3-1994

Type Checking and Modules for Multi-Methods

Craig Chambers
Iowa State University

Gary T. Leavens
Iowa State University

Type Checking and Modules for Multi-Methods

Abstract
Two major obstacles preventing the wider acceptance of multi-methods are concerns over the lack of encapsulation and modularity and the lack 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 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, the presence of abstract classes, and graph-based method lookup semantics. We prove formally that our algorithm fulfills its specification. 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.

Keywords
Multi-methods, object-oriented programming, encapsulation, modules, packages, static typechecking, typechecking algorithms, conformance, completeness, consistency, subtype, inheritance, abstract classes, Cecil language

Disciplines
Programming Languages and Compilers | Systems Architecture

Type Checking and Modules for Multi-Methods

Craig Chambers and Gary T. Leavens
TR #94-03
March 1994

A shorter version of this report has been submitted for publication..

Keywords: Multi-methods, object-oriented programming, encapsulation, modules, packages, static typechecking, typechecking algorithms, conformance, completeness, and consistency, subtype, inheritance abstract classes, Cecil language.

1994 CR Categories: D.3.2 [*Programming Languages*] Language Classifications — Object-oriented languages; D.3.3 [*Programming Language*] Language Constructs and Features — Modules, packages; D.3.m [*Programming Language*] Miscellaneous — type systems; F.2.m [*Analysis of Algorithms and Problem Complexity*] Miscellaneous — type checking algorithms; F.3.3 [*Logics and Meanings of Programs*] Studies of Program Constructs — type structure.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

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 94-03-01

ISU CS Technical Report #94-03

Abstract

Two major obstacles preventing the wider acceptance of multi-methods are concerns over the lack of encapsulation and modularity and the lack 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 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, the presence of abstract classes, and graph-based method lookup semantics. We prove formally that our algorithm fulfills its specification. 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 dynamic type or class of any of the arguments to a message, not just the dynamic type of the first as in singly-dispatched systems and not just the arguments' static type as in systems with static overloading. To illustrate, consider the following matrix multiplication implementations, written in a close approximation to Cecil syntax:*

```
abstract type matrix; -- matrix is the abstract superclass of all matrix implementations
   method index(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 index fns to do indexing
   method *(m1:matrix, m2:matrix):matrix {
      ... } -- multiply matrices, invoking implementation-specific index fns to do indexing
```

* 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.

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

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

let a, b: matrix := ...;
print(a + b*b); -- will invoke most specific + and * functions, depending on dynamic 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 *implementations*. Static type checking must guarantee that these interface requirements are satisfied.

In the matrix algebra example, the method `*` is unspecialized, and hence acts like a normal function. The methods named `index` 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 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 the wider acceptance of multi-methods:

- 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 preserves much of the flavor of object-centered programming [Chambers 92].
- The semantics of multi-method lookup is considered extremely 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 are seen to prevent object encapsulation. One approach to solving this problem was presented in the same paper, but that approach did not allow privileged access to be restricted to a single, well-defined area of program text.
- Multi-methods might be slower to implement efficiently than singly-dispatched methods. Work is progressing on this front, however, and we expect that the run-time performance difference between singly- and multiply-dispatched systems to become negligible in the near future.
- 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 separate 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 two points above:

- We describe a 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 show our algorithm to run in polynomial time.
- 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.

These two contributions are integrated: the module system helps make typechecking more practical, and the typechecking algorithm is compatible with and supports our module system.

The next section of this paper reviews related work. Section 3 describes the language model 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 assumptions about the language they typecheck:

- The multi-methods within a generic function can 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 matrix class and the index function earlier.
- Inheritance and subtyping is 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 and more flexible and extensible organizations of code can result if the two relations are allowed to be distinct [e.g. Snyder 86, Cook *et al.* 90, Leavens & Weihl 90], 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 graphs. 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*) to be defined independently of the set of multi-method implementations.
- 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, and 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]. 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 single-dispatching 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 only, and users may always circumvent the encapsulation of a package `p` by writing "`p::internal_sym`." Common Lisp does not include static type checking. Encapsulation can be enforced in our module system and our module system cooperates with our static typechecking algorithm.

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.

Several 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 supports these various 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

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 paragraphs elaborate on these assumptions and show how the Cecil language's constructs meet these assumptions. Appendix 3 formalizes these assumptions.

3.1 Classes and Inheritance

We assume that the program includes a fixed set of classes and a fixed implementation inheritance graph, potentially including multiple inheritance. Each class is marked as either abstract or concrete, with the implication that abstract classes cannot be directly instantiated at run-time. Abstract classes model pure virtual classes in C++ and deferred classes in Eiffel. We assume that there exists a class that is the ancestor of all other classes, and that this class is used as the specializer of "unspecialized" formals.

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`. `matrix_rep` is modeled as an abstract class, while `dense_matrix_rep` is considered 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` that is the ancestor of all other classes and used as the specializer of otherwise unspecialized formals of multi-methods. Cecil supports closures (first-class

function objects), and each distinct textual occurrence of a closure constructor expression (the Cecil equivalent of a lambda expression) is treated as a distinct new class.

3.2 Types and Subtyping

We assume that the program includes a fixed set of types related through subtyping. The type graph can be different than the class inheritance graph: types and classes can be independent, and inheritance and subtyping can differ. 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`. Cecil also includes the following special kinds of types:

- `void`, the type of functions that return no useful result to their callers, which is the supertype of all other types,
- `any`, which the supertype of all other non-`void` types,
- `none`, the type of functions that do not return to their callers, which is a subtype of all other types,
- $t_1 \mid t_2$, the most specific supertype (least upper bound) of two types,
- $t_1 \,\&\, t_2$, the most general subtype (greatest lower bound) of two types, and
- the types of closures, which use standard contravariant rules for subtyping.

3.3 Conformance of Classes to Types

The class and type graphs are related through conformance. A class *conforms* to a type when its direct instances may legally result from evaluating an expression of the type. We assume that the program indicates whether a class conforms to a type. In Cecil, conformance is derived from `conforms` clauses that are part of `representation` declarations and from subtyping of types:

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

These declarations indicate that the class `matrix_rep` conforms to the type `matrix_type` and to all supertypes of `matrix_type`, and that `dense_matrix_rep` conforms to `dense_matrix_type` and 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*. 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 is defined similarly.

3.5 Method Implementations

We assume a program contains a fixed set of method implementations. Each method implementation has a name, a vector of argument specializer classes, a vector of argument types, a result type, and a body. 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`.

The method overriding relationship is derived from the overriding relationship of the methods' argument specializer class vectors. This ordering reflects the message lookup semantics in Cecil: one method overrides another exactly when its argument specializers are more specific than the other's. 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*, where each signature has a name, a vector of argument types, and a result type. A signature represents a message that is considered legal to send, and consequently it places constraints on the set of method implementations supporting the signature. 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`.

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 an 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 an expression, 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.

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 and 4.5 filling in the details. Section 4.6 discusses the impact on the algorithm of some of the more sophisticated language features supported by our model.

This section presents our typechecking algorithm and correctness arguments informally. Appendix 4 formally specifies the typechecking problem, presents our algorithm formally, and includes a detailed complexity analysis of the algorithm.

4.1 Client-Side Typechecking

Client-side checks are fairly typical, and include checks like an expression of one type is only assigned to variables declared to be of a supertype and 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 that of 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. 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 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 `+` message is type-correct. The type of the result of this message is `int`, the most specific result type of the matching signatures.

Other client-side typechecks are straightforward and language-dependent, and we do not discuss them further here.

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 and only if every possible message that could be sent to concrete

arguments which 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, and that method's declared argument types can handle the actual arguments being passed and the method's result type is a subtype of that promised by the signature. Simply translating this specification into the isomorphic algorithm would lead to an algorithm whose execution time was exponential in the number of concrete classes in the program, which clearly is infeasible in a practical language. One of our main contributions is a polynomial-time algorithm for implementation-side typechecking for the class of languages that can be modeled as 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 a type/signature interface:

- For each signature, every method whose specializers could match a send that would typecheck against the signature must *conform* to the signature; i.e., the method's argument and result types must be compatible with those specified by the signature.
- For each signature, the methods implementing a signature must be *complete*; i.e., for any 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.
- For each signature, the methods implementing a signature must be *consistent*; i.e., for any concrete argument vector conforming to the signature's argument types, there must exist no more that one most-specific method that implements the message. If the methods are inconsistent, then a "message ambiguously defined" error could occur at run-time.

The following declarations illustrate these issues:

```
-- 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;
concrete representation int_rep conforms int inherits num_rep;
concrete representation fraction_rep conforms fraction inherits num_rep;
concrete 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 for two `float_rep` objects does not conform to the `+(fraction,fraction):fraction` signature, 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 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.

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 for any additional checks such as that a type really is a subtype of all its declared supertypes. If the method implementations are conformant, complete, and consistent with respect to all applicable signatures, then all classes are guaranteed to fully and correctly implement the types to which they conform. 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 satisfied.

4.4 Checking Conformance

Given a signature, the set of methods which must conform to the signature are those that might be invoked from a message that matches the signature. We say such a method is *covered* by the signature. A method is covered by a signature if there exists some vector of concrete classes that both conforms to the argument types of the signature and inherits from the method's argument specializers.

For every signature, for every method covered by the signature, we 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, restricted to unspecialized formals.

For each specialized formal, we need to ensure that, for every class of actual argument that might invoke the method, the class 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. A simple check would be that the specializer class conforms to the declared type. However, this check is not sufficient: 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 the declared type of the specialized formal. Consider the following example:

