow (\forall i.\ t_i' \leq_{sub} t_i) \wedge t_{result} \leq_{sub} t_{result}'$.

3.3 Conformance of Classes to Types

The class and type graphs are related through the notion of classes conforming to types. Informally, when a class *c conforms* to a type *t*, the class *c* implements the behavior specified by the type *t*; thus direct instances

* The notation $\vec{t}$ stands for a vector of types, as explained in subsection 3.4.

of class c may be stored in variables of type t. To model this relationship, we assume that the program defines a function direct-conforms: $C \rightarrow T$, which for each class gives the most specific type to which the class directly conforms. We then derive the full conformance relation between classes and types, ($<:$): $C \rightarrow T$, from direct-conforms and the subtyping relation $\leq_{sub}$ as follows:

$$c <: t \quad \equiv \quad \text{direct-conforms}(c) \leq_{sub} t.$$

Informally, whenever a class conforms to a type, it also conforms to all supertypes of the type.

Because subtyping and inheritance do not necessarily coincide, one class can inherit from another without being substitutable in place of the other. More formally, if $c \leq_{inh} c'$ and $c' <: t$, one *cannot* conclude that $c <: t$.

In Cecil, direct conformance is derived from `conforms` clauses that are part of `representation` declarations:

```
template representation dense_matrix_rep
  inherits matrix_rep
  conforms dense_matrix_type;
```

This declaration indicates that the class `dense_matrix_rep` conforms directly to the type `dense_matrix_type`. The `dense_matrix_rep` class will conform indirectly to all supertypes of `dense_matrix_type`.

3.4 Vectors of Classes and Types

To model argument lists, we form vectors of classes and types. It simplifies the discussion of the typechecking algorithm to assume an inheritance, subtyping, or conformance relation between vectors, derived by extending the appropriate relation on individual classes or types pointwise; i.e., for a relation $\leq_R$:

$$\vec{p} \leq_R \vec{q} \quad \equiv \quad |\vec{p}| = |\vec{q}| \wedge (\forall\, i \in \text{indexes}(\vec{q})\,.\, p_i \leq_R q_i).$$

Informally, a vector of classes is considered to override (inherit from) another equal-length vector of classes whenever each of the element classes of one vector overrides the corresponding element of the other vector; subtyping between two type vectors and conformance between a class vector and a type vector are defined similarly.

3.5 Method Implementations

We assume a program defines a finite set of message names, *MessageKey*, and a finite set of method implementations, M. Each method implementation has a name, a vector of argument specializer classes, a vector of argument types, a result type, and a body. We will use the following functions to access the components of a method implementation:

- A function msg: $M \longrightarrow \mathit{MessageKey}$, such that msg($m$) = μ is the message handled by m.
- A function specializers: $M \longrightarrow C^*$, such that specializers(m) = $\vec{c}$ is a vector of classes that are the argument specializers for method m.
- A function argtypes: $M \longrightarrow T^*$, such that argtypes(m) = $\vec{t}$ is a vector of types declared for the formals of method m.
- A function restype: $M \longrightarrow T$, such that restype(m) = t is result type of method m.

In Cecil, method implementations are derived from `implementation` declarations like the following:

```
implementation fetch(m@matrix_rep:matrix_type,
                     row@any:int, col@any:int):num { ... }
```

The name of this method is `fetch/3` (in Cecil, a method only applies to messages with the right number of arguments), its argument specializers are modeled with the class vector `<matrix_rep, any, any>`, its argument types are modeled with the type vector `<matrix_type, int, int>`, and its result type is `num`.

This method specializes only on its first argument, thereby stating that it is applicable to instances of classes that inherit from the class `matrix_rep`, but static type checking is required to ensure that the `row` and `col` formals are passed actual arguments that support the interface specified by the type `int`. (Static type checking is also needed to ensure that for each class `c` that inherits from `matrix_rep`, either `c` conforms to `matrix_type` or there is an overriding `fetch` method specialized on class `c`.)

We derive a partial ordering on methods, $\leq_{meth}$, modelling the method overriding relationship, from the methods' argument specializer classes (the type vector does not affect method overriding) as follows:

$$m_1 \leq_{meth} m_2 \quad \equiv \quad \mathsf{specializers}(m_1) \leq_{inh} \mathsf{specializers}(m_2).$$

This ordering reflects the message lookup semantics in Cecil: one method overrides another exactly when its argument specializers are at least as specific as the other's and moreover at least one of the overriding method's specializers is strictly more specific than the other's. We say that a vector of classes $\vec{c}$ *inherits* a method m when $\vec{c} \leq_{inh} \mathsf{specializers}(m)$.

Because vectors of classes are ordered pointwise, with no priority assigned to the position of the vector element, the specializers of a method are equally important in determining the method's overriding relationships [Touretzky 86]. This matches Cecil's semantics, but may not match other languages'. For example, CLOS prioritizes argument positions with earlier argument orderings completely dominating later argument orderings. It seems possible to extend our model to encompass other method overriding relationships, for example by ordering vectors of classes lexicographically rather than pointwise.

3.6 Signatures

The final component of a program is a set of *signatures*, S, where each signature has a name, a vector of argument types, and a result type. A signature declares that any message with a matching name and arguments that conform to the signature's argument types is considered legal to send, and consequently the signature places constraints on the set of method implementations that purport to support the signature. In our model, just as types and classes are distinct, signatures and method implementations are distinct. However, we will overload the function names for accessing a method implementation's components to access the analogous components of a signature:

- A function msg: $S \longrightarrow \mathit{MessageKey}$, such that $\mathsf{msg}(s) = \mu$ is the message handled by s.
- A function $\mathsf{argtypes}$: $S \longrightarrow T^*$, such that $\mathsf{argtypes}(s) = \vec{t}$ is a vector of types declared for the arguments of signature s.
- A function $\mathsf{restype}$: $S \longrightarrow T$, such that $\mathsf{restype}(s) = t$ is result type of signature s.

In Cecil, signatures are derived from `signature` declarations like the following:

```
signature fetch(matrix_type, int, int):num;
```

This signature specifies that it is legal to send the `fetch` message to three arguments that conform to the `matrix_type`, `int`, and `int` types, respectively. Additionally, such a message can be assumed to return an object that conforms to the type `num`.

Many signatures can be defined with the same name. Each can provide a different relationship between argument types and result types. For example, in the following declarations:

```
type num;
type int subtypes num;
type fraction subtypes num;
signature +(num,num):num;
signature +(int,int):int;
signature +(fraction,fraction):fraction;
```

several + signatures are defined. If a client knows only that it is adding two numbers, then it can assume only that the result is a number. However, if a client knows it is adding a pair of integers, then it will be able to infer that the result of the message is not just a number but also an integer. This ability to overload and extend signatures is an important part of our type system.

3.7 Syntactic Sugar

While Cecil supports independent specification of the class graph, the type graph, and the conforms relation, in practice these relations often take on very stylized forms. To make programming easier, Cecil includes the `object` declaration, which is syntactic sugar for a `representation` and a `type` declaration with a `conforms` clause linking the two, and the `isa` clause, which is syntactic sugar for an `inherits` clause and a `subtypes` clause. To illustrate, the following declarations more concisely define the same object, type, and conformance structures as the earlier `implementation` and `type` declarations:

```
abstract object matrix;
template object dense_matrix isa matrix;
```

Here `matrix` names both a representation and a type. Since in Cecil types and representations are in distinct name spaces, and it is clear by context which name space is used in a construct, no ambiguity can result.

As with representations and types, Cecil supports the `method` declaration which is syntactic sugar that allows implementations and signatures to be declared simultaneously when convenient. The following `method` declaration generates an implementation declaration and a signature similar to the ones illustrated above:

```
method fetch(m@:matrix, row:int, col:int):num { ... }
```

This declaration illustrates two final pieces of syntactic sugar in Cecil. If a formal's specializer is `@any`, this may be omitted, as in the `row` and `col` formals above. If a formal's specializer and its declared type have the same name, then the `@:` sugar is more concise, as with the `m` formal above.

With these sugars, Cecil programs are just as concise as other languages for the cases where code inheritance and subtyping coincide. However, the additional flexibility of independent inheritance and subtyping relations is always available when needed.

4 Typechecking Algorithm

The subtyping graph and the set of signatures together define an interface. We use this interface to divide the typechecking process for a program into two parts: *client-side* checking of expressions against the type/signature interface and *implementation-side* checking that class and method definitions properly implement the interface guaranteed to clients by the type and signature specifications. The next subsection briefly discusses client-side checking. The remaining subsections discuss the more difficult problem of implementation-side checking. Subsection 4.2 specifies the implementation-side typechecking problem. Subsection 4.3 presents an overview of our algorithm, with subsections 4.4 through 4.7 filling in the details. Subsection 4.8 discusses the impact on the algorithm of some of the more sophisticated language features supported by our model.

4.1 Client-Side Typechecking

Client-side checks are fairly typical, including checks such as that an expression of one type is only assigned to variables declared to be of a supertype and that a method only returns the results of expressions that are subtypes of the declared return type of the method. The most interesting of the client-side checks is for message sends, since sends are the only kind of expression whose checking depends on signatures. In our model, a message typechecks if there is a signature with the same name as the message whose argument types are supertypes of the static types of the send's argument expressions. To compute the type of the result of the message, all signatures that match the send in this way are collected, and then the most specific result type of any of the matching signatures is used as the result of the send. More precisely, a message μ sent to argument expressions of static type $\grave{t}$ typechecks iff the set of signatures

$$S_{match} = \{\, s \mid \mathsf{msg}(s) = \mu \wedge \grave{t} \leq_{sub} \mathsf{argtypes}(s) \,\}$$

is non-empty. The static type of the result of such a message is

$$\mathsf{glb}(\{\, \mathsf{restype}(s) \mid s \in S_{match} \,\})$$

For example, given the types and signatures

```
type num;
type int subtypes num;
type fraction subtypes num;
signature +(num,num):num;
signature +(int,int):int;
signature +(fraction,fraction):fraction;
```

and the send expression

```
3 + 4
```

whose argument types are `<int, int>`, the set of matching signatures is `{+(num,num):num, +(int,int):int}`. Because this set is non-empty, the expression is type-correct. The type of the result of this message is `int`, the most specific result type of the matching signatures.

To compute the type of the result of a message send expression from the set of matching signatures, the greatest lower bound of the signatures' result types is used. This selects the single most specific result type, if one exists, and otherwise computes the greatest lower bound of the most specific result types. One might have expected that the least upper bound would be used, as would happen if several applicable method implementations were being collected together. However, unlike method implementations, signatures are *not* selected at run-time. Rather, they are static guarantees about input-output typing properties of messages: if the client knows that the arguments to a + message are subtypes of `num`, then the client can assume that the result will be a subtype of `num`. A signature does not override another signature, but instead augments it with additional static type information. The following contrived example illustrates these properties:

```
signature *(num,num):int;
signature *(int,int):num;
```

These signatures promise clients that multiplying any two numbers returns an integer, and *additionally* that multiplying two integers can be assumed to return a number; the second signature provides no new information to clients about multiplying integers that isn't already known about multiplying numbers. Implementations of `*` will be required to satisfy all matching signatures. In particular, the implementation of `*` for integers will be required to return a subtype of both `int` and `num`, i.e., `int`. Because of this constraint on implementations, clients can safely assume that the result of a message will be a subtype of the declared result types of all matching signatures. One might expect more specific signatures to narrow result types in a covariant fashion [Reynolds 80], but by using the g.l.b. operation our model does not need to impose any such restrictions.

Other client-side checks are straightforward and language-dependent, and we do not discuss them further here. Of course, to prove type safety for a complete program, one would need to specify the kinds of expressions allowed in the language; describe sufficient conditions for expressions to be type-correct, assuming correctness of implementations; describe a typechecking algorithm for client-side expressions and prove that it implies the type-correctness conditions; and finally prove that the type-correctness of the client-side together with type-correctness of the implementation side implies a global type safety condition. Since our model of expressions and statements is fairly standard, we do not go through these steps, and instead focus on the new work of implementation-side typechecking.

4.2 Specification of Implementation-Side Typechecking

A set of classes and methods in a program is considered to correctly implement the interface guaranteed to clients by a set of types and signatures if every possible message that could be sent to concrete arguments that conform to the argument types of some signature would result in a legal message send with no message lookup errors. More precisely, the implementation-side checks are satisfied if for each signature, for each vector of concrete argument classes that conforms to the argument types of the signature, a single most specific method is inherited by that argument vector. (Because the inheritance ordering is acyclic, the unique most specific method is one that is least in the set of applicable methods.) Moreover, the vector of concrete argument classes must conform to the declared argument types of the method, and the method's result type must be a subtype of the signature's result type.

$$
\begin{array}{l}
\mathsf{ImplementationTypechecks} \equiv \\
\quad \forall\, s \in S.\;\; \forall\, \vec{c} \in (C_{concrete})^{*}. \\
\qquad \vec{c} <: \mathsf{argtypes}(s) \Rightarrow \\
\qquad\quad \exists\, m \in M. \\
\qquad\qquad m = \mathsf{least}(\mathsf{applicable\text{-}methods}(s,\ \vec{c})) \;\wedge \\
\qquad\qquad \vec{c} <: \mathsf{argtypes}(m) \;\wedge \\
\qquad\qquad \mathsf{restype}(m) \leq_{sub} \mathsf{restype}(s)
\end{array}
$$

where*

$$
\begin{array}{l}
\mathsf{applicable\text{-}methods}(s,\ \vec{c}) \equiv \\
\quad \{\, m \in M \mid \mathsf{msg}(m) = \mathsf{msg}(s) \;\wedge\; |\vec{c}| = |\mathsf{argtypes}(m)| \;\wedge\; \vec{c} \leq_{inh} \mathsf{specializers}(m) \,\}
\end{array}
$$

$$
\begin{array}{l}
m = \mathsf{least}(Ms) \;\equiv \\
\quad m \in Ms \;\wedge \\
\quad \forall\, m' \in Ms.\;\; m \leq_{meth} m'
\end{array}
$$

Note that the ImplementationTypechecks rule is looser than the standard contravariant method implementation overriding rule. An applicable method *m* can have argument types that are not supertypes of those in the corresponding signature *s*, as long as there are no concrete classes $\vec{c}$ that conform to the signature's argument types, argtypes(*s*), that do not also conform to the method's argument types, argtypes(*m*).

Directly executing this specification would lead to an algorithm with execution time on the order of $O(|S| \cdot |C_{concrete}|^{k} \cdot |M|)$, where *k* is the maximum number of arguments of any message in the program. We assume that *k* is bounded by a constant, and so does not grow with the size of a program. Since *S*, $C_{concrete}$, and *M* are likely to be large, such an algorithm would be unacceptably slow in a practical system. One of our main contributions is an algorithm that is much faster for what we believe are the commonly occuring cases for implementation-side typechecking. No such algorithm has been previously proposed for the class of languages that is described in section 3.

4.3 Overview of the Algorithm

We divide the implementation-side typechecking algorithm into checking for three separate properties of class/method implementations with respect to the type/signature interface: conformance, completeness, and consistency. We describe and specify each of these subproblems in turn below.

Conformance means that for each signature, the argument and result types of every method *covered* by the signature must be compatible with those specified by the signature. A method is covered by a signature if it could be invoked by a send that is type-correct according to the signature, i.e., if there exists some vector of concrete classes that conforms to the argument types of the signature, inherits from the method's argument

* In all our specifications, we assume that the sets and relations defined in section 3 are globally available.

specializers, and is not overridden by some other method. An entire implementation is conforming if for each signature, each of the methods in the signature's covered set conforms to the signature. Conformance ensures that the types of a method's formals are respected and that the type of the result of a message is respected, thus fulfilling the implementation side's part of the bargain with client expressions. This specification of conformance is formalized in the predicate ImplementationIsConforming.

$$
\begin{array}{l}
\text{ImplementationIsConforming} \equiv \\
\quad \forall\, s \in S.\ \ \forall\, \vec{c} \in (C_{concrete})^*. \\
\qquad \vec{c} <: \text{argtypes}(s) \Rightarrow \\
\qquad\quad \forall\, m \in \text{applicable-methods}(s, \vec{c}). \\
\qquad\qquad (\neg\, \exists\, m' \in \text{applicable-methods}(s, \vec{c}).\ m' \neq m \wedge m' \leq_{meth} m) \Rightarrow \\
\qquad\qquad\quad \vec{c} <: \text{argtypes}(m) \wedge \text{restype}(m) \leq_{sub} \text{restype}(s)
\end{array}
$$

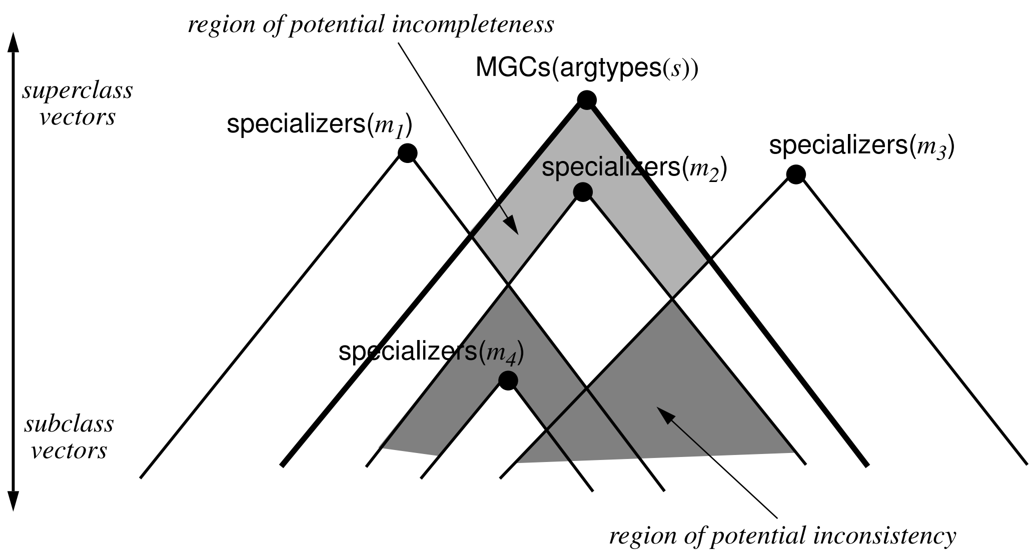
Informally, *completeness* means that each signature is fully implemented by a set of methods, i.e., for every concrete argument vector conforming to the signature's argument types, there must exist at least one method that implements the message. If the methods are incomplete, then a "message not understood" error might arise at run-time. Completeness is formalized as follows:

$$
\begin{array}{l}
\text{ImplementationIsComplete} \equiv \\
\quad \forall\, s \in S.\ \ \forall\, \vec{c} \in (C_{concrete})^*. \\
\qquad \vec{c} <: \text{argtypes}(s) \Rightarrow \\
\qquad\quad |\text{applicable-methods}(s, \vec{c})| > 0
\end{array}
$$

Conversely, *consistency* means that there are no ambiguities among the methods implementing a signature, i.e., for every concrete argument vector conforming to the signature's argument types, there must exist no more than one most-specific method that implements the message. If the methods are inconsistent, then a "message ambiguously defined" error could occur at run-time. In precise notation:

$$
\begin{array}{l}
\text{ImplementationIsConsistent} \equiv \\
\quad \forall\, s \in S.\ \ \forall\, \vec{c} \in (C_{concrete})^*. \\
\qquad \vec{c} <: \text{argtypes}(s) \Rightarrow \\
\qquad\quad \forall\, m_1, m_2 \in \text{applicable-methods}(s, \vec{c}). \\
\qquad\qquad \exists\, m \in \text{applicable-methods}(s, \vec{c}). \\
\qquad\qquad\quad m \leq_{meth} m_1 \wedge m \leq_{meth} m_2
\end{array}
$$

Completeness and consistency can be explained visually using the following "mountain top" diagram.



The diagram divides up regions of the space of vectors of classes, with one vector plotted below another if the first overrides the second (i.e., each of the elements of the first vector inherits from the corresponding element of the second). We have drawn cones below certain points in this space, enclosing the set of vectors that inherit from (override) the root of the cone; in the presence of multiple inheritance, a vector of classes may inherit from several mutually-unrelated vectors, leading to overlapping cones as in the diagram. The vector labeled MGCs(argtypes(s)) corresponds to the most general vector of classes that conforms to the argument types of the signature being checked.* The cone below this vector represents all class vectors that conform to the signature's argument types; the vectors in this cone are of interest because they are exactly the vectors that can be arguments of a message matching the signature. As an example, we have plotted four other class vectors representing the specializers of the methods that are covered by the signature. The cone below each specializer vector represents the class argument vectors that inherit that method.

Given this mountain-top diagram, the set of methods is complete with respect to the signature if there are no vectors of concrete classes in the region labeled as potentially incomplete. If such a vector existed, then it would be considered legal from the perspective of the signature but have no method implementation that it inherited. Similarly, a set of methods is consistent if there are no vectors of concrete classes in the region labeled as potentially inconsistent. If such a vector existed, then more than one method would be inherited by the vector but no single method would be most specific. The other regions under the signature's argtypes cone are completely and consistently implemented. The goal of the typechecking algorithm is to check for each signature whether there exist any vectors of concrete classes in either the incomplete or the inconsistent regions of the signature.

The following declarations exemplify the need for each of these three kinds of checks:

```
-- interface:
type num;
type int subtypes num;
type fraction subtypes num;
signature +(num,num):num;
signature +(int,int):int;
signature +(fraction,fraction):fraction;
-- implementation:
abstract representation num_rep conforms num;
template representation int_rep conforms int inherits num_rep;
template representation fraction_rep conforms fraction inherits num_rep;
template representation float_rep conforms fraction inherits num_rep;
implementation +(x@int_rep:int, y@int_rep:int):int {...}
implementation +(x@float_rep:fraction, y@float_rep:fraction):num {...}
implementation +(x@num_rep:num, y@fraction_rep:fraction):num {...}
implementation +(x@fraction_rep:fraction, y@num_rep:num):num {...}
```

This implementation fails all three criteria for type-correctness with respect to the interface. The + method given for two `float_rep` objects does not conform to the signature `+(fraction,fraction):fraction`, since its result type `num` is not a subtype of `fraction`. The implementations are incomplete, since addition for an `int_rep` object and a `float_rep` object, in either order, is not implemented. Finally, the implementations are inconsistent, since when adding two `fraction_rep` objects, two + methods apply but neither overrides the other. If these problems were corrected, then the implementations would become type-correct. In particular, because `num_rep` is abstract, no incompleteness results from not implementing addition of two `num_rep` objects.

* For simplicity in the two-dimensional diagram, we are assuming that there is one such most general vector of classes that corresponds to the argument types of the signature. In general this may not be true, but our algorithm does not depend on this assumption.

For the most part, conformance can be checked for each method declaration separately, similarly to the kinds of method interface checks that occur in other statically-typed object-oriented languages, although separating subtyping from inheritance introduces a subtlety that requires special care. Completeness and consistency must be checked globally, considering the combination of methods that together implement some signature. The requirement for a more global view for typechecking completeness and consistency stems from the presence of multi-methods, abstract classes, and the separation of code inheritance and subtyping.

There is no need to check explicitly that a type really is a subtype of all its declared supertypes. Our algorithm uses the declared conformance and subtyping relationships to determine which methods must be implemented for which classes. If these checks pass, then the declared conformance and subtyping relationships are structurally correct. (Of course, since the type system has no knowledge of specifications, it cannot guarantee the correctness of behavioral subtyping claims [America 87, America & van der Linden 90, Leavens 91, Leavens & Weihl 90].)

The following theorem states that the checks that constitute the parts of our algorithm are sufficient to ensure that the implementation typechecks. (As discussed in section 4.1, this is not sufficient to guarantee program type correctness, since the client-side checks also must be satisfied.)

Theorem 1. (ImplementationIsConforming $\wedge$ ImplementationIsComplete $\wedge$ ImplementationIsConsistent) $\Rightarrow$ ImplementationTypechecks

Proof sketch. Suppose, for the sake of contradiction, that ImplementationTypechecks is false but ImplementationIsConforming, ImplementationIsComplete, and ImplementationIsConsistent are all true. By definition of ImplementationTypechecks, there must be some signature, *s*, and some vector of concrete classes, $\vec{c}$, that conforms to the argument types of *s*, such that there is no method that satisfies the properties listed in the definition of ImplementationTypechecks. This can happen if the set of all applicable methods is empty, but this contradicts the assumption that ImplementationIsComplete is true. This also can happen if there is no single greatest lower bound in the set of applicable methods, but this contradicts the assumption that ImplementationIsConsistent is true. Finally, ImplementationTypechecks could fail if the greatest lower bound, *m*, in the set of applicable methods, has argument or result type declarations that are not compatible with the signature *s*, but this contradicts the assumption that ImplementationIsConforming is true. Hence the original assumption must be false and the theorem must hold.

(A formal, symbolic proof for this and all other theorems is contained in Appendix A.)

The next three subsections present algorithms that implement each part of the problem breakdown efficiently.

4.4 Checking Conformance

To check conformance, our algorithm considers each method in the program in turn. For each method, and for each signature such that the method could be invoked as a result of a call declared type-correct by the signature, we first verify the following two conditions:

- the type of each of the method's unspecialized formals must be a supertype of the corresponding type of the signature, and
- the result type of the method must be a subtype of the signature's result type.

This pair of checks is the standard contravariant rule for subtyping of functions [Cardelli 88, Cardelli & Wegner 85], restricted to unspecialized formals.

The final conformance check tests constrained formals. For each specialized formal, we need to ensure that every class of actual argument that might be passed to the method conforms to the declared type of the formal; because the formal is specialized, only classes that inherit from the specializing class need to be considered. As a first approximation, the typechecker could check that the formal's specializer class conforms to the formal's declared type. However, this check is insufficient: since subtyping and inheritance are independent, some class could inherit from the specializer class without conforming to the same set of types as the specializer class, in particular, without conforming to the declared type of the specialized formal. Consider the following example:*

```
type bag;
signature add(bag, int):void; -- allow duplicates
template representation bag_rep conforms bag;
method add(b@bag_rep:bag, x@any:int):void { ... } -- could create duplicates
type set; -- not a subtype of bag; add has incompatible specification
signature add(set, int):void; -- disallow diplicates
template representation set_rep inherits bag_rep conforms set;
... add(new set_rep, 3) ... -- violates conformance!
```

In the example above, `set_rep` inherits the `add` method, but `set_rep` objects do not conform to the `bag` type expected by the `add` method, and hence this implementation of the `bag` and `set` types is incorrect. If the error were not detected, then the no-duplicates property of sets could be broken, or run-time type errors could result if the `add` method sends messages to its `b` argument that are not understood by `set_rep` objects.

In our model, there are two ways to correct the type error. One way would be to use a more general type for the specialized formal of the `add` method for `bag_rep` objects, one that expressed the minimal requirements needed by the `add` method's body and to which both `bag_rep` and `set_rep` conformed:

```
type unordered_collection;
type bag subtypes unordered_collection;
signature add(bag, int):void; -- allow duplicates
template representation bag_rep conforms bag;
method add(b@bag_rep:unordered_collection, x@any:int):void { ... }
type set subtypes unordered_collection;
signature add(set, int):void;
template representation set_rep inherits bag_rep conforms set;
... add(new set_rep, 3) ... -- type-correct
```

This new implementation allows any class to inherit from `bag_rep` as long as the subclass conformed to `unordered_collection`. This does require some care when writing methods, however: the programmer should use types for specialized formals that are as general as possible, so as to enable future inheritance of code for subclasses that are not also subtypes.

Alternatively, our model allows the programmer to override the offending method for those subclasses that do not conform to the method's argument type declarations:

```
type bag;
signature add(bag, int):void; -- allow duplicates
template representation bag_rep conforms bag;
method add(b@bag_rep:bag, x@any:int):void { ... } -- could create duplicates
type set; -- not a subtype of bag; add has incompatible specification
signature add(set, int):void; -- disallow diplicates
template representation set_rep inherits bag_rep conforms set;
method add(s@set_rep:set, x@any:int):void { ... } -- no duplicates
... add(new set_rep, 3) ... -- type-correct
```

In this version, the `bag_rep` version of `add` is not obligated to work for `set_rep` objects, since it is overridden by a different method for `set_rep` objects. In general, overriding in this way can always be

* **new** is assumed to be a primitive function to instantiate a class.

used to overcome a problem with conformance. One would like the `set_rep` version of `add` to be implemented in terms of the `bag_rep` version of `add`, after checking that the element is not already present (Cecil has a `resend` construct to invoke an overridden method, analogous to Smalltalk's `super` construct). Unfortunately, such a resend is not type-correct: its argument, of type `set`, cannot be passed to `bag_rep`'s `add` method, which expects the incomparable type `bag`. So overriding is only useful for method implementations that are completely independent of the overridden methods.

Our conformance checking algorithm is as follows:

ComputeIsConforming ≡
  ComputeUnspecializedAndResultConform $\wedge$ ComputeSpecializedAreConforming

ComputeUnspecializedAndResultConform ≡
  $\forall\, m \in M.\ \forall\, s \in$ relevant-sigs(m, S).
    has-common-classes(m, argtypes(s)) $\Rightarrow$ contra-unspec-args-co-result(m, s)

ComputeSpecializedAreConforming ≡
  $\forall\, m \in M.$
    ($\forall\, i \in$ indexes(specializers(m)).
      specializers$(m)_i \neq any \Rightarrow$ specializers$(m)_i$ <: argtypes$(m)_i$) $\wedge$
    top-non-overridden-non-conforming-class-vecs(m) = $\varnothing$

where:

relevant-sigs(m, Ss) ≡ $\{ s \in Ss \mid$ msg(m) = msg(s) $\wedge$ |argtypes(s)| = |argtypes(m)| $\}$

has-common-classes($m, \vec{t}$) ≡
  let $TCSs = \{ \vec{c} \mid c_i \in$ top-concrete-conforming-subclasses(specializers$(m)_i$, $\vec{t}_i$) $\wedge$
    $i \in$ indexes(specializers(m)) $\}$ **in**
  $\exists \vec{c} \in TCSs$.
    ($\neg \exists\, m_1 \in M$. msg(m_1) = msg(m) $\wedge$ |argtypes(m_1)| = |argtypes(m)| $\wedge$
      $\vec{c} \leq_{inh}$ specializers(m_1) $\wedge\ m_1 \leq_{meth} m \wedge m_1 \neq m$)

top-concrete-conforming-subclasses(c, t) ≡
  top-classes(concrete-conforming-subclasses(c, t))

concrete-conforming-subclasses(c, t) ≡ $\{ c' \in C_{concrete} \mid c' \leq_{inh} c \wedge c' <: t \}$

top-classes(Cs) ≡ $\{ c \in Cs \mid \forall\, c' \in Cs.\ c' \neq c \Rightarrow \neg (c \leq_{inh} c') \}$

contra-unspec-args-co-result(m, s) ≡
  ($\forall\, i \in$ indexes(specializers(m)).
    specializers$(m)_i = any \Rightarrow$ argtypes$(s)_i \leq_{sub}$ argtypes$(m)_i$) $\wedge$
  restype(m) $\leq_{sub}$ restype(s)

top-non-overridden-non-conforming-class-vecs(m) ≡
  $\{ \vec{c} \in$ top-non-conforming-class-vecs(specializers(m), argtypes(m)) |
    $\neg \exists\, m' \in M$.
      msg(m') = msg(m) $\wedge$ |argtypes(m')| = |argtypes(m)| $\wedge\ m' \leq_{meth} m \wedge m' \neq m \wedge$
      $\vec{c} \leq_{inh}$ specializers(m') $\}$

top-non-conforming-class-vecs($\vec{c}$, $\vec{t}$) ≡
$\{ \vec{c}\,' \in C^* \mid |\vec{c}\,'| = |\vec{c}| \wedge$
$\quad \forall\, i \in$ indexes($\vec{c}$).
$\quad\quad (c_i = any \;\Rightarrow\; c_i' = c_i) \;\wedge$
$\quad\quad (c_i \neq any \;\Rightarrow$
$\quad\quad\quad$ **let** Cs = top-non-conforming-classes(c_i, t_i) **in**
$\quad\quad\quad\quad (Cs = \emptyset \;\Rightarrow\; c_i' = c_i) \;\wedge$
$\quad\quad\quad\quad (Cs \neq \emptyset \;\Rightarrow\; c_i' \in Cs)) \} - \{ \vec{c} \}$

top-non-conforming-classes(c, t) ≡
top-classes($\{ c' \in C_{concrete} \mid c' \leq_{inh} c \;\wedge\; \neg\,(c' <: t) \}$)

To check unspecialized formals and results, the algorithm checks each method against each covering signature. To check specialized formals, the algorithm searches for potential conformance errors of each method. The algorithm locates the most general concrete classes that inherit from each of the method's specializers but do not conform to the declared argument type of the method (top-non-conforming-classes), forms all combinations of these potentially illegal argument classes (top-non-conforming-class-vecs), and then filters out any argument class vectors that invoke a more specific method (top-non-overridden-non-conforming-class-vecs). If any vectors remain, then the method has a conformance error. The specializers themselves are treated as a special case: they are checked explicitly for conformance to the method's argument types so that top-non-conforming-class-vecs can be restricted to vectors that have at least one proper (and therefore non-conforming) subclass of the corresponding specializer.

Theorem 2. ComputeIsConforming ⇒ ImplementationIsConforming

Proof sketch. To show that the algorithm correctly computes whether the set of methods conforms to the set of signatures, assume for the sake of contradiction that there exists a vector of concrete classes that conforms to the argument types of some signature but that invokes a method where either the vector of concrete classes does not conform to the method's argument types or the method returns a result that is not a subtype of the signature's result type. First, assume that the method result type is wrong. But this is contradicted by the algorithm's ComputeUnspecializedAndResultConform test. Second, assume that one of the concrete argument classes does not conform to the declared type of one of the method's unspecialized formals. But this is again contradicted by the ComputeUnspecializedAndResultConform test: since the concrete class conforms to the signature's argument type, and the algorithm ensures that the signature's argument type is a subtype of the method's argument type, the concrete class must therefore conform to the method's argument type. Finally, assume that one of the concrete argument classes does not conform to the declared type of one of the method's specialized arguments. Then the argument class must either be a member of top-non-conforming-classes, or there must exist a concrete superclass of the argument class that also leads to a conformance error. Without loss of generality, we assume that the non-conforming concrete argument class being considered does not have any non-conforming concrete superclasses that also inherit from the method's specializer class; this class is a member of top-non-conforming-classes. However, we assumed that this vector of top non-conforming concrete classes invoked the method in question, not some more specific method, but the algorithm verified that there are no such vectors of classes, by checking that top-non-overridden-non-conforming-class-vecs was empty and that the specializers themselves all conform to the declared argument types. Hence the assumption that there exists a vector of concrete classes that illustrates that a method does not conform to a signature must be wrong.

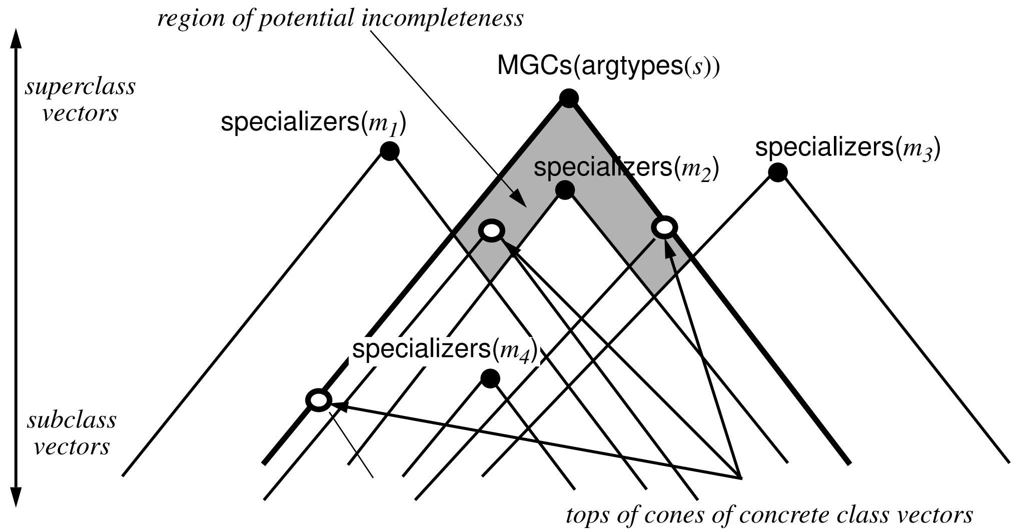
Complexity. (In our complexity analyses, we assume that basic operations over the subtyping and inheritance partial orders, such as testing whether one element is less than another element or finding lower bounds, can be performed in constant time. Various sources have described efficient data structures and algorithms for testing subtyping in a lattice and for testing whether there exists a common descendant of two members of a partial order [e.g. Caseau 93, Agrawal *et al.* 91].)

Prior to typechecking, we precompute for each class the partial order of its concrete subclasses. With this set-up, the worst-case time complexity of computing top-concrete-conforming-subclasses(c, t) is $O(|C_{concrete}|)$. The worst-case complexity of has-common-classes(m, $\grave{t}$) is $O(|C_{concrete}|^k \cdot |M|)$, which would only occur when the class inheritance order is flat; we would expect this to be more like $O(|C_{concrete}| \cdot |M|)$ in practice. So checking ComputeUnspecializedAndResultConform requires time $O(|C_{concrete}|^k \cdot |M|^2 \cdot |S|)$ in the worst-case; but we expect $O(|C_{concrete}| \cdot |M|^2 \cdot |S|)$. Computing top-non-conforming-classes takes $O(|C_{concrete}|)$, using a downwards topological traversal of the inheritance graph to locate the top non-conforming classes in a single pass. In the worst case, there can be $|C_{concrete}|$ such classes, which occurs if the inheritance graph is flat and no classes conform to the method's declared argument type; we would expect that in practice inheritance without subtyping would be relatively rare, and so the number of top-non-conforming-classes would be a small fraction of $|C_{concrete}|$. Computing top-non-conforming-class-vecs depends directly on the number of top-non-conforming-classes, requiring $O(|C_{concrete}|^k)$ time in the worst case, where k is, again the maximum number of arguments to a method. (Recall that we assume k is bounded by a constant.) By precomputing a partial ordering over methods based on the method overriding relation, the time required for top-non-overridden-non-conforming-class-vecs is $O(|M| \cdot |C_{concrete}|^k)$ in the worst case, but more like $O(|M|)$ in practice. Consequently, the time to compute ComputeSpecializedAreConforming is $O(|M|^2 \cdot |C_{concrete}|^k)$ in the worst case, leading to an overall worst-case time for ComputeIsConforming of $O(|M|^2 \cdot |C_{concrete}|^k \cdot |S|)$. However, we believe that in practice the algorithm would have a performance on the order of $O(|M|^2 \cdot |C_{concrete}| \cdot |S|)$.

If the pruning based on overridden methods is omitted, then conformance checking is much faster. ComputeSpecializedAreConforming would require only constant time to determine whether a method has a conformance error, by precomputing the most specific type to which each class and its subclasses conform and then testing this type against the declared type of each specialized formal. This simpler version of ComputeIsConforming would require only $O(|M| \cdot (|S|+|T|))$ time overall [Chambers & Leavens 94].

4.5 Checking Completeness

To check the completeness of a set of method implementations with respect to a signature, we first compute the set of concrete class vectors that are the tops of those cones that conform to the argument types of the signature, i.e., the set of concrete class vectors that conform to the argument types of the signature and do not inherit from any other such vectors. For each member of this set, we verify that there exists a method inherited by this vector. The following diagram illustrates the check, showing with open circles the three top concrete class vectors that conform to the argument types of the signature. The two tops in the region of potential incompleteness are flagged as errors.



Our algorithm iterates over all signatures, verifying completeness for each signature. As described in precise notation below, to check the completeness of a set of method implementations *M* with respect to a set of class vectors *Cs* and a signature *s*, our algorithm first locates the set of concrete class vectors *TCSs* that are the tops of those that both inherit from a member of *Cs* and conform to the argument types of *s*.* It then verifies that each member of *TCSs* inherits a method in *M*.

ComputeIsComplete ≡
 $\forall\, s \in S.$
  let *any-vector* = $\{\ \vec{c}\ \}$ where $|\vec{c}| = |\text{argtypes}(s)|$ and each $c_i = any$ **in**
   IsComplete(relevant(*M, s*), *any-vector, s*)

where:

IsComplete(*M, Cs, s*) ≡
 $\forall\ \vec{c} \in Cs.$
  let $TCSs = \{\ \vec{c}\,'\ |\ c_i' \in \text{top-concrete-conforming-subclasses}(c_i, \text{argtypes}(s)_i) \wedge i \in \text{indexes}(\vec{c})\ \}$ **in**
   $\forall\ \vec{c}\,' \in TCSs.\ \exists\, m \in M.\ \vec{c}\,' \leq_{inh} \text{specializers}(m)$

relevant(*M*s, *s*) ≡ $\{\ m \in Ms\ |\ \text{msg}(m) = \text{msg}(s)\ \wedge\ |\text{argtypes}(m)| = |\text{argtypes}(s)|\ \}$

Theorem 3. ComputeIsComplete ⇒ ImplementationIsComplete

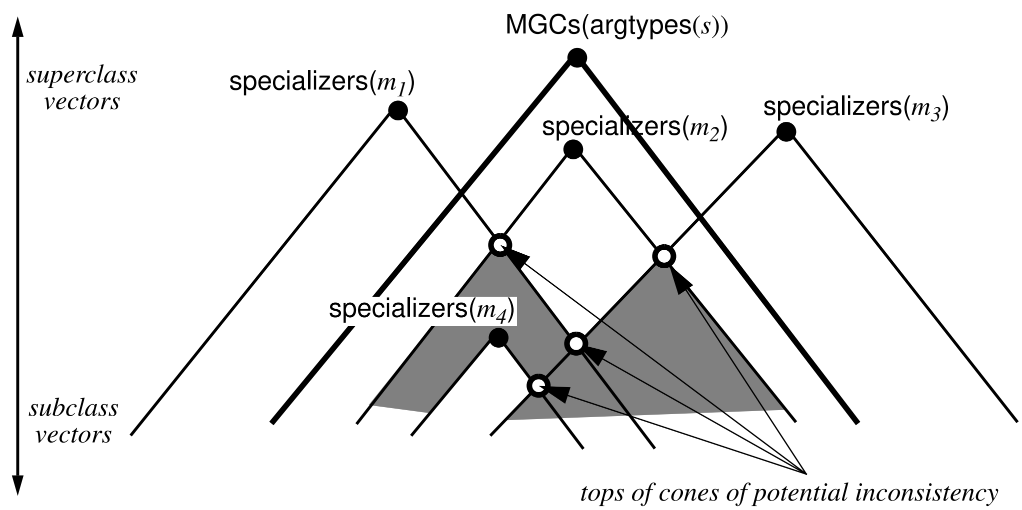
Proof sketch. To show that this algorithm correctly detects incompleteness in a set of method implementations with respect to a signature, assume for the sake of contradiction that the algorithm reports that the methods are complete but that they really are incomplete. Then there must exist a vector of concrete classes which conforms to the argument types of the signature but does not inherit a method (by definition of incompleteness). This class vector must inherit from at least one of the top vectors computed above (by definition of top). However, each of these top vectors has been verified to inherit at least one method (by assumption that the check was successful), and this method must therefore be inherited by the concrete class vector (by definition of inheritance). Hence the assumption that the system was incomplete must be wrong.

* IsComplete is more general than needed for ComputeIsComplete because it will be reused as part of consistency checking.

Complexity. Prior to typechecking, we precompute for each class the partial order of its concrete subclasses. With this set-up, the time complexity of computing top-concrete-conforming-subclasses(c, t) is $O(|C_{concrete}|)$. All methods are checked against each of the top concrete class vectors, leading to an overall time complexity for IsComplete(M, Cs, s) of $O(|M| \cdot |Cs| \cdot |C_{concrete}|)$ and for the entire ComputeIsComplete algorithm of $O(|S| \cdot |M| \cdot |C_{concrete}|)$.* In practice, we believe this algorithm can be sped up by checking all signatures with the same name in a single pass.

4.6 Checking Consistency

To check the consistency of a set of method implementations with respect to a signature, we need to show the absence of any regions of potential inconsistency where two method implementations are inherited by a vector of concrete classes without an intervening method resolving the ambiguity. Our algorithm tackles this problem by first computing the set of all pairs of mutually incomparable method implementations (i.e., all pairs of methods where neither method overrides the other). This set defines all those pairs of methods that have the potential to be mutually ambiguous. For each pair, we then construct the set of class vectors that are the tops of the lower bounds of the argument specializers of the two methods, i.e., the set of class vectors that inherit from both specializer class vectors and are not overridden by any other such vectors. Each of these vectors is the root of a cone of potential inconsistency. The following diagram highlights with open circles the four top lower bounds constructed from the four incomparable combinations of methods from the earlier diagram:



We wish to determine whether there exists a concrete argument vector in any of these cones that does not inherit some other method resolving the ambiguity. To help us solve this problem, we observe that determining the absence of concrete class vectors in a region of potential inconsistency is similar to the problem of determining the absence of concrete class vectors in a region of potential incompleteness. Accordingly, for each pair of incomparable methods, our algorithm first constructs a new set of methods comprised of those methods in the original set that override both of the two incomparable methods, and then it tests for completeness of this reduced set of methods with respect to the set of top lower bound class vectors constructed above. If this subgraph is complete, then the two mutually-ambiguous methods are not a source of inconsistency. In precise notation:

ComputeIsConsistent $\equiv$
 $\forall\, s \in S.$ IsConsistent(relevant(M, s), s)

* $|Cs| = 1$ for ComputeIsComplete.

To check the consistency of a set of method implementations M with respect to a signature s, we first compute the set MP of all pairs of incomparable methods in M. For each pair (m_1, m_2) in MP, we construct the set of class vectors $TLBs$ that are the tops of the lower bounds of the argument specializers of m_1 and m_2.* We then construct the set of methods $M\text{-}reduced$ that override both m_1 and m_2. Finally, we test for completeness of $M\text{-}reduced$ with respect to $TLBs$ and s.

$$
\begin{aligned}
&\mathsf{IsConsistent}(M, s) \equiv \\
&\quad \forall\, (m_1, m_2) \in \mathsf{incomparable\text{-}pairs}(M). \\
&\qquad \mathbf{let}\ \ TLBs = \mathsf{tlb}(\mathsf{specializers}(m_1), \mathsf{specializers}(m_2), s), \\
&\qquad\qquad M\text{-}reduced = \{\, m \in M \mid m \le_{meth} m_1 \wedge m \le_{meth} m_2 \,\}\ \mathbf{in} \\
&\qquad\qquad\quad \mathsf{IsComplete}(M\text{-}reduced, TLBs, s)
\end{aligned}
$$

where

$$\mathsf{incomparable\text{-}pairs}(M) \equiv \{\, (m_1, m_2) \in M{\times}M \mid \neg(m_1 \le_{meth} m_2) \wedge \neg(m_2 \le_{meth} m_1) \,\}$$

$$\mathsf{tlb}(\vec{c}, \vec{c}\,', s) \equiv \mathsf{top\text{-}classes}(\mathsf{lb}(\vec{c}, \vec{c}\,', s))$$

$$\mathsf{lb}(\vec{c}, \vec{c}\,', s) \equiv \{\, \vec{c}\,'' \mid \vec{c}\,'' \le_{inh} \vec{c} \wedge \vec{c}\,'' \le_{inh} \vec{c}\,' \wedge \vec{c}\,'' <: \mathsf{argtypes}(s) \,\}$$

Theorem 4. ComputeIsConsistent $\Rightarrow$ ImplementationIsConsistent

Proof sketch. To show that our algorithm correctly detects inconsistencies in a set of methods with respect to a signature, assume for the sake of contradiction that the algorithm reports success but that the methods really are inconsistent. Then there must exist a vector of concrete classes that conforms to the argument types of the signature but inherits no single most specific method implementation (by definition of inconsistency). This vector must inherit at least two methods that are mutually unordered but are not overridden by a third method that is inherited by the concrete class vector (by definition of "inheriting no single most specific method"). The concrete class vector must inherit from a vector that is a top lower bound of the specializers of the two methods (by definition of top). But there are no concrete class vectors that inherit from this top class vector that do not also inherit from some other method that overrides the two mutually-ambiguous methods (by the definition of completeness of the subgraph). Hence the original assumption of inconsistency must be wrong.

Complexity. There can be at most $O(|M|^2)$ incompatible pairs, each requiring constant time to produce. This worst-case scenario occurs with completely flat sets of method implementations. There can be at most $O(|C|^k)$ top lower bounds of each of these pairs, where k is the number of arguments of the message, each computed in constant time; since we assume k is bounded by a constant, this term is polynomial. This worst case scenario is quite unlikely in practice, however, and can only occur in a program with many type errors; we would expect a small constant number of top lower bounds in normal programs with few type errors. The reduced method set can be constructed in $O(|M|)$ time, and its size can be on the same order as M; note that, if there are $O(|M|^2)$ incompatible pairs, then the size of $M\text{-}reduced$ will be a small constant, so we are being overly conservative by assuming it always is of size $O(|M|)$. Therefore, each call to IsComplete(M-$reduced$, $TLBs$, s) can require time $O(|M| \cdot |C|^k \cdot |C_{concrete}|)$ (although we would expect something more like $O(|M|)$ in practice). This leads to a worst-case time complexity for IsConsistent(M, s) of $O(|M|^3 \cdot (|C|^{2k} + |M|) \cdot |C_{concrete}|)$, although $O(|M|^4 \cdot |C_{concrete}|)$ is a more likely worst-case time in practice. The overall time for consistency checking of an entire program, ComputeIsConsistent, is therefore $O(|S| \cdot |M|^3 \cdot (|C|^{2k} + |M|) \cdot |C_{concrete}|)$ in the worst case and $O(|S| \cdot |M|^4 \cdot |C_{concrete}|)$ in more reasonable expected cases. As with completeness checking, we suspect that checking all signatures with the same name in one pass will lead to faster typechecking in practice.

* We do not require the object inheritance partial order to be a downward semi-lattice, and so we cannot assume g.l.b.'s of argument specializers exist. For our algorithm, g.l.b.'s are not required. We only need the set of "top lower bounds," i.e., those lower bounds that are not strictly less than any other lower bound.

4.7 Algorithm Summary

Implementation-side typechecking ensures that the class inheritance graph and the method implementations correctly implement the interface defined by the type graph and the signatures. We broke the problem of implementation-side typechecking into three components: checking conformance, checking completeness, and checking consistency. Solving these three subproblems is sufficient for showing that the implementation is correct (Theorem 1). For each of these subproblems, an algorithm was presented and shown to imply its specification.

Checking conformance of all method implementations against all signatures requires worst-case time of $O(|M| \cdot (|M| \cdot |C_{concrete}|^k + |S|))$ (where k, the maximum number of message arguments, is bounded by a constant), if inheritance without subtyping and shallow inheritance hierarchies are commonplace. We expect that in practice the performance would be on the order of $O(|M| \cdot (|M| + |S|))$. (A simpler but stricter version of conformance checking would require only $O(|M| \cdot |S|)$ time.) Checking completeness of all method implementations against all signatures requires $O(|S| \cdot |M| \cdot |C_{concrete}|)$. Checking consistency of all method implementations against all signatures requires $O(|S| \cdot |M|^3 \cdot (|C|^{2k} + |M|) \cdot |C_{concrete}|)$ in contrived worst-case scenarios and $O(|S| \cdot |M|^4 \cdot |C_{concrete}|)$ in more realistic situations.

Although the worst-case running time of our algorithm is worse than simply executing the specification given for ImpementationTypeChecks, which has a worst-case running time of $O(|S| \cdot |C_{concrete}|^k \cdot |M|)$, we believe that the the elimination of the factor $|C_{concrete}|^k$ in practical cases is worthwhile. We leave the checking of this belief for future work.

4.8 Discussion

The independence of inheritance and subtyping has a major impact on our algorithm. In conformance checking of a specialized formal, our algorithm seeks out inheriting subclasses that do not conform to the declared type of the formal and that do not invoke more specific methods. If inheritance and subtyping were joined, then the checking of specialized formals would be trivial, requiring only constant time. Moreover, a consequence of our type system is that the addition of a new subclass can create type errors in methods inherited by the subclass, if the new subclass does not conform to some specialized formal's argument type. Our type system allows the use of overriding to remove these type errors, but this solution adds implementation complexity and may not be considered desirable from a language design perspective.* These issues are not limited to multi-method-based languages, however. It seems that the combination of allowing inheritance without subtyping and avoiding retypechecking of inherited methods in subclasses leads to a type system with these properties, whether or not multi-methods are present. For example, the TOOPLE singly-dispatched language places restrictions on inheritance to avoid conformance errors [Bruce *et al.* 93]. TOOPLE includes a ClassType that represents the interface to "self" within a method; subclasses are required to preserve this interface. In our type system, there is no central ClassType, but instead each method describes its own local constraints on the types of all its specialized arguments, in a decentralized fashion more appropriate to multi-methods.

During checking of completeness and consistency, our algorithm deals with the independence of subtyping and code inheritance by passing the signature being checked to all the various subproblems. Each of these subproblems restricts the set of classes under consideration to those that also conform to the appropriate argument type of the signature. This has the same effect as producing a new class and inheritance graph containing only those classes that conform to the signature, and then processing this reduced graph as if inheritance and subtyping were the same.

* It is easy to omit this feature from our type system. In fact, earlier versions of this work did exactly that [Chambers & Leavens 94].

Our programming language model also distinguishes abstract and concrete classes. This distinction appears in the completeness and consistency checking algorithms where the tops of the set of vectors of concrete classes are calculated from a vector of (potentially abstract) classes. We feel that handling this distinction in the typechecking algorithm is of crucial importance in being able to typecheck realistic programs. Our current body of Cecil code includes more than 250 abstract classes, nearly a third of all classes, and virtually all of the abstract classes would be rejected as incompletely implemented if our algorithm did not treat them specially.

We allow each multi-method to decide independently which formals are specialized and which are not; multi-methods are completely independent and not restricted by a "congruent lambda list" rule as are CLOS multi-methods. This flexibility also allows our language model to include singly-dispatched languages as a special case, enabling more direct comparisons of type systems. Mixing specialized and unspecialized formals is fairly easy to accommodate in our algorithm. Unspecialized formals are modeled as specialized on a class any that is a superclass of all other classes. During conformance checking, unspecialized formals are checked against signatures using normal contravariant rules, while specialized formals are checked independently of covering signatures. Completeness and consistency checking are unaffected by the difference between specialized and unspecialized formals.

5 Modules

Object-oriented methods encourage programmers to develop reusable libraries of code. However, multi-methods can pose obstacles to smoothly integrating code that was developed independently. Unlike with singly-dispatched systems, if two classes that subclass a common superclass are combined in a program, it is possible for incompleteness or inconsistency to result where none occurs with either subclass alone. The additional expressiveness and flexibility of multi-methods creates new pitfalls for integration.

Standard module systems, such as the Common Lisp package system, help to manage the global name space, and in some circumstances the name hiding they provide can serve to avoid integration problems. But Common Lisp packages do not allow a CLOS multi-method to be added to a global generic function within a particular package, without exposing the presence of the multi-method to all invokers of the generic function.* As CLOS resolves method ambiguities automatically, independently-developed CLOS packages can work in isolation but silently fail to give correct results when combined. No prior module system for a multi-method language allows a library module to be certified as free of static type errors, independently of its use in a program.

Encapsulation and modularity of multi-methods is a related problem. To support careful reasoning and to ease maintenance, a data structure's implementation may be encapsulated [Parnas71, Parnas72, Liskov & Zilles 74]. But previous multi-method languages do not provide the same support for encapsulation as abstract data type-based languages such as CLU [Liskov *et al.* 77, Liskov *et al.* 81] or singly-dispatched object-oriented languages such as C++ and even Smalltalk. In ADT-based or singly-dispatched languages, direct access to an object's representation can be limited to a statically-determined region of the program. An earlier approach to encapsulation in Cecil suffered from the problem that privileged access could always be gained by writing methods that specialized on the desired data structures [Chambers 92].

We have developed a module system for Cecil that addresses these shortcomings of existing multi-method languages. This system can restrict access to parts of an implementation to a statically-determined region of program text while preserving the flexibility of multi-methods. Individual modules can be reasoned about and typechecked in isolation from modules not explicitly imported. Modules can *extend* existing modules with subclasses, subtypes, and augmenting multi-methods. If any conflicts arise between independent

* CLOS does allow an entire generic function to be private to a single package, but CLOS does not support generic functions whose member multi-methods have different visibilities.

extensions, they are resolved through *resolving modules* that extend each of the conflicting modules. A simple check for the presence of the necessary resolving modules is all that is needed at link-time to guarantee type safety.

In this paper we present our module system informally. We defer a more formal treatment of modules and multi-methods to a future paper.

5.1 Module Basics

The core of our module system provides standard name space management, as in Modula-2 [Wirth 88]. Like Common Lisp and Oberon-2, we do not tie the module notion to the notion of classes or types [Szyperski 92]. A program is a sequence of one or more modules, one of which is called `Main`. Each module contains a group of declarations; there is no code that appears outside of a module, and for simplicity modules do not nest. The declarations in a module are tagged `public` (the default) or `private`. A module may explicitly import another module, which has the effect of making the imported module's public declarations visible in the importing module. Private declarations are encapsulated within a module and are invisible to other modules.* Import declarations themselves can be tagged `public` or `private`. The declarations imported through a public import declaration are visible in the module's public interface, while declarations imported through a private import declaration are hidden from clients.

We illustrate the core of our module system with the following example:

```
module Complex {
  type complex subtypes num;
  signature +(complex, complex):complex;
  method new_complex(x:real, y:real):complex {...}
}
module Main {
  private import Complex;
  method main():void {
    let c := new_complex(3.5, 4.25);
    ... }
}
```

The visibility of declarations determines the set of method implementations considered during method lookup. All declarations visible at the call site, either by being declared in the current module or by being imported as a public declaration from another module (potentially through a chain of public imports declarations), are considered in effect for the purposes of resolving method lookup. All other declarations are invisible and do not affect method lookup. This guarantees that unrelated code, even code that defines methods with the same name as the message being sent, has no effect on method lookup and can be ignored when reasoning about the behavior of the program or when statically typechecking it. The scope of a private declaration is limited to the enclosing module, and consequently no other module can be affected by a private declaration.

Using the sending scope to determine the set of potentially callable methods allows a module to extend and customize imported types and representations without affecting unrelated modules or requiring changes to the source code of imported modules [Hölzle 93, Harrison & Ossher 93]. For example, a text-processing module can add tab-expansion behavior to string data structures without polluting the general interface to strings as seen by unrelated modules. This local extension feature of multi-methods resolves a tension observed in singly-dispatched languages of whether to add functionality as operations within the class or external to the class.

* Our module system also includes a notion of explicitly-named friend modules which are able to access the private declarations of a module, much as in C++.

To typecheck a program, each module in the program is typechecked separately. Typechecking a module involves performing both client-side typechecks of the expressions in the module and implementation-side typechecks of conformance, completeness, and consistency, with respect to the declarations in the current module and the public declarations of any explicitly imported modules. Because each module can be typechecked independently, examining only a small portion of the declarations in a large program, typechecking can run much faster. Moreover, the public interface of each module can be typechecked in isolation, allowing the compiler to assume that each module's public interface is type-correct when typechecking modules that import it, potentially speeding typechecking further.

5.2 Subtyping and Extensions of Modules

Unfortunately, subtyping creates a problem for the basic module design presented above. Consider the following example in which a `CartComplex` module implements the `complex` type:*

```
module Complex {
  type complex subtypes num;
  signature +(complex, complex):complex;
}
module CartComplex {
  import Complex;
  template representation cartesian conforms complex;
  private field x(c@cartesian:cartesian):real;
  private field y(c@cartesian:cartesian):real;
  method +(c1@cartesian:complex,c2@cartesian:complex):complex {
    new_cartesian(c1.x + c2.x, c1.y + c2.y) }
  method new_cartesian(r,i:real):complex {
    new cartesian(x:=r, y:=i) }
}
module Storage {
  import Complex;
  private import CartComplex;  -- hide this use of CartComplex from clients
  var c1, c2: complex;  -- variables visible to modules importing Storage
  method store() {
    c1 := new_cartesian(3.14, 15.9);
    c2 := new_cartesian(-2.5, 227.0);
  }
}
module Main {
  private import Storage;
  private import Complex;
  method main() {
    store();
    ... c1 + c2 ...;    -- message not understood!
  }
}
```

In this example, the method + for two cartesian objects is not visible where it is called in the `main` routine. The cartesian objects have "outrun" the scope of their methods, passing through the module `Storage` which hides its use of `CartComplex` from its clients. We could fix the problem by requiring `Main` to explicitly import `CartComplex`, but there is no particular reason that `Main` should know about that module. Alternatively, we could alter our visibility rules so that the set of potentially callable methods is based on the module that defines the dynamic classes of the argument objects rather than the sending module; this approach is effectively how singly-dispatched systems such as C++ and Smalltalk determine

* The **`field`** declaration introduces the Cecil equivalent of instance variables.

the operation to invoke. However, if the classes of the arguments of a multi-method are defined in separate modules, then these different perspectives on the set of available methods need to be reconciled somehow. Moreover, an object-centered approach would sacrifice the ability of the sending module to customize its view of the interfaces of the objects it manipulates.

The key insight underlying our solution to this problem is to observe that if the `Main` module imported the `CartComplex` module (and every other module that defined a class conforming to the `complex` type), then the appropriate implementations of the + signature would be visible at the call site. The trick is to adjust the visibility rules so that the declarations in `CartComplex` are considered visible at method-lookup time without requiring `Main` or `Complex` to explicitly list `CartComplex` or any other implementation of `complex` at program-definition time.

Our solution achieves this implicit importing of declarations through the notion of *extension modules*. If a module *m* declares a class or type that conforms or subtypes, respectively, from a type declared in another module *n*, then we require that *m* be defined as an extension of *n*. In the complex number example, `CartComplex` must be declared as an extension of `Complex`, since `cartesian` in `CartComplex` conforms to `complex` in `Complex`:

```
module Complex { ... }
module CartComplex extends Complex { ... }
```

For the purposes of determining which declarations are visible *dynamically* at message-lookup time, the public declarations in an extension module are imported automatically whenever the extended module is imported (either explicitly or recursively through additional layers of module extension). However, for the purposes of reasoning *statically* about code or typechecking clients such as `Main`, only the public interfaces of the explicitly imported modules need to be examined. For example, to statically typecheck the body of the `main` function, only the public interface of `Complex` needs to be considered; the presence (or absence) of `CartComplex` is irrelevant. This distinction preserves the ability to easily extend existing code without rewriting or even retypechecking clients. Typechecking the `CartComplex` module will ensure that the interface assumed by clients of `Complex` is conformingly, completely, and consistently implemented. This split between checking clients against explicitly imported interfaces and checking extensions of the interface resembles the “modularity” obtained by the use of legal subtyping in the verification of object-oriented languages with subtyping [Leavens & Weihl 90, Leavens 91].