```
type a_type;
representation a_rep conforms a_type;
method foo(x@a_rep:a_type) { ... }
type b_type; -- is not a subtype of a_type
representation b_rep inherits a_rep conforms b_type;
... foo(new b_rep) ... -- violates conformance!
```

Here the `b_rep` class inherits the `foo` method, but `b_rep` objects do not conform to the `a_type` expected in the `foo` method. This could lead to dynamic type errors if messages are sent to `x` that are supported by `a_type` but not `b_type`. To detect these kinds of problems, our algorithm computes the most specific types to which the specializer and its subclasses conform, and then verifies that these types conform to the

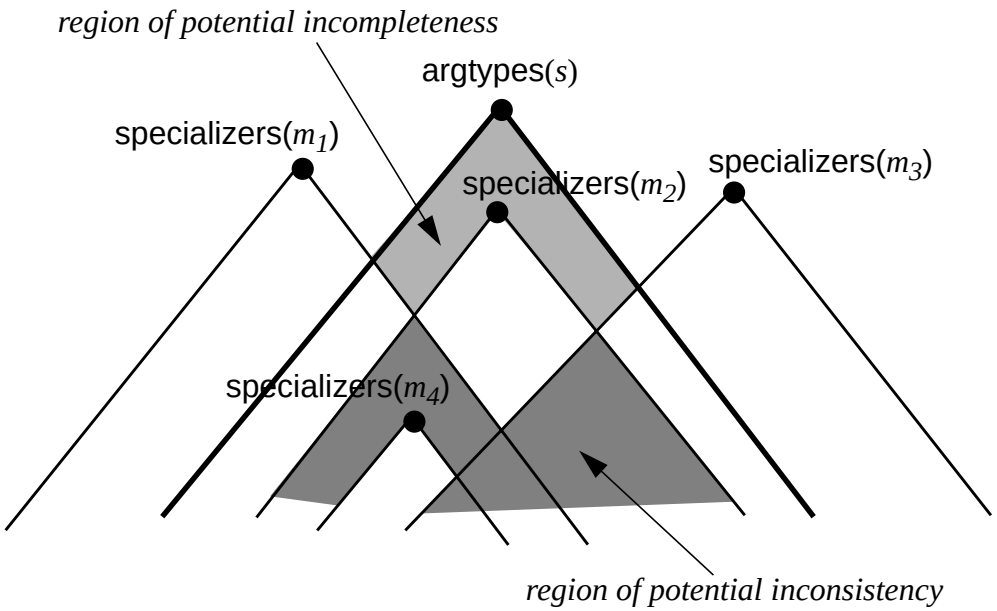
declared type of the specialized formal. By precomputing the most specific such types for each class, this check can be fast.

To show that this algorithm correctly determines whether the set of methods conforms to the set of signatures, we consider each vector of concrete classes that conforms to a signature and show that for each of the methods inherited by this vector, the vector of concrete classes conforms to the declared argument types of the method and the declared result type of the method is a subtype of the signature's declared result type. The algorithm directly ensures that this property is satisfied with respect to result types. Checking conformance of each actual concrete argument class against the corresponding type of the method's formal is more subtle. If the corresponding formal is unspecified, then the algorithm's contravariance check verifies that the formal's type is a supertype of the corresponding type of the signature, thereby ensuring that the actual argument class will conform to the declared type of the formal. If the formal is specialized, then the class of the actual must be a descendant of the specializing class (otherwise the method would not be applicable and not be considered covered by the signature). The algorithm has calculated the most specific set of types to which all subclasses of the specializer conform and verified that these types are subtypes of the formal's declared type. Since the actual class must be a subclass of the specializer, it must conform to one of the types in the set, and hence it must conform to the declared type of the formal. This completes the correctness argument.

The algorithmic complexity of conformance checking of a program is proportional to the number of methods and the larger of the number of signatures and the number of types.

4.5 Checking Completeness and Consistency

Completeness and consistency checking forms the heart of our algorithm. The following "mountain top" diagram illustrates the key issues:



The diagram divides up regions of the space of vectors of classes, with one vector plotted below another if the first overrides (i.e., its elements inherit from the elements of) the second. We have drawn cones below certain points in this space, modeling the set of vectors that 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 argtypes(*s*) corresponds to the most general vector

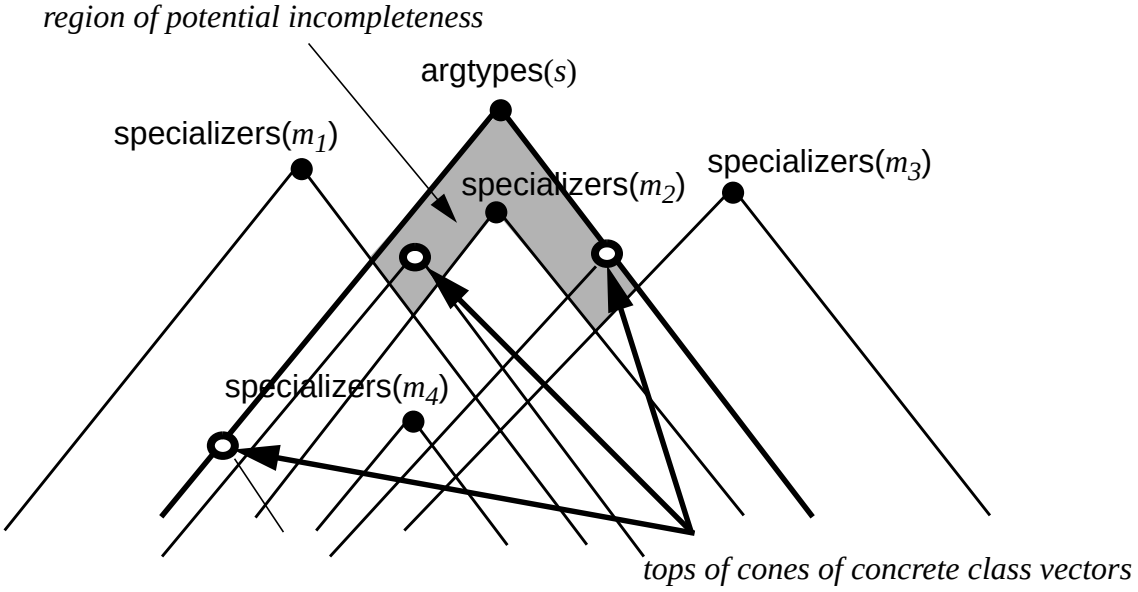
of classes that conforms to the argument types of the signature being checked.* The cone below this vector represents all class vectors that conform to the signature's argument types; the vectors in this cone are of interest because they are exactly the vectors that can be arguments of a message matching the signature. Four other class vectors represent 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" picture, the meaning of completeness and consistency can be made clear. A set of methods is complete with respect to a 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 whether there exist any vectors of concrete classes in either the incomplete or the inconsistent regions.

Our completeness and consistency checking algorithm iterates over all signatures, for each signature verifying completeness and consistency of the set of method implementations with names that match the signature. This algorithm can be performed in polynomial time, as shown in Appendix 4.

4.5.1 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. For each member of this set, we verify that there exists a method inherited by this vector. The following diagram illustrates the check, by showing with open circles the tops of three concrete class vectors that conform to the argument types of a signature. The two tops in the region of potential incompleteness are flagged as errors.



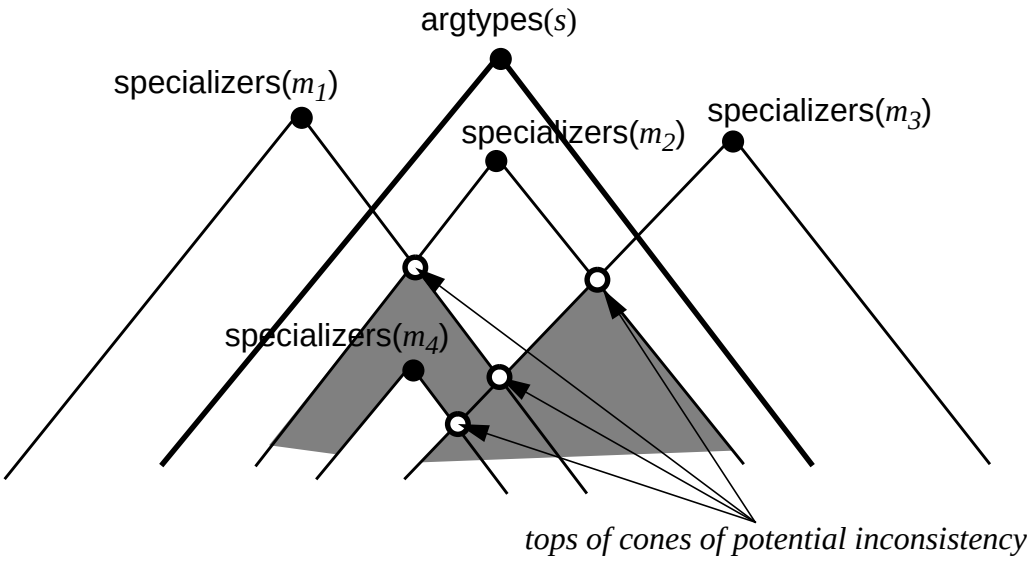
* For simplicity in the diagram we are assuming that there is one vector of classes that corresponds to the argument types of the signature. In general that may not be true, and our algorithm does not depend on this assumption.

This algorithm requires time proportional to the number of methods and the number of concrete classes, for every signature in the program. In practice, we believe this algorithm can be sped up by checking all signatures with the same name in a single pass.

To show that this algorithm correctly determines whether a set of method implementations is complete 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.

4.5.2 Checking Consistency

To check the consistency of a set of method implementations with respect to a signature, we need show the absence of any regions of potential incompleteness 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 one method does not override 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 “mountain top” 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 incompleteness is related to the problem of determining the absence of concrete class vectors in a region of potential incompleteness. Accordingly, our algorithm first constructs a new set of methods comprised of those methods in the original set that override both of the two mutually ambiguous 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.

The complexity of this check is a polynomial function of the number of methods and classes in the program, to be performed for each signature. As with completeness checking, we suspect that checking all signatures with the same name in one pass will lead to faster typechecking in practice. The real cost of this check depends on the kinds of inheritance structures that occur in practice. We expect that for most kinds of program structures, the time required to verify consistency will be acceptable. Modules as described in the next section will serve to further reduce the time required for typechecking.

To show that our algorithm correctly determines whether a set of methods is consistent 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.

4.6 Discussion

The independence of inheritance and subtyping has a major impact on our algorithm. In conformance checking, our algorithm must explicitly calculate the set of most specific types to which a class conforms, to ensure that methods are not inherited by classes that do not conform to the types of specialized formals. If inheritance and subtyping were the joined, then the set would be just the specializing class and the check of specialized formals would be trivial. These two degrees of complexity also appear in singly-dispatched languages, where languages that link subtyping and inheritance make no check of the implicit receiver argument, while languages that separate the two require additional checking in subclasses or place restrictions on inheritance to ensure that subclasses do not misuse inherited methods [Bruce *et al.* 93].

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 interesting classes (typically classes that inherit from some particular class such as a method argument's specializer class) 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.

Our programming language model distinguishes abstract and concrete classes. This distinction shows up 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 180 abstract classes, and virtually all of them would be rejected as incompletely implemented if our algorithm did not treat them specially.

We allow each multi-method to independently decide 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. Mixed specialized and unspecialized formals is fairly easy to accommodate in our algorithm. Unspecialized formals are modelled as specialized on a top class that is a superclass of all other classes. During conformance checking, unspecialized formals are checked against signatures using normal contravariant rules, while specialized formals can be 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 class are included into a program, it is possible for incompleteness or inconsistency to result. 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 existing 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 existing 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 bounded 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 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.

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

* 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.

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;
   signature new_complex(x:real, y:real):complex;
   method new_complex(x:real, y:real):complex {...}
}
module Main {
   private import Complex;
   method main():void {
      let c := new_complex(3, 4);
      ...}
}
```

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. 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.

To typecheck a program, each module in the program is typechecked. 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 and thereby

* 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++.

avoid rechecking any parts of the imported interface that are not affected its use in the importing module, further speeding typechecking.

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;
   class cartesian conforms complex;
   private field x(c@:cartesian):real;
   private field y(c@:cartesian):real;
   method +(c1@cartesian:complex, c2@:cartesian:complex):cartesian {
      new_cart_complex(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 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),

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

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 inherits or subtypes from a class or 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 exact 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 in 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 subtypes and subclasses can only be defined in the same module or in an extension module, fixes the problem of objects outrunning their methods and preserves 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 incompletely or inconsistently. For example, consider writing a `PolarComplex` module with a different representation for complex numbers:

```
module PolarComplex extends Complex {
   class 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` or `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 the `+` 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 (this set is defined as those transitively reachable through `import` declarations from the `Main` module): for each module *m* in the program, there must exist a *single most-extending module n*. Formally,

ProgramIsWellFormed ≡
$\forall\, m \in Program.$
$\exists\, n \in Program.$
$n \leq_{module} m \;\wedge$
$\forall\, n' \in Program.\; n' \leq_{module} m \;\Rightarrow\; n \leq_{module} n'$

where $\leq_{module}$ is the reflexive, transitive closure of the `extends` relation.

Such a most extending module will import all other extensions, and consequently that module will be responsible for resolving any ambiguities between other independently-developed 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`. So when the programmer combines the two independent 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 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 developed a polynomial-time algorithm for statically typechecking multi-methods. This algorithm supports 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 into our algorithm is to divide up the space of concrete class vectors conforming to a signature into cone-shaped regions, such that 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 feature of our design is extension 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 subclassing 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. At least two issues remain open: 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. To date, over 30,000 lines of Cecil code has been written, in a version of the language lacking modules or static type checking, and we expect that this code base will be an effective test of our design.

Acknowledgments

We thank Piaw Na for discussions about the typechecking algorithm and its complexity. Thanks to William Cook for discussions about the modularity problems of multi-methods. We thank Jens Palsberg, Tim Wahls, David Fernandez-Baca, Jeffrey Dean, David Grove, and Charles Garrett for their helpful comments on earlier drafts.

Chambers's work is supported in part by a National Science Foundation Research Initiation Award (contract number CCR-9210990) and several gifts from Sun Microsystems. Leavens's work is supported in part by a National Science Foundation grant (contract number CCR-9108654).

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.

[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. In *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. Mutagh, 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.

[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 & Wegner 85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. In *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.

[Caseau 93] Yves Caseau. Efficient Handling of Multiple Inheritance Hierarchies. In *OOPSLA '93 Conference Proceedings*, pp. 271-287, Washington, D.C., September, 1993. Published as *SIGPLAN Notices 28(10)*, October, 1993.

[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.

[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.

[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. In *IEEE Transactions on Software Engineering 10(5)*, pp. 528-543, September, 1984.

[Gries 91] David Gries. Teaching Calculation and Discrimination: A More Effective Curriculum. In *Communications of the ACM 34(3)*, pp. 44-55, March, 1991.

[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. In *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. In *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.

[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 Formal Programming Model

This appendix formalizes the programming model discussed in Section 3.

We assume a finite set of classes, C, a set $C_{concrete}$ of concrete classes which is a subset of C, and an associated binary relation direct-inherits on C modeling direct inheritance of implementation between classes. We extend this to a relation $\leq_{inh}$ on C by defining $\leq_{inh}$ as the reflexive, transitive closure of direct-inherits; we require $\leq_{inh}$ to be a partial order. We further assume that there exists a greatest element *top* of the $\leq_{inh}$ partial order.

We assume a finite set of types, T, and an associated downward semi-lattice* $\leq_{sub}$ 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. We further assume that the following relationship between subtyping and the g.l.b. of two types holds:

$$\forall\, t, t_1, t_2 \in T.\quad t \leq_{sub} t_1 \;\wedge\; t \leq_{sub} t_2 \;\Leftrightarrow\; t \leq_{sub} \mathsf{glb}(t_1, t_2)$$

We assume a function direct-conforms: $C \to T$, which for each class gives the most specific type to which the class directly conforms. We derive the full conformance relation between classes and types, $<:$, from direct-conforms and the subtyping relation $\leq_{sub}$ as follows:

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

Because subtyping and inheritance do not necessarily coincide, if $c' \leq_{inh} c$ and $c <: t$, one *cannot* conclude that $c' <: t$.

We extend the $\leq_{inh}$, $\leq_{sub}$, and $<:$ relations to equal-length vectors of classes and types, pointwise; i.e., for a relation $\leq$:

$$\vec{p} \leq \vec{q} \quad\equiv\quad \forall\, i.\, p_i \leq q_i.$$

We assume a finite set of message keys, *MessageKey*.

We assume a finite set of method implementations, M, with the following accessor functions:

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

We derive a partial ordering on methods, $\leq_{meth}$, modelling the method overriding relationship, from the methods' specializers as follows:

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

We assume a finite set of signatures, S, with the following accessor functions:

- A function msg: $S \longrightarrow$ *MessageKey*, such that $\mathsf{msg}(s) = \mu$ is the message handled by s.
- A function 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 restype: $S \longrightarrow T$, such that $\mathsf{restype}(s) = t$ is result type of signature s.

* A downward semi-lattice is a partial order where every pair of elements has a single greatest lower bound in the order.

Appendix B Formal Details of the Typechecking Algorithm

This appendix formalizes the typechecking algorithm presented in Section 4 and analyzes its complexity. In our complexity analyses, we assume that basic operations over the subtyping and inheritance partial orders 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]. We also assume that the maximum number of arguments to a message is bounded by a constant (does not grow with the size of the program).

B.1 Client-Side Typechecking

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 the g.l.b. of $\{\, \mathsf{restype}(s) \mid s \in S_{match} \,\}$.*

B.2 Specification of Implementation-Side Typechecking

The implementation-side typechecking problem can be specified formally as follows:

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

where

$$\mathsf{applicable\text{-}methods}(s, \grave{c}) \equiv \{\, m \in M \mid \mathsf{msg}(m) = \mathsf{msg}(s) \wedge \grave{c} \leq_{inh} \mathsf{specializers}(m) \,\}$$

$$m = \mathsf{glb}(M) \equiv m \in M \ \wedge\ \forall\, m' \in M.\ m \leq_{meth} m'$$

(In all our specifications, we assume that the sets and relations defined in Appendix 3 are globally available.)

Simply translating this specification into an isomorphic algorithm would lead to an algorithm with execution time equal to $O(|S| \cdot k^{|Cconcrete|} \cdot |M|)$, where k is the maximum number of arguments of any message in the program. Such an exponential algorithm is unacceptable in a practical system.

B.3 Overview of the Algorithm

Our algorithm breaks down the implementation-side typechecking problem into three pieces:

- conformance checking, specified by ImplementationIsConforming and implemented using the algorithm ComputeIsConforming,
- completeness checking, specified by ImplementationIsComplete and implemented using the algorithm ComputeIsComplete, and