To provide more control over the interface seen by extension modules, declarations in a module may be tagged `protected`. A `protected` declaration is not visible to clients that import the module explicitly, but it is visible to extension modules; in this respect it is analogous to the `protected` construct in C++. Extension modules automatically import the public and protected declarations of the module(s) they extend. For example, the `x` and `y` fields in `CartComplex` would probably be tagged `protected`, to allow future extensions of cartesian complex numbers access to the representation of cartesian complex numbers.

The extension mechanism, together with the restriction that a subtype of a type can only be defined in the same module or in an extension module of the module that defines the type being subtyped, fixes the problem of objects outrunning their methods while preserving the ability of each scope to extend and customize a set of methods. Furthermore, it does not require changes to existing modules when new extension modules are added to a program, and extension modules do not have to be considered when reasoning statically about a module.

5.3 Resolving Module Conflicts

Unfortunately, multi-methods create a final problem with this module design. Two independently-developed modules can extend a common module correctly in isolation by incompletely or inconsistently in combination. For example, consider writing a `PolarComplex` module with a different representation for complex numbers:

```
module PolarComplex extends Complex {
  template representation polar conforms complex;
  private field rho(c@polar:complex):real;
  private field theta(c@polar:complex):real;
  method +(c1@polar:complex, c2@polar:complex):complex {...}
  method new_polar(r,t:real):complex {
    new polar(rho:=r,theta:=t) }
}
```

If only one of the `CartComplex` or `PolarComplex` modules is linked into a program, then no conflicts arise. However, if both modules are used, then any variable of type `complex`, such as `c1` and `c2` in `Storage`, might hold an instance of either the `cartesian` or `polar` classes. When sending the + message in `main`, if at run-time `c1` was an instance of `cartesian` while `c2` was an instance of `polar`, then + will not be understood; the program is incomplete. But viewed independently, each module is type-correct.

To solve this problem, we impose a well-formedness condition on the set of modules comprising a program: for each module *m* in the program, there must exist a *single most-extending module n* which extends, directly or indirectly, all other modules that extend *m*; a module with no extensions is its own single most extending module. More precisely,

$$\begin{array}{l} \textsf{ProgramIsWellFormed} \equiv \\ \quad \forall\, m \in \mathit{Program}. \\ \quad\quad \exists\, n \in \mathit{Program}. \\ \quad\quad\quad n = \textsf{most-extending-module}(\{\, m' \mid m' \leq_{module} m \,\}) \end{array}$$

where

$$\begin{array}{l} m = \textsf{most-extending-module}(Ms) \;\equiv \\ \quad m \in Ms \;\wedge \\ \quad \forall\, m' \in Ms.\;\; m \leq_{module} m' \end{array}$$

and where $\leq_{module}$ is the reflexive, transitive closure of the module-extends relation and *Program* is the set of modules in the program. Such a most extending module will import all other extensions, statically witness all implementations of types declared in the modules, and consequently be responsible for resolving any ambiguities among the various extension modules.

In our running example, if neither or only one of `CartComplex` or `PolarComplex` is present, then the system of modules in this example is well-formed. However, when both are present, then there is no single most extending module for `Complex`. To combine the two representations of complex numbers into a single program, the programmer must also create a new *resolving module* that extends both:

```
module CPComplex extends CartComplex, PolarComplex {
  method +(c1@cartesian:complex, c2@polar:complex):complex {
    ... }
  method +(c1@polar:complex, c2@cartesian:complex):complex {
    c2 + c1 }
}
```

This module extends the two representations and adds the necessary "glue" methods to make the two representations interoperate. For the purposes of run-time method lookup, the declarations in this module are visible to any module that imports `Complex`, through the rules for extension modules. When the `CPComplex` module is typechecked, it will ensure that the combination of the two representations forms a conformant, complete, and consistent implementation of the `complex` type, again according to the normal rules for typechecking a module. By requiring such a most extending module that statically witnesses and checks all other extensions of a module, we guarantee that a complete program can have no message errors.

As the programmer combines independently-developed code into larger libraries, the programmer creates the necessary resolving modules. At link-time, the linker can test quickly for the existence of the necessary resolving modules. No typechecking is performed at link-time; resolving modules are written and typechecked independently during program development just like other modules. A programming environment could automatically create and typecheck any omitted resolving modules, reporting whenever new methods need to be written to eliminate incompleteness or inconsistency.

To summarize, by requiring the existence of single most extending modules which resolve incompleteness or inconsistency problems arising from the combination of independently-developed multi-methods, we ensure that there exist modules whose static checking ensures that the program has no message lookup errors. Checking for the existence of such modules must be done at link-time, but creating and typechecking the resolving modules can be done as part of normal program development.

6 Conclusions

The work presented in this paper targets problems that arise when large programs are constructed in languages based on multi-methods. To secure the benefits of static typechecking for multi-methods, we designed a flexible static type system and developed a supporting typechecking algorithm. We addressed a broader class of languages than previous work, including those that incorporate mutable state, separate subtyping and code inheritance, abstract classes, mixed specialized and unspecialized formals, and graph-based multi-method lookup semantics. Our algorithm breaks down the typechecking problem into client-side and implementation-side checking, then further subdivides implementation-side checking into conformance, completeness, and consistency checking. A key insight underlying our algorithm is that the space of concrete class vectors conforming to a signature can be divided into cone-shaped regions, where the correctness of the tops of the cones implies correctness of the class vectors contained in the cones.

To help organize programs with multi-methods, we designed a module system that enables portions of a program to be encapsulated within modules, protecting this code from unwanted external access and insulating clients from the details of the hidden code. Our design retains the advantages of multi-methods, including allowing clients to extend and customize an existing set of methods, while enabling each module to be typechecked independently. The key new features of our design are extension modules and resolving modules. The declarations in an extension module are automatically imported into the extended module, for the purposes of run-time method lookup. By restricting subtyping and conformance to cross only extension module boundaries, and by requiring the final program to include for each module a single most extending module which can ensure the completeness and consistency of independently-developed extensions, we retain the ability to typecheck client code using only the public interfaces of explicitly imported modules.

We believe that these two contributions are important steps towards modular development of robust software in multi-method-based languages. In particular, the combination of typechecking, modules, and multi-methods allows programmers to integrate the advantages of both abstract data types and object-oriented programming [Cook 90]. One can simulate the ADT mechanisms of languages like CLU and Ada by hiding the data representation in a module. Binary methods in the module can directly access the representation of two objects of the same class (or any other class in the module), something not possible in Smalltalk, while disallowing access from outside the module. Moreover, multiple implementations of an ADT can coexist and interoperate in a single program, which is easily done in Smalltalk but not CLU or Ada. One can simulate single-dispatching object-oriented languages such as Smalltalk by specializing only on the first argument of a method. But even when the programmer chooses to specialize on more than one argument, independently developed classes may still be combined in a single program in a type-safe manner, using resolving modules verifying type safety statically.

At least two questions remain: will typechecking of individual modules be fast enough in practice, and will the restrictions placed on module extensions be too severe in practice? To gain the necessary experience with which to answer these questions, we are implementing our typechecking algorithm and module system in the context of the Cecil language. At present, over 50,000 lines of Cecil code have been written, in a version of the language lacking modules and full static type checking, and we expect that revising this code base to use modules and respect the restrictions of static type checking will be an effective test of the practicality of our design.

Acknowledgments

We would like to thank to William Cook for discussions about the modularity problems of multi-methods. Thanks also to Giuseppe Castagna, Jens Palsberg, Tim Wahls, David Fernandez-Baca, Jeffrey Dean, David Grove, Charles Garrett, and the anonymous OOPSLA'94 and TOPLAS referees for their helpful comments on earlier versions of this paper and for their encouragement.

Chambers's work is supported in part by a National Science Foundation Research Initiation Award (contract number CCR-9210990), an NSF Young Investigator award (contract number CCR-9457767), and gifts from Sun Microsystems, IBM, Pure Software, and Edison Design Group. Leavens's work is supported in part by a National Science Foundation Research Initiation Award (contract number CCR-9108654) and an NSF grant (contract number CCR-9503168).

More information on the Cecil language and implementation project can be found through the World Wide Web under http://www.cs.washington.edu/research/projects/cecil and via anonymous `ftp` from cs.washington.edu:/pub/chambers.

References

[Ada 83] *Reference Manual for the Ada Programming Language,* ANSI/MIL-STD 1815A, 1983.

[Agrawal *et al.* 91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. Static Type Checking of Multi-Methods. In *OOPSLA '91 Conference Proceedings*, pp. 113-128, Phoenix, AZ, October, 1991. Published as *SIGPLAN Notices 26(11)*, November, 1991.

[America 87] Pierre America. Inheritance and Subtyping in a Parallel Object-Oriented Language. In *ECOOP '87 Conference Proceedings*, pp. 234-242, Paris, France, June, 1987. Published as *Lecture Notes in Computer Science 276*, Springer-Verlag, Berlin, 1987.

[America & van der Linden 90] Pierre America and Frank van der Linden. A Parallel Object-Oriented Language with Inheritance and Subtyping. In *OOPSLA/ECOOP '90 Conference Proceedings*, pp. 161-168, Ottawa, Canada, October, 1990. Published as *SIGPLAN Notices 25(10)*, October, 1990.

[Amiel *et al.* 94] Eric Amiel, Olivier Gruber, and Eric Simon. Optimizing Multi-Method Dispatch Using Compressed Dispatch Tables. In *OOPSLA '94 Conference Proceedings*, Portland, OR, October, 1994.

[Barnes 91] J. G. P. Barnes. *Programming in Ada (third edition).* Addison-Wesley, Wokingham, England, 1991.

[Bobrow *et al.* 88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, D. A. Moon. Common Lisp Object System Specification X3J13. *SIGPLAN Notices 23(Special Issue)*, September, 1988.

[Bracha & Griswold 93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *OOPSLA '93 Conference Proceedings*, pp. 215-230, Washington, D.C., September, 1993. Published as *SIGPLAN Notices 28(10)*, October, 1993.

[Bruce *et al.* 93] Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, Robert van Gent, Allyn Dimock, and Robert Muller. Safe and Decidable Type Checking in an Object-Oriented Language. In *OOPSLA '93 Conference Proceedings*, pp. 29-46, Washington, D.C., September, 1993. Published as *SIGPLAN Notices 28(10)*, October, 1993.

[Bruce *et al.* 95] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On Binary Methods. Technical report #95-08, Department of Computer Science, Iowa State University, May, 1995.

[Canning *et al.* 89] Peter S. Canning, William R. Cook, Walter L. Hill, and Walter G. Olthoff. Interfaces for Strongly-Typed Object-Oriented Programming. In *OOPSLA '89 Conference Proceedings*, pp. 457-467, New Orleans, LA, October, 1989. Published as *SIGPLAN Notices 24(10)*, October, 1989.

[Cardelli 88] Luca Cardelli. A Semantics of Multiple Inheritance. *Information and Computation 76(2/3)*, pp. 138-164, February/March, 1988.

[Cardelli & Wegner 85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys 17(4)*, pp. 471-522, December, 1985.

[Cardelli & Mitchell 89] Luca Cardelli and John C. Mitchell. Operations on Records. In *Proceedings of the International Conference on the Mathematical Foundation of Programming Semantics*, New Orleans, LA, 1989.

[Castagna *et al.* 92] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A Calculus for Overloaded Functions with Subtyping. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pp. 182-192, San Francisco, June, 1992. Published as *Lisp Pointers 5(1)*, January-March, 1992.

[Chambers 92] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In *ECOOP '92 Conference Proceedings*, pp. 33-56, Utrecht, the Netherlands, June/July, 1992. Published as *Lecture Notes in Computer Science 615*, Springer-Verlag, Berlin, 1992.

[Chambers 93] Craig Chambers. The Cecil Language: Specification and Rationale. Technical report #93-03-05, Department of Computer Science and Engineering, University of Washington, March, 1993.

[Chambers & Leavens 94] Craig Chambers and Gary T. Leavens. Typechecking and Modules for Multi-Methods. In *OOPSLA '94 Conference Proceedings*, pp. 1-15, Portland, OR, October, 1994. Published as *SIGPLAN Notices 29(10)*, October, 1994.

[Chambers & Leavens 95] Craig Chambers and Gary T. Leavens. Typechecking and Modules for Multi-Methods. Technical report #95-19, Department of Computer Science, Iowa State University, June, 1995.

[Chen & Turau 94] Weimin Chen and Volker Turau. Efficient Dynamic Look-Up Strategy for Multi-Methods. In *ECOOP '94 Conference Proceedings*, Bologna, Italy, July, 1994.

[Cook *et al.* 90] William Cook, Walter Hill, and Peter Canning. Inheritance is not Subtyping. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January, 1990.

[Cook 90] William Cook. Object-Oriented Programming versus Abstract Data Types. In *Proceedings of the Workshop on the Foundations of Object-Oriented Languages*, pp. 151-178, Noordwijkerhout, the Netherlands, May/June, 1990. Published as *Lecture Notes in Computer Science 489*, Springer-Verlag, New York, 1991.

[Cook 92] William R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *OOPSLA '92 Conference Proceedings*, pp. 1-15, Vancouver, Canada, October 1992. Published as *SIGPLAN Notices 27(10)*, October 1992.

[Ghelli 91] Giorgio Ghelli. A Static Type System for Message Passing. In *OOPSLA '91 Conference Proceedings*, pp. 129-145, Phoenix, AZ, October, 1991. Published as *SIGPLAN Notices 26(11)*, November, 1991.

[Goldberg & Robson 83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Mass., 1983.

[Goguen 84] Joseph A. Goguen. Parameterized Programming. *IEEE Transactions on Software Engineering 10(5)*, pp. 528-543, September, 1984.

[Harrison & Ossher 93] William Harrison and Harold Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *OOPSLA '93 Conference Proceedings*, pp. 411-428, Washington, D.C., September, 1993. Published as *SIGPLAN Notices 28(10)*, October, 1993.

[Hölzle 93] Urs Hölzle. Integrating Independently-Developed Components in Object-Oriented Languages. In *ECOOP '93 Conference Proceedings*, pp. 36-56, Kaiserslautern, Germany, July, 1993. Published as *Lecture Notes in Computer Science 707*, Springer-Verlag, Berlin, 1993.

[Hudak *et al.* 92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. *Report on the Programming Language Haskell: A Non-strict, Purely Functional Language, Version 1.2*. In *SIGPLAN Notices 27(5)*, May, 1992.

[Leavens 91] Gary T. Leavens. Modular Specification and Verification of Object-Oriented Programs. *IEEE Software 8(4)*, pp. 72-80, July, 1991.

[Leavens & Weihl 90] Gary T. Leavens and William E. Weihl. Reasoning about Object-Oriented Programs that use Subtypes. In *OOPSLA/ECOOP '90 Conference Proceedings*, pp. 212-223, Ottawa, Canada, October, 1990. Published as *SIGPLAN Notices 25(10)*, October, 1990.

[Liskov *et al.* 77] Barbara Liskov, Alan Snyder, Russell Atkinson, and J. Craig Schaffert. Abstraction Mechanisms in CLU. *Communications of the ACM 20(8)*, pp. 564-576, August, 1977.

[Liskov *et al.* 81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*. Lecture Notes in Computer Science, volume 114, Springer-Verlag, New York, NY, 1981.

[Liskov & Zilles 74] Barbara H. Liskov and Stephen N. Zilles. Programming with Abstract Data Types. In *Proceedings of the ACM SIGPLAN Conference on Very High Level Languages*, pp. 50-59, April, 1974. Published as *SIGPLAN Notices 9(4)*, 1974.

[Meyer 88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, 1998.

[Meyer 92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, New York, 1992.

[Milner *et al.* 90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[Mössenböck & Wirth 91] H. Mössenböck and Niklaus Wirth. The Programming Language Oberon-2. *Structured Programming 12(4)*, 1991.

[Mugridge *et al.* 91] W. B. Mugridge, J. G. Hosking, and J. Hamer. Multi-Methods in a Statically-Typed Programming Language. Technical report #50, Department of Computer Science, University of Auckland, 1991. Also appears in *ECOOP '91 Conference Proceedings*, Geneva, Switzerland, July, 1991.

[Nelson 91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Paepcke 93] Andreas Paepcke. *Object-Oriented Programming: The CLOS Perspective*. MIT Press, 1993.

[Palsberg & Schwartzbach 94] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.

[Parnas 71] D. L. Parnas. Information Distribution Aspects of Design Methodology. *Proceedings of IFIP Congress 71*. IFIP, 1971.

[Parnas 72] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM 15(5),* pp. 330-336, May, 1972.

[Paulson 91] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[Pierce & Turner 92] Benjamin C. Pierce and David N. Turner. Statically Typed Multi-Methods via Partially Abstract Types. Unpublished manuscript, October, 1992.

[Reynolds 80] John C. Reynolds. Using Category Theory to Design Implicit Conversions and Generic Operators. In Neil D. Jones (ed.), *Semantics-Directed Compiler Generation, Proceedings of a Workshop,* pp. 211-258, Aarhus, Denmark, January, 1980. Lecture Notes in Computer Science, volume 94, Springer-Verlag, New York, NY, 1980.

[Rouaix 90] Francois Rouaix. Safe Run-Time Overloading. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pp. 355-366, San Francisco, CA, January, 1990.

[Schaffert *et al.* 86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An Introduction to Trellis/Owl. In *OOPSLA '86 Conference Proceedings*, pp. 9-16, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Snyder 86] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *OOPSLA '86 Conference Proceedings*, pp. 38-45, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Steele 90] Guy L. Steele Jr. *Common Lisp: The Language (second edition)*. Digital Press, Bedford, MA, 1990.

[Stroustrup 91] Bjarne Stroustrup. *The C++ Programming Language (second edition)*. Addison-Wesley, Reading, MA, 1991.

[Szyperski 92] Clemens A. Szyperski. Import is Not Inheritance - Why We Need Both: Modules and Classes. In *ECOOP '92 Conference Proceedings*, pp. 19-32, Utrecht, the Netherlands, June/July, 1992. Published as *Lecture Notes in Computer Science 615*, Springer-Verlag, Berlin, 1992.

[Touretzky 86] D. Touretzky. *The Mathematics of Inheritance Systems*. Morgan-Kaufmann, 1986.

[Wirfs-Brock & Wilkerson 88] Allen Wirfs-Brock and Brian Wilkerson. An Overview of Modular Smalltalk. In *OOPSLA '88 Conference Proceedings*, pp. 123-134, San Diego, CA, October, 1988. Published as *SIGPLAN Notices 23(11)*, November, 1988.

[Wirth 88] Niklaus Wirth. *Programming in Modula-2 (fourth edition)*. Springer-Verlag, Berlin, 1988.

Appendix A Correctness Theorems and Proofs

This appendix gives formal correctness proofs for our typechecking algorithm. The proofs use the calculational format described by David Gries in his article "Teaching Calculation and Discrimination: A More Effective Curriculum" (*Communications of the ACM 34(3)*, pp. 44-55, March, 1991).

A.1 Problem Breakdown

We use two lemmas to help in the proof of Theorem 1.

Lemma 1. Let *Ms* be a finite set of methods. Then

$$|Ms| > 0 \;\wedge\; (\forall\, m_1, m_2 \in Ms.\ (\exists\, m \in Ms.\ m \leq_{meth} m_1 \;\wedge\; m \leq_{meth} m_2))$$
$$\Rightarrow (\exists\, m.\ m = \mathsf{least}(Ms))$$

Proof: We proceed by induction on the size of *Ms*.

It will be convenient to consider two base cases. If the size of *Ms* is zero, then the result follows trivially, because the first conjunct in the antecedent is false. If the size of *Ms* is one, then the single element of *Ms* is the least element, and so the consequent is true.

For the inductive step, suppose that the size of *Ms* is $n > 1$ and that the lemma holds for all proper subsets of *Ms*. Since *Ms* is non-empty, there is an element $p \in Ms$ and a proper subset *Ms'*, such that $Ms = Ms' \cup \{p\}$. We calculate as follows.

$$|Ms| > 0 \wedge (\forall\, m_1, m_2 \in Ms.\ (\exists\, m \in Ms.\ m \leq_{meth} m_1 \;\wedge\; m \leq_{meth} m_2))$$

$\Longrightarrow$ ⟨by assumption of $|Ms| > 1$, by instantiation of m_1 to $\mathsf{least}(Ms')$ (which exists because $n>1$), and by instantiation of m_2 to p, and predicate calculus⟩

$$\exists\, m \in Ms.\ m \leq_{meth} \mathsf{least}(Ms') \;\wedge\; m \leq_{meth} p$$

$\Longleftrightarrow$ ⟨by the law of the excluded middle⟩

$$\begin{aligned}&\exists\, m \in Ms.\\ &\quad (m \in Ms' \Longrightarrow m \leq_{meth} \mathsf{least}(Ms') \;\wedge\; m \leq_{meth} p) \;\wedge\\ &\quad (m \notin Ms' \Longrightarrow m \leq_{meth} \mathsf{least}(Ms') \;\wedge\; m \leq_{meth} p)\end{aligned}$$

$\Longleftrightarrow$ ⟨by definition of least⟩

$$\begin{aligned}&\exists\, m \in Ms.\\ &\quad (m \in Ms' \Longrightarrow m = \mathsf{least}(Ms') \;\wedge\; m \leq_{meth} p) \;\wedge\\ &\quad (m \notin Ms' \Longrightarrow m \leq_{meth} \mathsf{least}(Ms') \;\wedge\; m \leq_{meth} p)\end{aligned}$$

$\Longleftrightarrow$ ⟨by transitivity of $\leq_{meth}$ and definition of least⟩

$$\begin{aligned}&\exists\, m \in Ms.\\ &\quad (m \in Ms' \Longrightarrow m = \mathsf{least}(Ms' \cup \{p\})) \;\wedge\\ &\quad (m \notin Ms' \Longrightarrow m \leq_{meth} \mathsf{least}(Ms') \;\wedge\; m \leq_{meth} p)\end{aligned}$$

$\Longleftrightarrow$ ⟨by assumption that p is the only element of *Ms* not in *Ms'*⟩

$$\begin{aligned}&\exists\, m \in Ms.\\ &\quad (m \in Ms' \Longrightarrow m = \mathsf{least}(Ms' \cup \{p\})) \;\wedge\\ &\quad (m \notin Ms' \Longrightarrow m = p \;\wedge\; m \leq_{meth} \mathsf{least}(Ms') \;\wedge\; m \leq_{meth} p)\end{aligned}$$

$\Longleftrightarrow$ ⟨by assumption that p is the only element of *Ms* not in *Ms'*⟩

$$\begin{aligned}&\exists\, m \in Ms.\\ &\quad (m \in Ms' \Longrightarrow m = \mathsf{least}(Ms' \cup \{p\})) \;\wedge\\ &\quad (m \notin Ms' \Longrightarrow m = p \;\wedge\; m \leq_{meth} \mathsf{least}(Ms') \;\wedge\; m \leq_{meth} p)\end{aligned}$$

$\Longrightarrow \langle$by definition of $\leq_{meth}$ and least$\rangle$
$\exists\, m \in Ms.$
$(m \in Ms' \Longrightarrow m = \mathsf{least}(Ms' \cup \{p\}))\ \wedge$
$(m \notin Ms' \Longrightarrow m = \mathsf{least}(Ms' \cup \{p\}))$

$\Longleftrightarrow \langle$by the law of the excluded middle and $Ms = Ms' \cup \{p\}\rangle$
$\exists\, m \in Ms.\ m = \mathsf{least}(Ms)$

$\Longleftrightarrow \langle$by the range type of least$\rangle$
$\exists\, m.\ m = \mathsf{least}(Ms)$

Lemma 2. Let $s \in S$ and $\vec{c} \in (C_{concrete})^*$ be such that $\vec{c}$ <: argtypes(s). Then

m = least(applicable-methods(s, $\vec{c}$))
$\Longrightarrow m$ = least(applicable-methods(s, $\vec{c}$))
$\wedge\ (\neg \exists\, m' \in$ applicable-methods(s, $\vec{c}$). $m' \leq_{meth} m \wedge m' \neq m)$

Proof: We calculate as follows.

m = least(applicable-methods(s, $\vec{c}$))

$\Longleftrightarrow \langle$by predicate calculus$\rangle$
m = least(applicable-methods(s, $\vec{c}$))
$\wedge\ m$ = least(applicable-methods(s, $\vec{c}$))

$\Longleftrightarrow \langle$by definition of least$\rangle$
m = least(applicable-methods(s, $\vec{c}$))
$\wedge\ m \in$ applicable-methods(s, $\vec{c}$) $\wedge$ ($\forall\, m' \in$ applicable-methods(s, $\vec{c}$). $m \leq_{meth} m'$)

$\Longrightarrow \langle$by $A \wedge B \wedge C \Rightarrow A \wedge C\rangle$
m = least(applicable-methods(s, $\vec{c}$))
$\wedge$ ($\forall\, m' \in$ applicable-methods(s, $\vec{c}$). $m \leq_{meth} m'$)

$\Longleftrightarrow \langle$by law of the excluded middle, to do a case analysis$\rangle$
m = least(applicable-methods(s, $\vec{c}$))
$\wedge$ ($\forall\, m' \in$ applicable-methods(s, $\vec{c}$). $m \leq_{meth} m' \wedge (m \neq m' \vee m = m')$)

$\Longleftrightarrow \langle$by conjunction distributes over disjunction$\rangle$
m = least(applicable-methods(s, $\vec{c}$))
$\wedge$ ($\forall\, m' \in$ applicable-methods(s, $\vec{c}$). $(m \leq_{meth} m' \wedge m \neq m') \vee (m \leq_{meth} m' \wedge m = m')$)

$\Longrightarrow \langle$by the fact that $\leq_{meth}$ is an acyclic partial order$\rangle$
m = least(applicable-methods(s, $\vec{c}$))
$\wedge$ ($\forall\, m' \in$ applicable-methods(s, $\vec{c}$). $\neg (m' \leq_{meth} m) \vee m = m'$)

$\Longleftrightarrow \langle$by predicate calculus$\rangle$
m = least(applicable-methods(s, $\vec{c}$))
$\wedge\ (\neg \exists\, m' \in$ applicable-methods(s, $\vec{c}$). $m' \leq_{meth} m \wedge m' \neq m)$

Theorem 1.
(ImplementationIsComplete $\wedge$ ImplementationIsConsistent $\wedge$ ImplementationIsConforming) $\Rightarrow$ ImplementationTypechecks

Proof: We prove this theorem by the following calculation.

ImplementationIsComplete $\wedge$ ImplementationIsConsistent $\wedge$ ImplementationIsConforming

$\Leftrightarrow \langle$by commutivity$\rangle$
$\quad$ ImplementationIsConforming $\wedge$ ImplementationIsComplete $\wedge$ ImplementationIsConsistent

$\Leftrightarrow \langle$by definition$\rangle$

$$
\begin{aligned}
&(\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*.\\
&\quad \vec{c} <: \text{argtypes}(s) \Rightarrow\\
&\qquad \forall\, m \in \text{applicable-methods}(s, \vec{c}).\\
&\qquad\quad (\neg\, \exists\, m' \in \text{applicable-methods}(s, \vec{c}).\ m' \neq m \ \wedge\ m' \leq_{meth} m) \Rightarrow\\
&\qquad\qquad \vec{c} <: \text{argtypes}(m) \ \wedge\ \text{restype}(m) \leq_{sub} \text{restype}(s))\\
&\wedge (\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*.\\
&\quad \vec{c} <: \text{argtypes}(s) \Rightarrow\\
&\qquad |\text{applicable-methods}(s, \vec{c})| > 0)\\
&\wedge (\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*.\\
&\quad \vec{c} <: \text{argtypes}(s) \Rightarrow\\
&\qquad \forall\, m_1, m_2 \in \text{applicable-methods}(s, \vec{c}).\\
&\qquad\quad \exists\, m \in \text{applicable-methods}(s, \vec{c}).\\
&\qquad\qquad m \leq_{meth} m_1 \ \wedge\ m \leq_{meth} m_2)
\end{aligned}
$$

$\Leftrightarrow \langle$by $(\forall x.P(x)) \wedge (\forall x.Q(x)) \Leftrightarrow \forall x.(P(x) \wedge Q(x))$, twice; and $((P \Rightarrow Q) \wedge (P \Rightarrow R)) \Leftrightarrow (P \Rightarrow (Q \wedge R))$, twice$\rangle$

$$
\begin{aligned}
&\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*.\\
&\quad \vec{c} <: \text{argtypes}(s) \Rightarrow\\
&\qquad (\forall\, m \in \text{applicable-methods}(s, \vec{c}).\\
&\qquad\quad (\neg\, \exists\, m' \in \text{applicable-methods}(s, \vec{c}).\ m' \neq m \ \wedge\ m' \leq_{meth} m) \Rightarrow\\
&\qquad\qquad \vec{c} <: \text{argtypes}(m) \ \wedge\ \text{restype}(m) \leq_{sub} \text{restype}(s))\\
&\qquad \wedge (|\text{applicable-methods}(s, \vec{c})| > 0)\\
&\qquad \wedge (\forall\, m_1, m_2 \in \text{applicable-methods}(s, \vec{c}).\\
&\qquad\quad \exists\, m \in \text{applicable-methods}(s, \vec{c}).\\
&\qquad\qquad m \leq_{meth} m_1 \ \wedge\ m \leq_{meth} m_2)
\end{aligned}
$$

$\Rightarrow \langle$by Lemma 1 and the ranges of applicable-methods and least$\rangle$

$$
\begin{aligned}
&\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*.\\
&\quad \vec{c} <: \text{argtypes}(s) \Rightarrow\\
&\qquad (\forall\, m \in \text{applicable-methods}(s, \vec{c}).\\
&\qquad\quad (\neg\, \exists\, m' \in \text{applicable-methods}(s, \vec{c}).\ m' \neq m \ \wedge\ m' \leq_{meth} m) \Rightarrow\\
&\qquad\qquad \vec{c} <: \text{argtypes}(m) \ \wedge\ \text{restype}(m) \leq_{sub} \text{restype}(s))\\
&\qquad \wedge (\exists\, m \in M.\ m = \text{least}(\text{applicable-methods}(s, \vec{c})))
\end{aligned}
$$

$\Leftrightarrow \langle$by renaming m to $m''\rangle$

$$
\begin{aligned}
&\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*.\\
&\quad \vec{c} <: \text{argtypes}(s) \Rightarrow\\
&\qquad (\forall\, m'' \in \text{applicable-methods}(s, \vec{c}).\\
&\qquad\quad (\neg\, \exists\, m' \in \text{applicable-methods}(s, \vec{c}).\ m' \neq m'' \ \wedge\ m' \leq_{meth} m'') \Rightarrow\\
&\qquad\qquad \vec{c} <: \text{argtypes}(m'') \ \wedge\ \text{restype}(m'') \leq_{sub} \text{restype}(s))\\
&\qquad \wedge (\exists\, m \in M.\ m = \text{least}(\text{applicable-methods}(s, \vec{c})))
\end{aligned}
$$

$\Leftrightarrow \langle$by predicate calculus, as m is not free in the universally quantified expression$\rangle$

$$
\begin{array}{l}
\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*. \\
\quad \vec{c} <: \mathsf{argtypes}(s) \Rightarrow \\
\quad\quad \exists\, m \in M. \\
\quad\quad\quad m = \mathsf{least}(\mathsf{applicable\text{-}methods}(s, \vec{c})) \wedge \\
\quad\quad\quad (\forall\, m'' \in \mathsf{applicable\text{-}methods}(s, \vec{c}). \\
\quad\quad\quad\quad (\neg \exists\, m' \in \mathsf{applicable\text{-}methods}(s, \vec{c}).\ m' \neq m'' \wedge m' \leq_{meth} m'') \Rightarrow \\
\quad\quad\quad\quad\quad \vec{c} <: \mathsf{argtypes}(m'') \wedge \mathsf{restype}(m'') \leq_{sub} \mathsf{restype}(s))
\end{array}
$$

$\Rightarrow \langle$by instantiation of m'' to $m\rangle$

$$
\begin{array}{l}
\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*. \\
\quad \vec{c} <: \mathsf{argtypes}(s) \Rightarrow \\
\quad\quad \exists\, m \in M. \\
\quad\quad\quad m = \mathsf{least}(\mathsf{applicable\text{-}methods}(s, \vec{c})) \wedge \\
\quad\quad\quad ((\neg \exists\, m' \in \mathsf{applicable\text{-}methods}(s, \vec{c}).\ m' \neq m \wedge m' \leq_{meth} m) \Rightarrow \\
\quad\quad\quad\quad \vec{c} <: \mathsf{argtypes}(m) \wedge \mathsf{restype}(m) \leq_{sub} \mathsf{restype}(s))
\end{array}
$$

$\Rightarrow \langle$by Lemma 2$\rangle$

$$
\begin{array}{l}
\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*. \\
\quad \vec{c} <: \mathsf{argtypes}(s) \Rightarrow \\
\quad\quad \exists\, m \in M. \\
\quad\quad\quad m = \mathsf{least}(\mathsf{applicable\text{-}methods}(s, \vec{c})) \wedge \\
\quad\quad\quad (\neg \exists\, m' \in \mathsf{applicable\text{-}methods}(s, \vec{c}).\ m' \neq m \wedge m' \leq_{meth} m) \wedge \\
\quad\quad\quad ((\neg \exists\, m' \in \mathsf{applicable\text{-}methods}(s, \vec{c}).\ m' \neq m \wedge m' \leq_{meth} m) \Rightarrow \\
\quad\quad\quad\quad \vec{c} <: \mathsf{argtypes}(m) \wedge \mathsf{restype}(m) \leq_{sub} \mathsf{restype}(s))
\end{array}
$$

$\Rightarrow \langle$by $(Q \wedge (Q \Rightarrow R)) \Rightarrow R\rangle$

$$
\begin{array}{l}
\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*. \\
\quad \vec{c} <: \mathsf{argtypes}(s) \Rightarrow \\
\quad\quad \exists\, m \in M. \\
\quad\quad\quad m = \mathsf{least}(\mathsf{applicable\text{-}methods}(s, \vec{c})) \wedge \\
\quad\quad\quad \vec{c} <: \mathsf{argtypes}(m) \wedge \mathsf{restype}(m) \leq_{sub} \mathsf{restype}(s)
\end{array}
$$

$\Leftrightarrow \langle$by definition$\rangle$

ImplementationTypechecks

A.2 Correctness of Conformance Checking Algorithm

We prove the correctness of the conformance checking algorithm with several lemmas and an auxiliary definition. The auxuiliary definition, of AbstractComputeIsConforming, is used as an intermediate stage in the proof. It uses a definition of conformance that is similar to ComputeIsConforming, but abstracts away the computational details. This definition is given below.

$$
\begin{array}{l}
\mathsf{AbstractComputeIsConforming} \equiv \\
\quad \mathsf{UnspecializedAndResultConform} \wedge \mathsf{SpecializersAreConforming}
\end{array}
$$

where:

$$
\begin{array}{l}
\mathsf{UnspecializedAndResultConform} \equiv \\
\quad \forall\, m \in M.\ \forall\, s \in \mathsf{relevant\text{-}sigs}(m, S). \\
\quad\quad \mathsf{abs\text{-}has\text{-}common\text{-}classes}(m, \mathsf{argtypes}(s)) \Rightarrow \mathsf{contra\text{-}unspec\text{-}args\text{-}co\text{-}result}(m, s)
\end{array}
$$

$$\text{abs-has-common-classes}(m, \vec{t}\,) \equiv \exists\, \vec{c} \in \text{covered-class-vecs}(m)\,.\, \vec{c} <: \vec{t}$$

$$\begin{aligned}
&\text{covered-class-vecs}(m) \equiv \\
&\quad \{\, \vec{c} \in (C_{concrete})^* \mid \vec{c} \leq_{inh} \text{specializers}(m) \\
&\qquad\qquad \wedge (\neg \exists\, m_1 \in M.\ \text{msg}(m_1) = \text{msg}(m) \wedge |\text{argtypes}(m_1)| = |\text{argtypes}(m)| \\
&\qquad\qquad\qquad \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m) \,\}
\end{aligned}$$

$$\begin{aligned}
&\text{SpecializersAreConforming} \equiv \\
&\quad \forall\, m \in M.\ \forall\, \vec{c} \in \text{covered-class-vecs}(m). \\
&\qquad \forall\, i \in \text{indexes}(\text{specializers}(m)). \\
&\qquad\quad \text{specializers}(m)_i \neq any \ \Rightarrow\ \vec{c}_i <: \text{argtypes}(m)_i
\end{aligned}$$

The definitions of relevant-sigs and contra-unspec-args-co-result are the same as given in the main body, but are repeated here for clarity.

$$\begin{aligned}
&\text{contra-unspec-args-co-result}(m, s) \equiv \\
&\quad (\forall\, i \in \text{indexes}(\text{specializers}(m)). \\
&\qquad \text{specializers}(m)_i = any \ \Rightarrow\ \text{argtypes}(s)_i \leq_{sub} \text{argtypes}(m)_i) \\
&\quad \wedge\ \text{restype}(m) \leq_{sub} \text{restype}(s)
\end{aligned}$$

$$\text{relevant-sigs}(m, Ss) \equiv \{\, s \in Ss \mid \text{msg}(m) = \text{msg}(s) \wedge |\text{argtypes}(s)| = |\text{argtypes}(m)| \,\}$$

Theorem 2. ComputeIsConforming $\Rightarrow$ ImplementationIsConforming

Proof: We use the lemmas below, as follows.

ComputeIsConforming

$\Rightarrow$ $\langle$by Lemma 4.$\rangle$
AbstractComputeIsConforming

$\Rightarrow$ $\langle$by Lemma 3.$\rangle$
ImplementationIsConforming

Lemma 3. AbstractComputeIsConforming $\Rightarrow$ ImplementationIsConforming

Proof: If either the set of signatures, S, or the set of concrete classes, $C_{concrete}$, is empty, then the consequent is true, and so the theorem follows. We show that the lemma holds in the case where both S and $C_{concrete}$ are nonempty, by the following calculation.

AbstractComputeIsConforming

$\Leftrightarrow$ $\langle$by definition$\rangle$
UnspecializedAndResultConform $\wedge$ SpecializersAreConforming

$\Leftrightarrow$ $\langle$by definition$\rangle$

$$\begin{aligned}
&(\forall\, m \in M.\ \forall\, s \in \text{relevant-sigs}(m, S). \\
&\quad \text{abs-has-common-classes}(m, \text{argtypes}(s)) \Rightarrow \text{contra-unspec-args-co-result}(m, s)) \\
&\wedge\ (\forall\, m \in M.\ \forall\, \vec{c} \in \text{covered-class-vecs}(m). \\
&\quad \forall\, i \in \text{indexes}(\text{specializers}(m)). \\
&\qquad \text{specializers}(m)_i \neq any \ \Rightarrow\ \vec{c}_i <: \text{argtypes}(m)_i)
\end{aligned}$$

$\Leftrightarrow \langle$by definition of $\text{relevant-sigs}\rangle$

$(\forall\, m \in M.\ \forall\, s \in \{\, s \in S \mid \text{msg}(m) = \text{msg}(s)\ \wedge |\text{argtypes}(s)| = |\text{argtypes}(m)|\,\}.$

$\quad \text{abs-has-common-classes}(m, \text{argtypes}(s)) \Rightarrow \text{contra-unspec-args-co-result}(m, s))$

$\wedge\ (\forall\, m \in M.\ \forall\, \vec{c} \in \text{covered-class-vecs}(m).$

$\quad \forall\, i \in \text{indexes}(\text{specializers}(m)).$

$\qquad \text{specializers}(m)_i \neq any\ \Rightarrow\ \vec{c}_i <: \text{argtypes}(m)_i)$

$\Leftrightarrow \langle$by set theory$\rangle$

$(\forall\, m \in M.\ \forall\, s \in S\,.$

$\quad \text{msg}(m) = \text{msg}(s)\ \wedge |\text{argtypes}(s)| = |\text{argtypes}(m)|$

$\qquad \Rightarrow (\text{abs-has-common-classes}(m, \text{argtypes}(s))$

$\qquad\quad \Rightarrow \text{contra-unspec-args-co-result}(m, s)))$

$\wedge\ (\forall\, m \in M.\ \forall\, \vec{c} \in \text{covered-class-vecs}(m).$

$\quad \forall\, i \in \text{indexes}(\text{specializers}(m)).$

$\qquad \text{specializers}(m)_i \neq any\ \Rightarrow\ \vec{c}_i <: \text{argtypes}(m)_i)$

$\Leftrightarrow \langle$by interchange of quantifiers and the constant term rule, assuming S is nonempty$\rangle$

$(\forall\, s \in S\,.\ \forall\, m \in M.$

$\quad \text{msg}(m) = \text{msg}(s)\ \wedge |\text{argtypes}(s)| = |\text{argtypes}(m)|$

$\qquad \Rightarrow (\text{abs-has-common-classes}(m, \text{argtypes}(s))$

$\qquad\quad \Rightarrow \text{contra-unspec-args-co-result}(m, s)))$

$\wedge\ (\forall\, s \in S\,.\ \forall\, m \in M.\ \forall\, \vec{c} \in \text{covered-class-vecs}(m).$

$\quad \forall\, i \in \text{indexes}(\text{specializers}(m)).$

$\qquad \text{specializers}(m)_i \neq any\ \Rightarrow\ \vec{c}_i <: \text{argtypes}(m)_i)$

$\qquad\qquad \text{contra-unspec-args-co-result}(m, s)))$

$\wedge\ (\forall\, s \in S\,.\ \forall\, m \in M.\ \forall\, \vec{c} \in \text{covered-class-vecs}(m).$

$\quad \forall\, i \in \text{indexes}(\text{specializers}(m)).$

$\qquad \text{specializers}(m)_i \neq any\ \Rightarrow\ \vec{c}_i <: \text{argtypes}(m)_i)$

$\Leftrightarrow \langle$by joining the term, twice$\rangle$

$\forall\, s \in S\,.\ \forall\, m \in M.$

$\quad (\text{msg}(m) = \text{msg}(s)\ \wedge |\text{argtypes}(s)| = |\text{argtypes}(m)|$

$\qquad \Rightarrow (\text{abs-has-common-classes}(m, \text{argtypes}(s))$

$\qquad\quad \Rightarrow \text{contra-unspec-args-co-result}(m, s)))$

$\quad \wedge\ (\forall\, \vec{c} \in \text{covered-class-vecs}(m).$

$\qquad \forall\, i \in \text{indexes}(\text{specializers}(m)).$

$\qquad\quad \text{specializers}(m)_i \neq any\ \Rightarrow\ \vec{c}_i <: \text{argtypes}(m)_i)$

$\Leftrightarrow \langle$by definition of $\text{covered-class-vecs}(m)\rangle$

$\forall\, s \in S\,.\ \forall\, m \in M.$

$\quad (\text{msg}(m) = \text{msg}(s)\ \wedge |\text{argtypes}(s)| = |\text{argtypes}(m)|$

$\qquad \Rightarrow (\text{abs-has-common-classes}(m, \text{argtypes}(s))$

$\qquad\quad \Rightarrow \text{contra-unspec-args-co-result}(m, s)))$

$\quad \wedge\ (\forall\, \vec{c} \in \{\, \vec{c} \in (C_{concrete})^* \mid \vec{c} \leq_{inh} \text{specializers}(m)$

$\qquad\qquad \wedge (\neg \exists\, m_1 \in M.\ \text{msg}(m_1) = \text{msg}(m)\ \wedge |\text{argtypes}(m_1)| = |\text{argtypes}(m)|$

$\qquad\qquad\quad \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)\,\}.$

$\qquad \forall\, i \in \text{indexes}(\text{specializers}(m)).$

$\qquad\quad \text{specializers}(m)_i \neq any\ \Rightarrow\ \vec{c}_i <: \text{argtypes}(m)_i)$

$\Leftrightarrow \langle$by set theory$\rangle$
$\forall\, s \in S \,.\, \forall\, m \in M.$
(msg(m) = msg(s) $\wedge$ |argtypes(s)| = |argtypes(m)|
$\Rightarrow$ (abs-has-common-classes(m, argtypes(s))
$\Rightarrow$ contra-unspec-args-co-result(m, s)))
$\wedge$ ($\forall\, \vec{c} \in (C_{concrete})^*$.
($\vec{c} \leq_{inh}$ specializers(m)
$\wedge$ ($\neg\, \exists\, m_1 \in M.$ msg(m_1) = msg(m) $\wedge$ |argtypes(m_1)| = |argtypes(m)|
$\wedge\ \vec{c} \leq_{inh}$ specializers(m_1) $\wedge\ m_1 \leq_{meth} m \wedge m_1 \neq m$))
$\Rightarrow$ ($\forall\, i \in$ indexes(specializers(m)).
specializers(m)$_i \neq any \Rightarrow\ \vec{c}_i$ <: argtypes(m)$_i$))

$\Leftrightarrow \langle$by the constant term rule, assuming $C_{concrete}$ is nonempty$\rangle$
$\forall\, s \in S \,.\, \forall\, m \in M.\ \forall\, \vec{c} \in (C_{concrete})^*$.
(msg(m) = msg(s) $\wedge$ |argtypes(s)| = |argtypes(m)|
$\Rightarrow$ (abs-has-common-classes(m, argtypes(s))
$\Rightarrow$ contra-unspec-args-co-result(m, s)))
$\wedge$ (($\vec{c} \leq_{inh}$ specializers(m)
$\wedge$ ($\neg\, \exists\, m_1 \in M.$ msg(m_1) = msg(m) $\wedge$ |argtypes(m_1)| = |argtypes(m)|
$\wedge\ \vec{c} \leq_{inh}$ specializers(m_1) $\wedge\ m_1 \leq_{meth} m \wedge m_1 \neq m$))
$\Rightarrow$ ($\forall\, i \in$ indexes(specializers(m)).
specializers(m)$_i \neq any \Rightarrow\ \vec{c}_i$ <: argtypes(m)$_i$))

$\Leftrightarrow \langle$by definition of abs-has-common-classes(m, argtypes(s))$\rangle$
$\forall\, s \in S \,.\, \forall\, m \in M.\ \forall\, \vec{c} \in (C_{concrete})^*$.
(msg(m) = msg(s) $\wedge$ |argtypes(s)| = |argtypes(m)|
$\Rightarrow$ (($\exists\, \vec{c} \in$ covered-class-vecs(m) .$\vec{c}$ <: argtypes(s))
$\Rightarrow$ contra-unspec-args-co-result(m, s)))
$\wedge$ (($\vec{c} \leq_{inh}$ specializers(m)
$\wedge$ ($\neg\, \exists\, m_1 \in M.$ msg(m_1) = msg(m) $\wedge$ |argtypes(m_1)| = |argtypes(m)|
$\wedge\ \vec{c} \leq_{inh}$ specializers(m_1) $\wedge\ m_1 \leq_{meth} m \wedge m_1 \neq m$))
$\Rightarrow$ ($\forall\, i \in$ indexes(specializers(m)).
specializers(m)$_i \neq any \Rightarrow\ \vec{c}_i$ <: argtypes(m)$_i$))

$\Leftrightarrow \langle$by definition of covered-class-vecs(m)$\rangle$
$\forall\, s \in S \,.\, \forall\, m \in M.\ \forall\, \vec{c} \in (C_{concrete})^*$.
(msg(m) = msg(s) $\wedge$ |argtypes(s)| = |argtypes(m)|
$\Rightarrow$ (($\exists\, \vec{c} \in \{\, \vec{c} \in (C_{concrete})^* \mid \vec{c} \leq_{inh}$ specializers(m)
$\wedge$ ($\neg\, \exists\, m_1 \in M.$ msg(m_1) = msg(m) $\wedge$ |argtypes(m_1)| = |argtypes(m)|
$\wedge\ \vec{c} \leq_{inh}$ specializers(m_1) $\wedge\ m_1 \leq_{meth} m \wedge m_1 \neq m$) $\}$.
$\vec{c}$ <: argtypes(s))
$\Rightarrow$ contra-unspec-args-co-result(m, s)))
$\wedge$ (($\vec{c} \leq_{inh}$ specializers(m)
$\wedge$ ($\neg\, \exists\, m_1 \in M.$ msg(m_1) = msg(m) $\wedge$ |argtypes(m_1)| = |argtypes(m)|
$\wedge\ \vec{c} \leq_{inh}$ specializers(m_1) $\wedge\ m_1 \leq_{meth} m \wedge m_1 \neq m$))
$\Rightarrow$ ($\forall\, i \in$ indexes(specializers(m)).
specializers(m)$_i \neq any \Rightarrow\ \vec{c}_i$ <: argtypes(m)$_i$))

$\Leftrightarrow \langle$by set theory$\rangle$

$$
\begin{aligned}
&\forall\, s \in S\,.\ \forall\, m \in M.\ \forall\, \vec{c} \in (C_{concrete})^*\,.\\
&\quad (\text{msg}(m) = \text{msg}(s)\ \wedge |\text{argtypes}(s)| = |\text{argtypes}(m)|\\
&\qquad \Rightarrow ((\exists\, \vec{c} \in (C_{concrete})^*\,.\\
&\qquad\qquad \vec{c} \leq_{inh} \text{specializers}(m)\\
&\qquad\qquad \wedge (\neg \exists\, m_1 \in M.\ \text{msg}(m_1) = \text{msg}(m)\ \wedge |\text{argtypes}(m_1)| = |\text{argtypes}(m)|\\
&\qquad\qquad\qquad \wedge\, \vec{c} \leq_{inh} \text{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)\\
&\qquad\qquad \wedge\, \vec{c} <: \text{argtypes}(s))\\
&\qquad\quad \Rightarrow \text{contra-unspec-args-co-result}(m, s)))\\
&\quad \wedge ((\vec{c} \leq_{inh} \text{specializers}(m)\\
&\qquad\quad \wedge (\neg \exists\, m_1 \in M.\ \text{msg}(m_1) = \text{msg}(m)\ \wedge |\text{argtypes}(m_1)| = |\text{argtypes}(m)|\\
&\qquad\qquad\qquad \wedge\, \vec{c} \leq_{inh} \text{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)\ )\\
&\qquad \Rightarrow (\forall\, i \in \text{indexes}(\text{specializers}(m)).\\
&\qquad\qquad \text{specializers}(m)_i \neq any\ \Rightarrow\ \vec{c}_i <: \text{argtypes}(m)_i))
\end{aligned}
$$

$\Leftrightarrow \langle$by definition of contra-unspec-args-co-result$(m, s)\rangle$

$$
\begin{aligned}
&\forall\, s \in S\,.\ \forall\, m \in M.\ \forall\, \vec{c} \in (C_{concrete})^*\,.\\
&\quad (\text{msg}(m) = \text{msg}(s)\ \wedge |\text{argtypes}(s)| = |\text{argtypes}(m)|\\
&\qquad \Rightarrow ((\exists\, \vec{c} \in (C_{concrete})^*\,.\\
&\qquad\qquad \vec{c} \leq_{inh} \text{specializers}(m)\\
&\qquad\qquad \wedge (\neg \exists\, m_1 \in M.\ \text{msg}(m_1) = \text{msg}(m)\ \wedge |\text{argtypes}(m_1)| = |\text{argtypes}(m)|\\
&\qquad\qquad\qquad \wedge\, \vec{c} \leq_{inh} \text{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)\\
&\qquad\qquad \wedge\, \vec{c} <: \text{argtypes}(s))\\
&\qquad\quad \Rightarrow ((\forall\, i \in \text{indexes}(\text{specializers}(m)).\\
&\qquad\qquad\quad \text{specializers}(m)_i = any\ \Rightarrow\ \text{argtypes}(s)_i \leq_{sub} \text{argtypes}(m)_i)\\
&\qquad\qquad \wedge\, \text{restype}(m) \leq_{sub} \text{restype}(s)))))\\
&\quad \wedge ((\vec{c} \leq_{inh} \text{specializers}(m)\\
&\qquad\quad \wedge (\neg \exists\, m_1 \in M.\ \text{msg}(m_1) = \text{msg}(m)\ \wedge |\text{argtypes}(m_1)| = |\text{argtypes}(m)|\\
&\qquad\qquad\qquad \wedge\, \vec{c} \leq_{inh} \text{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)\ )\\
&\qquad \Rightarrow (\forall\, i \in \text{indexes}(\text{specializers}(m)).\\
&\qquad\qquad \text{specializers}(m)_i \neq any\ \Rightarrow\ \vec{c}_i <: \text{argtypes}(m)_i))
\end{aligned}
$$

$\Leftrightarrow \langle$by $(P \Rightarrow (Q \Rightarrow R)) \Leftrightarrow (P \wedge Q \Rightarrow R)\rangle$

$$
\begin{aligned}
&\forall\, s \in S\,.\ \forall\, m \in M.\ \forall\, \vec{c} \in (C_{concrete})^*\,.\\
&\quad (\text{msg}(m) = \text{msg}(s)\ \wedge |\text{argtypes}(s)| = |\text{argtypes}(m)|\\
&\quad \wedge (\exists\, \vec{c} \in (C_{concrete})^*\,.\\
&\qquad\quad \vec{c} \leq_{inh} \text{specializers}(m)\\
&\qquad\quad \wedge (\neg \exists\, m_1 \in M.\ \text{msg}(m_1) = \text{msg}(m)\ \wedge |\text{argtypes}(m_1)| = |\text{argtypes}(m)|\\
&\qquad\qquad\qquad \wedge\, \vec{c} \leq_{inh} \text{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)\\
&\qquad\quad \wedge\, \vec{c} <: \text{argtypes}(s)))
\end{aligned}
$$

$$
\begin{array}{l}
\qquad\qquad \Rightarrow ((\forall\, i \in \text{indexes(specializers}(m)). \\
\qquad\qquad\qquad\quad \text{specializers}(m)_i = any \;\Rightarrow\; \text{argtypes}(s)_i \leq_{sub} \text{argtypes}(m)_i) \\
\qquad\qquad\quad \wedge\ \text{restype}(m) \leq_{sub} \text{restype}(s)))) \\
\qquad \wedge\ ((\vec{c} \leq_{inh} \text{specializers}(m) \\
\qquad\qquad\quad \wedge\ (\neg\, \exists\, m_1 \in M.\ \text{msg}(m_1) = \text{msg}(m)\ \wedge |\text{argtypes}(m_1)| = |\text{argtypes}(m)| \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)\ ) \\
\qquad\qquad \Rightarrow (\forall\, i \in \text{indexes(specializers}(m)). \\
\qquad\qquad\qquad\quad \text{specializers}(m)_i \neq any \;\Rightarrow\; \vec{c}_i <: \text{argtypes}(m)_i))
\end{array}
$$

$$
\begin{array}{l}
\Rightarrow \langle \text{by } (\mathrm{P} \Rightarrow \mathrm{Q}) \Rightarrow (\mathrm{P} \wedge \mathrm{R} \Rightarrow \mathrm{Q}), \text{ twice} \rangle \\
\quad \forall\, s \in S\,.\ \forall\, m \in M.\ \forall\, \vec{c} \in (C_{concrete})^*\,. \\
\qquad \text{msg}(m) = \text{msg}(s)\ \wedge |\text{argtypes}(s)| = |\text{argtypes}(m)| \\
\qquad \wedge\ \vec{c} \leq_{inh} \text{specializers}(m) \\
\qquad \wedge\ (\neg\, \exists\, m_1 \in M.\ \text{msg}(m_1) = \text{msg}(m)\ \wedge |\text{argtypes}(m_1)| = |\text{argtypes}(m)| \\
\qquad\qquad\qquad\qquad \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m) \\
\qquad \wedge\ (\exists\, \vec{c} \in (C_{concrete})^*\,. \\
\qquad\qquad\quad \vec{c} \leq_{inh} \text{specializers}(m) \\
\qquad\qquad\quad \wedge\ (\neg\, \exists\, m_1 \in M.\ \text{msg}(m_1) = \text{msg}(m)\ \wedge |\text{argtypes}(m_1)| = |\text{argtypes}(m)| \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m) \\
\qquad\qquad\quad \wedge\ \vec{c} <: \text{argtypes}(s)) \\
\qquad\qquad \Rightarrow ((\forall\, i \in \text{indexes(specializers}(m)). \\
\qquad\qquad\qquad\quad \text{specializers}(m)_i = any \;\Rightarrow\; \text{argtypes}(s)_i \leq_{sub} \text{argtypes}(m)_i) \\
\qquad\qquad\quad \wedge\ \text{restype}(m) \leq_{sub} \text{restype}(s)))) \\
\qquad \text{msg}(m) = \text{msg}(s)\ \wedge |\text{argtypes}(s)| = |\text{argtypes}(m)| \\
\qquad \wedge\ \vec{c} \leq_{inh} \text{specializers}(m) \\
\qquad \wedge\ (\neg\, \exists\, m_1 \in M.\ \text{msg}(m_1) = \text{msg}(m)\ \wedge |\text{argtypes}(m_1)| = |\text{argtypes}(m)| \\
\qquad\qquad\qquad\qquad \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m) \\
\qquad \wedge\ (\exists\, \vec{c} \in (C_{concrete})^*\,. \\
\qquad\qquad\quad \vec{c} \leq_{inh} \text{specializers}(m) \\
\qquad\qquad\quad \wedge\ (\neg\, \exists\, m_1 \in M.\ \text{msg}(m_1) = \text{msg}(m)\ \wedge |\text{argtypes}(m_1)| = |\text{argtypes}(m)| \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m) \\
\qquad\qquad\quad \wedge\ \vec{c} <: \text{argtypes}(s)) \\
\qquad\qquad \Rightarrow (\forall\, i \in \text{indexes(specializers}(m)). \\
\qquad\qquad\qquad\quad \text{specializers}(m)_i \neq any \;\Rightarrow\; \vec{c}_i <: \text{argtypes}(m)_i))
\end{array}
$$

$\Leftrightarrow \langle$by $((P \Rightarrow Q) \wedge (P \Rightarrow R)) \Leftrightarrow (P \Rightarrow Q \wedge R)\rangle$

$\forall s \in S\,.\, \forall m \in M.\, \forall \vec{c} \in (C_{concrete})^*\,.$

$\quad \mathsf{msg}(m) = \mathsf{msg}(s) \wedge |\mathsf{argtypes}(s)| = |\mathsf{argtypes}(m)| \wedge \vec{c} \le_{inh} \mathsf{specializers}(m)$

$\quad \wedge (\neg \exists\, m_1 \in M.\, \mathsf{msg}(m_1) = \mathsf{msg}(m) \wedge |\mathsf{argtypes}(m_1)| = |\mathsf{argtypes}(m)|$

$\qquad\qquad \wedge \vec{c} \le_{inh} \mathsf{specializers}(m_1) \wedge m_1 \le_{meth} m \wedge m_1 \ne m)$

$\quad \wedge (\exists\, \vec{c} \in (C_{concrete})^*\,.$

$\qquad \vec{c} \le_{inh} \mathsf{specializers}(m)$

$\qquad \wedge (\neg \exists\, m_1 \in M.\, \mathsf{msg}(m_1) = \mathsf{msg}(m) \wedge |\mathsf{argtypes}(m_1)| = |\mathsf{argtypes}(m)|$

$\qquad\qquad \wedge \vec{c} \le_{inh} \mathsf{specializers}(m_1) \wedge m_1 \le_{meth} m \wedge m_1 \ne m)$

$\qquad \wedge \vec{c} <: \mathsf{argtypes}(s))$

$\qquad \Rightarrow ((\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)).$

$\qquad\qquad \mathsf{specializers}(m)_i = any \Rightarrow \mathsf{argtypes}(s)_i \le_{sub} \mathsf{argtypes}(m)_i)$

$\qquad \wedge (\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)).$

$\qquad\qquad \mathsf{specializers}(m)_i \ne any \Rightarrow \vec{c}_i <: \mathsf{argtypes}(m)_i)$

$\qquad \wedge \mathsf{restype}(m) \le_{sub} \mathsf{restype}(s))$

$\Leftrightarrow \langle$by joining the term$\rangle$

$\forall s \in S\,.\, \forall m \in M.\, \forall \vec{c} \in (C_{concrete})^*\,.$

$\quad \mathsf{msg}(m) = \mathsf{msg}(s) \wedge |\mathsf{argtypes}(s)| = |\mathsf{argtypes}(m)| \wedge \vec{c} \le_{inh} \mathsf{specializers}(m)$

$\quad \wedge (\neg \exists\, m_1 \in M.\, \mathsf{msg}(m_1) = \mathsf{msg}(m) \wedge |\mathsf{argtypes}(m_1)| = |\mathsf{argtypes}(m)|$

$\qquad\qquad \wedge \vec{c} \le_{inh} \mathsf{specializers}(m_1) \wedge m_1 \le_{meth} m \wedge m_1 \ne m)$

$\quad \wedge (\exists\, \vec{c} \in (C_{concrete})^*\,.$

$\qquad \vec{c} \le_{inh} \mathsf{specializers}(m)$

$\qquad \wedge (\neg \exists\, m_1 \in M.\, \mathsf{msg}(m_1) = \mathsf{msg}(m) \wedge |\mathsf{argtypes}(m_1)| = |\mathsf{argtypes}(m)|$

$\qquad\qquad \wedge \vec{c} \le_{inh} \mathsf{specializers}(m_1) \wedge m_1 \le_{meth} m \wedge m_1 \ne m)$

$\qquad \wedge \vec{c} <: \mathsf{argtypes}(s))$

$\qquad \Rightarrow ((\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)).$

$\qquad\qquad (\mathsf{specializers}(m)_i = any \Rightarrow \mathsf{argtypes}(s)_i \le_{sub} \mathsf{argtypes}(m)_i)$

$\qquad\qquad \wedge (\mathsf{specializers}(m)_i \ne any \Rightarrow \vec{c}_i <: \mathsf{argtypes}(m)_i))$

$\qquad \wedge \mathsf{restype}(m) \le_{sub} \mathsf{restype}(s))$

$\Rightarrow \langle$by $P \Leftrightarrow P \wedge P$, the instantiation rule for existential quantifiers, and transitivity of implication$\rangle$

$\forall s \in S\,.\, \forall m \in M.\, \forall \vec{c} \in (C_{concrete})^*\,.$

$\quad \mathsf{msg}(m) = \mathsf{msg}(s) \wedge |\mathsf{argtypes}(s)| = |\mathsf{argtypes}(m)| \wedge \vec{c} \le_{inh} \mathsf{specializers}(m)$

$\quad \wedge (\neg \exists\, m_1 \in M.\, \mathsf{msg}(m_1) = \mathsf{msg}(m) \wedge |\mathsf{argtypes}(m_1)| = |\mathsf{argtypes}(m)|$

$\qquad\qquad \wedge \vec{c} \le_{inh} \mathsf{specializers}(m_1) \wedge m_1 \le_{meth} m \wedge m_1 \ne m)$

$\quad \wedge \vec{c} <: \mathsf{argtypes}(s)$

$\qquad \Rightarrow ((\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)).$

$\qquad\qquad (\mathsf{specializers}(m)_i = any \Rightarrow \mathsf{argtypes}(s)_i \le_{sub} \mathsf{argtypes}(m)_i)$

$\qquad\qquad \wedge (\mathsf{specializers}(m)_i \ne any \Rightarrow \vec{c}_i <: \mathsf{argtypes}(m)_i))$