- conformance checking, specified by ImplementationIsConsistent and implemented using the algorithm ComputeIsConsistent.

* The g.l.b. is used, rather than the l.u.b., because clients can assume the information promised by every matching signature, including their result types. Signatures are not selected at run-time, in contrast to method implementations. Implementation-side checking will ensure that each signature's guarantee is fulfilled.

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

Proof: See appendix C.1.

B.4 Checking Conformance

Given a signature, the set of methods which must conform to the signature are those whose argument specializers are ancestors of some concrete classes that conform to the argument types of the signature; this set is called the *covered set* of the signature. An entire implementation is conforming if for each signature, each of the methods in the signature's covered set conforms to the signature. Formally, we can specify conformance as follows:

ImplementationIsConforming $\equiv$
$\forall\ s \in S.\ \ \forall\ \vec{c} \in (C_{concrete})^*.$
$\vec{c}$ <: argtypes(s) $\Rightarrow$
$\forall\ m \in$ applicable-methods(s, $\vec{c}$).
$\vec{c}$ <: argtypes(m) $\wedge$
restype(m) $\leq_{sub}$ restype(s)

We divide implementing this specification into two pieces: a check of each method's specialized formals ensuring that all classes that inherit from a formal's specializer also conform to formal's declared type, and a check of each method's unspecialized formals and result types against all signatures that cover it, applying contravariance-like rules. Formally:

ComputeIsConforming $\equiv$
SpecializersAreConforming $\wedge$ ConformsToSignatures

SpecializersAreConforming $\equiv$
$\forall\ m \in M.\ \ \forall\ i \in$ indexes(specializers(m)).
let c = specializers(m)$_i$ **in**
$c \neq top\ \Rightarrow$
($\forall\ t \in$ conformed-types(c). $t \leq_{sub}$ argtypes(m)$_i$)

ConformsToSignatures $\equiv$
$\forall\ s \in S.\ \ \forall\ m \in$ relevant(M, s).
$m \in$ covered-methods(s) $\Rightarrow$ conforms(m, s)

where

conformed-types(c) $\equiv$
let $Cs = \{\ c' \mid \exists\ c'.\ \ c' \leq_{inh} c\ \wedge\ (c' \neq c\ \Rightarrow\ \neg$ is-parallel-type-and-class(c')) $\}$,
$Ts = \{\ t \mid \exists\ c' \in Cs.\ \ t =$ direct-conforms(c') $\}$ **in**
tops(Ts)

is-parallel-type-and-class(c) $\equiv$
$\forall\ c'.\ \ c$ direct-inherits $c'\ \wedge\ c$ <: direct-conforms(c')

tops(T) $\equiv \{\ t \in T \mid t' \in T \wedge t' \neq t \Rightarrow \neg\ (t \leq_{sub} t')\ \}$

relevant(M, s) $\equiv \{\ m \in M \mid$ msg(m) = msg(s) $\}$

covered-methods(s) $\equiv$
  $\{ m \in$ relevant(M, s) $\mid$ has-common-classes(specializers(m), argtypes(s)) $\}$

has-common-classes($\grave{c}$, $\grave{t}$) $\equiv \exists\, \grave{c}\,' \in (C_{concrete})^*.\ \grave{c}\,' \leq_{inh} \grave{c} \wedge \grave{c}\,' <: \grave{t}.$

conforms(m, s) $\equiv$
  $(\forall\, i \in$ indexes(specializers(m)).
    specializers(m)$_i$ = *top* $\Rightarrow$ argtypes(s)$_i \leq_{sub}$ argtypes(m)$_i$) $\wedge$
  restype(m) $\leq_{sub}$ restype(s)

Theorem 2. ComputeIsConforming $\Rightarrow$ ImplementationIsConforming

Proof: See appendix C.2.

Complexity. We precompute the set of tops of the specific types to which a class and its subclasses conform (conformed-types) for every class. This precomputation can use breadth first search on the direct-inherits relation, making the worst case time $O(|C| + |\text{direct-inherits}|)$. Each of these sets is of size $O(|T|)$ at worst. Then to check conformance for the specialized arguments of all methods (SpecializersAreConforming) requires time $O(|M| \cdot |T|)$. The time complexity of checking conformance of relevant methods against all signatures (ConformsToSignatures) is $O(|M| \cdot |S|)$. Therefore, the overall time required to check conformance of a program is $O(|M| \cdot (|S| + |T|))$.

B.5 Checking Completeness

Formally, completeness of a program can be specified as follows:

ImplementationIsComplete $\equiv$
  $\forall\, s \in S.\ \neg \exists\, \grave{c} \in (C_{concrete})^*.$
    $\grave{c} <:$ argtypes(s) $\wedge$
    $\neg \exists\, m \in$ relevant(M, s). $\grave{c} \leq_{inh}$ specializers(m).

Our algorithm recasts this problem as follows:

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

To check the completeness of a set of method implementations M with respect to a set of class vectors Cs and a signature s, this 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.

IsComplete(M, Cs, s) $\equiv$
  $\forall\, \grave{c} \in Cs.$
    let $TCSs = \{ \grave{c}\,' \mid c_i' \in$ top-concrete-subclasses(c_i, argtypes(s)$_i$) $\}$ **in**
      $\forall\, \grave{c}\,' \in TCSs.\ \exists\, m \in M.\ \grave{c}\,' \leq_{inh}$ specializers(m)

where

top-concrete-subclasses(c, t) $\equiv$ tops(concrete-subclasses(c, t))

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

tops(C) $\equiv \{ c \in C \mid c' \in C \wedge c' \neq c \Rightarrow \neg (c \leq_{inh} c') \}$

Theorem 3. ComputeIsComplete ⇒ ImplementationIsComplete

Proof: See appendix C.3.

Complexity. We precompute the partial order of concrete subclasses of every class prior to typechecking. With this set-up, the time complexity of computing top-concrete-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 $O(|M| \cdot |Cs| \cdot |C_{concrete}|)$.

B.6 Checking Consistency

Formally, consistency of a program can be specified as follows:

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

Our algorithm recasts this problem as follows:

$$\begin{array}{l}\text{ComputeIsConsistent} \equiv \\ \quad \forall\, s \in S.\ \text{IsConsistent}(\text{relevant}(M, s), s)\end{array}$$

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-reduced* that override both m_1 and m_2. Finally, we test for completeness of *M-reduced* with respect to *TLBs* and s.

$$\begin{array}{l}\text{IsConsistent}(M, s) \equiv \\ \quad \forall\, (m_1, m_2) \in \text{incomparable-pairs}(M). \\ \qquad \textbf{let}\ \ TLBs = \text{tlb}(\text{specializers}(m_1), \text{specializers}(m_2), s), \\ \qquad\quad M\text{-reduced} = \{\, m \in M \mid m \leq_{meth} m_1 \wedge m \leq_{meth} m_2 \,\}\ \textbf{in} \\ \qquad\qquad \text{IsComplete}(M\text{-reduced}, TLBs, s)\end{array}$$

where

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

$$\text{tlb}(\grave{c}, \grave{c}', s) \equiv \text{tops}(\text{lb}(\grave{c}, \grave{c}', s))$$

$$\text{lb}(\grave{c}, \grave{c}', s) \equiv \{\, \grave{c}'' \mid \grave{c}'' \leq_{inh} \grave{c} \wedge \grave{c}'' \leq_{inh} \grave{c}' \wedge \grave{c}'' <: \text{argtypes}(s) \,\}$$

$$\text{tops}(Cs) \equiv \{\, \grave{c} \in Cs \mid \grave{c}' \in Cs \wedge \grave{c}' \neq \grave{c} \Rightarrow \neg\, (\grave{c} \leq_{inh} \grave{c}') \,\}$$

Theorem 4. ComputeIsConsistent ⇒ ImplementationIsConsistent

Proof: See appendix C.4.

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

* 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 "great lower bounds," i.e., those lower bounds that are not strictly less than any other lower bound.

$O(|C^k|)$ 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 extremely 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-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 $O(|M|^3 \cdot (|C^k| + |M|) \cdot |C^k| \cdot |C_{concrete}|)$, although $O(|M|^4 \cdot |C_{concrete}|)$ is a more likely worst-case time in practice. Note that *M* here refers to the subset of methods that have the same name as the signature being checked, not all methods in the program.

B.7 Complexity Summary

Checking conformance of all method implementations against all signatures requires worst-case time of $O(|M| \cdot (|S| + |T|))$. Checking completeness of all method implementations against all signatures requires $O(|M| \cdot |S| \cdot |C_{concrete}|)$. Checking consistency of all method implementations against all signatures requires $O(|M|^3 \cdot |S| \cdot (|C^k| + |M|) \cdot |C^k| \cdot |C_{concrete}|)$ (where k, the maximum number of message arguments, is bounded by a constant) in contrived worst-case scenarios and $O(|M|^4 \cdot |S| \cdot |C_{concrete}|)$ in somewhat more realistic situations. Consistency checking dominates the overall complexity of implementation-side typechecking.

Appendix C Correctness Theorems and Proofs

This appendix gives formal correctness proofs for our typechecking algorithm, using the specifications defined in Appendix B. The proofs use the calculational format described in [Gries 91].

C.1 Problem Breakdown

This theorem shows that our conformance, completeness, and consistency checks are sufficient to guarantee that an implementation type checks.

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 definition$\rangle$

$\forall\, s \in S.\ \neg\, \exists\, \grave{c} \in (C_{concrete})^*.$
  $\grave{c} <:$ argtypes(s) $\wedge\ \neg\, \exists\, m \in$ relevant(M, s). $\grave{c} \leq_{inh}$ specializers(m)
$\wedge\ \forall\, s \in S.\ \neg\, \exists\, \grave{c} \in (C_{concrete})^*.$
  $\grave{c} <:$ argtypes(s) $\wedge$
  $\exists\, m_1, m_2 \in$ relevant(M, s).
    $\grave{c} \leq_{inh}$ specializers(m_1) $\wedge$ $\grave{c} \leq_{inh}$ specializers(m_2) $\wedge$