$\qquad \wedge \mathsf{restype}(m) \le_{sub} \mathsf{restype}(s))$

$$\begin{array}{l}
\Rightarrow \langle \text{by } (P \Rightarrow Q) \Leftrightarrow (P \Rightarrow P \wedge Q)\text{, and } (A \Rightarrow B \wedge C) \Rightarrow (A \Rightarrow B) \rangle \\
\quad \forall\, s \in S\,.\ \forall\, m \in M.\ \forall\, \vec{c} \in (C_{concrete})^*\,. \\
\qquad \mathrm{msg}(m) = \mathrm{msg}(s) \ \wedge |\mathrm{argtypes}(s)| = |\mathrm{argtypes}(m)| \wedge \vec{c} \leq_{inh} \mathrm{specializers}(m) \\
\qquad \wedge\, (\neg\, \exists\, m_1 \in M.\ \mathrm{msg}(m_1) = \mathrm{msg}(m) \ \wedge |\mathrm{argtypes}(m_1)| = |\mathrm{argtypes}(m)| \\
\qquad\qquad\qquad\qquad \wedge\, \vec{c} \leq_{inh} \mathrm{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m) \\
\qquad \wedge\, \vec{c} <: \mathrm{argtypes}(s) \\
\qquad\quad \Rightarrow (\vec{c} <: \mathrm{argtypes}(s) \\
\qquad\qquad \wedge\, (\forall\, i \in \mathrm{indexes}(\mathrm{specializers}(m)). \\
\qquad\qquad\quad (\mathrm{specializers}(m)_i = any \ \Rightarrow\ \mathrm{argtypes}(s)_i \leq_{sub} \mathrm{argtypes}(m)_i) \\
\qquad\qquad\quad \wedge\, (\mathrm{specializers}(m)_i \neq any \ \Rightarrow\ \vec{c}_i <: \mathrm{argtypes}(m)_i)) \\
\qquad\qquad \wedge\, \mathrm{restype}(m) \leq_{sub} \mathrm{restype}(s))
\end{array}$$

$$\begin{array}{l}
\Leftrightarrow \langle \text{by definition of } <: \text{ for vectors } |\vec{c}| = |\mathrm{argtypes}(s)|\text{, and law of excluded middle} \rangle \\
\quad \forall\, s \in S\,.\ \forall\, m \in M.\ \forall\, \vec{c} \in (C_{concrete})^*\,. \\
\qquad \mathrm{msg}(m) = \mathrm{msg}(s) \ \wedge |\mathrm{argtypes}(s)| = |\mathrm{argtypes}(m)| \wedge \vec{c} \leq_{inh} \mathrm{specializers}(m) \\
\qquad \wedge\, (\neg\, \exists\, m_1 \in M.\ \mathrm{msg}(m_1) = \mathrm{msg}(m) \ \wedge |\mathrm{argtypes}(m_1)| = |\mathrm{argtypes}(m)| \\
\qquad\qquad\qquad\qquad \wedge\, \vec{c} \leq_{inh} \mathrm{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m) \\
\qquad \wedge\, \vec{c} <: \mathrm{argtypes}(s) \wedge |\vec{c}| = |\mathrm{argtypes}(m)| \\
\qquad\quad \Rightarrow (\vec{c} <: \mathrm{argtypes}(s) \\
\qquad\qquad \wedge\, (\mathrm{indexes}(\mathrm{specializers}(m) = \varnothing \vee \mathrm{indexes}(\mathrm{specializers}(m) \neq \varnothing) \\
\qquad\qquad \wedge\, (\forall\, i \in \mathrm{indexes}(\mathrm{specializers}(m)). \\
\qquad\qquad\quad (\mathrm{specializers}(m)_i = any \ \Rightarrow\ \mathrm{argtypes}(s)_i \leq_{sub} \mathrm{argtypes}(m)_i) \\
\qquad\qquad\quad \wedge\, (\mathrm{specializers}(m)_i \neq any \ \Rightarrow\ \vec{c}_i <: \mathrm{argtypes}(m)_i)) \\
\qquad\qquad \wedge\, \mathrm{restype}(m) \leq_{sub} \mathrm{restype}(s))
\end{array}$$

$$\begin{array}{l}
\Leftrightarrow \langle \text{by distribution of conjunction over disjunction} \rangle \\
\quad \forall\, s \in S\,.\ \forall\, m \in M.\ \forall\, \vec{c} \in (C_{concrete})^*\,. \\
\qquad \mathrm{msg}(m) = \mathrm{msg}(s) \ \wedge |\mathrm{argtypes}(s)| = |\mathrm{argtypes}(m)| \wedge \vec{c} \leq_{inh} \mathrm{specializers}(m) \\
\qquad \wedge\, (\neg\, \exists\, m_1 \in M.\ \mathrm{msg}(m_1) = \mathrm{msg}(m) \ \wedge |\mathrm{argtypes}(m_1)| = |\mathrm{argtypes}(m)| \\
\qquad\qquad\qquad\qquad \wedge\, \vec{c} \leq_{inh} \mathrm{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m) \\
\qquad \wedge\, \vec{c} <: \mathrm{argtypes}(s) \wedge |\vec{c}| = |\mathrm{argtypes}(m)| \\
\qquad\quad \Rightarrow ((\vec{c} <: \mathrm{argtypes}(s) \wedge \mathrm{indexes}(\mathrm{specializers}(m) = \varnothing \\
\qquad\qquad\quad \wedge\, (\forall\, i \in \mathrm{indexes}(\mathrm{specializers}(m)). \\
\qquad\qquad\qquad (\mathrm{specializers}(m)_i = any \ \Rightarrow\ \mathrm{argtypes}(s)_i \leq_{sub} \mathrm{argtypes}(m)_i) \\
\qquad\qquad\qquad \wedge\, (\mathrm{specializers}(m)_i \neq any \ \Rightarrow\ \vec{c}_i <: \mathrm{argtypes}(m)_i))) \\
\qquad\qquad \vee\, (\vec{c} <: \mathrm{argtypes}(s) \wedge \mathrm{indexes}(\mathrm{specializers}(m) \neq \varnothing) \\
\qquad\qquad\quad \wedge\, (\forall\, i \in \mathrm{indexes}(\mathrm{specializers}(m)). \\
\qquad\qquad\qquad (\mathrm{specializers}(m)_i = any \ \Rightarrow\ \mathrm{argtypes}(s)_i \leq_{sub} \mathrm{argtypes}(m)_i) \\
\qquad\qquad\qquad \wedge\, (\mathrm{specializers}(m)_i \neq any \ \Rightarrow\ \vec{c}_i <: \mathrm{argtypes}(m)_i)))) \\
\qquad\qquad \wedge\, \mathrm{restype}(m) \leq_{sub} \mathrm{restype}(s))
\end{array}$$

$\Leftrightarrow \langle$by conjunction distributes over universal quantification with non-empty range$\rangle$

$$
\begin{aligned}
&\forall\, s \in S\,.\ \forall\, m \in M.\ \forall\, \vec{c} \in (C_{concrete})^*\ .\\
&\quad \mathsf{msg}(m) = \mathsf{msg}(s)\ \wedge |\mathsf{argtypes}(s)| = |\mathsf{argtypes}(m)| \wedge \vec{c} \le_{inh} \mathsf{specializers}(m)\\
&\quad \wedge (\neg\, \exists\, m_1 \in M.\ \mathsf{msg}(m_1) = \mathsf{msg}(m)\ \wedge |\mathsf{argtypes}(m_1)| = |\mathsf{argtypes}(m)|\\
&\qquad\qquad\qquad \wedge\ \vec{c} \le_{inh} \mathsf{specializers}(m_1) \wedge m_1 \le_{meth} m \wedge m_1 \neq m)\\
&\quad \wedge\ \vec{c}\ \text{<:}\ \mathsf{argtypes}(s) \wedge |\vec{c}| = |\mathsf{argtypes}(m)|\\
&\qquad \Rightarrow ((\vec{c}\ \text{<:}\ \mathsf{argtypes}(s) \wedge \mathsf{indexes}(\mathsf{specializers}(m) = \emptyset\\
&\qquad\qquad \wedge (\forall\, i \in\ \mathsf{indexes}(\mathsf{specializers}(m)).\\
&\qquad\qquad\quad (\mathsf{specializers}(m)_i = any\ \Rightarrow\ \mathsf{argtypes}(s)_i \le_{sub} \mathsf{argtypes}(m)_i)\\
&\qquad\qquad\quad \wedge (\mathsf{specializers}(m)_i \neq any\ \Rightarrow\ \ \vec{c}_i\ \text{<:}\ \mathsf{argtypes}(m)_i)))\\
&\qquad\quad \vee (\mathsf{indexes}(\mathsf{specializers}(m) \neq \emptyset)\\
&\qquad\qquad \wedge (\forall\, i \in\ \mathsf{indexes}(\mathsf{specializers}(m)).\\
&\qquad\qquad\quad \vec{c}\ \text{<:}\ \mathsf{argtypes}(s)\\
&\qquad\qquad\quad \wedge (\mathsf{specializers}(m)_i = any\ \Rightarrow\ \mathsf{argtypes}(s)_i \le_{sub} \mathsf{argtypes}(m)_i))\\
&\qquad\qquad\quad \wedge (\mathsf{specializers}(m)_i \neq any\ \Rightarrow\ \ \vec{c}_i\ \text{<:}\ \mathsf{argtypes}(m)_i))))\\
&\qquad\quad \wedge\ \mathsf{restype}(m) \le_{sub} \mathsf{restype}(s))
\end{aligned}
$$

$\Leftrightarrow \langle$by definition of $\vec{c}$ <: argtypes$(s)\rangle$

$$
\begin{aligned}
&\forall\, s \in S\,.\ \forall\, m \in M.\ \forall\, \vec{c} \in (C_{concrete})^*\ .\\
&\quad \mathsf{msg}(m) = \mathsf{msg}(s)\ \wedge |\mathsf{argtypes}(s)| = |\mathsf{argtypes}(m)| \wedge \vec{c} \le_{inh} \mathsf{specializers}(m)\\
&\quad \wedge (\neg\, \exists\, m_1 \in M.\ \mathsf{msg}(m_1) = \mathsf{msg}(m)\ \wedge |\mathsf{argtypes}(m_1)| = |\mathsf{argtypes}(m)|\\
&\qquad\qquad\qquad \wedge\ \vec{c} \le_{inh} \mathsf{specializers}(m_1) \wedge m_1 \le_{meth} m \wedge m_1 \neq m)\\
&\quad \wedge\ \vec{c}\ \text{<:}\ \mathsf{argtypes}(s) \wedge |\vec{c}| = |\mathsf{argtypes}(m)|\\
&\qquad \Rightarrow ((\vec{c}\ \text{<:}\ \mathsf{argtypes}(s) \wedge \mathsf{indexes}(\mathsf{specializers}(m) = \emptyset\\
&\qquad\qquad \wedge (\forall\, i \in\ \mathsf{indexes}(\mathsf{specializers}(m)).\\
&\qquad\qquad\quad (\mathsf{specializers}(m)_i = any\ \Rightarrow\ \mathsf{argtypes}(s)_i \le_{sub} \mathsf{argtypes}(m)_i)\\
&\qquad\qquad\quad \wedge (\mathsf{specializers}(m)_i \neq any\ \Rightarrow\ \ \vec{c}_i\ \text{<:}\ \mathsf{argtypes}(m)_i)))\\
&\qquad\quad \vee (\mathsf{indexes}(\mathsf{specializers}(m) \neq \emptyset)\\
&\qquad\qquad \wedge (\forall\, i \in\ \mathsf{indexes}(\mathsf{specializers}(m)).\\
&\qquad\qquad\quad |\vec{c}| =\ |\mathsf{argtypes}(s)| \wedge \vec{c}_i\ \text{<:}\ \mathsf{argtypes}(s)_i\\
&\qquad\qquad\quad \wedge (\mathsf{specializers}(m)_i = any\ \Rightarrow\ \mathsf{argtypes}(s)_i \le_{sub} \mathsf{argtypes}(m)_i)\\
&\qquad\qquad\quad \wedge (\mathsf{specializers}(m)_i \neq any\ \Rightarrow\ \ \vec{c}_i\ \text{<:}\ \mathsf{argtypes}(m)_i))))\\
&\qquad\quad \wedge\ \mathsf{restype}(m) \le_{sub} \mathsf{restype}(s))
\end{aligned}
$$

$\Leftrightarrow \langle$by (A $\wedge$ (B$\Rightarrow$ C)) $\Leftrightarrow$ (A $\wedge$ (B $\Rightarrow$ (A $\wedge$ C))$\rangle$

$$
\begin{aligned}
&\forall\, s \in S\,.\ \forall\, m \in M.\ \forall\, \vec{c} \in (C_{concrete})^*\ .\\
&\quad \mathsf{msg}(m) = \mathsf{msg}(s)\ \wedge |\mathsf{argtypes}(s)| = |\mathsf{argtypes}(m)| \wedge \vec{c} \le_{inh} \mathsf{specializers}(m)\\
&\quad \wedge (\neg\, \exists\, m_1 \in M.\ \mathsf{msg}(m_1) = \mathsf{msg}(m)\ \wedge |\mathsf{argtypes}(m_1)| = |\mathsf{argtypes}(m)|\\
&\qquad\qquad\qquad \wedge\ \vec{c} \le_{inh} \mathsf{specializers}(m_1) \wedge m_1 \le_{meth} m \wedge m_1 \neq m)\\
&\quad \wedge\ \vec{c}\ \text{<:}\ \mathsf{argtypes}(s) \wedge |\vec{c}| = |\mathsf{argtypes}(m)|\\
&\qquad \Rightarrow ((\vec{c}\ \text{<:}\ \mathsf{argtypes}(s) \wedge \mathsf{indexes}(\mathsf{specializers}(m) = \emptyset\\
&\qquad\qquad \wedge (\forall\, i \in\ \mathsf{indexes}(\mathsf{specializers}(m)).\\
&\qquad\qquad\quad (\mathsf{specializers}(m)_i = any\ \Rightarrow\ \mathsf{argtypes}(s)_i \le_{sub} \mathsf{argtypes}(m)_i)
\end{aligned}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad \wedge\ (\mathsf{specializers}(m)_i \neq any \ \Rightarrow\ \vec{c}_i <: \mathsf{argtypes}(m)_i))) \\
\qquad\quad \vee\ (\mathsf{indexes}(\mathsf{specializers}(m) \neq \varnothing) \\
\qquad\qquad \wedge\ (\forall\ i \in\ \mathsf{indexes}(\mathsf{specializers}(m)). \\
\qquad\qquad\qquad |\vec{c}| =\ |\mathsf{argtypes}(s)| \wedge \vec{c}_i <: \mathsf{argtypes}(s)_i \\
\qquad\qquad\qquad \wedge\ (\mathsf{specializers}(m)_i = any \\
\qquad\qquad\qquad\qquad \Rightarrow\ \vec{c}_i <: \mathsf{argtypes}(s)_i \wedge \mathsf{argtypes}(s)_i \leq_{sub} \mathsf{argtypes}(m)_i) \\
\qquad\qquad\qquad \wedge\ (\mathsf{specializers}(m)_i \neq any \ \Rightarrow\ \vec{c}_i <: \mathsf{argtypes}(m)_i)))) \\
\qquad\quad \wedge\ \mathsf{restype}(m) \leq_{sub} \mathsf{restype}(s))
\end{array}
$$

$\Rightarrow \langle$by definition of $\vec{c}$ <: argtypes(s) and <:$\rangle$

$$
\begin{array}{l}
\forall\ s \in S\ .\ \forall\ m \in M.\ \forall\ \vec{c} \in (C_{concrete})^*\ . \\
\quad \mathsf{msg}(m) = \mathsf{msg}(s)\ \wedge |\mathsf{argtypes}(s)| = |\mathsf{argtypes}(m)| \wedge \vec{c} \leq_{inh} \mathsf{specializers}(m) \\
\quad \wedge\ (\neg\ \exists\ m_1 \in M.\ \mathsf{msg}(m_1) = \mathsf{msg}(m)\ \wedge |\mathsf{argtypes}(m_1)| = |\mathsf{argtypes}(m)| \\
\qquad\qquad\qquad \wedge\ \vec{c} \leq_{inh} \mathsf{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m) \\
\quad \wedge\ \vec{c}\ <: \mathsf{argtypes}(s) \wedge |\vec{c}| = |\mathsf{argtypes}(m)| \\
\qquad \Rightarrow ((\vec{c}\ <: \mathsf{argtypes}(s) \wedge \mathsf{indexes}(\mathsf{specializers}(m) = \varnothing \\
\qquad\qquad \wedge\ (\forall\ i \in\ \mathsf{indexes}(\mathsf{specializers}(m)). \\
\qquad\qquad\qquad (\mathsf{specializers}(m)_i = any\ \Rightarrow\ \mathsf{argtypes}(s)_i \leq_{sub} \mathsf{argtypes}(m)_i) \\
\qquad\qquad\qquad \wedge\ (\mathsf{specializers}(m)_i \neq any \ \Rightarrow\ \vec{c}_i <: \mathsf{argtypes}(m)_i))) \\
\qquad\quad \vee\ (\mathsf{indexes}(\mathsf{specializers}(m) \neq \varnothing) \\
\qquad\qquad \wedge\ (\forall\ i \in\ \mathsf{indexes}(\mathsf{specializers}(m)). \\
\qquad\qquad\qquad \vec{c}\ <: \mathsf{argtypes}(s) \\
\qquad\qquad\qquad \wedge\ (\mathsf{specializers}(m)_i = any\ \Rightarrow\ \vec{c}_i <: \mathsf{argtypes}(m)_i) \\
\qquad\qquad\qquad \wedge\ (\mathsf{specializers}(m)_i \neq any \ \Rightarrow\ \vec{c}_i <: \mathsf{argtypes}(m)_i)))) \\
\qquad\quad \wedge\ \mathsf{restype}(m) \leq_{sub} \mathsf{restype}(s))
\end{array}
$$

$\Leftrightarrow \langle$by conjunction distributes over universal quantification with non-empty range$\rangle$

$$
\begin{array}{l}
\forall\ s \in S\ .\ \forall\ m \in M.\ \forall\ \vec{c} \in (C_{concrete})^*\ . \\
\quad \mathsf{msg}(m) = \mathsf{msg}(s)\ \wedge |\mathsf{argtypes}(s)| = |\mathsf{argtypes}(m)| \wedge \vec{c} \leq_{inh} \mathsf{specializers}(m) \\
\quad \wedge\ (\neg\ \exists\ m_1 \in M.\ \mathsf{msg}(m_1) = \mathsf{msg}(m)\ \wedge |\mathsf{argtypes}(m_1)| = |\mathsf{argtypes}(m)| \\
\qquad\qquad\qquad \wedge\ \vec{c} \leq_{inh} \mathsf{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m) \\
\quad \wedge\ \vec{c}\ <: \mathsf{argtypes}(s) \wedge |\vec{c}| = |\mathsf{argtypes}(m)| \\
\qquad \Rightarrow ((\vec{c}\ <: \mathsf{argtypes}(s) \wedge \mathsf{indexes}(\mathsf{specializers}(m) = \varnothing \\
\qquad\qquad \wedge\ (\forall\ i \in\ \mathsf{indexes}(\mathsf{specializers}(m)). \\
\qquad\qquad\qquad (\mathsf{specializers}(m)_i = any\ \Rightarrow\ \mathsf{argtypes}(s)_i \leq_{sub} \mathsf{argtypes}(m)_i) \\
\qquad\qquad\qquad \wedge\ (\mathsf{specializers}(m)_i \neq any \ \Rightarrow\ \vec{c}_i <: \mathsf{argtypes}(m)_i))) \\
\qquad\quad \vee\ (\vec{c}\ <: \mathsf{argtypes}(s) \wedge \mathsf{indexes}(\mathsf{specializers}(m) \neq \varnothing) \\
\qquad\qquad \wedge\ (\forall\ i \in\ \mathsf{indexes}(\mathsf{specializers}(m)). \\
\qquad\qquad\qquad (\mathsf{specializers}(m)_i = any\ \Rightarrow\ \vec{c}_i <: \mathsf{argtypes}(m)_i) \\
\qquad\qquad\qquad \wedge\ (\mathsf{specializers}(m)_i \neq any \ \Rightarrow\ \vec{c}_i <: \mathsf{argtypes}(m)_i)))) \\
\qquad\quad \wedge\ \mathsf{restype}(m) \leq_{sub} \mathsf{restype}(s))
\end{array}
$$

$\Leftrightarrow \langle$by empty range rule$\rangle$

$\forall s \in S .\ \forall m \in M .\ \forall \vec{c} \in (C_{concrete})^* .$

$\quad \mathsf{msg}(m) = \mathsf{msg}(s) \wedge |\mathsf{argtypes}(s)| = |\mathsf{argtypes}(m)| \wedge \vec{c} \leq_{inh} \mathsf{specializers}(m)$

$\quad \wedge (\neg \exists m_1 \in M .\ \mathsf{msg}(m_1) = \mathsf{msg}(m) \wedge |\mathsf{argtypes}(m_1)| = |\mathsf{argtypes}(m)|$

$\qquad\qquad \wedge \vec{c} \leq_{inh} \mathsf{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)$

$\quad \wedge \vec{c} <: \mathsf{argtypes}(s) \wedge |\vec{c}| = |\mathsf{argtypes}(m)|$

$\qquad \Rightarrow ((\vec{c} <: \mathsf{argtypes}(s) \wedge \mathsf{indexes}(\mathsf{specializers}(m) = \varnothing$

$\qquad\qquad \wedge \text{true})$

$\qquad\quad \vee (\vec{c} <: \mathsf{argtypes}(s) \wedge \mathsf{indexes}(\mathsf{specializers}(m) \neq \varnothing)$

$\qquad\qquad \wedge (\forall i \in \mathsf{indexes}(\mathsf{specializers}(m)).$

$\qquad\qquad\quad (\mathsf{specializers}(m)_i = any \Rightarrow \vec{c}_i <: \mathsf{argtypes}(m)_i)$

$\qquad\qquad\quad \wedge (\mathsf{specializers}(m)_i \neq any \Rightarrow \vec{c}_i <: \mathsf{argtypes}(m)_i))))$

$\qquad\quad \wedge \mathsf{restype}(m) \leq_{sub} \mathsf{restype}(s))$

$\Leftrightarrow \langle$by empty range rule$\rangle$

$\forall s \in S .\ \forall m \in M .\ \forall \vec{c} \in (C_{concrete})^* .$

$\quad \mathsf{msg}(m) = \mathsf{msg}(s) \wedge |\mathsf{argtypes}(s)| = |\mathsf{argtypes}(m)| \wedge \vec{c} \leq_{inh} \mathsf{specializers}(m)$

$\quad \wedge (\neg \exists m_1 \in M .\ \mathsf{msg}(m_1) = \mathsf{msg}(m) \wedge |\mathsf{argtypes}(m_1)| = |\mathsf{argtypes}(m)|$

$\qquad\qquad \wedge \vec{c} \leq_{inh} \mathsf{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)$

$\quad \wedge \vec{c} <: \mathsf{argtypes}(s) \wedge |\vec{c}| = |\mathsf{argtypes}(m)|$

$\qquad \Rightarrow ((\vec{c} <: \mathsf{argtypes}(s) \wedge \mathsf{indexes}(\mathsf{specializers}(m) = \varnothing$

$\qquad\qquad \wedge (\forall i \in \mathsf{indexes}(\mathsf{specializers}(m)).$

$\qquad\qquad\quad (\mathsf{specializers}(m)_i = any \Rightarrow \vec{c}_i <: \mathsf{argtypes}(m)_i)$

$\qquad\qquad\quad \wedge (\mathsf{specializers}(m)_i \neq any \Rightarrow \vec{c}_i <: \mathsf{argtypes}(m)_i)))$

$\qquad\quad \vee (\vec{c} <: \mathsf{argtypes}(s) \wedge \mathsf{indexes}(\mathsf{specializers}(m) \neq \varnothing)$

$\qquad\qquad \wedge (\forall i \in \mathsf{indexes}(\mathsf{specializers}(m)).$

$\qquad\qquad\quad (\mathsf{specializers}(m)_i = any \Rightarrow \vec{c}_i <: \mathsf{argtypes}(m)_i)$

$\qquad\qquad\quad \wedge (\mathsf{specializers}(m)_i \neq any \Rightarrow \vec{c}_i <: \mathsf{argtypes}(m)_i))))$

$\qquad\quad \wedge \mathsf{restype}(m) \leq_{sub} \mathsf{restype}(s))$

$\Leftrightarrow \langle$by conjunction distributes over disjunction (backwards)$\rangle$

$\forall s \in S .\ \forall m \in M .\ \forall \vec{c} \in (C_{concrete})^* .$

$\quad \mathsf{msg}(m) = \mathsf{msg}(s) \wedge |\mathsf{argtypes}(s)| = |\mathsf{argtypes}(m)| \wedge \vec{c} \leq_{inh} \mathsf{specializers}(m)$

$\quad \wedge (\neg \exists m_1 \in M .\ \mathsf{msg}(m_1) = \mathsf{msg}(m) \wedge |\mathsf{argtypes}(m_1)| = |\mathsf{argtypes}(m)|$

$\qquad\qquad \wedge \vec{c} \leq_{inh} \mathsf{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)$

$\quad \wedge \vec{c} <: \mathsf{argtypes}(s) \wedge |\vec{c}| = |\mathsf{argtypes}(m)|$

$\qquad \Rightarrow (\vec{c} <: \mathsf{argtypes}(s)$

$\qquad\quad \wedge (\mathsf{indexes}(\mathsf{specializers}(m) = \varnothing \vee \mathsf{indexes}(\mathsf{specializers}(m) \neq \varnothing)$

$\qquad\quad \wedge (\forall i \in \mathsf{indexes}(\mathsf{specializers}(m)).$

$\qquad\qquad (\mathsf{specializers}(m)_i = any \Rightarrow \vec{c}_i <: \mathsf{argtypes}(m)_i)$

$\qquad\qquad \wedge (\mathsf{specializers}(m)_i \neq any \Rightarrow \vec{c}_i <: \mathsf{argtypes}(m)_i))$

$\qquad\quad \wedge \mathsf{restype}(m) \leq_{sub} \mathsf{restype}(s))$

$\Leftrightarrow \langle$by law of excluded middle$\rangle$

$$
\begin{array}{l}
\forall\, s \in S\,.\ \forall\, m \in M.\ \forall\, \vec{c} \in (C_{concrete})^*\,.\\
\quad \text{msg}(m) = \text{msg}(s)\ \wedge |\text{argtypes}(s)| = |\text{argtypes}(m)| \wedge \vec{c} \leq_{inh} \text{specializers}(m)\\
\quad \wedge (\neg\, \exists\, m_1 \in M.\ \text{msg}(m_1) = \text{msg}(m)\ \wedge |\text{argtypes}(m_1)| = |\text{argtypes}(m)|\\
\qquad\qquad\qquad \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)\\
\quad \wedge\ \vec{c}\ <:\ \text{argtypes}(s) \wedge |\vec{c}\,| = |\text{argtypes}(m)|\\
\qquad \Rightarrow (\vec{c}\ <:\ \text{argtypes}(s)\\
\qquad\quad \wedge (\forall\, i \in \text{indexes}(\text{specializers}(m)).\\
\qquad\qquad (\text{specializers}(m)_i = any\ \Rightarrow\ \vec{c}_i <: \text{argtypes}(m)_i)\\
\qquad\qquad \wedge (\text{specializers}(m)_i \neq any\ \Rightarrow\ \vec{c}_i <: \text{argtypes}(m)_i))\\
\qquad\quad \wedge\ \text{restype}(m) \leq_{sub} \text{restype}(s))
\end{array}
$$

$\Rightarrow \langle$by $(A \Rightarrow B \wedge C) \Rightarrow (A \Rightarrow B))\rangle$

$$
\begin{array}{l}
\forall\, s \in S.\ \ \forall\, m \in M\,.\ \forall\, \vec{c} \in (C_{concrete})^*.\\
\quad \text{msg}(m) = \text{msg}(s)\ \wedge\ |\text{argtypes}(s)| = |\text{argtypes}(m)| \wedge \vec{c} \leq_{inh} \text{specializers}(m)\\
\quad \wedge (\neg\, \exists\, m_1 \in M\,.\ \text{msg}(m_1) = \text{msg}(s)\ \wedge\ |\vec{c}\,| = |\text{argtypes}(m_1)|\\
\qquad\quad \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)\\
\quad \wedge\ \vec{c}\ <:\ \text{argtypes}(s)\ \wedge\ |\vec{c}\,| = |\text{argtypes}(m)|\\
\qquad \Rightarrow ((\forall\, i \in \text{indexes}(\text{specializers}(m)).\\
\qquad\qquad (\text{specializers}(m)_i = any\ \Rightarrow\ \vec{c}_i <: \text{argtypes}(m)_i)\\
\qquad\qquad \wedge (\text{specializers}(m)_i \neq any\ \Rightarrow\ \vec{c}_i <: \text{argtypes}(m)_i))\\
\qquad\quad \wedge\ \text{restype}(m) \leq_{sub} \text{restype}(s))
\end{array}
$$

$\Leftrightarrow \langle$by definition of $\leq_{inh}$ for vectors, and commutivity of conjunction$\rangle$

$$
\begin{array}{l}
\forall\, s \in S.\ \ \forall\, m \in M\,.\ \forall\, \vec{c} \in (C_{concrete})^*.\\
\quad \text{msg}(m) = \text{msg}(s)\ \wedge\ |\vec{c}\,| = |\text{argtypes}(m)| \wedge \vec{c} \leq_{inh} \text{specializers}(m)\\
\quad \wedge (\neg\, \exists\, m_1 \in M\,.\ \text{msg}(m_1) = \text{msg}(s)\ \wedge\ |\vec{c}\,| = |\text{argtypes}(m_1)|\\
\qquad\quad \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)\\
\quad \wedge\ \vec{c}\ <:\ \text{argtypes}(s)\\
\qquad \Rightarrow ((\forall\, i \in \text{indexes}(\text{specializers}(m)).\\
\qquad\qquad (\text{specializers}(m)_i = any\ \Rightarrow\ \vec{c}_i <: \text{argtypes}(m)_i)\\
\qquad\qquad \wedge (\text{specializers}(m)_i \neq any\ \Rightarrow\ \vec{c}_i <: \text{argtypes}(m)_i))\\
\qquad\quad \wedge\ \text{restype}(m) \leq_{sub} \text{restype}(s))
\end{array}
$$

$\Leftrightarrow \langle$by $(P \Rightarrow (Q \Rightarrow R)) \Leftrightarrow (P \wedge Q \Rightarrow R)$, and $\wedge$ is commutative$\rangle$

$$
\begin{array}{l}
\forall\, s \in S.\ \ \forall\, m \in M\,.\ \forall\, \vec{c} \in (C_{concrete})^*.\\
\quad \vec{c}\ <:\ \text{argtypes}(s)\\
\quad \wedge\ \text{msg}(m) = \text{msg}(s)\ \wedge\ |\vec{c}\,| = |\text{argtypes}(m)|\ \wedge\ \vec{c} \leq_{inh} \text{specializers}(m)\\
\qquad \Rightarrow ((\neg\, \exists\, m_1 \in M\,.\\
\qquad\qquad\qquad \text{msg}(m_1) = \text{msg}(s)\ \wedge\ |\vec{c}\,| = |\text{argtypes}(m_1)|\\
\qquad\qquad\qquad \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)\\
\qquad\quad \Rightarrow ((\forall\, i \in \text{indexes}(\text{specializers}(m)).\\
\qquad\qquad\qquad (\text{specializers}(m)_i = any\\
\qquad\qquad\qquad\quad \Rightarrow\ \vec{c}_i <: \text{argtypes}(s)_i \wedge \text{argtypes}(s)_i \leq_{sub} \text{argtypes}(m)_i)\\
\qquad\qquad\quad \wedge (\text{specializers}(m)_i \neq any\ \Rightarrow\ \vec{c}_i <: \text{argtypes}(m)_i))\\
\qquad\qquad \wedge\ \text{restype}(m) \leq_{sub} \text{restype}(s))
\end{array}
$$

$\Longrightarrow \langle$by definition of <:$\rangle$

$$
\begin{array}{l}
\forall\, s \in S.\ \ \forall\, m \in M\,.\ \forall\, \vec{c} \in (C_{concrete})^*. \\
\quad \vec{c} <: \mathsf{argtypes}(s) \\
\quad \wedge\ \mathsf{msg}(m) = \mathsf{msg}(s)\ \wedge\ |\vec{c}| = |\mathsf{argtypes}(m)|\ \wedge\ \vec{c} \le_{inh} \mathsf{specializers}(m) \\
\qquad \Rightarrow ((\neg\, \exists\, m_1 \in M\,. \\
\qquad\qquad\qquad \mathsf{msg}(m_1) = \mathsf{msg}(s)\ \wedge\ |\vec{c}| = |\mathsf{argtypes}(m_1)| \\
\qquad\qquad\qquad \wedge\ \vec{c} \le_{inh} \mathsf{specializers}(m_1) \wedge m_1 \le_{meth} m \wedge m_1 \ne m) \\
\qquad\quad \Rightarrow ((\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)). \\
\qquad\qquad\qquad (\mathsf{specializers}(m)_i = any\ \Rightarrow\ \vec{c}_i <: \mathsf{argtypes}(m)_i) \\
\qquad\qquad\qquad \wedge\ (\mathsf{specializers}(m)_i \ne any\ \Rightarrow\ \vec{c}_i <: \mathsf{argtypes}(m)_i)) \\
\qquad\qquad \wedge\ \mathsf{restype}(m) \le_{sub} \mathsf{restype}(s))
\end{array}
$$

$\Longleftrightarrow \langle$by excluded middle$\rangle$

$$
\begin{array}{l}
\forall\, s \in S.\ \ \forall\, m \in M\,.\ \forall\, \vec{c} \in (C_{concrete})^*. \\
\quad \vec{c} <: \mathsf{argtypes}(s) \\
\quad \wedge\ \mathsf{msg}(m) = \mathsf{msg}(s)\ \wedge\ |\vec{c}| = |\mathsf{argtypes}(m)|\ \wedge\ \vec{c} \le_{inh} \mathsf{specializers}(m) \\
\qquad \Rightarrow ((\neg\, \exists\, m_1 \in M\,. \\
\qquad\qquad\qquad \mathsf{msg}(m_1) = \mathsf{msg}(s)\ \wedge\ |\vec{c}| = |\mathsf{argtypes}(m_1)| \\
\qquad\qquad\qquad \wedge\ \vec{c} \le_{inh} \mathsf{specializers}(m_1) \wedge m_1 \le_{meth} m \wedge m_1 \ne m) \\
\qquad\quad \Rightarrow ((\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)).\ \vec{c}_i <: \mathsf{argtypes}(m)_i) \\
\qquad\qquad \wedge\ \mathsf{restype}(m) \le_{sub} \mathsf{restype}(s))
\end{array}
$$

$\Longleftrightarrow \langle$by definition of $<:$ for vectors$\rangle$

$$
\begin{array}{l}
\forall\, s \in S.\ \ \forall\, m \in M\,.\ \forall\, \vec{c} \in (C_{concrete})^*. \\
\quad \vec{c} <: \mathsf{argtypes}(s) \\
\quad \wedge\ \mathsf{msg}(m) = \mathsf{msg}(s)\ \wedge\ |\vec{c}| = |\mathsf{argtypes}(m)|\ \wedge\ \vec{c} \le_{inh} \mathsf{specializers}(m) \\
\qquad \Rightarrow ((\neg\, \exists\, m_1 \in M\,. \\
\qquad\qquad\qquad \mathsf{msg}(m_1) = \mathsf{msg}(s)\ \wedge\ |\vec{c}| = |\mathsf{argtypes}(m_1)| \\
\qquad\qquad\qquad \wedge\ \vec{c} \le_{inh} \mathsf{specializers}(m_1) \wedge m_1 \le_{meth} m \wedge m_1 \ne m) \\
\qquad\quad \Rightarrow (\vec{c} <: \mathsf{argtypes}(m) \\
\qquad\qquad \wedge\ \mathsf{restype}(m) \le_{sub} \mathsf{restype}(s))
\end{array}
$$

$\Longleftrightarrow \langle$by predicate calculus$\rangle$

$$
\begin{array}{l}
\forall\, s \in S.\ \ \forall\, m \in M\,.\ \forall\, \vec{c} \in (C_{concrete})^*. \\
\quad \vec{c} <: \mathsf{argtypes}(s) \\
\quad \Rightarrow ((\mathsf{msg}(m) = \mathsf{msg}(s)\ \wedge\ |\vec{c}| = |\mathsf{argtypes}(m)|\ \wedge\ \vec{c} \le_{inh} \mathsf{specializers}(m)) \\
\qquad \Rightarrow ((\neg\, \exists\, m_1 \in M\,. \\
\qquad\qquad\qquad \mathsf{msg}(m_1) = \mathsf{msg}(s)\ \wedge\ |\vec{c}| = |\mathsf{argtypes}(m_1)| \\
\qquad\qquad\qquad \wedge\ \vec{c} \le_{inh} \mathsf{specializers}(m_1) \wedge m_1 \le_{meth} m \wedge m_1 \ne m) \\
\qquad\quad \Rightarrow (\vec{c} <: \mathsf{argtypes}(m) \\
\qquad\qquad \wedge\ \mathsf{restype}(m) \le_{sub} \mathsf{restype}(s)))
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \langle \text{by interchange of quantifiers} \rangle \\
\quad \forall\, s \in S.\ \ \forall\, \vec{c} \in (C_{concrete})^*. \\
\qquad (\forall\, m \in M\,. \\
\qquad\quad \vec{c}\ <:\ \mathsf{argtypes}(s) \\
\qquad\quad \Rightarrow ((\mathsf{msg}(m) = \mathsf{msg}(s)\ \wedge\ |\vec{c}| = |\mathsf{argtypes}(m)|\ \wedge\ \vec{c} \leq_{inh} \mathsf{specializers}(m)) \\
\qquad\qquad \Rightarrow ((\neg\, \exists\, m_1 \in M\,. \\
\qquad\qquad\qquad\qquad \mathsf{msg}(m_1) = \mathsf{msg}(s)\ \wedge\ |\vec{c}| = |\mathsf{argtypes}(m_1)| \\
\qquad\qquad\qquad\qquad \wedge\ \vec{c} \leq_{inh} \mathsf{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m) \\
\qquad\qquad\quad \Rightarrow (\vec{c}\ <:\ \mathsf{argtypes}(m) \\
\qquad\qquad\qquad \wedge\ \mathsf{restype}(m) \leq_{sub} \mathsf{restype}(s)))) \\
\Leftrightarrow \langle \text{by implication distributes over universal quantification} \rangle \\
\quad \forall\, s \in S.\ \ \forall\, \vec{c} \in (C_{concrete})^*. \\
\qquad \vec{c}\ <:\ \mathsf{argtypes}(s) \Rightarrow \\
\qquad\quad (\forall\, m \in M\,. \\
\qquad\qquad (\mathsf{msg}(m) = \mathsf{msg}(s)\ \wedge\ |\vec{c}| = |\mathsf{argtypes}(m)|\ \wedge\ \vec{c} \leq_{inh} \mathsf{specializers}(m)) \\
\qquad\qquad \Rightarrow ((\neg\, \exists\, m_1 \in M\,. \\
\qquad\qquad\qquad\qquad \mathsf{msg}(m_1) = \mathsf{msg}(s)\ \wedge\ |\vec{c}| = |\mathsf{argtypes}(m_1)| \\
\qquad\qquad\qquad\qquad \wedge\ \vec{c} \leq_{inh} \mathsf{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m) \\
\qquad\qquad\quad \Rightarrow (\vec{c}\ <:\ \mathsf{argtypes}(m) \\
\qquad\qquad\qquad \wedge\ \mathsf{restype}(m) \leq_{sub} \mathsf{restype}(s))) \\
\Leftrightarrow \langle \text{by set theory and predicate logic} \rangle \\
\quad \forall\, s \in S.\ \ \forall\, \vec{c} \in (C_{concrete})^*. \\
\qquad \vec{c}\ <:\ \mathsf{argtypes}(s) \Rightarrow \\
\qquad\quad \forall\, m \in \{\, m \in M \mid \mathsf{msg}(m) = \mathsf{msg}(s)\ \wedge\ |\vec{c}| = |\mathsf{argtypes}(m)|\ \wedge\ \vec{c} \leq_{inh} \mathsf{specializers}(m)\,\}. \\
\qquad\qquad (\neg\, \exists\, m_1 \in \{\, m_1 \in M \mid \\
\qquad\qquad\qquad\qquad\qquad \mathsf{msg}(m_1) = \mathsf{msg}(s)\ \wedge\ |\vec{c}| = |\mathsf{argtypes}(m_1)| \\
\qquad\qquad\qquad\qquad\qquad \wedge\ \vec{c} \leq_{inh} \mathsf{specializers}(m_1)\,\}. \\
\qquad\qquad\qquad\qquad \vec{c} \leq_{inh} \mathsf{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m) \\
\qquad\qquad \Rightarrow (\vec{c}\ <:\ \mathsf{argtypes}(m) \\
\qquad\qquad\quad \wedge\ \mathsf{restype}(m) \leq_{sub} \mathsf{restype}(s)) \\
\Leftrightarrow \langle \text{by definition of } \mathsf{applicable\text{-}methods}(s,\ \vec{c}) \rangle \\
\quad \forall\, s \in S.\ \ \forall\, \vec{c} \in (C_{concrete})^*. \\
\qquad \vec{c}\ <:\ \mathsf{argtypes}(s) \Rightarrow \\
\qquad\quad \forall\, m \in \mathsf{applicable\text{-}methods}(s,\ \vec{c}). \\
\qquad\qquad (\neg\, \exists\, m_1 \in \mathsf{applicable\text{-}methods}(s,\ \vec{c}). \\
\qquad\qquad\qquad\qquad \vec{c} \leq_{inh} \mathsf{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m) \\
\qquad\qquad \Rightarrow (\vec{c}\ <:\ \mathsf{argtypes}(m) \\
\qquad\qquad\quad \wedge\ \mathsf{restype}(m) \leq_{sub} \mathsf{restype}(s)) \\
\Leftrightarrow \langle \text{by definition} \rangle \\
\quad \mathsf{ImplementationIsConforming}
\end{array}
$$

The proof of the following lemma uses two lemmas, one for each conjunct in the definition.

Lemma 4. ComputeIsConforming $\Rightarrow$ AbstractComputeIsConforming

Proof: The proof relies on Lemmas 5 and 15.

ComputeIsConforming

$\Leftrightarrow$ ⟨by definition⟩
ComputeUnspecializedAndResultConform $\wedge$ ComputeSpecializedAreConforming

$\Rightarrow$ ⟨by Lemmas 5 and 15⟩
UnspecializedAndResultConform $\wedge$ SpecializersAreConforming

$\Leftrightarrow$ ⟨by definition⟩
AbstractComputeIsConforming

Lemma 5. ComputeUnspecializedAndResultConform $\Leftrightarrow$ UnspecializedAndResultConform

Proof: We calculate as follows.

ComputeUnspecializedAndResultConform

$\Leftrightarrow$ ⟨by definition⟩
$\forall\, m \in M.\ \forall\, s \in$ relevant-sigs(m, S).
has-common-classes(m, argtypes(s)) $\Rightarrow$ contra-unspec-args-co-result(m, s)

$\Leftrightarrow$ ⟨by Lemma 6.⟩
$\forall\, m \in M.\ \forall\, s \in$ relevant-sigs(m, S).
abs-has-common-classes(m, argtypes(s)) $\Rightarrow$ contra-unspec-args-co-result(m, s)

$\Leftrightarrow$ ⟨by definition⟩
UnspecializedAndResultConform

Lemma 6. For all $m \in M$ and $\vec{t} \in T^*$, has-common-classes(m, $\vec{t}$) $\Leftrightarrow$ abs-has-common-classes(m, $\vec{t}$).

Proof: Let m and $\vec{t}$ be given. We calculate as follows.

has-common-classes(m, $\vec{t}$)

$\Leftrightarrow$ ⟨by definition⟩
let $TCSs$ = { $\vec{c}$ | $c_i \in$ top-concrete-conforming-subclasses(specializers(m)$_i$, $\vec{t}_i$)
$\wedge\ i \in$ indexes(specializers(m)) } **in**
$\exists \vec{c} \in TCSs$.
($\neg\, \exists\, m_1 \in M$. msg($m_1$) = msg($m$) $\wedge$ |argtypes(m_1)| = |argtypes(m)|
$\wedge\ \vec{c} \leq_{inh}$ specializers(m_1) $\wedge\ m_1 \leq_{meth} m \wedge m_1 \neq m$)

$\Leftrightarrow$ ⟨by definition of top-concrete-conforming-subclasses⟩
let $TCSs$ = { $\vec{c}$ | $c_i \in$ top-classes(concrete-conforming-subclasses(specializers(m)$_i$, $\vec{t}_i$))
$\wedge\ i \in$ indexes(specializers(m)) } **in**
$\exists \vec{c} \in TCSs$.
($\neg\, \exists\, m_1 \in M$. msg($m_1$) = msg($m$) $\wedge$ |argtypes(m_1)| = |argtypes(m)|
$\wedge\ \vec{c} \leq_{inh}$ specializers(m_1) $\wedge\ m_1 \leq_{meth} m \wedge m_1 \neq m$)

$\Leftrightarrow$ ⟨by definition of concrete-conforming-subclasses⟩
let $TCSs$ = { $\vec{c}$ | $c_i \in$ top-classes({ $c' \in C_{concrete}$ | $c' \leq_{inh}$ specializers(m)$_i \wedge c' <: \vec{t}_i$ })
$\wedge\ i \in$ indexes(specializers(m)) } **in**
$\exists \vec{c} \in TCSs$.
($\neg\, \exists\, m_1 \in M$. msg($m_1$) = msg($m$) $\wedge$ |argtypes(m_1)| = |argtypes(m)|
$\wedge\ \vec{c} \leq_{inh}$ specializers(m_1) $\wedge\ m_1 \leq_{meth} m \wedge m_1 \neq m$)

$\Leftrightarrow \langle$by definition of $TCSs\rangle$
$\exists \vec{c} \in \{\, \vec{c} \mid c_i \in$ top-classes$(\{\, c' \in C_{concrete} \mid c' \leq_{inh}$ specializers$(m)_i \wedge c' <: \vec{t}_i \,\})$
$\wedge\, i \in$ indexes(specializers(m)) $\}$.
$(\neg \exists\, m_1 \in M.$ msg(m_1) = msg(m) $\wedge$ |argtypes(m_1)| = |argtypes(m)|
$\wedge\, \vec{c} \leq_{inh}$ specializers$(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)$

$\Leftrightarrow \langle$by vector notation $\rangle$
$\exists \vec{c} \in$ top-classes$(\{\, \vec{c} \in (C_{concrete})^* \mid \vec{c} \leq_{inh}$ specializers$(m) \wedge \vec{c} <: \vec{t}\})$.
$(\neg \exists\, m_1 \in M.$ msg(m_1) = msg(m) $\wedge$ |argtypes(m_1)| = |argtypes(m)|
$\wedge\, \vec{c} \leq_{inh}$ specializers$(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)$

$\Leftrightarrow \langle$by de Morgan$\rangle$
$\exists \vec{c} \in$ top-classes$(\{\, \vec{c} \in (C_{concrete})^* \mid \vec{c} \leq_{inh}$ specializers$(m) \wedge \vec{c} <: \vec{t}\})$.
$\wedge\, (\forall\, m_1 \in \{\, m_1 \in M.$ msg(m_1) = msg(m) $\wedge$ |argtypes(m_1)| = |argtypes(m)|
$\wedge\, m_1 \leq_{meth} m \wedge m_1 \neq m \,\}$.
$\neg\, (\vec{c} \leq_{inh}$ specializers$(m_1)))$

$\Leftrightarrow \langle$by Lemma 7.$\rangle$
$\exists \vec{c} \in \{\, \vec{c} \in (C_{concrete})^* \mid \vec{c} \leq_{inh}$ specializers$(m) \wedge \vec{c} <: \vec{t}\}$.
$\wedge\, (\forall\, m_1 \in \{\, m_1 \in M.$ msg(m_1) = msg(m) $\wedge$ |argtypes(m_1)| = |argtypes(m)|
$\wedge\, m_1 \leq_{meth} m \wedge m_1 \neq m \,\}$.
$\neg\, (\vec{c} \leq_{inh}$ specializers$(m_1)))$

$\Leftrightarrow \langle$by de Morgan$\rangle$
$\exists \vec{c} \in \{\, \vec{c} \in (C_{concrete})^* \mid \vec{c} \leq_{inh}$ specializers$(m) \wedge \vec{c} <: \vec{t}\}$.
$(\neg \exists\, m_1 \in M.$ msg(m_1) = msg(m) $\wedge$ |argtypes(m_1)| = |argtypes(m)|
$\wedge\, \vec{c} \leq_{inh}$ specializers$(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)$

$\Leftrightarrow \langle$by set theory and commutivity$\rangle$
$\exists\, \vec{c} \in \{\, \vec{c} \in (C_{concrete})^* \mid \vec{c} \leq_{inh}$ specializers(m)
$\wedge\, (\neg \exists\, m_1 \in M.$ msg(m_1) = msg(m) $\wedge$ |argtypes(m_1)| = |argtypes(m)|
$\wedge\, \vec{c} \leq_{inh}$ specializers$(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m) \,\}$.
$\vec{c} <: \vec{t}$

$\Leftrightarrow \langle$by definition of covered-class-vecs$(m)\rangle$
$\exists\, \vec{c} \in$ covered-class-vecs(m) . $\vec{c} <: \vec{t}$

$\Leftrightarrow \langle$by definition$\rangle$
abs-has-common-classes$(m, \vec{t})$

Lemma 7. Let $EMSs$ and Cs be sets of class vectors. Then

$(\exists\, \vec{c} \in$ top-classes(Cs). $\forall \vec{c}\,'' \in EMSs\,.\neg\, (\vec{c} \leq_{inh} \vec{c}\,'')) \Leftrightarrow (\exists\, \vec{c} \in Cs.\ \forall \vec{c}\,'' \in EMSs\,.\neg\, (\vec{c} \leq_{inh} \vec{c}\,''))$

Proof: Because top-classes(Cs) is a subset of Cs, the left hand side immediately implies the right. For the converse, suppose that the right hand side is true. Let $\vec{c}$ be such that $\forall \vec{c}\,'' \in EMSs\,.\neg\, (\vec{c} \leq_{inh} \vec{c}\,'')$. Consider a vector $\vec{c}\,'''$ such that $\vec{c} \leq_{inh} \vec{c}\,'''$ and $\vec{c}\,''' \in$ top-classes(Cs). Such a vector exists by definition of top-classes. We can then prove the converse as follows.

$\vec{c}\,''' \in$ top-classes(Cs)
$\wedge\, \vec{c} \leq_{inh} \vec{c}\,''' \wedge \forall \vec{c}\,'' \in EMSs\,.\neg\, (\vec{c} \leq_{inh} \vec{c}\,'')$

$\Rightarrow \langle$by transitivity of inheritance, if for some $\vec{c}\,'' \in EMSs$, $(\vec{c}\,''' \leq_{inh} \vec{c}\,'')$, then also $(\vec{c} \leq_{inh} \vec{c}\,'')\rangle$
$\vec{c}\,''' \in$ top-classes(Cs)
$\wedge\, \forall \vec{c}\,'' \in EMSs\,.\neg\, (\vec{c}\,''' \leq_{inh} \vec{c}\,'')$

$\Rightarrow \langle$by instantiation rule$\rangle$
$(\exists\ \vec{c} \in \text{top-classes}(Cs).\ \forall \vec{c}\,'' \in EMSs\ .\neg\ (\vec{c} \leq_{inh} \vec{c}\,''))$

The following lemmas are used in the proof of Lemma 15.

Lemma 8. Let Cs be a finite set of classes. Then $\text{top-classes}(Cs) = \varnothing \Leftrightarrow Cs = \varnothing$

Proof: We calculate as follows.

$\text{top-classes}(Cs) = \varnothing$

$\Leftrightarrow \langle$by definition$\rangle$
$\{ c \in Cs \mid \forall\ c' \in Cs.\ \ c' \neq c\ \Rightarrow \neg\ (c \leq_{inh} c') \} = \varnothing$

$\Leftrightarrow \langle$by set theory$\rangle$
$\neg\ (\exists\ c \in Cs\ .\ (\forall\ c' \in Cs.\ \ c' \neq c\ \Rightarrow\ \neg\ (c \leq_{inh} c')))$

$\Leftrightarrow \langle$by de Morgan, twice and $\neg(A \Rightarrow B) \Leftrightarrow (A \wedge \neg B)\rangle$
$(\forall\ c \in Cs\ .\ (\exists\ c' \in Cs.\ \ c' \neq c \wedge (c \leq_{inh} c')))$

$\Leftrightarrow \langle$by Cs is finite and $\leq_{inh}$ is acyclic, so the above is false except when Cs is empty$\rangle$
$Cs = \varnothing$

Lemma 9. Let $c \in C$ be a class, $c1 \in C_{concrete}$ be a concrete class, and $t \in T$ be a type. Then

$c1 \in \text{top-non-conforming-classes}(c,\ t)$
$\Leftrightarrow$
$c1 \leq_{inh} c\ \wedge\ \neg\ (c1 <: t)$
$\wedge\ (\forall\ c' \in C_{concrete}\ .\ (c' \leq_{inh} c\ \wedge\ \neg\ (c' <: t)) \Rightarrow (c' \neq c1\ \Rightarrow\ \neg\ (c1 \leq_{inh} c'))).$

Proof: We brgin the proof by calculating as follows.

$c1 \in \text{top-non-conforming-classes}(c,\ t))$

$\Leftrightarrow \langle$by definition$\rangle$
$c1 \in \text{top-classes}(\{ c' \in C_{concrete} \mid\ c' \leq_{inh} c\ \wedge\ \neg\ (c' <: t)\ \})$

$\Leftrightarrow \langle$by renaming c' to $c2\rangle$
$c1 \in \text{top-classes}(\{ c2 \in C_{concrete} \mid\ c2 \leq_{inh} c\ \wedge\ \neg\ (c2 <: t)\ \})$

$\Leftrightarrow \langle$by definition of top-classes, using $c3$ for c in the definition to avoid capture$\rangle$
$c1 \in \{ c3 \in \{ c2 \in C_{concrete} \mid\ c2 \leq_{inh} c\ \wedge\ \neg\ (c2 <: t)\ \} \mid$
$\forall\ c' \in \{ c2 \in C_{concrete} \mid\ c2 \leq_{inh} c\ \wedge\ \neg\ (c2 <: t)\ \}.$
$c' \neq c3\ \Rightarrow\ \neg\ (c3 \leq_{inh} c') \}$

$\Leftrightarrow \langle$by set theory$\rangle$
$c1 \in \{ c3 \in C_{concrete} \mid\ c3 \leq_{inh} c\ \wedge\ \neg\ (c3 <: t)$
$\wedge\ (\forall\ c' \in C_{concrete}\ .$
$(c' \leq_{inh} c\ \wedge\ \neg\ (c' <: t)) \Rightarrow (c' \neq c3\ \Rightarrow\ \neg\ (c3 \leq_{inh} c')))\ \}$

$\Leftrightarrow \langle$by set theory$\rangle$
$c1 \in C_{concrete} \wedge c1 \leq_{inh} c\ \wedge\ \neg\ (c1 <: t)$
$\wedge\ (\forall\ c' \in C_{concrete}\ .(c' \leq_{inh} c\ \wedge\ \neg\ (c' <: t)) \Rightarrow (c' \neq c1\ \Rightarrow\ \neg\ (c1 \leq_{inh} c')))$

$\Leftrightarrow \langle$by the assumption that $c1 \in C_{concrete}\rangle$
$c1 \leq_{inh} c\ \wedge\ \neg\ (c1 <: t)$
$\wedge\ (\forall\ c' \in C_{concrete}\ .\ (c' \leq_{inh} c\ \wedge\ \neg\ (c' <: t)) \Rightarrow (c' \neq c1\ \Rightarrow\ \neg\ (c1 \leq_{inh} c')))$

Lemma 10. Let $c \in C$ be a class, $c1 \in C_{concrete}$ be a concrete class, and $t \in T$ be a type. Then

$$c1 \in \text{top-non-conforming-classes}(c, t) \Rightarrow c1 \leq_{inh} c \ \wedge \ \neg\, (c1 <: t)$$

Proof: This follows directly from Lemma 9 by A ∧ B ⇒ A.

The following lemma extends the previous lemma to provide a kind of converse to it.

Lemma 11. Let $c \in C$ be a class and $t \in T$ be a type. Then

$$(\exists\, c1 \in C_{concrete} \,.\, c1 \leq_{inh} c \ \wedge \ \neg\, (c1 <: t))$$
$$\Leftrightarrow$$
$$(\exists\, c1 \in C_{concrete} \,.\, c1 \in \text{top-non-conforming-classes}(c, t))$$

Proof: We proceed by mutual implication.

$$(\exists\, c1 \in C_{concrete} \,.\, c1 \in \text{top-non-conforming-classes}(c, t))$$
$$\Rightarrow \langle\text{by Lemma 10}\rangle$$
$$(\exists\, c1 \in C_{concrete} \,.\, c1 \leq_{inh} c \ \wedge \ \neg\, (c1 <: t))$$

The converse is shown as follows.

$$(\exists\, c1 \in C_{concrete} \,.\, c1 \leq_{inh} c \ \wedge \ \neg\, (c1 <: t))$$
$$\Rightarrow \langle\text{by the finiteness of } C_{concrete} \text{ and the acyclic nature of } \leq_{inh}\text{, there are maximal such } c1\rangle$$
$$\begin{aligned}(\exists\, c1 \in C_{concrete} \,.\, c1 \leq_{inh} c \ \wedge \ \neg\, (c1 <: t) \\ \wedge\, (\forall\, c' \in C_{concrete} \,. \\ (c' \leq_{inh} c \ \wedge \ \neg\, (c' <: t)) \Rightarrow (c' \neq c1 \ \Rightarrow \ \neg\, (c1 \leq_{inh} c'))))\end{aligned}$$
$$\Leftrightarrow \langle\text{by Lemma 9}\rangle$$
$$(\exists\, c1 \in C_{concrete} \,.\, c1 \in \text{top-non-conforming-classes}(c, t))$$

This completes the proof.

The following lemma can be proved as a corrollary to Lemma 11, but we give a direct proof below.

Lemma 12. Let $c \in C$ be a class and $t \in T$ be a type. Then

$$\text{top-non-conforming-classes}(c, t) = \varnothing \Leftrightarrow (\forall\, c' \in C_{concrete} \,.\, c' \leq_{inh} c \Rightarrow c' <: t)$$

Proof: We calculate as follows.

$$\text{top-non-conforming-classes}(c, t) = \varnothing$$
$$\Leftrightarrow \langle\text{by definition}\rangle$$
$$\text{top-classes}(\{\, c' \in C_{concrete} \mid c' \leq_{inh} c \ \wedge \ \neg\, (c' <: t) \,\}) = \varnothing$$
$$\Leftrightarrow \langle\text{by Lemma 8.}\rangle$$
$$\{\, c' \in C_{concrete} \mid c' \leq_{inh} c \ \wedge \ \neg\, (c' <: t) \,\} = \varnothing$$
$$\Leftrightarrow \langle\text{by set theory}\rangle$$
$$(\forall\, c' \in C_{concrete} \,.\, \neg\, (c' \leq_{inh} c \ \wedge \ \neg\, (c' <: t)))$$
$$\Leftrightarrow \langle\text{by } \neg(A \Rightarrow B) \Leftrightarrow (A \wedge \neg B) \text{ and double negation}\rangle$$
$$(\forall\, c' \in C_{concrete} \,.\, c' \leq_{inh} c \Rightarrow c' <: t)$$

We now move from lemmas about individual classes to lemmas about class vectors.