    $\neg\, \exists\, m \in$ relevant(M, s). $\grave{c} \leq_{inh}$ specializers(m) $\wedge$ $m \leq_{meth} m_1$ $\wedge$ $m \leq_{meth} m_2$
$\wedge\ \forall\, s \in S.\ \forall\, \grave{c} \in (C_{concrete})^*.$
  $\grave{c} <:$ argtypes(s) $\Rightarrow$
    $\forall\, m \in$ applicable-methods(s, $\grave{c}$).
      $\grave{c} <:$ argtypes(m) $\wedge$ restype(m) $\leq_{sub}$ restype(s)

$\Leftrightarrow$ $\langle$by $\neg\, \exists\, x.\ P(x) \Leftrightarrow \forall x.\ \neg P(x)$, twice; predicate calculus and definition of implication, twice$\rangle$

$\forall\, s \in S.\ \forall\, \grave{c} \in (C_{concrete})^*.$
  $\grave{c} <:$ argtypes(s) $\Rightarrow \exists\, m \in$ relevant(M, s). $\grave{c} \leq_{inh}$ specializers(m)
$\wedge\ \forall\, s \in S.\ \forall\, \grave{c} \in (C_{concrete})^*.$
  $\grave{c} <:$ argtypes(s) $\Rightarrow$
    $\neg\, \exists\, m_1, m_2 \in$ relevant(M, s).
      $\grave{c} \leq_{inh}$ specializers(m_1) $\wedge$ $\grave{c} \leq_{inh}$ specializers(m_2) $\wedge$
      $\neg\, \exists\, m \in$ relevant(M, s). $\grave{c} \leq_{inh}$ specializers(m) $\wedge$ $m \leq_{meth} m_1$ $\wedge$ $m \leq_{meth} m_2$
$\wedge\ \forall\, s \in S.\ \forall\, \grave{c} \in (C_{concrete})^*.$
  $\grave{c} <:$ argtypes(s) $\Rightarrow$
    $\forall\, m \in$ applicable-methods(s, $\grave{c}$).
      $\grave{c} <:$ argtypes(m) $\wedge$ restype(m) $\leq_{sub}$ restype(s)

$\Leftrightarrow$ ⟨by $(\forall x.P(x)) \wedge (\forall x.Q(x)) \Leftrightarrow \forall x.(P(x) \wedge Q(x))$, 4 times; and $((P \Rightarrow Q) \wedge (P \Rightarrow R)) \Leftrightarrow (P \Rightarrow (Q \wedge R))$, twice⟩

$\forall\, s \in S.\ \forall\, \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \text{argtypes}(s) \Rightarrow$
$((\exists\, m \in \text{relevant}(M, s).\ \grave{c} \leq_{inh} \text{specializers}(m))$
$\wedge\ (\neg\, \exists\, m_1, m_2 \in \text{relevant}(M, s).$
$\grave{c} \leq_{inh} \text{specializers}(m_1) \wedge \grave{c} \leq_{inh} \text{specializers}(m_2) \wedge$
$\neg\, \exists\, m \in \text{relevant}(M, s).\ \grave{c} \leq_{inh} \text{specializers}(m) \wedge m \leq_{meth} m_1 \wedge m \leq_{meth} m_2)$
$\wedge\ (\forall\, m \in \text{applicable-methods}(s, \grave{c}).$
$\grave{c} <: \text{argtypes}(m) \wedge \text{restype}(m) \leq_{sub} \text{restype}(s)))$

$\Leftrightarrow$ ⟨by $\neg\exists x.P(x) \Leftrightarrow \forall x.\neg P(x)$ and definition of $\Rightarrow$⟩

$\forall\, s \in S.\ \forall\, \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \text{argtypes}(s) \Rightarrow$
$((\exists\, m \in \text{relevant}(M, s).\ \grave{c} \leq_{inh} \text{specializers}(m))$
$\wedge\ (\forall\, m_1, m_2 \in \text{relevant}(M, s).$
$\grave{c} \leq_{inh} \text{specializers}(m_1) \wedge \grave{c} \leq_{inh} \text{specializers}(m_2) \Rightarrow$
$\exists\, m \in \text{relevant}(M, s).\ \grave{c} \leq_{inh} \text{specializers}(m) \wedge m \leq_{meth} m_1 \wedge m \leq_{meth} m_2)$
$\wedge\ (\forall\, m \in \text{applicable-methods}(s, \grave{c}).$
$\grave{c} <: \text{argtypes}(m) \wedge \text{restype}(m) \leq_{sub} \text{restype}(s)))$

$\Rightarrow$ ⟨by instantiation of the quantifier $\forall\, m_2 \in \text{relevant}(M, s)$ to m_1, renaming m_1 to m', and predicate calculus⟩

$\forall\, s \in S.\ \forall\, \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \text{argtypes}(s) \Rightarrow$
$((\exists\, m \in \text{relevant}(M, s).\ \grave{c} \leq_{inh} \text{specializers}(m))$
$\wedge\ (\forall\, m' \in \text{relevant}(M, s).\ \grave{c} \leq_{inh} \text{specializers}(m') \Rightarrow$
$\exists\, m \in \text{relevant}(M, s).\ \grave{c} \leq_{inh} \text{specializers}(m) \wedge m \leq_{meth} m')$
$\wedge\ (\forall\, m \in \text{applicable-methods}(s, \grave{c}).$
$\grave{c} <: \text{argtypes}(m) \wedge \text{restype}(m) \leq_{sub} \text{restype}(s)))$

$\Leftrightarrow$ ⟨by definition of $\text{glb}(\{\, m \in \text{relevant}(M, s) \mid \grave{c} \leq_{inh} \text{specializers}(m) \,\})$⟩

$\forall\, s \in S.\ \forall\, \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \text{argtypes}(s) \Rightarrow$
$((\exists\, m \in \text{relevant}(M, s).\ \grave{c} \leq_{inh} \text{specializers}(m))$
$\wedge\ (\exists\, m.\ m = \text{glb}(\{\, m \in \text{relevant}(M, s) \mid \grave{c} \leq_{inh} \text{specializers}(m) \,\}))$
$\wedge\ (\forall\, m \in \text{applicable-methods}(s, \grave{c}).$
$\grave{c} <: \text{argtypes}(m) \wedge \text{restype}(m) \leq_{sub} \text{restype}(s)))$

$\Rightarrow$ ⟨by $P \wedge Q \Rightarrow Q$⟩

$\forall\, s \in S.\ \forall\, \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \text{argtypes}(s) \Rightarrow$
$(\exists\, m.\ m = \text{glb}(\{\, m \in \text{relevant}(M, s) \mid \grave{c} \leq_{inh} \text{specializers}(m) \,\}))$
$\wedge\ (\forall\, m \in \text{applicable-methods}(s, \grave{c}).$
$\grave{c} <: \text{argtypes}(m) \wedge \text{restype}(m) \leq_{sub} \text{restype}(s)))$

$\Leftrightarrow$ ⟨by definition of relevant⟩
$\forall\, s \in S.\ \forall\, \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \text{argtypes}(s) \Rightarrow$
$((\exists\, m.\ m = \text{glb}(\{\ m \in M \mid \text{msg}(m) = \text{msg}(s) \wedge \grave{c} \leq_{inh} \text{specializers}(m)\ \}))$
$\wedge\ (\forall\, m \in \text{applicable-methods}(s, \grave{c}).$
$\grave{c} <: \text{argtypes}(m)\ \wedge \text{restype}(m) \leq_{sub} \text{restype}(s)))$

$\Leftrightarrow$ ⟨by definition of applicable-methods⟩
$\forall\, s \in S.\ \forall\, \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \text{argtypes}(s) \Rightarrow$
$((\exists\, m.\ m = \text{glb}(\text{applicable-methods}(s, \grave{c})))$
$\wedge\ (\forall\, m \in \text{applicable-methods}(s, \grave{c}).$
$\grave{c} <: \text{argtypes}(m)\ \wedge \text{restype}(m) \leq_{sub} \text{restype}(s)))$

$\Rightarrow$ ⟨by instantiation of the quantifier $\forall\, m \in$ applicable-methods$(s, \grave{c})$ to the glb⟩
$\forall\, s \in S.\ \forall\, \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \text{argtypes}(s) \Rightarrow$
$\exists\, m.\ m = \text{glb}(\text{applicable-methods}(s, \grave{c}))$
$\wedge\ \grave{c} <: \text{argtypes}(m)\ \wedge \text{restype}(m) \leq_{sub} \text{restype}(s)$

$\Leftrightarrow$ ⟨by definition⟩
ImplementationTypechecks

C.2 Correctness of Conformance Checking Algorithm

Theorem 2 below says that our algorithm for computing completeness is sufficient to ensure completeness. To prove this theorem we first prove the following lemma. This lemma handles the case of specialized arguments as checked by our algorithm.

Lemma 1. Let *c* and *c'* be classes. Then

$c \leq_{inh} c' \ \Rightarrow\ \exists\, t \in \text{conformed-types}(c').\ \text{direct-conforms}(c) \leq_{sub} t$

Proof: We proceed by induction on the length of the (shortest) chain of inheritance relationships that connects *c* to *c'* to show that the last formula above is implied by $c \leq_{inh} c'$.

For the basis, suppose $c = c'$. Then we can calculate as follows.

$c = c'$

$\Rightarrow$ ⟨by Leibnitz's rule⟩
direct-conforms(*c*) = direct-conforms(*c'*)

$\Rightarrow$ ⟨by reflexivity of $\leq_{sub}$⟩
direct-conforms(*c*) $\leq_{sub}$ direct-conforms(*c'*)

$\Leftrightarrow$ ⟨by set theory⟩
$\exists\, t \in \{$ direct-conforms(*c'*) $\}.$ direct-conforms(*c*) $\leq_{sub} t$

$\Leftrightarrow$ ⟨by set theory⟩
$\exists\, t \in \{\ t \mid t =$ direct-conforms(*c'*) $\}.$ direct-conforms(*c*) $\leq_{sub} t$

$\Leftrightarrow$ ⟨by reflexivity of $\leq_{inh}$ and $P \Rightarrow P \wedge true$⟩
$\exists\, t \in \{\ t \mid c' \leq_{inh} c'\ \wedge\ t =$ direct-conforms(*c'*) $\}.$
direct-conforms(*c*) $\leq_{sub} t$

$\Leftrightarrow \langle$by $P \Leftrightarrow (P \wedge (\mathit{false} \Rightarrow Q))\rangle$
$\exists\, t \in \{\, t \mid c' \leq_{inh} c' \wedge (c' \neq c' \Rightarrow \neg$ is-parallel-type-and-class(c')) $\wedge\ t =$ direct-conforms(c') $\}$.
direct-conforms(c) $\leq_{sub} t$

$\Rightarrow \langle$by $P(x) \Rightarrow \exists x'.\ P(x')\rangle$
$\exists\, t \in \{\, t \mid \exists\, c''.\ \ c'' \leq_{inh} c' \wedge (c'' \neq c' \Rightarrow \neg$ is-parallel-type-and-class(c'')) $\wedge$
$t =$ direct-conforms(c'') $\}$.
direct-conforms(c) $\leq_{sub} t$

$\Leftrightarrow \langle$by definition of tops and $\leq_{sub}\rangle$
$\exists\, t \in$ tops($\{\, t \mid \exists\, c''.\ \ c'' \leq_{inh} c' \wedge (c'' \neq c' \Rightarrow \neg$ is-parallel-type-and-class(c'')) $\wedge$
$t =$ direct-conforms(c'') $\}$).
direct-conforms(c) $\leq_{sub} t$

$\Leftrightarrow \langle$by definition of conformed-types(c')$\rangle$
$\exists\, t \in$ conformed-types(c'). direct-conforms(c) $\leq_{sub} t$

For the inductive case, suppose that $\exists\, c''.\ c$ direct-inherits $c'' \wedge c'' \leq_{inh} c'$. The inductive assumption is that the lemma holds for all shorter inheritance chains, such as $c'' \leq_{inh} c'$. We calculate as follows.

$\exists\, c''.\ c$ direct-inherits $c'' \wedge c'' \leq_{inh} c'$

$\Leftrightarrow \langle$by law of excluded middle$\rangle$
($\exists\, c''.\ c$ direct-inherits $c'' \wedge c'' \leq_{inh} c' \wedge$ is-parallel-type-and-class(c))
$\vee$ ($\exists\, c''.\ c$ direct-inherits $c'' \wedge c'' \leq_{inh} c' \wedge \neg$ (is-parallel-type-and-class(c)))

$\Leftrightarrow \langle$by definition of is-parallel-type-and-class(c)$\rangle$
($\exists\, c''.\ c$ direct-inherits $c'' \wedge c'' \leq_{inh} c'$
$\wedge$ ($\forall\, c'''.\ c$ direct-inherits $c''' \wedge c <:$ direct-conforms(c''')))
$\vee$ ($\exists\, c''.\ c$ direct-inherits $c'' \wedge\ c'' \leq_{inh} c' \wedge \neg$ (is-parallel-type-and-class(c)))

$\Rightarrow \langle$by instantiating c''' to c'' and idempotency of $\wedge\rangle$
($\exists\, c''.\ c$ direct-inherits $c'' \wedge\ c'' \leq_{inh} c' \wedge c <:$ direct-conforms(c''))
$\vee$ ($\exists\, c''.\ c$ direct-inherits $c'' \wedge\ c'' \leq_{inh} c' \wedge \neg$ (is-parallel-type-and-class(c)))

$\Rightarrow \langle$by the inductive hypothesis, with c'' for $c\rangle$
($\exists\, c''.\ c$ direct-inherits $c'' \wedge\ c'' \leq_{inh} c' \wedge c <:$ direct-conforms(c'')
$\wedge$ ($\exists\, t \in$ conformed-types(c'). direct-conforms(c'') $\leq_{sub} t$))
$\vee$ ($\exists\, c''.\ c$ direct-inherits $c'' \wedge\ c'' \leq_{inh} c' \wedge \neg$ (is-parallel-type-and-class(c)))

$\Leftrightarrow \langle$by definition of $<:\rangle$
($\exists\, c''.\ c$ direct-inherits $c'' \wedge\ c'' \leq_{inh} c' \wedge$ direct-conforms(c) $\leq_{sub}$ direct-conforms(c'')
$\wedge$ ($\exists\, t \in$ conformed-types(c'). direct-conforms(c'') $\leq_{sub} t$))
$\vee$ ($\exists\, c''.\ c$ direct-inherits $c'' \wedge\ c'' \leq_{inh} c' \wedge \neg$ (is-parallel-type-and-class(c)))

$\Rightarrow \langle$by transitivity of $\leq_{sub}\rangle$
($\exists\, c''.\ c$ direct-inherits $c'' \wedge\ c'' \leq_{inh} c'$
$\wedge$ ($\exists\, t \in$ conformed-types(c'). direct-conforms(c) $\leq_{sub} t$))
$\vee$ ($\exists\, c''.\ c$ direct-inherits $c'' \wedge\ c'' \leq_{inh} c' \wedge \neg$ (is-parallel-type-and-class(c)))

$\Rightarrow \langle$by logic, as c'' does not occur in ($\exists\, t \in$ conformed-types(c'). direct-conforms(c) $\leq_{sub} t$)$\rangle$
($\exists\, t \in$ conformed-types(c'). direct-conforms(c) $\leq_{sub} t$)
$\vee$ ($\exists\, c''.\ c$ direct-inherits $c'' \wedge\ c'' \leq_{inh} c' \wedge \neg$ (is-parallel-type-and-class(c)))

$\Leftrightarrow$ ⟨by definition of direct-inherits and $\leq_{inh}$⟩
(∃ $t \in$ conformed-types(c'). direct-conforms(c) $\leq_{sub}$ t)
$\vee$ (∃ c". $c \leq_{inh} c' \wedge c \neq c' \wedge \neg$ (is-parallel-type-and-class(c)))

$\Leftrightarrow$ ⟨by logic, as c" does not occur in the formula above⟩
(∃ $t \in$ conformed-types(c'). direct-conforms(c) $\leq_{sub}$ t)
$\vee$ ($c \leq_{inh} c' \wedge c \neq c' \wedge \neg$ (is-parallel-type-and-class(c)))

$\Leftrightarrow$ ⟨by $(P \wedge Q) \Leftrightarrow (R \Rightarrow Q)$⟩
(∃ $t \in$ conformed-types(c'). direct-conforms(c) $\leq_{sub}$ t)
$\vee$ ($c \leq_{inh} c' \wedge$ ($c \neq c' \Rightarrow \neg$ (is-parallel-type-and-class(c))))

$\Leftrightarrow$ ⟨by reflexivity of $\leq_{sub}$, the added clause is a tautology⟩
(∃ $t \in$ conformed-types(c'). direct-conforms(c) $\leq_{sub}$ t)
$\vee$ ($c \leq_{inh} c' \wedge (c \neq c' \Rightarrow \neg$ (is-parallel-type-and-class(c)))
$\wedge$ direct-conforms(c) $\leq_{sub}$ direct-conforms(c))

$\Leftrightarrow$ ⟨by set theory⟩
(∃ $t \in$ conformed-types(c'). direct-conforms(c) $\leq_{sub}$ t)
$\vee$ (∃ $t \in \{ t \mid c \leq_{inh} c' \wedge (c \neq c' \Rightarrow \neg$ (is-parallel-type-and-class(c))) $\wedge$
$t =$ direct-conforms(c) }.
direct-conforms(c) $\leq_{sub}$ t)

$\Rightarrow$ ⟨by predicate calculus, where c satisfies the existentially quantified c"⟩
(∃ $t \in$ conformed-types(c'). direct-conforms(c) $\leq_{sub}$ t)
$\vee$ (∃ $t \in \{ t \mid \exists c''.\ c'' \leq_{inh} c' \wedge (c'' \neq c' \Rightarrow \neg$ is-parallel-type-and-class(c")) $\wedge$
$t =$ direct-conforms(c") }.
direct-conforms(c) $\leq_{sub}$ t)

$\Leftrightarrow$ ⟨by definition of tops and $\leq_{sub}$⟩
(∃ $t \in$ conformed-types(c'). direct-conforms(c) $\leq_{sub}$ t)
$\vee$ (∃ $t \in$ tops({ $t \mid \exists c''.\ c'' \leq_{inh} c' \wedge (c'' \neq c' \Rightarrow \neg$ is-parallel-type-and-class(c")) $\wedge$
$t =$ direct-conforms(c") }).
direct-conforms(c) $\leq_{sub}$ t)

$\Leftrightarrow$ ⟨by definition of conformed-types(c')⟩
(∃ $t \in$ conformed-types(c'). direct-conforms(c) $\leq_{sub}$ t)
$\vee$ (∃ $t \in$ conformed-types(c'). direct-conforms(c) $\leq_{sub}$ t)

$\Leftrightarrow$ ⟨by idempotence of $\vee$⟩
∃ $t \in$ conformed-types(c'). direct-conforms(c) $\leq_{sub}$ t

Now that the lemma is proved, we can prove the main theorem about our conformance checking algorithm.

Theorem 2. ComputeIsConforming $\Rightarrow$ ImplementationIsConforming

Proof: We prove this theorem by the following calculation.

ComputeIsConforming

$\Leftrightarrow$ ⟨by definition⟩
SpecializersAreConforming $\wedge$ ConformsToSignatures

$\Leftrightarrow \langle$by definition$\rangle$
$(\forall\ m \in M.\ \forall\ i \in \text{indexes}(\text{specializers}(m)).$
let $c = \text{specializers}(m)_i$ **in**
$c \neq top\ \Rightarrow$
$(\forall\ t \in \text{conformed-types}(c).\ t \leq_{sub} \text{argtypes}(m)_i))$
$\wedge\ (\forall\ s \in S.\ \forall\ m \in \text{relevant}(M, s).$
$m \in \text{covered-methods}(s)\ \Rightarrow\ \text{conforms}(m, s))$

$\Leftrightarrow \langle$by logic, as s does not occur in the first conjunct$\rangle$
$\forall\ s \in S.$
$(\forall\ m \in M.\ \forall\ i \in \text{indexes}(\text{specializers}(m)).$
let $c = \text{specializers}(m)_i$ **in**
$c \neq top\ \Rightarrow$
$(\forall\ t \in \text{conformed-types}(c).\ t \leq_{sub} \text{argtypes}(m)_i))$
$\wedge\ (\forall\ m \in \text{relevant}(M, s).$
$m \in \text{covered-methods}(s)\ \Rightarrow\ \text{conforms}(m, s))$

$\Rightarrow \langle$by set theory, as $\text{relevant}(M, s)$ is a subset of $M\rangle$
$\forall\ s \in S.\ \forall\ m \in \text{relevant}(M, s).$
$(\forall\ i \in \text{indexes}(\text{specializers}(m)).$
let $c = \text{specializers}(m)_i$ **in**
$c \neq top\ \Rightarrow$
$(\forall\ t \in \text{conformed-types}(c).\ t \leq_{sub} \text{argtypes}(m)_i))$
$\wedge\ (m \in \text{covered-methods}(s)\ \Rightarrow\ \text{conforms}(m, s))$

$\Leftrightarrow \langle$by definition of c in the **let**$\rangle$
$\forall\ s \in S.\ \forall\ m \in \text{relevant}(M, s).$
$(\forall\ i \in \text{indexes}(\text{specializers}(m)).$
$\text{specializers}(m)_i \neq top\ \Rightarrow$
$(\forall\ t \in \text{conformed-types}(\text{specializers}(m)_i).\ t \leq_{sub} \text{argtypes}(m)_i))$
$\wedge\ (m \in \text{covered-methods}(s)\ \Rightarrow\ \text{conforms}(m, s))$

$\Leftrightarrow \langle$by definition of $\text{covered-methods}(s)$ and $\text{conforms}(m, s)\rangle$
$\forall\ s \in S.\ \forall\ m \in \text{relevant}(M, s).$
$(\forall\ i \in \text{indexes}(\text{specializers}(m)).$
$\text{specializers}(m)_i \neq top\ \Rightarrow$
$(\forall\ t \in \text{conformed-types}(\text{specializers}(m)_i).\ t \leq_{sub} \text{argtypes}(m)_i))$
$\wedge\ (m \in \{m \in \text{relevant}(M, s) \mid \text{has-common-classes}(\text{specializers}(m), \text{argtypes}(s))\} \Rightarrow$
$((\forall i \in \text{indexes}(\text{specializers}(m)).$
$\text{specializers}(m)_i = top\ \Rightarrow\ \text{argtypes}(s)_i \leq_{sub} \text{argtypes}(m)_i)$
$\wedge\ \text{restype}(m) \leq_{sub} \text{restype}(s)))$

$\Leftrightarrow$ ⟨by set theory, as each m is already in relevant(M, s)⟩
$\forall\, s \in S.\ \forall\, m \in$ relevant(M, s).
($\forall\, i \in$ indexes(specializers(m)).
specializers(m)$_i \neq top \Rightarrow$
($\forall\, t \in$ conformed-types(specializers(m)$_i$). $t \leq_{sub}$ argtypes(m)$_i$))
$\wedge$ (has-common-classes(specializers(m), argtypes(s)) $\Rightarrow$
((∀$i \in$ indexes(specializers(m)).
specializers(m)$_i = top \Rightarrow$ argtypes(s)$_i \leq_{sub}$ argtypes(m)$_i$)
$\wedge$ restype(m) $\leq_{sub}$ restype(s)))

$\Leftrightarrow$ ⟨by logic, as i does not occur in has-common-classes(specializers(m), argtypes(s))⟩
$\forall\, s \in S.\ \forall\, m \in$ relevant(M, s).
($\forall\, i \in$ indexes(specializers(m)).
specializers(m)$_i \neq top \Rightarrow$
($\forall\, t \in$ conformed-types(specializers(m)$_i$). $t \leq_{sub}$ argtypes(m)$_i$))
$\wedge$ (∀$i \in$ indexes(specializers(m)).
has-common-classes(specializers(m), argtypes(s)) $\Rightarrow$
((specializers(m)$_i = top \Rightarrow$ argtypes(s)$_i \leq_{sub}$ argtypes(m)$_i$)
$\wedge$ restype(m) $\leq_{sub}$ restype(s)))

$\Leftrightarrow$ ⟨by $(\forall x.P(x)) \wedge (\forall x.Q(x)) \Leftrightarrow \forall x.P(x) \wedge Q(x)$⟩
$\forall\, s \in S.\ \forall\, m \in$ relevant(M, s). $\forall\, i \in$ indexes(specializers(m)).
(specializers(m)$_i \neq top \Rightarrow$
($\forall\, t \in$ conformed-types(specializers(m)$_i$). $t \leq_{sub}$ argtypes(m)$_i$))
$\wedge$ (has-common-classes(specializers(m), argtypes(s)) $\Rightarrow$
((specializers(m)$_i = top \Rightarrow$ argtypes(s)$_i \leq_{sub}$ argtypes(m)$_i$)
$\wedge$ restype(m) $\leq_{sub}$ restype(s)))

$\Leftrightarrow$ ⟨by definition of has-common-classes(specializers(m), argtypes(s))⟩
$\forall\, s \in S.\ \forall\, m \in$ relevant(M, s). $\forall\, i \in$ indexes(specializers(m)).
(specializers(m)$_i \neq top \Rightarrow$
($\forall\, t \in$ conformed-types(specializers(m)$_i$). $t \leq_{sub}$ argtypes(m)$_i$))