Lemma 13. Let $EMSs$ be a set of class vectors. Let $\vec{c} \in C^*$ be a class vector, and let $\vec{t} \in T^*$ be a type vector. Suppose that $|\vec{c}| = |\vec{t}|$. Then

$$
\begin{array}{l}
(\exists \vec{d} \in C^*. \\
\quad \vec{d} \in (C_{concrete})^* \wedge \vec{d} \neq \vec{c} \wedge \vec{d} \leq_{inh} \vec{c} \\
\quad \wedge (\exists\, i \in \mathsf{indexes}(\vec{c})\,.\, \vec{c}_i \neq any \;\wedge \neg (\vec{d}_i <: \vec{t}_i) \\
\quad \wedge (\forall \vec{c}\,'' \in EMSs\,.\, \neg (\vec{d} \leq_{inh} \vec{c}\,'')) \\
\Rightarrow \\
(\exists \vec{d} \in C^*. \\
\quad \vec{d} \in \mathsf{top\text{-}non\text{-}conforming\text{-}class\text{-}vecs}(\vec{c}, \vec{t}) \\
\quad \wedge (\forall \vec{c}\,'' \in EMSs\,.\neg (\vec{d} \leq_{inh} \vec{c}\,''))
\end{array}
$$

Proof: We calculate as follows.

$$
\begin{array}{l}
(\exists \vec{d} \in C^*. \\
\quad \vec{d} \in (C_{concrete})^* \wedge \vec{d} \neq \vec{c} \wedge \vec{d} \leq_{inh} \vec{c} \\
\quad \wedge (\exists\, i \in \mathsf{indexes}(\vec{c})\,.\, \vec{c}_i \neq any \;\wedge \neg (\vec{d}_i <: \vec{t}_i) \\
\quad \wedge (\forall \vec{c}\,'' \in EMSs\,.\, \neg (\vec{d} \leq_{inh} \vec{c}\,''))
\end{array}
$$

$\Longrightarrow \langle$by A $\wedge$ B $\Rightarrow$ A and alternate notation for vector elements ($c_i = \vec{c}_i$)$\rangle$

$$
\begin{array}{l}
(\exists \vec{d} \in C^*. \\
\quad \vec{d} \neq \vec{c} \wedge \vec{d} \leq_{inh} \vec{c} \\
\quad \wedge (\exists\, i \in \mathsf{indexes}(\vec{c})\,.\, c_i \neq any \;\wedge \neg (\vec{d}_i <: \vec{t}_i) \\
\quad \wedge (\forall \vec{c}\,'' \in EMSs\,.\, \neg (\vec{d} \leq_{inh} \vec{c}\,''))
\end{array}
$$

$\Longrightarrow \langle$by reflexivity of inheritance, can choose $\vec{d}_i$ for positions where $c_i = any$ to be $c_i\rangle$

$$
\begin{array}{l}
(\exists \vec{d} \in C^*. \\
\quad \vec{d} \neq \vec{c} \wedge \vec{d} \leq_{inh} \vec{c} \\
\quad \wedge (\forall\, i \in \mathsf{indexes}(\vec{c}). \\
\quad\quad (c_i = any \;\Rightarrow\; \vec{d}_i = c_i)) \\
\quad \wedge (\exists\, i \in \mathsf{indexes}(\vec{c})\,.\, c_i \neq any \;\wedge \neg (\vec{d}_i <: \vec{t}_i) \\
\quad \wedge (\forall \vec{c}\,'' \in EMSs\,.\, \neg (\vec{d} \leq_{inh} \vec{c}\,''))
\end{array}
$$

$\Longrightarrow \langle$by reflexivity of inheritance, can choose $\vec{d}_i$ for positions where $c_i \neq any$ to be c_i when there is no non-conforming concrete subclass, and to be a non-conforming concrete subclass otherwise$\rangle$

$$
\begin{array}{l}
(\exists \vec{d} \in C^*. \\
\quad \vec{d} \neq \vec{c} \wedge \vec{d} \leq_{inh} \vec{c} \\
\quad \wedge (\forall\, i \in \mathsf{indexes}(\vec{c}). \\
\quad\quad (c_i = any \;\Rightarrow\; \vec{d}_i = c_i) \\
\quad\quad \wedge (c_i \neq any \\
\quad\quad\quad \Rightarrow (((\forall\, c'' \in C_{concrete}\,.\, c'' \leq_{inh} c_i \Rightarrow c'' <: t_i) \;\Rightarrow\; \vec{d}_i = c_i) \\
\quad\quad\quad\quad \wedge ((\exists\, c'' \in C_{concrete}\,.\, c'' \leq_{inh} c_i \;\wedge\; \neg (c'' <: t_i)) \\
\quad\quad\quad\quad\quad \Rightarrow (\vec{d}_i \leq_{inh} c_i \;\wedge\; \neg (\vec{d}_i <: t_i)))))) \\
\quad \wedge (\exists\, i \in \mathsf{indexes}(\vec{c})\,.\, c_i \neq any \;\wedge \neg (\vec{d}_i <: \vec{t}_i) \\
\quad \wedge (\forall \vec{c}\,'' \in EMSs\,.\, \neg (\vec{d} \leq_{inh} \vec{c}\,''))
\end{array}
$$

$$
\begin{array}{l}
\Rightarrow \langle \text{by } A \wedge B \Rightarrow A \rangle \\
\quad (\exists \vec{d} \in C^*. \\
\qquad \vec{d} \neq \vec{c} \wedge \vec{d} \leq_{inh} \vec{c} \\
\qquad \wedge (\forall i \in \text{indexes}(\vec{c}). \\
\qquad\quad (c_i = any \Rightarrow \vec{d}_i = c_i) \\
\qquad\quad \wedge (c_i \neq any \\
\qquad\qquad \Rightarrow (((\forall c'' \in C_{concrete} \,.\, c'' \leq_{inh} c_i \Rightarrow c'' <: t_i) \Rightarrow \vec{d}_i = c_i) \\
\qquad\qquad\quad \wedge ((\exists c'' \in C_{concrete} \,.\, c'' \leq_{inh} c_i \wedge \neg (c'' <: t_i)) \\
\qquad\qquad\qquad \Rightarrow (\vec{d}_i \leq_{inh} c_i \wedge \neg (\vec{d}_i <: t_i)))))) \\
\qquad \wedge (\forall \vec{c}'' \in EMSs \,.\neg (\vec{d} \leq_{inh} \vec{c}''))
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \langle \text{by definition of inheritance for vectors} \rangle \\
\quad (\exists \vec{d} \in C^*. \\
\qquad \vec{d} \neq \vec{c} \\
\qquad \wedge |\vec{d}| = |\vec{c}| \\
\qquad \wedge (\forall i \in \text{indexes}(\vec{c}). \vec{d}_i \leq_{inh} c_i) \\
\qquad \wedge (\forall i \in \text{indexes}(\vec{c}). \\
\qquad\quad (c_i = any \Rightarrow \vec{d}_i = c_i) \\
\qquad\quad \wedge (c_i \neq any \\
\qquad\qquad \Rightarrow (((\forall c'' \in C_{concrete} \,.\, c'' \leq_{inh} c_i \Rightarrow c'' <: t_i) \Rightarrow \vec{d}_i = c_i) \\
\qquad\qquad\quad \wedge ((\exists c'' \in C_{concrete} \,.\, c'' \leq_{inh} c_i \wedge \neg (c'' <: t_i)) \\
\qquad\qquad\qquad \Rightarrow (\vec{d}_i \leq_{inh} c_i \wedge \neg (\vec{d}_i <: t_i)))))) \\
\qquad \wedge (\forall \vec{c}'' \in EMSs \,.\neg (\vec{d} \leq_{inh} \vec{c}''))
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \langle \text{by joining the term} \rangle \\
\quad (\exists \vec{d} \in C^*. \\
\qquad \vec{d} \neq \vec{c} \\
\qquad \wedge |\vec{d}| = |\vec{c}| \\
\qquad \wedge (\forall i \in \text{indexes}(\vec{c}). \\
\qquad\quad \vec{d}_i \leq_{inh} c_i \\
\qquad\quad \wedge (c_i = any \Rightarrow \vec{d}_i = c_i) \\
\qquad\quad \wedge (c_i \neq any \\
\qquad\qquad \Rightarrow (((\forall c'' \in C_{concrete} \,.\, c'' \leq_{inh} c_i \Rightarrow c'' <: t_i) \Rightarrow \vec{d}_i = c_i) \\
\qquad\qquad\quad \wedge ((\exists c'' \in C_{concrete} \,.\, c'' \leq_{inh} c_i \wedge \neg (c'' <: t_i)) \\
\qquad\qquad\qquad \Rightarrow (\vec{d}_i \leq_{inh} c_i \wedge \neg (\vec{d}_i <: t_i)))))) \\
\qquad \wedge (\forall \vec{c}'' \in EMSs \,.\neg (\vec{d} \leq_{inh} \vec{c}''))
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \langle \text{by conjunction distributes over conjunction and implication (twice)} \rangle \\
\quad (\exists \vec{d} \in C^*. \\
\qquad \vec{d} \neq \vec{c} \\
\qquad \wedge |\vec{d}| = |\vec{c}| \\
\qquad \wedge (\forall i \in \text{indexes}(\vec{c}). \\
\qquad\quad (c_i = any \Rightarrow \vec{d}_i = c_i \wedge \vec{d}_i \leq_{inh} c_i) \\
\qquad\quad \wedge (c_i \neq any \\
\qquad\qquad \Rightarrow (((\forall c'' \in C_{concrete} \,.\, c'' \leq_{inh} c_i \Rightarrow c'' <: t_i) \Rightarrow \vec{d}_i = c_i \wedge \vec{d}_i \leq_{inh} c_i) \\
\qquad\qquad\quad \wedge ((\exists c'' \in C_{concrete} \,.\, c'' \leq_{inh} c_i \wedge \neg (c'' <: t_i)) \\
\qquad\qquad\qquad \Rightarrow (\vec{d}_i \leq_{inh} c_i \wedge \neg (\vec{d}_i <: t_i)))))) \\
\qquad \wedge (\forall \vec{c}'' \in EMSs \,.\neg (\vec{d} \leq_{inh} \vec{c}''))
\end{array}
$$

$\Leftrightarrow \langle$by inheritance is reflexive$\rangle$

$$
\begin{array}{l}
(\exists \vec{d} \in C^*. \\
\quad \vec{d} \neq \vec{c} \\
\quad \wedge |\vec{d}| = |\vec{c}| \\
\quad \wedge (\forall\, i \in \text{indexes}(\vec{c}). \\
\qquad (c_i = any \;\; \Rightarrow \;\; \vec{d}_i = c_i) \\
\qquad \wedge (c_i \neq any \\
\qquad\quad \Rightarrow (((\forall\, c'' \in C_{concrete} \,.\, c'' \leq_{inh} c_i \Rightarrow c'' <: t_i) \;\; \Rightarrow \;\; \vec{d}_i = c_i) \\
\qquad\qquad \wedge ((\exists\, c'' \in C_{concrete} \,.\, c'' \leq_{inh} c_i \;\wedge\; \neg\, (c'' <: t_i)) \\
\qquad\qquad\quad \Rightarrow (\vec{d}_i \leq_{inh} c_i \;\wedge\; \neg\, (\vec{d}_i <: t_i)))))) \\
\quad \wedge (\forall \vec{c}\,'' \in EMSs \,.\neg\, (\vec{d} \leq_{inh} \vec{c}\,''))
\end{array}
$$

$\Rightarrow \langle$by the finiteness of C, and the acyclic nature of $\leq_{inh}$, there are maximal such vectors $\vec{d}\,\rangle$

$$
\begin{array}{l}
(\exists \vec{d} \in C^*. \\
\quad \vec{d} \neq \vec{c} \\
\quad \wedge |\vec{d}| = |\vec{c}| \\
\quad \wedge (\forall\, i \in \text{indexes}(\vec{c}). \\
\qquad (c_i = any \;\; \Rightarrow \;\; \vec{d}_i = c_i) \\
\qquad \wedge (c_i \neq any \\
\qquad\quad \Rightarrow (((\forall\, c'' \in C_{concrete} \,.\, c'' \leq_{inh} c_i \Rightarrow c'' <: t_i) \;\; \Rightarrow \;\; \vec{d}_i = c_i) \\
\qquad\qquad \wedge ((\exists\, c'' \in C_{concrete} \,.\, c'' \leq_{inh} c_i \;\wedge\; \neg\, (c'' <: t_i)) \\
\qquad\qquad\quad \Rightarrow (\vec{d}_i \leq_{inh} c_i \;\wedge\; \neg\, (\vec{d}_i <: t_i) \\
\qquad\qquad\qquad \wedge (\forall\, c' \in C_{concrete} \,.\, (c' \leq_{inh} c_i \;\wedge\; \neg\, (c' <: t_i)) \\
\qquad\qquad\qquad\quad \Rightarrow (c' \neq \vec{d}_i \;\Rightarrow\; \neg\, (\vec{d}_i \leq_{inh} c')))))))) \\
\quad \wedge (\forall \vec{c}\,'' \in EMSs \,.\neg\, (\vec{d} \leq_{inh} \vec{c}\,''))
\end{array}
$$

$\Leftrightarrow \langle$by Lemmas 11, 12, and 9$\rangle$

$$
\begin{array}{l}
(\exists \vec{d} \in C^*. \\
\quad \vec{d} \neq \vec{c} \\
\quad \wedge |\vec{d}| = |\vec{c}| \\
\quad \wedge (\forall\, i \in \text{indexes}(\vec{c}). \\
\qquad (c_i = any \;\; \Rightarrow \;\; \vec{d}_i = c_i) \\
\qquad \wedge (c_i \neq any \\
\qquad\qquad \Rightarrow ((\text{top-non-conforming-classes}(c_i, t_i) = \varnothing \;\; \Rightarrow \;\; \vec{d}_i = c_i) \\
\qquad\qquad\quad \wedge (\text{top-non-conforming-classes}(c_i, t_i) \neq \varnothing \\
\qquad\qquad\qquad \Rightarrow \vec{d}_i \in \text{top-non-conforming-classes}(c_i, t_i)))) \\
\quad \wedge (\forall \vec{c}\,'' \in EMSs \,.\neg\, (\vec{d} \leq_{inh} \vec{c}\,''))
\end{array}
$$

$\Leftrightarrow$ $\langle$by conjunction is commutative and **let**-abstraction and the range of the existential quantifier$\rangle$

$$
\begin{array}{l}
(\exists \vec{d} \in C^*. \\
\quad \vec{d} \in C^* \\
\quad \wedge |\vec{d}| = |\vec{c}| \\
\quad \wedge (\forall i \in \text{indexes}(\vec{c}). \\
\quad\quad (c_i = any \;\Rightarrow\; \vec{d}_i = c_i) \\
\quad\quad \wedge (c_i \neq any \;\Rightarrow \\
\quad\quad\quad \textbf{let } Cs = \text{top-non-conforming-classes}(c_i, t_i) \textbf{ in} \\
\quad\quad\quad\quad (Cs = \varnothing \;\Rightarrow\; \vec{d}_i = c_i) \\
\quad\quad\quad\quad \wedge (Cs \neq \varnothing \;\Rightarrow\; \vec{d}_i \in Cs))) \\
\quad \wedge \vec{d} \neq \vec{c} \\
\quad \wedge (\forall \vec{c}\,'' \in EMSs . \neg (\vec{d} \leq_{inh} \vec{c}\,''))
\end{array}
$$

$\Leftrightarrow$ $\langle$by set theory$\rangle$

$$
\begin{array}{l}
(\exists \vec{d} \in C^*. \\
\quad \vec{d} \in (\{ \vec{c}\,' \in C^* \mid |\vec{c}\,'| = |\vec{c}| \\
\quad\quad\quad\quad \wedge (\forall i \in \text{indexes}(\vec{c}). \\
\quad\quad\quad\quad\quad (c_i = any \;\Rightarrow\; c_i' = c_i) \\
\quad\quad\quad\quad\quad \wedge (c_i \neq any \;\Rightarrow \\
\quad\quad\quad\quad\quad\quad \textbf{let } Cs = \text{top-non-conforming-classes}(c_i, t_i) \textbf{ in} \\
\quad\quad\quad\quad\quad\quad\quad (Cs = \varnothing \;\Rightarrow\; c_i' = c_i) \\
\quad\quad\quad\quad\quad\quad\quad \wedge (Cs \neq \varnothing \;\Rightarrow\; c_i' \in Cs))) \} \; - \{ \vec{c} \}) \\
\quad \wedge (\forall \vec{c}\,'' \in EMSs . \neg (\vec{d} \leq_{inh} \vec{c}\,''))
\end{array}
$$

$\Leftrightarrow$ $\langle$by definition$\rangle$

$$
\begin{array}{l}
(\exists \vec{d} \in C^*. \\
\quad \vec{d} \in \text{top-non-conforming-class-vecs}(\vec{c}, \vec{t}) \\
\quad \wedge (\forall \vec{c}\,'' \in EMSs . \neg (\vec{d} \leq_{inh} \vec{c}\,''))
\end{array}
$$

Lemma 14. Let $m \in M$ be a method. Then

$$
\begin{array}{l}
\text{top-non-overridden-non-conforming-class-vecs}(m) = \varnothing \\
\Rightarrow (\forall \vec{c} \in \text{covered-class-vecs}(m) . \\
\quad\quad \vec{c} \neq \text{specializers}(m) \\
\quad\quad\quad \Rightarrow (\forall i \in \text{indexes}(\text{specializers}(m)). \\
\quad\quad\quad\quad\quad \text{specializers}(m)_i \neq any \;\Rightarrow\; \vec{c}_i <: \text{argtypes}(m)_i))
\end{array}
$$

Proof: We prove the contrapositive by calculating as follows.

$$
\begin{array}{l}
\neg (\forall \vec{c} \in \text{covered-class-vecs}(m) . \\
\quad \vec{c} \neq \text{specializers}(m) \\
\quad\quad \Rightarrow (\forall i \in \text{indexes}(\text{specializers}(m)). \\
\quad\quad\quad\quad \text{specializers}(m)_i \neq any \;\Rightarrow\; \vec{c}_i <: \text{argtypes}(m)_i))
\end{array}
$$

$\Leftrightarrow$ $\langle$by de Morgan and $\neg(A \Rightarrow B) \Leftrightarrow (A \wedge \neg B)$$\rangle$

$$
\begin{array}{l}
(\exists \vec{c} \in \text{covered-class-vecs}(m) . \\
\quad \vec{c} \neq \text{specializers}(m) \\
\quad \wedge \neg (\forall i \in \text{indexes}(\text{specializers}(m)). \\
\quad\quad\quad \text{specializers}(m)_i \neq any \;\Rightarrow\; \vec{c}_i <: \text{argtypes}(m)_i))
\end{array}
$$

$\Leftrightarrow \langle$by de Morgan and $\neg(A \Rightarrow B) \Leftrightarrow (A \wedge \neg B)\rangle$

$$\begin{array}{l}
(\exists\, \vec{c} \in \text{covered-class-vecs}(m)\,.\\
\quad \vec{c} \neq \text{specializers}(m)\\
\quad \wedge (\exists\, i \in \text{indexes}(\text{specializers}(m)).\\
\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\wedge \neg\, (\vec{c}_i <: \text{argtypes}(m)_i)))
\end{array}$$

$\Leftrightarrow \langle$by definition$\rangle$

$$\begin{array}{l}
(\exists\, \vec{c} \in \{\, \vec{c} \in (C_{concrete})^* \mid \vec{c} \leq_{inh} \text{specializers}(m)\\
\qquad\qquad \wedge (\neg \exists\, m_1 \in M.\ \text{msg}(m_1) = \text{msg}(m) \;\wedge |\text{argtypes}(m_1)| = |\text{argtypes}(m)|\\
\qquad\qquad\qquad \wedge\, \vec{c} \leq_{inh} \text{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)\,\}\,.\\
\quad \vec{c} \neq \text{specializers}(m)\\
\quad \wedge (\exists\, i \in \text{indexes}(\text{specializers}(m)).\\
\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\wedge \neg\, (\vec{c}_i <: \text{argtypes}(m)_i)))
\end{array}$$

$\Leftrightarrow \langle$by set theory$\rangle$

$$\begin{array}{l}
(\exists\, \vec{c} \in (C_{concrete})^*\,.\\
\quad \vec{c} \leq_{inh} \text{specializers}(m)\\
\quad \wedge (\neg \exists\, m_1 \in M.\ \text{msg}(m_1) = \text{msg}(m) \;\wedge |\text{argtypes}(m_1)| = |\text{argtypes}(m)|\\
\qquad\qquad \wedge\, \vec{c} \leq_{inh} \text{specializers}(m_1) \wedge m_1 \leq_{meth} m \wedge m_1 \neq m)\\
\quad \wedge\, \vec{c} \neq \text{specializers}(m)\\
\quad \wedge (\exists\, i \in \text{indexes}(\text{specializers}(m)).\\
\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\wedge \neg\, (\vec{c}_i <: \text{argtypes}(m)_i)))
\end{array}$$

$\Leftrightarrow \langle$by commutivity of conjunction$\rangle$

$$\begin{array}{l}
(\exists\, \vec{c} \in (C_{concrete})^*\,.\\
\quad \vec{c} \neq \text{specializers}(m)\\
\quad \wedge\, \vec{c} \leq_{inh} \text{specializers}(m)\\
\quad \wedge (\exists\, i \in \text{indexes}(\text{specializers}(m)).\\
\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\wedge \neg\, (\vec{c}_i <: \text{argtypes}(m)_i)))\\
\quad \wedge (\neg \exists\, m_1 \in M.\\
\qquad\qquad \text{msg}(m_1) = \text{msg}(m) \;\wedge |\text{argtypes}(m_1)| = |\text{argtypes}(m)| \wedge m_1 \leq_{meth} m \wedge m_1 \neq m\\
\qquad\qquad\qquad \wedge\, \vec{c} \leq_{inh} \text{specializers}(m_1))
\end{array}$$

$\Leftrightarrow \langle$by $(C_{concrete})^* \subset C^*$ and range rule$\rangle$

$$\begin{array}{l}
(\exists\, \vec{c} \in C^*\,.\\
\quad \vec{c} \in (C_{concrete})^*\\
\quad \wedge\, \vec{c} \neq \text{specializers}(m)\\
\quad \wedge\, \vec{c} \leq_{inh} \text{specializers}(m)\\
\quad \wedge (\exists\, i \in \text{indexes}(\text{specializers}(m)).\\
\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\wedge \neg\, (\vec{c}_i <: \text{argtypes}(m)_i)))\\
\quad \wedge (\neg \exists\, m_1 \in M.\\
\qquad\qquad \text{msg}(m_1) = \text{msg}(m) \;\wedge |\text{argtypes}(m_1)| = |\text{argtypes}(m)| \wedge m_1 \leq_{meth} m \wedge m_1 \neq m\\
\qquad\qquad\qquad \wedge\, \vec{c} \leq_{inh} \text{specializers}(m_1))
\end{array}$$

$\Leftrightarrow \langle$by set theory and de Morgan$\rangle$

$(\exists\ \vec{c} \in C^*$.
$\vec{c} \in (C_{concrete})^*$
$\wedge\ \vec{c} \neq$ specializers(m)
$\wedge\ \vec{c} \leq_{inh}$ specializers(m)
$\wedge$ ($\exists\ i \in$ indexes(specializers(m)).
specializers$(m)_i \neq any\ \wedge \neg\ (\vec{c}_i$ <: argtypes$(m)_i$)))
$\wedge$ ($\forall\ m_1 \in \{m_1 \in M$ | msg(m_1) = msg(m) $\wedge$ |argtypes(m_1)| = |argtypes(m)|
$\wedge\ m_1 \leq_{meth} m \wedge m_1 \neq m\ \}$.
$\neg\ (\vec{c} \leq_{inh}$ specializers(m_1)))

$\Rightarrow \langle$by Lemma 13$\rangle$

$(\exists \vec{c} \in C^*$.
$\vec{c} \in$ top-non-conforming-class-vecs(specializers(m), argtypes(m))
$\wedge$ ($\forall\ m_1 \in \{m_1 \in M$ | msg(m_1) = msg(m) $\wedge$ |argtypes(m_1)| = |argtypes(m)|
$\wedge\ m_1 \leq_{meth} m \wedge m_1 \neq m\ \}$.
$\neg\ (\vec{c} \leq_{inh}$ specializers(m_1)))

$\Leftrightarrow \langle$by set theory and de Morgan$\rangle$

$(\exists \vec{c} \in C^*$.
$\vec{c} \in$ top-non-conforming-class-vecs(specializers(m), argtypes(m))
$\wedge$ ($\neg\ \exists\ m' \in M$.
msg(m') = msg(m) $\wedge$ |argtypes(m')| = |argtypes(m)| $\wedge\ m' \leq_{meth} m\ \wedge\ m' \neq m$
$\wedge\ \vec{c} \leq_{inh}$ specializers(m')))

$\Leftrightarrow \langle$by set theory$\rangle$

{ $\vec{c} \in$ top-non-conforming-class-vecs(specializers(m), argtypes(m)) |
$\neg\ \exists\ m' \in M$.
msg(m') = msg(m) $\wedge$ |argtypes(m')| = |argtypes(m)| $\wedge\ m' \leq_{meth} m\ \wedge\ m' \neq m$
$\wedge\ \vec{c} \leq_{inh}$ specializers(m') } $\neq \varnothing$

$\Leftrightarrow \langle$by definition$\rangle$

top-non-overridden-non-conforming-class-vecs(m) $\neq \varnothing$

We now turn to the second main lemma of this section.

Lemma 15. ComputeSpecializedAreConforming $\Rightarrow$ SpecializersAreConforming

Proof: We calculate as follows.

ComputeSpecializedAreConforming

$\Leftrightarrow \langle$by definition$\rangle$

$\forall\ m \in M$.
($\forall\ i \in$ indexes(specializers(m)).
specializers$(m)_i \neq any\ \Rightarrow$ specializers$(m)_i$ <: argtypes$(m)_i$)
$\wedge$ top-non-overridden-non-conforming-class-vecs(m) = $\varnothing$

$\Longrightarrow \langle \text{by Lemma 14} \rangle$
$\quad \forall\, m \in M.$
$\qquad (\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)).$
$\qquad\quad \mathsf{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \mathsf{specializers}(m)_i <: \mathsf{argtypes}(m)_i)$
$\qquad \wedge\, (\forall\, \vec{c} \in \mathsf{covered\text{-}class\text{-}vecs}(m)\,.$
$\qquad\quad \vec{c} \neq \mathsf{specializers}(m)$
$\qquad\qquad \Rightarrow (\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)).$
$\qquad\qquad\qquad \mathsf{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \vec{c}_i <: \mathsf{argtypes}(m)_i))$

$\Longleftrightarrow \langle \text{by the law of excluded middle} \rangle$
$\quad \forall\, m \in M.$
$\qquad (\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)).$
$\qquad\quad \mathsf{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \mathsf{specializers}(m)_i <: \mathsf{argtypes}(m)_i)$
$\qquad \wedge\, (\mathsf{covered\text{-}class\text{-}vecs}(m) = \varnothing \;\vee \mathsf{covered\text{-}class\text{-}vecs}(m) \neq \varnothing)$
$\qquad \wedge\, (\forall\, \vec{c} \in \mathsf{covered\text{-}class\text{-}vecs}(m)\,.$
$\qquad\quad \vec{c} \neq \mathsf{specializers}(m)$
$\qquad\qquad \Rightarrow (\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)).$
$\qquad\qquad\qquad \mathsf{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \vec{c}_i <: \mathsf{argtypes}(m)_i))$

$\Longleftrightarrow \langle \text{by distribution of conjunction over disjunction} \rangle$
$\quad \forall\, m \in M.$
$\qquad (\mathsf{covered\text{-}class\text{-}vecs}(m) = \varnothing$
$\qquad\quad \wedge\, (\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)).$
$\qquad\qquad \mathsf{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \mathsf{specializers}(m)_i <: \mathsf{argtypes}(m)_i)$
$\qquad\quad \wedge\, (\forall\, \vec{c} \in \mathsf{covered\text{-}class\text{-}vecs}(m)\,.$
$\qquad\qquad \vec{c} \neq \mathsf{specializers}(m)$
$\qquad\qquad\quad \Rightarrow (\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)).$
$\qquad\qquad\qquad\quad \mathsf{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \vec{c}_i <: \mathsf{argtypes}(m)_i)))$
$\qquad \vee\, (\mathsf{covered\text{-}class\text{-}vecs}(m) \neq \varnothing$
$\qquad\quad \wedge\, (\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)).$
$\qquad\qquad \mathsf{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \mathsf{specializers}(m)_i <: \mathsf{argtypes}(m)_i)$
$\qquad\quad \wedge\, (\forall\, \vec{c} \in \mathsf{covered\text{-}class\text{-}vecs}(m)\,.$
$\qquad\qquad \vec{c} \neq \mathsf{specializers}(m)$
$\qquad\qquad\quad \Rightarrow (\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)).$
$\qquad\qquad\qquad\quad \mathsf{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \vec{c}_i <: \mathsf{argtypes}(m)_i)))$

$\Longleftrightarrow \langle \text{by empty range rule and distribution of conjunction over universal with nonempty range} \rangle$
$\quad \forall\, m \in M.$
$\qquad (\mathsf{covered\text{-}class\text{-}vecs}(m) = \varnothing$
$\qquad\quad \wedge\, (\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)).$
$\qquad\qquad \mathsf{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \mathsf{specializers}(m)_i <: \mathsf{argtypes}(m)_i)$
$\qquad\quad \wedge\, \text{true})$
$\qquad \vee\, (\mathsf{covered\text{-}class\text{-}vecs}(m) \neq \varnothing$
$\qquad\quad \wedge\, (\forall\, \vec{c} \in \mathsf{covered\text{-}class\text{-}vecs}(m)\,.$
$\qquad\qquad (\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)).$
$\qquad\qquad\quad \mathsf{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \mathsf{specializers}(m)_i <: \mathsf{argtypes}(m)_i)$
$\qquad\qquad \wedge\, (\vec{c} \neq \mathsf{specializers}(m)$
$\qquad\qquad\quad \Rightarrow (\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)).$
$\qquad\qquad\qquad\quad \mathsf{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \vec{c}_i <: \mathsf{argtypes}(m)_i))))$

$$
\begin{array}{l}
\Leftrightarrow \langle \text{by empty range rule and definition of implication} \rangle \\
\quad \forall\, m \in M. \\
\qquad (\text{covered-class-vecs}(m) = \varnothing \\
\qquad\quad \wedge (\forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \text{specializers}(m)_i <: \text{argtypes}(m)_i) \\
\qquad\quad \wedge (\forall\, \vec{c} \in \text{covered-class-vecs}(m)\, . \\
\qquad\qquad\quad \forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \vec{c}_i <: \text{argtypes}(m)_i)) \\
\qquad \vee (\text{covered-class-vecs}(m) \neq \varnothing \\
\qquad\quad \wedge (\forall\, \vec{c} \in \text{covered-class-vecs}(m)\, . \\
\qquad\qquad (\forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\quad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \text{specializers}(m)_i <: \text{argtypes}(m)_i) \\
\qquad\qquad \wedge (\vec{c} = \text{specializers}(m) \\
\qquad\qquad\quad \vee (\forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\qquad\quad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \vec{c}_i <: \text{argtypes}(m)_i))))
\end{array}
$$

$$
\begin{array}{l}
\Rightarrow \langle \text{by A} \wedge \text{B} \Rightarrow \text{A} \rangle \\
\quad \forall\, m \in M. \\
\qquad (\text{covered-class-vecs}(m) = \varnothing \\
\qquad\quad \wedge (\forall\, \vec{c} \in \text{covered-class-vecs}(m)\, . \\
\qquad\qquad\quad \forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \vec{c}_i <: \text{argtypes}(m)_i)) \\
\qquad \vee (\text{covered-class-vecs}(m) \neq \varnothing \\
\qquad\quad \wedge (\forall\, \vec{c} \in \text{covered-class-vecs}(m)\, . \\
\qquad\qquad (\forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\quad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \text{specializers}(m)_i <: \text{argtypes}(m)_i) \\
\qquad\qquad \wedge (\vec{c} = \text{specializers}(m) \\
\qquad\qquad\quad \vee (\forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\qquad\quad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \vec{c}_i <: \text{argtypes}(m)_i))))
\end{array}
$$

$$
\begin{array}{l}
\Leftrightarrow \langle \text{by distribution of conjunction over disjunction} \rangle \\
\quad \forall\, m \in M. \\
\qquad (\text{covered-class-vecs}(m) = \varnothing \\
\qquad\quad \wedge (\forall\, \vec{c} \in \text{covered-class-vecs}(m)\, . \\
\qquad\qquad\quad \forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \vec{c}_i <: \text{argtypes}(m)_i)) \\
\qquad \vee (\text{covered-class-vecs}(m) \neq \varnothing \\
\qquad\quad \wedge (\forall\, \vec{c} \in \text{covered-class-vecs}(m)\, . \\
\qquad\qquad ((\forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \text{specializers}(m)_i <: \text{argtypes}(m)_i) \\
\qquad\qquad\quad \wedge\, \vec{c} = \text{specializers}(m)) \\
\qquad\qquad \vee ((\forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \text{specializers}(m)_i <: \text{argtypes}(m)_i) \\
\qquad\qquad\quad \wedge (\forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\qquad\quad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \vec{c}_i <: \text{argtypes}(m)_i))))
\end{array}
$$

$$
\begin{array}{l}
\Longrightarrow \langle \text{by } \mathrm{A} \wedge \mathrm{B} \Rightarrow \mathrm{A} \rangle \\
\quad \forall\, m \in M. \\
\qquad (\text{covered-class-vecs}(m) = \varnothing \\
\qquad\quad \wedge\, (\forall\, \vec{c} \in \text{covered-class-vecs}(m)\,. \\
\qquad\qquad\quad \forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \vec{c}_i <: \text{argtypes}(m)_i)) \\
\qquad \vee\, (\text{covered-class-vecs}(m) \neq \varnothing \\
\qquad\quad \wedge\, (\forall\, \vec{c} \in \text{covered-class-vecs}(m)\,. \\
\qquad\qquad ((\forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \text{specializers}(m)_i <: \text{argtypes}(m)_i) \\
\qquad\qquad\quad \wedge\, \vec{c} = \text{specializers}(m)) \\
\qquad\qquad \vee\, (\forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \vec{c}_i <: \text{argtypes}(m)_i)))) \\
\Longleftrightarrow \langle \text{by defintion of equality for vectors} \rangle \\
\quad \forall\, m \in M. \\
\qquad (\text{covered-class-vecs}(m) = \varnothing \\
\qquad\quad \wedge\, (\forall\, \vec{c} \in \text{covered-class-vecs}(m)\,. \\
\qquad\qquad\quad \forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \vec{c}_i <: \text{argtypes}(m)_i)) \\
\qquad \vee\, (\text{covered-class-vecs}(m) \neq \varnothing \\
\qquad\quad \wedge\, (\forall\, \vec{c} \in \text{covered-class-vecs}(m)\,. \\
\qquad\qquad ((\forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \text{specializers}(m)_i <: \text{argtypes}(m)_i) \\
\qquad\qquad\quad \wedge\, (\forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\qquad\; \vec{c}_i = \text{specializers}(m)_i)) \\
\qquad\qquad \vee\, (\forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \vec{c}_i <: \text{argtypes}(m)_i)))) \\
\Longleftrightarrow \langle \text{by joining the term} \rangle \\
\quad \forall\, m \in M. \\
\qquad (\text{covered-class-vecs}(m) = \varnothing \\
\qquad\quad \wedge\, (\forall\, \vec{c} \in \text{covered-class-vecs}(m)\,. \\
\qquad\qquad\quad \forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \vec{c}_i <: \text{argtypes}(m)_i)) \\
\qquad \vee\, (\text{covered-class-vecs}(m) \neq \varnothing \\
\qquad\quad \wedge\, (\forall\, \vec{c} \in \text{covered-class-vecs}(m)\,. \\
\qquad\qquad (\forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\quad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \text{specializers}(m)_i <: \text{argtypes}(m)_i \\
\qquad\qquad\quad \wedge\, \vec{c}_i = \text{specializers}(m)_i) \\
\qquad\qquad \vee\, (\forall\, i \in \text{indexes}(\text{specializers}(m)). \\
\qquad\qquad\qquad \text{specializers}(m)_i \neq \mathit{any} \;\Rightarrow\; \vec{c}_i <: \text{argtypes}(m)_i))))
\end{array}
$$