$\wedge$ (($\exists\, \grave{c}\,' \in (C_{concrete})^*.\ \grave{c}\,' \leq_{inh}$ specializers(m) $\wedge\ \grave{c}\,' <:$ argtypes(s)) $\Rightarrow$
((specializers(m)$_i = top \Rightarrow$ argtypes(s)$_i \leq_{sub}$ argtypes(m)$_i$)
$\wedge$ restype(m) $\leq_{sub}$ restype(s)))

$\Rightarrow$ ⟨by logic as $\grave{c}$ does not occur in the above formula, and then $Q \Rightarrow (P \Rightarrow Q)$, twice⟩
$\forall\, s \in S.\ \forall\, m \in$ relevant(M, s). $\forall\, \grave{c} \in (C_{concrete})^*.$
$\grave{c} <:$ argtypes(s) $\Rightarrow$
$\grave{c} \leq_{inh}$ specializers(m) $\Rightarrow$
$\forall\, i \in$ indexes(specializers(m)).
(specializers(m)$_i \neq top \Rightarrow$
($\forall\, t \in$ conformed-types(specializers(m)$_i$). $t \leq_{sub}$ argtypes(m)$_i$))
$\wedge$ (($\exists\, \grave{c}\,' \in (C_{concrete})^*.\ \grave{c}\,' \leq_{inh}$ specializers(m) $\wedge\ \grave{c}\,' <:$ argtypes(s)) $\Rightarrow$
((specializers(m)$_i = top \Rightarrow$ argtypes(s)$_i \leq_{sub}$ argtypes(m)$_i$)
$\wedge$ restype(m) $\leq_{sub}$ restype(s)))

$\Longrightarrow$ ⟨by assumptions about $\grave{c}$, which satisfy the hypothesis about $\grave{c}$' in the last main conjunct⟩
$\forall\ s \in S.\ \forall\ m \in$ relevant(M, s). $\forall\ \grave{c} \in (C_{concrete})^*$.
$\grave{c}$ <: argtypes(s) $\Rightarrow$
$\grave{c} \leq_{inh}$ specializers(m) $\Rightarrow$
$\forall\ i \in$ indexes(specializers(m)).
(specializers(m)$_i \neq top\ \Rightarrow$
($\forall\ t \in$ conformed-types(specializers(m)$_i$). $t \leq_{sub}$ argtypes(m)$_i$))
$\wedge$ (specializers(m)$_i = top\ \Rightarrow$ argtypes(s)$_i \leq_{sub}$ argtypes(m)$_i$)
$\wedge$ restype(m) $\leq_{sub}$ restype(s))

$\Longrightarrow$ ⟨by definition of <: and assumption that $\grave{c}$ <: argtypes(s)⟩
$\forall\ s \in S.\ \forall\ m \in$ relevant(M, s). $\forall\ \grave{c} \in (C_{concrete})^*$.
$\grave{c}$ <: argtypes(s) $\Rightarrow$
$\grave{c} \leq_{inh}$ specializers(m) $\Rightarrow$
$\forall\ i \in$ indexes(specializers(m)).
(specializers(m)$_i \neq top\ \Rightarrow$
($\forall\ t \in$ conformed-types(specializers(m)$_i$). $t \leq_{sub}$ argtypes(m)$_i$))
$\wedge$ (specializers(m)$_i = top\ \Rightarrow\ c_i <:$ argtypes(m)$_i$)
$\wedge$ restype(m) $\leq_{sub}$ restype(s)

$\Longrightarrow$ ⟨by lemma 1 and assumption that $\grave{c} \leq_{inh}$ specializers(m)⟩
$\forall\ s \in S.\ \forall\ m \in$ relevant(M, s). $\forall\ \grave{c} \in (C_{concrete})^*$.
$\grave{c}$ <: argtypes(s) $\Rightarrow$
$\grave{c} \leq_{inh}$ specializers(m) $\Rightarrow$
$\forall\ i \in$ indexes(specializers(m)).
(specializers(m)$_i \neq top\ \Rightarrow$
($\forall\ t \in$ conformed-types(specializers(m)$_i$).
($\exists\ t' \in$ conformed-types(specializers(m)$_i$).
direct-conforms(c_i) $\leq_{sub}\ t'$)
$\wedge\ t \leq_{sub}$ argtypes(m)$_i$))
$\wedge$ (specializers(m)$_i = top\ \Rightarrow\ c_i <:$ argtypes(m)$_i$)
$\wedge$ restype(m) $\leq_{sub}$ restype(s)

$\Longrightarrow$ ⟨by instantiating t to t', which is legal because the result of conformed-types is always non-empty⟩
$\forall\ s \in S.\ \forall\ m \in$ relevant(M, s). $\forall\ \grave{c} \in (C_{concrete})^*$.
$\grave{c}$ <: argtypes(s) $\Rightarrow$
$\grave{c} \leq_{inh}$ specializers(m) $\Rightarrow$
$\forall\ i \in$ indexes(specializers(m)).
(specializers(m)$_i \neq top\ \Rightarrow$
($\exists\ t' \in$ conformed-types(specializers(m)$_i$).
direct-conforms(c_i) $\leq_{sub}\ t'$
$\wedge\ t' \leq_{sub}$ argtypes(m)$_i$))
$\wedge$ (specializers(m)$_i = top\ \Rightarrow\ c_i <:$ argtypes(m)$_i$)
$\wedge$ restype(m) $\leq_{sub}$ restype(s)

$\Leftrightarrow$ ⟨by definition of $<:$ and transitivity of $\leq_{sub}$⟩
$\forall\ s \in S.\ \forall\ m \in \text{relevant}(M, s).\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \text{argtypes}(s) \Rightarrow$
$\grave{c} \leq_{inh} \text{specializers}(m) \Rightarrow$
$\forall\ i \in \text{indexes}(\text{specializers}(m)).$
$(\text{specializers}(m)_i \neq top \Rightarrow$
$(\exists\ t' \in \text{conformed-types}(\text{specializers}(m)_i).$
$c_i <: \text{argtypes}(m)_i))$
$\wedge\ (\text{specializers}(m)_i = top \Rightarrow c_i <: \text{argtypes}(m)_i)$
$\wedge\ \text{restype}(m) \leq_{sub} \text{restype}(s)$

$\Rightarrow$ ⟨by logic, as t' does not occur in $c_i <: \text{argtypes}(m)_i$⟩
$\forall\ s \in S.\ \forall\ m \in \text{relevant}(M, s).\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \text{argtypes}(s) \Rightarrow$
$\grave{c} \leq_{inh} \text{specializers}(m) \Rightarrow$
$\forall\ i \in \text{indexes}(\text{specializers}(m)).$
$(\text{specializers}(m)_i \neq top \Rightarrow c_i <: \text{argtypes}(m)_i)$
$\wedge\ (\text{specializers}(m)_i = top \Rightarrow c_i <: \text{argtypes}(m)_i)$
$\wedge\ \text{restype}(m) \leq_{sub} \text{restype}(s)$

$\Leftrightarrow$ ⟨by law of excluded middle⟩
$\forall\ s \in S.\ \forall\ m \in \text{relevant}(M, s).\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \text{argtypes}(s) \Rightarrow$
$\grave{c} \leq_{inh} \text{specializers}(m) \Rightarrow$
$\forall\ i \in \text{indexes}(\text{specializers}(m)).$
$c_i <: \text{argtypes}(m)_i$
$\wedge\ \text{restype}(m) \leq_{sub} \text{restype}(s)$

$\Leftrightarrow$ ⟨by logic, as i does not occur in $\text{restype}(m) \leq_{sub} \text{restype}(s)$⟩
$\forall\ s \in S.\ \forall\ m \in \text{relevant}(M, s).\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \text{argtypes}(s) \Rightarrow$
$\grave{c} \leq_{inh} \text{specializers}(m) \Rightarrow$
$(\forall\ i \in \text{indexes}(\text{specializers}(m)).$
$c_i <: \text{argtypes}(m)_i)$
$\wedge\ \text{restype}(m) \leq_{sub} \text{restype}(s)$

$\Leftrightarrow$ ⟨by definition of $<:$ for vectors⟩
$\forall\ s \in S.\ \forall\ m \in \text{relevant}(M, s).\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \text{argtypes}(s) \Rightarrow$
$\grave{c} \leq_{inh} \text{specializers}(m) \Rightarrow$
$\grave{c} <: \text{argtypes}(m)$
$\wedge\ \text{restype}(m) \leq_{sub} \text{restype}(s)$

$\Leftrightarrow$ ⟨by logic, as m does not occur in $\grave{c} <: \text{argtypes}(s)$⟩
$\forall\ s \in S.\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \text{argtypes}(s) \Rightarrow$
$\forall\ m \in \text{relevant}(M, s).$
$\grave{c} \leq_{inh} \text{specializers}(m) \Rightarrow$
$\grave{c} <: \text{argtypes}(m)\ \wedge\ \text{restype}(m) \leq_{sub} \text{restype}(s)$

$\Leftrightarrow$ ⟨by definition of relevant(M, s) and set theory⟩
$\forall\ s \in S.\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\grave{c}$ <: argtypes(s) $\Rightarrow$
$\forall\ m \in \{\ m \in M \mid \text{msg}(m) = \text{msg}(s) \wedge \grave{c} \leq_{inh} \text{specializers}(m)\ \}.$
$\grave{c}$ <: argtypes(m) $\wedge$ restype(m) $\leq_{sub}$ restype(s)

$\Leftrightarrow$ ⟨by definition of applicable-methods(s, $\grave{c}$)⟩
$\forall\ s \in S.\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\grave{c}$ <: argtypes(s) $\Rightarrow$
$\forall\ m \in$ applicable-methods(s, $\grave{c}$).
$\grave{c}$ <: argtypes(m) $\wedge$ restype(m) $\leq_{sub}$ restype(s)

$\Leftrightarrow$ ⟨by definition⟩
ImplementationIsConforming

C.3 Correctness of Completeness Checking Algorithm

The following theorem says that our algorithm for computing completeness is sufficient to ensure completeness.

Theorem 3. ComputeIsComplete $\Rightarrow$ ImplementationIsComplete

Proof: We prove this theorem by the following calculation.

ComputeIsComplete

$\Leftrightarrow$ ⟨by definition⟩
$\forall\ s \in S.$
let *top-vector* = { $\grave{c}$ } where $|\grave{c}| = |$argtypes(s)$|$ and each c_i = *top* **in**
IsComplete(relevant(M, s), *top-vector*, s)

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

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

$\Leftrightarrow$ ⟨by definition of top-concrete-subclasses(*top*, argtypes(s)$_i$)⟩
$\forall\ s \in S.$
let *TCSs* = { $\grave{c}$' $\mid$ c_i' $\in$ tops(concrete-subclasses(*top*, argtypes(s)$_i$)) } **in**
$\forall\ \grave{c}$' $\in$ *TCSs*. $\exists\ m \in$ relevant(M, s). $\grave{c}$' $\leq_{inh}$ specializers(m)

$\Leftrightarrow$ ⟨by definition of concrete-subclasses(*top*, argtypes(s)$_i$)⟩
$\forall\ s \in S.$
let *TCSs* = { $\grave{c}$' $\mid$ c_i' $\in$ tops({ c' $\in C_{concrete} \mid c' \leq_{inh} top \wedge c'$<: argtypes($s$)$_i$ }) } **in**
$\forall\ \grave{c}$' $\in$ *TCSs*. $\exists\ m \in$ relevant(M, s). $\grave{c}$' $\leq_{inh}$ specializers(m)

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

$\Leftrightarrow$ ⟨by the definition of *TCSs* in the **let**⟩
$\forall\ s \in S.$
$\forall\ \grave{c}\,' \in \{\ \grave{c}\,' \mid c_i' \in \text{tops}(\{\ c' \in C_{concrete} \mid c' <: \text{argtypes}(s)_i\ \})\ \}.$
$\exists\ m \in \text{relevant}(M, s).\ \grave{c}\,' \leq_{inh} \text{specializers}(m)$

$\Leftrightarrow$ ⟨by definition of tops and $<:$ for vectors⟩
$\forall\ s \in S.$
$\forall\ \grave{c}\,' \in \text{tops}(\{\ \grave{c}\,' \in (C_{concrete})^* \mid \grave{c}\,' <: \text{argtypes}(s)\ \}).$
$\exists\ m \in \text{relevant}(M, s).\ \grave{c}\,' \leq_{inh} \text{specializers}(m)$

$\Leftrightarrow$ ⟨by definition of tops and $\leq_{inh}$⟩
$\forall\ s \in S.$
$\forall\ \grave{c}\,' \in \{\ \grave{c}\,' \in (C_{concrete})^* \mid \grave{c}\,' <: \text{argtypes}(s)\ \}.$