$\Rightarrow \langle$by substituting $\vec{c}_i$ for $\mathsf{specializers}(m)_i\rangle$
$$\begin{array}{l}
\forall\, m \in M. \\
\quad (\mathsf{covered\text{-}class\text{-}vecs}(m) = \varnothing \\
\quad\quad \wedge (\forall\, \vec{c} \in \mathsf{covered\text{-}class\text{-}vecs}(m)\,. \\
\quad\quad\quad\quad \forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)). \\
\quad\quad\quad\quad\quad \mathsf{specializers}(m)_i \neq any \;\Rightarrow\; \vec{c}_i <: \mathsf{argtypes}(m)_i)) \\
\quad \vee (\mathsf{covered\text{-}class\text{-}vecs}(m) \neq \varnothing \\
\quad\quad \wedge (\forall\, \vec{c} \in \mathsf{covered\text{-}class\text{-}vecs}(m)\,. \\
\quad\quad\quad (\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)). \\
\quad\quad\quad\quad \mathsf{specializers}(m)_i \neq any \;\Rightarrow\; \vec{c}_i <: \mathsf{argtypes}(m)_i) \\
\quad\quad\quad \vee (\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)). \\
\quad\quad\quad\quad\quad \mathsf{specializers}(m)_i \neq any \;\Rightarrow\; \vec{c}_i <: \mathsf{argtypes}(m)_i))))
\end{array}$$

$\Leftrightarrow \langle$by idempotence of disjunction$\rangle$
$$\begin{array}{l}
\forall\, m \in M. \\
\quad (\mathsf{covered\text{-}class\text{-}vecs}(m) = \varnothing \\
\quad\quad \wedge (\forall\, \vec{c} \in \mathsf{covered\text{-}class\text{-}vecs}(m)\,. \\
\quad\quad\quad\quad \forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)). \\
\quad\quad\quad\quad\quad \mathsf{specializers}(m)_i \neq any \;\Rightarrow\; \vec{c}_i <: \mathsf{argtypes}(m)_i)) \\
\quad \vee (\mathsf{covered\text{-}class\text{-}vecs}(m) \neq \varnothing \\
\quad\quad \wedge (\forall\, \vec{c} \in \mathsf{covered\text{-}class\text{-}vecs}(m)\,. \\
\quad\quad\quad (\forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)). \\
\quad\quad\quad\quad \mathsf{specializers}(m)_i \neq any \;\Rightarrow\; \vec{c}_i <: \mathsf{argtypes}(m)_i)))
\end{array}$$

$\Leftrightarrow \langle$by distribution of conjunction over disjunction $\rangle$
$$\begin{array}{l}
\forall\, m \in M. \\
\quad (\mathsf{covered\text{-}class\text{-}vecs}(m) = \varnothing \vee \mathsf{covered\text{-}class\text{-}vecs}(m) \neq \varnothing) \\
\quad \wedge (\forall\, \vec{c} \in \mathsf{covered\text{-}class\text{-}vecs}(m)\,. \\
\quad\quad \forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)). \\
\quad\quad\quad \mathsf{specializers}(m)_i \neq any \;\Rightarrow\; \vec{c}_i <: \mathsf{argtypes}(m)_i)
\end{array}$$

$\Leftrightarrow \langle$by the law of the excluded middle and true is the unit of conjunction$\rangle$
$$\begin{array}{l}
\forall\, m \in M.\ \forall\, \vec{c} \in \mathsf{covered\text{-}class\text{-}vecs}(m)\,. \\
\quad \forall\, i \in \mathsf{indexes}(\mathsf{specializers}(m)). \\
\quad\quad \mathsf{specializers}(m)_i \neq any \;\Rightarrow\; \vec{c}_i <: \mathsf{argtypes}(m)_i
\end{array}$$

$\Leftrightarrow \langle$by definition$\rangle$
SpecializersAreConforming

A.3 Correctness of Completeness Checking Algorithm

We use the following simple lemma in the proofs of completeness checking.

Lemma 16. Let $s \in S$ and $\vec{c} \in C^*$.

$$\begin{array}{l}
\quad (\vec{c} <: \mathsf{argtypes}(s) \;\Rightarrow \\
\quad\quad \exists\, m \in \mathsf{relevant}(M, s).\ \vec{c} \leq_{inh} \mathsf{specializers}(m)) \\
\Leftrightarrow \\
\quad (\vec{c} <: \mathsf{argtypes}(s) \;\Rightarrow \\
\quad\quad \exists\, m \in \mathsf{applicable\text{-}methods}(s, \vec{c}))
\end{array}$$

Proof:

$\vec{c}$ <: argtypes(s) $\Rightarrow$
  $\exists\, m \in$ relevant(M, s). $\vec{c} \leq_{inh}$ specializers(m)

$\Leftrightarrow$ $\langle$by definition of relevant$\rangle$
  $\vec{c}$ <: argtypes(s) $\Rightarrow$
    $\exists\, m \in \{ m \in M \mid$ msg(m) = msg(s) $\wedge$ |argtypes(m)| = |argtypes(s)| }.
      $\vec{c} \leq_{inh}$ specializers(m)

$\Leftrightarrow$ $\langle$by set theory$\rangle$
  $\vec{c}$ <: argtypes(s) $\Rightarrow$
    $\exists\, m \in \{ m \in M \mid$
      msg(m) = msg(s) $\wedge$ |argtypes(m)| = |argtypes(s)| $\wedge$ $\vec{c} \leq_{inh}$ specializers(m) }

$\Leftrightarrow$ $\langle$by definition of <: which is true only of equal-length vectors$\rangle$
  $\vec{c}$ <: argtypes(s) $\Rightarrow$
    $\exists\, m \in \{ m \in M \mid$
      msg(m) = msg(s) $\wedge$ |argtypes(m)| = $|\vec{c}|$ $\wedge$ $\vec{c} \leq_{inh}$ specializers(m) }

$\Leftrightarrow$ $\langle$by definition of applicable-methods$\rangle$
  $\vec{c}$ <: argtypes(s) $\Rightarrow$
    $\exists\, m \in$ applicable-methods(s, $\vec{c}$)

The proof of the following lemma, which is used in the proof of conformance checking, is identical to the proof of Lemma 16.

Lemma 17. Let $s \in S$ and $\vec{c} \in C^*$.

  ($\vec{c}$ <: argtypes(s) $\Rightarrow$
    $\neg\exists\, m \in$ relevant(M, s). $\vec{c} \leq_{inh}$ specializers(m))

$\Leftrightarrow$

  ($\vec{c}$ <: argtypes(s) $\Rightarrow$
    $\neg\exists\, m \in$ applicable-methods(s, $\vec{c}$))

The following theorem says that our algorithm for computing completeness is sufficient to ensure completeness. Note that the proof also shows the converse.

Theorem 3. ComputeIsComplete $\Rightarrow$ ImplementationIsComplete

Proof: We prove this theorem by the following calculation.

  ComputeIsComplete

$\Leftrightarrow$ $\langle$by definition$\rangle$
  $\forall\, s \in S$.
    let *any-vector* = { $\vec{c}$ } where $|\vec{c}|$ = |argtypes(s)| and each c_i = *any* **in**
      IsComplete(relevant(M, s), *any-vector*, s)

$\Leftrightarrow$ $\langle$by definition of IsComplete(relevant(M, s), *any-vector*, s)$\rangle$
  $\forall\, s \in S$.
    let *any-vector* = { $\vec{c}$ } where $|\vec{c}|$ = |argtypes(s)| and each c_i = *any* **in**
      $\forall\, \vec{c} \in$ *any-vector*.
        let *TCSs* = { $\vec{c}\,'$ | $c_i' \in$ top-concrete-conforming-subclasses(c_i, argtypes(s)$_i$) } **in**
          $\forall\, \vec{c}\,' \in$ *TCSs*. $\exists\, m \in$ relevant(M, s). $\vec{c}\,' \leq_{inh}$ specializers(m)

$\Leftrightarrow$ $\langle$by definition of *any-vector*, there is only one $\vec{c}$ in *any-vector* and each element of $\vec{c}$ is *any*$\rangle$
$\forall\, s \in S.$
  let $TCSs = \{\, \vec{c}\,' \mid c_i' \in \text{top-concrete-conforming-subclasses}(any, \text{argtypes}(s)_i) \,\}$ **in**
    $\forall\, \vec{c}\,' \in TCSs.\ \exists\, m \in \text{relevant}(M, s).\ \vec{c}\,' \leq_{inh} \text{specializers}(m)$

$\Leftrightarrow$ $\langle$by definition of top-concrete-conforming-subclasses(any, argtypes$(s)_i$)$\rangle$
$\forall\, s \in S.$
  let $TCSs = \{\, \vec{c}\,' \mid c_i' \in \text{top-classes}(\text{concrete-conforming-subclasses}(any, \text{argtypes}(s)_i)) \,\}$
  in
    $\forall\, \vec{c}\,' \in TCSs.\ \exists\, m \in \text{relevant}(M, s).\ \vec{c}\,' \leq_{inh} \text{specializers}(m)$

$\Leftrightarrow$ $\langle$by definition of concrete-conforming-subclasses(any, argtypes$(s)_i$)$\rangle$
$\forall\, s \in S.$
  let $TCSs = \{\, \vec{c}\,' \mid c_i' \in \text{top-classes}(\{\, c' \in C_{concrete} \mid c' \leq_{inh} any \wedge c' <: \text{argtypes}(s)_i \,\}) \,\}$ **in**
    $\forall\, \vec{c}\,' \in TCSs.\ \exists\, m \in \text{relevant}(M, s).\ \vec{c}\,' \leq_{inh} \text{specializers}(m)$

$\Leftrightarrow$ $\langle$by definition of *any*, every class c' is such that $c' \leq_{inh} any$$\rangle$
$\forall\, s \in S.$
  let $TCSs = \{\, \vec{c}\,' \mid c_i' \in \text{top-classes}(\{\, c' \in C_{concrete} \mid c' <: \text{argtypes}(s)_i \,\}) \,\}$ **in**
    $\forall\, \vec{c}\,' \in TCSs.\ \exists\, m \in \text{relevant}(M, s).\ \vec{c}\,' \leq_{inh} \text{specializers}(m)$

$\Leftrightarrow$ $\langle$by the definition of *TCSs* in the **let**$\rangle$
$\forall\, s \in S.$
  $\forall\, \vec{c}\,' \in \{\, \vec{c}\,' \mid c_i' \in \text{top-classes}(\{\, c' \in C_{concrete} \mid c' <: \text{argtypes}(s)_i \,\}) \,\}.$
    $\exists\, m \in \text{relevant}(M, s).\ \vec{c}\,' \leq_{inh} \text{specializers}(m)$

$\Leftrightarrow$ $\langle$by definition of top-classes and $<:$ for vectors$\rangle$
$\forall\, s \in S.$
  $\forall\, \vec{c}\,' \in \text{top-classes}(\{\, \vec{c}\,' \in (C_{concrete})^* \mid \vec{c}\,' <: \text{argtypes}(s) \,\}).$
    $\exists\, m \in \text{relevant}(M, s).\ \vec{c}\,' \leq_{inh} \text{specializers}(m)$

$\Leftrightarrow$ $\langle$by definition of top-classes and $\leq_{inh}$$\rangle$
$\forall\, s \in S.$
  $\forall\, \vec{c}\,' \in \{\, \vec{c}\,' \in (C_{concrete})^* \mid \vec{c}\,' <: \text{argtypes}(s) \,\}.$
    $\exists\, m \in \text{relevant}(M, s).\ \vec{c}\,' \leq_{inh} \text{specializers}(m)$

$\Leftrightarrow$ $\langle$by set theory$\rangle$
$\forall\, s \in S.\ \forall \vec{c}\,' \in (C_{concrete})^*.$
  $\vec{c}\,' <: \text{argtypes}(s) \Rightarrow$
    $\exists\, m \in \text{relevant}(M, s).\ \vec{c}\,' \leq_{inh} \text{specializers}(m)$

$\Leftrightarrow$ $\langle$by lemma 16$\rangle$
$\forall\, s \in S.\ \forall \vec{c}\,' \in (C_{concrete})^*.$
  $\vec{c}\,' <: \text{argtypes}(s) \Rightarrow$
    $\exists\, m \in \text{applicable-methods}(s, \vec{c}\,')$

$\Leftrightarrow$ $\langle$by set theory$\rangle$
$\forall\, s \in S.\ \forall \vec{c}\,' \in (C_{concrete})^*.$
  $\vec{c}\,' <: \text{argtypes}(s) \Rightarrow$
    $|\text{applicable-methods}(s, \vec{c}\,')| > 0$

$\Leftrightarrow \langle$by definition, renaming $\vec{c}\,'$ to $\vec{c}\rangle$
ImplementationIsComplete

A.4 Correctness of Consistency Checking Algorithm

Theorem 4. ComputeIsConsistent $\Rightarrow$ ImplementationIsConsistent

Proof: We prove this theorem by the following calculation.

ComputeIsConsistent

$\Leftrightarrow \langle$by definition$\rangle$
$\forall\, s \in S.\ \text{IsConsistent}(\text{relevant}(M, s), s)$

$\Leftrightarrow \langle$by definition of $\text{IsConsistent}(\text{relevant}(M, s), s)\rangle$
$\forall\, s \in S.\ \forall\, (m_1, m_2) \in \text{incomparable-pairs}(\text{relevant}(M, s)).$
let $TLBs = \text{tlb}(\text{specializers}(m_1), \text{specializers}(m_2), s),$
$M\text{-}reduced = \{\, m \in \text{relevant}(M, s) \mid m \leq_{meth} m_1 \wedge m \leq_{meth} m_2 \,\}$ **in**
$\text{IsComplete}(M\text{-}reduced, TLBs, s)$

$\Leftrightarrow \langle$by definition of $\text{incomparable-pairs}(\text{relevant}(M, s))\rangle$
$\forall\, s \in S.\ \forall m_1, m_2 \in \text{relevant}(M, s).\ \neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1) \Rightarrow$
let $TLBs = \text{tlb}(\text{specializers}(m_1), \text{specializers}(m_2), s),$
$M\text{-}reduced = \{\, m \in \text{relevant}(M, s) \mid m \leq_{meth} m_1 \wedge m \leq_{meth} m_2 \,\}$ **in**
$\text{IsComplete}(M\text{-}reduced, TLBs, s)$

$\Leftrightarrow \langle$by definition of $\text{IsComplete}(M\text{-}reduced, TLBs, s)\rangle$
$\forall\, s \in S.\ \forall m_1, m_2 \in \text{relevant}(M, s).\ \neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1) \Rightarrow$
let $TLBs = \text{tlb}(\text{specializers}(m_1), \text{specializers}(m_2), s),$
$M\text{-}reduced = \{\, m \in \text{relevant}(M, s) \mid m \leq_{meth} m_1 \wedge m \leq_{meth} m_2 \,\}$ **in**
$\forall\, \vec{c} \in TLBs.$
let $TCSs = \{\, \vec{c}\,' \mid c_i' \in \text{top-concrete-conforming-subclasses}(c_i, \text{argtypes}(s)_i) \,\}$ **in**
$\forall\, \vec{c}\,' \in TCSs.\ \exists\, m \in M\text{-}reduced.\ \vec{c}\,' \leq_{inh} \text{specializers}(m)$

$\Leftrightarrow \langle$by definition of $M\text{-}reduced$ and set theory$\rangle$
$\forall\, s \in S.\ \forall m_1, m_2 \in \text{relevant}(M, s).\ \neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1) \Rightarrow$
let $TLBs = \text{tlb}(\text{specializers}(m_1), \text{specializers}(m_2), s)$ **in**
$\forall\, \vec{c} \in TLBs.$
let $TCSs = \{\, \vec{c}\,' \mid c_i' \in \text{top-concrete-conforming-subclasses}(c_i, \text{argtypes}(s)_i) \,\}$ **in**
$\forall\, \vec{c}\,' \in TCSs.\ \exists\, m \in \text{relevant}(M, s).$
$\vec{c}\,' \leq_{inh} \text{specializers}(m) \wedge m \leq_{meth} m_1 \wedge m \leq_{meth} m_2$

$\Rightarrow \langle$by definition of $TLBs$, $TCSs$, and inheritance$\rangle$
$\forall\, s \in S.\ \forall m_1, m_2 \in \text{relevant}(M, s).\ \neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1) \Rightarrow$
$\forall\, \vec{c} \in (C_{concrete})^*.$
$\vec{c} <: \text{argtypes}(s) \ \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_1) \ \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_2) \Rightarrow$
$\exists\, m \in \text{relevant}(M, s).\ \vec{c}\,' \leq_{inh} \text{specializers}(m) \wedge m \leq_{meth} m_1 \wedge m \leq_{meth} m_2$

$\Leftrightarrow \langle$by logic, as $\vec{c}$ does not occur in $\forall m_1, m_2 \in \text{relevant}(M, s).\ (\neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1))\rangle$

$$\begin{array}{l}
\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*. \\
\quad \forall m_1, m_2 \in \text{relevant}(M, s). \\
\quad\quad \neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1) \ \Rightarrow \\
\quad\quad\quad \vec{c} <: \text{argtypes}(s) \ \wedge \vec{c} \leq_{inh} \text{specializers}(m_1) \ \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_2) \ \Rightarrow \\
\quad\quad\quad\quad \exists\, m \in \text{relevant}(M, s).\ \vec{c}\,' \leq_{inh} \text{specializers}(m) \wedge m \leq_{meth} m_1 \wedge m \leq_{meth} m_2
\end{array}$$

$\Leftrightarrow \langle$by definition of $\Rightarrow$, twice, and predicate calculus$\rangle$

$$\begin{array}{l}
\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*. \\
\quad \forall m_1, m_2 \in \text{relevant}(M, s). \\
\quad\quad \neg(\neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1)) \ \vee \\
\quad\quad \neg(\vec{c} <: \text{argtypes}(s)) \ \vee\ \neg(\vec{c} \leq_{inh} \text{specializers}(m_1) \ \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_2)) \ \vee \\
\quad\quad \exists\, m \in \text{relevant}(M, s).\ \vec{c}\,' \leq_{inh} \text{specializers}(m) \wedge m \leq_{meth} m_1 \wedge m \leq_{meth} m_2
\end{array}$$

$\Leftrightarrow \langle$by associativity of $\vee\rangle$

$$\begin{array}{l}
\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*. \\
\quad \forall m_1, m_2 \in \text{relevant}(M, s). \\
\quad\quad \neg(\vec{c} <: \text{argtypes}(s)) \ \vee \\
\quad\quad \neg(\neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1)) \ \vee \\
\quad\quad \neg(\vec{c} \leq_{inh} \text{specializers}(m_1) \ \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_2)) \ \vee \\
\quad\quad \exists\, m \in \text{relevant}(M, s).\ \vec{c}\,' \leq_{inh} \text{specializers}(m) \wedge m \leq_{meth} m_1 \wedge m \leq_{meth} m_2
\end{array}$$

$\Leftrightarrow \langle$by definition of $\Rightarrow$, three times$\rangle$

$$\begin{array}{l}
\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*. \\
\quad \forall\, m_1, m_2 \in \text{relevant}(M, s). \\
\quad\quad \vec{c} <: \text{argtypes}(s) \ \Rightarrow \\
\quad\quad\quad \neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1) \ \Rightarrow \\
\quad\quad\quad\quad \vec{c} \leq_{inh} \text{specializers}(m_1) \ \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_2) \ \Rightarrow \\
\quad\quad\quad\quad\quad \exists\, m \in \text{relevant}(M, s).\ \vec{c} \leq_{inh} \text{specializers}(m) \ \wedge\ m \leq_{meth} m_1 \ \wedge\ m \leq_{meth} m_2
\end{array}$$

$\Leftrightarrow \langle$by logic, as m_1, m_2 do not occur in $\vec{c} <: \text{argtypes}(s)\rangle$

$$\begin{array}{l}
\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*. \\
\quad \vec{c} <: \text{argtypes}(s) \ \Rightarrow \\
\quad\quad \forall\, m_1, m_2 \in \text{relevant}(M, s). \\
\quad\quad\quad \neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1) \ \Rightarrow \\
\quad\quad\quad\quad \vec{c} \leq_{inh} \text{specializers}(m_1) \ \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_2) \ \Rightarrow \\
\quad\quad\quad\quad\quad \exists\, m \in \text{relevant}(M, s).\ \vec{c} \leq_{inh} \text{specializers}(m) \ \wedge\ m \leq_{meth} m_1 \ \wedge\ m \leq_{meth} m_2
\end{array}$$

$\Leftrightarrow \langle$by predicate calculus$\rangle$

$$\begin{array}{l}
\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*. \\
\quad \vec{c} <: \text{argtypes}(s) \ \Rightarrow \\
\quad\quad \forall\, m_1, m_2 \in \text{relevant}(M, s). \\
\quad\quad\quad (\neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1) \ \Rightarrow \\
\quad\quad\quad\quad \vec{c} \leq_{inh} \text{specializers}(m_1) \ \wedge\ \vec{c} \leq_{inh} \text{specializers}(m_2) \ \Rightarrow \\
\quad\quad\quad\quad\quad \exists\, m \in \text{relevant}(M, s).\ \vec{c} \leq_{inh} \text{specializers}(m) \ \wedge\ m \leq_{meth} m_1 \ \wedge\ m \leq_{meth} m_2) \\
\quad\quad\quad \wedge\ true \wedge true
\end{array}$$

$\Leftrightarrow \langle$by reflexivity of $\leq_{meth}$ and predicate calculus$\rangle$

$\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*.$

$\quad \vec{c} <: \mathsf{argtypes}(s) \Rightarrow$

$\quad\quad \forall\, m_1, m_2 \in \mathsf{relevant}(M, s).$

$\quad\quad\quad (\neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1) \Rightarrow$

$\quad\quad\quad\quad \vec{c} \leq_{inh} \mathsf{specializers}(m_1)) \wedge \vec{c} \leq_{inh} \mathsf{specializers}(m_2) \Rightarrow$

$\quad\quad\quad\quad\quad \exists\, m \in \mathsf{relevant}(M, s).\ \vec{c} \leq_{inh} \mathsf{specializers}(m) \wedge m \leq_{meth} m_1 \wedge m \leq_{meth} m_2)$

$\quad\quad\quad \wedge\, (m_1 \leq_{meth} m_2 \Rightarrow$

$\quad\quad\quad\quad \vec{c} \leq_{inh} \mathsf{specializers}(m_1) \Rightarrow$

$\quad\quad\quad\quad\quad \vec{c} \leq_{inh} \mathsf{specializers}(m_1) \wedge m_1 \leq_{meth} m_1 \wedge m_1 \leq_{meth} m_2)$

$\quad\quad\quad \wedge\, (m_2 \leq_{meth} m_1 \Rightarrow$

$\quad\quad\quad\quad \vec{c} \leq_{inh} \mathsf{specializers}(m_2) \Rightarrow$

$\quad\quad\quad\quad\quad \vec{c} \leq_{inh} \mathsf{specializers}(m_2) \wedge m_2 \leq_{meth} m_2 \wedge m_2 \leq_{meth} m_1)$

$\Rightarrow \langle$by existentially quantifying over m_1 in one clause and m_2 in another clause$\rangle$

$\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*.$

$\quad \vec{c} <: \mathsf{argtypes}(s) \Rightarrow$

$\quad\quad \forall\, m_1, m_2 \in \mathsf{relevant}(M, s).$

$\quad\quad\quad (\neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1) \Rightarrow$

$\quad\quad\quad\quad \vec{c} \leq_{inh} \mathsf{specializers}(m_1)) \wedge \vec{c} \leq_{inh} \mathsf{specializers}(m_2) \Rightarrow$

$\quad\quad\quad\quad\quad \exists\, m \in \mathsf{relevant}(M, s).\ \vec{c} \leq_{inh} \mathsf{specializers}(m) \wedge m \leq_{meth} m_1 \wedge m \leq_{meth} m_2)$

$\quad\quad\quad \wedge\, (m_1 \leq_{meth} m_2 \Rightarrow$

$\quad\quad\quad\quad \vec{c} \leq_{inh} \mathsf{specializers}(m_1) \wedge \vec{c} \leq_{inh} \mathsf{specializers}(m_2) \Rightarrow$

$\quad\quad\quad\quad\quad \exists\, m \in \mathsf{relevant}(M, s).\ \vec{c} \leq_{inh} \mathsf{specializers}(m) \wedge m \leq_{meth} m_1 \wedge m \leq_{meth} m_2)$

$\quad\quad\quad \wedge\, (m_2 \leq_{meth} m_1 \Rightarrow$

$\quad\quad\quad\quad \vec{c} \leq_{inh} \mathsf{specializers}(m_1) \wedge \vec{c} \leq_{inh} \mathsf{specializers}(m_2) \Rightarrow$

$\quad\quad\quad\quad\quad \exists\, m \in \mathsf{relevant}(M, s).\ \vec{c} \leq_{inh} \mathsf{specializers}(m) \wedge m \leq_{meth} m_2 \wedge m \leq_{meth} m_1)$

$\Leftrightarrow \langle$by $((P \Rightarrow R) \wedge (Q \Rightarrow R)) \Leftrightarrow ((P \vee Q) \Rightarrow R)$ and predicate calculus$\rangle$

$\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*.$

$\quad \vec{c} <: \mathsf{argtypes}(s) \Rightarrow$

$\quad\quad \forall\, m_1, m_2 \in \mathsf{relevant}(M, s).$

$\quad\quad\quad (\neg(m_1 \leq_{meth} m_2 \vee m_2 \leq_{meth} m_1) \Rightarrow$

$\quad\quad\quad\quad \vec{c} \leq_{inh} \mathsf{specializers}(m_1) \wedge \vec{c} \leq_{inh} \mathsf{specializers}(m_2) \Rightarrow$

$\quad\quad\quad\quad\quad \exists\, m \in \mathsf{relevant}(M, s).\ \vec{c} \leq_{inh} \mathsf{specializers}(m) \wedge m \leq_{meth} m_1 \wedge m \leq_{meth} m_2)$

$\quad\quad\quad \wedge\, (m_1 \leq_{meth} m_2 \vee m_2 \leq_{meth} m_1 \Rightarrow$

$\quad\quad\quad\quad \vec{c} \leq_{inh} \mathsf{specializers}(m_1) \wedge \vec{c} \leq_{inh} \mathsf{specializers}(m_2) \Rightarrow$

$\quad\quad\quad\quad\quad \exists\, m \in \mathsf{relevant}(M, s).\ \vec{c} \leq_{inh} \mathsf{specializers}(m) \wedge m \leq_{meth} m_1 \wedge m \leq_{meth} m_2)$

$\Leftrightarrow \langle$by the law of the excluded middle$\rangle$

$\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*.$

$\quad \vec{c} <: \mathsf{argtypes}(s) \Rightarrow$

$\quad\quad \forall\, m_1, m_2 \in \mathsf{relevant}(M, s).$

$\quad\quad\quad \vec{c} \leq_{inh} \mathsf{specializers}(m_1) \wedge \vec{c} \leq_{inh} \mathsf{specializers}(m_2) \Rightarrow$

$\quad\quad\quad\quad \exists\, m \in \mathsf{relevant}(M, s).\ \vec{c} \leq_{inh} \mathsf{specializers}(m) \wedge m \leq_{meth} m_1 \wedge m \leq_{meth} m_2$

$\Leftrightarrow \langle$by definition of $\Rightarrow\rangle$
$\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*.$
$\quad \vec{c}$ <: argtypes(s) $\Rightarrow$
$\quad\quad \forall\, m_1, m_2 \in$ applicable-methods($s, \vec{c}$).
$\quad\quad\quad \neg\, (\vec{c} \leq_{inh}$ specializers(m_1) $\wedge\ \vec{c} \leq_{inh}$ specializers(m_2)) $\vee$
$\quad\quad\quad (\exists\, m \in$ relevant(M, s). $\vec{c} \leq_{inh}$ specializers(m) $\wedge\ m \leq_{meth} m_1\ \wedge\ m \leq_{meth} m_2$)

$\Leftrightarrow \langle$by $\forall x.P(x) \Leftrightarrow \neg\exists x.\neg P(x)$ and predicate calculus$\rangle$
$\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*.$
$\quad \vec{c}$ <: argtypes(s) $\Rightarrow$
$\quad\quad \neg\, \exists\, m_1, m_2 \in$ relevant(M, s).
$\quad\quad\quad \vec{c} \leq_{inh}$ specializers(m_1) $\wedge\ \vec{c} \leq_{inh}$ specializers(m_2) $\wedge$
$\quad\quad\quad \neg\, (\exists\, m \in$ relevant(M, s). $\vec{c} \leq_{inh}$ specializers(m) $\wedge\ m \leq_{meth} m_1\ \wedge\ m \leq_{meth} m_2$)

$\Leftrightarrow \langle$by lemma 17, three times$\rangle$
$\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*.$
$\quad \vec{c}$ <: argtypes(s) $\Rightarrow$
$\quad\quad \neg\, \exists\, m_1, m_2 \in$ applicable-methods($s, \vec{c}$).
$\quad\quad\quad \neg\, (\exists\, m \in$ applicable-methods($s, \vec{c}$). $m \leq_{meth} m_1\ \wedge\ m \leq_{meth} m_2$)

$\Leftrightarrow \langle$by $\neg\exists x.\neg P(x) \Leftrightarrow \forall x.P(x)\rangle$
$\forall\, s \in S.\ \forall\, \vec{c} \in (C_{concrete})^*.$
$\quad \vec{c}$ <: argtypes(s) $\Rightarrow$
$\quad\quad \forall\, m_1, m_2 \in$ applicable-methods($s, \vec{c}$).
$\quad\quad\quad \exists\, m \in$ applicable-methods($s, \vec{c}$). $m \leq_{meth} m_1\ \wedge\ m \leq_{meth} m_2$

$\Leftrightarrow \langle$by definition$\rangle$
ImplementationIsConsistent