$\exists\ m \in \text{relevant}(M, s).\ \grave{c}\,' \leq_{inh} \text{specializers}(m)$

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

$\Leftrightarrow$ ⟨by definition of implication, predicate calculus, and $\forall x.\neg P(x) \Leftrightarrow \neg\exists x.P(x)$⟩
$\forall\ s \in S.\ \neg\ \exists\ \grave{c}\,' \in (C_{concrete})^*.$
$\grave{c}\,' <: \text{argtypes}(s)\ \wedge$
$\neg\ \exists\ m \in \text{relevant}(M, s).\ \grave{c}\,' \leq_{inh} \text{specializers}(m)$

$\Leftrightarrow$ ⟨by definition, renaming $\grave{c}\,'$ to $\grave{c}$⟩
ImplementationIsComplete

C.4 Correctness of Consistency Checking Algorithm

Theorem 4. ComputeIsConsistent $\Rightarrow$ ImplementationIsConsistent

Proof: We prove this theorem by the following calculation.

ComputeIsConsistent

$\Leftrightarrow$ ⟨by definition⟩
$\forall\ s \in S.$ IsConsistent(relevant(M, s), s)

$\Leftrightarrow$ ⟨by definition of IsConsistent(relevant(M, s), s)⟩
$\forall\ s \in S.\ \forall\ (m_1, m_2) \in$ incomparable-pairs(relevant(M, s)).
let $TLBs$ = tlb(specializers(m_1), 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**
IsComplete(M-*reduced*, *TLBs*, s)

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

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

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

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

$\Leftrightarrow$ ⟨by logic, as $\grave{c}$ does not occur in $\forall m_1, m_2 \in \text{relevant}(M, s).\ (\neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1))$⟩
$\forall\, s \in S.\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\forall m_1, m_2 \in \text{relevant}(M, s).$
$\neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1)\ \Rightarrow$
$\grave{c} <: \text{argtypes}(s)\ \wedge \grave{c} \leq_{inh} \text{specializers}(m_1)\ \wedge\ \grave{c} \leq_{inh} \text{specializers}(m_2)\ \Rightarrow$
$\exists\, m \in \text{relevant}(M, s).\ \grave{c}\,' \leq_{inh} \text{specializers}(m) \wedge m \leq_{meth} m_1 \wedge m \leq_{meth} m_2$

$\Leftrightarrow$ ⟨by definition of $\Rightarrow$, twice, and predicate calculus⟩
$\forall\, s \in S.\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\forall m_1, m_2 \in \text{relevant}(M, s).$
$\neg(\neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1))\ \vee$
$\neg(\grave{c} <: \text{argtypes}(s))\ \vee\ \neg(\grave{c} \leq_{inh} \text{specializers}(m_1)\ \wedge\ \grave{c} \leq_{inh} \text{specializers}(m_2))\ \vee$
$\exists\, m \in \text{relevant}(M, s).\ \grave{c}\,' \leq_{inh} \text{specializers}(m) \wedge m \leq_{meth} m_1 \wedge m \leq_{meth} m_2$

$\Leftrightarrow$ ⟨by associativity of $\vee$⟩
$\forall\, s \in S.\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\forall m_1, m_2 \in \text{relevant}(M, s).$
$\neg(\grave{c} <: \text{argtypes}(s))\ \vee$
$\neg(\neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1))\ \vee$
$\neg(\grave{c} \leq_{inh} \text{specializers}(m_1)\ \wedge\ \grave{c} \leq_{inh} \text{specializers}(m_2))\ \vee$
$\exists\, m \in \text{relevant}(M, s).\ \grave{c}\,' \leq_{inh} \text{specializers}(m) \wedge m \leq_{meth} m_1 \wedge m \leq_{meth} m_2$

$\Leftrightarrow$ ⟨by definition of $\Rightarrow$, three times⟩
$\forall\ s \in S.\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\forall\ m_1, m_2 \in \mathsf{relevant}(M, s).$
$\grave{c} <: \mathsf{argtypes}(s)\ \Rightarrow$
$\neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1)\ \Rightarrow$
$\grave{c} \leq_{inh} \mathsf{specializers}(m_1)\ \wedge\ \grave{c} \leq_{inh} \mathsf{specializers}(m_2)\ \Rightarrow$
$\exists\ m \in \mathsf{relevant}(M, s).\ \grave{c} \leq_{inh} \mathsf{specializers}(m)\ \wedge\ m \leq_{meth} m_1\ \wedge\ m \leq_{meth} m_2$

$\Leftrightarrow$ ⟨by logic, as m_1, m_2 do not occur in $\grave{c} <: \mathsf{argtypes}(s)$⟩
$\forall\ s \in S.\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \mathsf{argtypes}(s)\ \Rightarrow$
$\forall\ m_1, m_2 \in \mathsf{relevant}(M, s).$
$\neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1)\ \Rightarrow$
$\grave{c} \leq_{inh} \mathsf{specializers}(m_1)\ \wedge\ \grave{c} \leq_{inh} \mathsf{specializers}(m_2)\ \Rightarrow$
$\exists\ m \in \mathsf{relevant}(M, s).\ \grave{c} \leq_{inh} \mathsf{specializers}(m)\ \wedge\ m \leq_{meth} m_1\ \wedge\ m \leq_{meth} m_2$

$\Leftrightarrow$ ⟨by predicate calculus⟩
$\forall\ s \in S.\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \mathsf{argtypes}(s)\ \Rightarrow$
$\forall\ m_1, m_2 \in \mathsf{relevant}(M, s).$
$(\neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1)\ \Rightarrow$
$\grave{c} \leq_{inh} \mathsf{specializers}(m_1)\ \wedge\ \grave{c} \leq_{inh} \mathsf{specializers}(m_2)\ \Rightarrow$
$\exists\ m \in \mathsf{relevant}(M, s).\ \grave{c} \leq_{inh} \mathsf{specializers}(m)\ \wedge\ m \leq_{meth} m_1\ \wedge\ m \leq_{meth} m_2)$
$\wedge\ true \wedge\ true$

$\Leftrightarrow$ ⟨by reflexivity of $\leq_{meth}$ and predicate calculus⟩
$\forall\ s \in S.\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \mathsf{argtypes}(s)\ \Rightarrow$
$\forall\ m_1, m_2 \in \mathsf{relevant}(M, s).$
$(\neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1)\ \Rightarrow$
$\grave{c} \leq_{inh} \mathsf{specializers}(m_1))\ \wedge\ \grave{c} \leq_{inh} \mathsf{specializers}(m_2)\ \Rightarrow$
$\exists\ m \in \mathsf{relevant}(M, s).\ \grave{c} \leq_{inh} \mathsf{specializers}(m)\ \wedge\ m \leq_{meth} m_1\ \wedge\ m \leq_{meth} m_2)$
$\wedge\ (m_1 \leq_{meth} m_2\ \Rightarrow$
$\grave{c} \leq_{inh} \mathsf{specializers}(m_1)\ \Rightarrow$
$\grave{c} \leq_{inh} \mathsf{specializers}(m_1)\ \wedge\ m_1 \leq_{meth} m_1\ \wedge\ m_1 \leq_{meth} m_2)$
$\wedge\ (m_2 \leq_{meth} m_1\ \Rightarrow$
$\grave{c} \leq_{inh} \mathsf{specializers}(m_2)\ \Rightarrow$
$\grave{c} \leq_{inh} \mathsf{specializers}(m_2)\ \wedge\ m_2 \leq_{meth} m_2\ \wedge\ m_2 \leq_{meth} m_1)$

$\Longrightarrow$ ⟨by existentially quantifying over m_1 in one clause and m_2 in another clause⟩
$\forall\ s \in S.\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \mathsf{argtypes}(s)\ \Rightarrow$
$\forall\ m_1, m_2 \in \mathsf{relevant}(M, s).$
$(\neg(m_1 \leq_{meth} m_2) \wedge \neg(m_2 \leq_{meth} m_1)\ \Rightarrow$
$\grave{c} \leq_{inh} \mathsf{specializers}(m_1))\ \wedge\ \grave{c} \leq_{inh} \mathsf{specializers}(m_2)\ \Rightarrow$
$\exists\ m \in \mathsf{relevant}(M, s).\ \grave{c} \leq_{inh} \mathsf{specializers}(m)\ \wedge\ m \leq_{meth} m_1\ \wedge\ m \leq_{meth} m_2)$
$\wedge\ (m_1 \leq_{meth} m_2\ \Rightarrow$
$\grave{c} \leq_{inh} \mathsf{specializers}(m_1)\ \wedge\ \grave{c} \leq_{inh} \mathsf{specializers}(m_2)\ \Rightarrow$
$\exists\ m \in \mathsf{relevant}(M, s).\ \grave{c} \leq_{inh} \mathsf{specializers}(m)\ \wedge\ m \leq_{meth} m_1\ \wedge\ m \leq_{meth} m_2)$
$\wedge\ (m_2 \leq_{meth} m_1\ \Rightarrow$
$\grave{c} \leq_{inh} \mathsf{specializers}(m_1)\ \wedge\ \grave{c} \leq_{inh} \mathsf{specializers}(m_2)\ \Rightarrow$
$\exists\ m \in \mathsf{relevant}(M, s).\ \grave{c} \leq_{inh} \mathsf{specializers}(m)\ \wedge\ m \leq_{meth} m_2\ \wedge\ m \leq_{meth} m_1)$

$\Longleftrightarrow$ ⟨by $((P \Rightarrow R) \wedge (Q \Rightarrow R))\ \Leftrightarrow\ ((P \vee Q) \Rightarrow R)$ and predicate calculus⟩
$\forall\ s \in S.\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \mathsf{argtypes}(s)\ \Rightarrow$
$\forall\ m_1, m_2 \in \mathsf{relevant}(M, s).$
$(\neg(m_1 \leq_{meth} m_2\ \vee\ m_2 \leq_{meth} m_1)\ \Rightarrow$
$\grave{c} \leq_{inh} \mathsf{specializers}(m_1)\ \wedge\ \grave{c} \leq_{inh} \mathsf{specializers}(m_2)\ \Rightarrow$
$\exists\ m \in \mathsf{relevant}(M, s).\ \grave{c} \leq_{inh} \mathsf{specializers}(m)\ \wedge\ m \leq_{meth} m_1\ \wedge\ m \leq_{meth} m_2)$
$\wedge\ (m_1 \leq_{meth} m_2\ \vee\ m_2 \leq_{meth} m_1\ \Rightarrow$
$\grave{c} \leq_{inh} \mathsf{specializers}(m_1)\ \wedge\ \grave{c} \leq_{inh} \mathsf{specializers}(m_2)\ \Rightarrow$
$\exists\ m \in \mathsf{relevant}(M, s).\ \grave{c} \leq_{inh} \mathsf{specializers}(m)\ \wedge\ m \leq_{meth} m_1\ \wedge\ m \leq_{meth} m_2)$

$\Longleftrightarrow$ ⟨by law of excluded middle⟩
$\forall\ s \in S.\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \mathsf{argtypes}(s)\ \Rightarrow$
$\forall\ m_1, m_2 \in \mathsf{relevant}(M, s).$
$\grave{c} \leq_{inh} \mathsf{specializers}(m_1)\ \wedge\ \grave{c} \leq_{inh} \mathsf{specializers}(m_2)\ \Rightarrow$
$\exists\ m \in \mathsf{relevant}(M, s).\ \grave{c} \leq_{inh} \mathsf{specializers}(m)\ \wedge\ m \leq_{meth} m_1\ \wedge\ m \leq_{meth} m_2$

$\Longleftrightarrow$ ⟨by $\forall x.\neg P(x) \Leftrightarrow \neg\exists x.P(x)$ and definition of $\Rightarrow$⟩
$\forall\ s \in S.\ \forall\ \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \mathsf{argtypes}(s)\ \Rightarrow$
$\neg\ \exists\ m_1, m_2 \in \mathsf{relevant}(M, s).$
$\grave{c} \leq_{inh} \mathsf{specializers}(m_1)\ \wedge\ \grave{c} \leq_{inh} \mathsf{specializers}(m_2)\ \wedge$
$\neg\ \exists\ m \in \mathsf{relevant}(M, s).\ \grave{c} \leq_{inh} \mathsf{specializers}(m)\ \wedge\ m \leq_{meth} m_1\ \wedge\ m \leq_{meth} m_2$

$\Longleftrightarrow$ ⟨by $\forall x.\neg P(x) \Leftrightarrow \neg\exists x.P(x)$ and definition of $\Rightarrow$⟩
$\forall\ s \in S.\ \neg\ \exists\ \grave{c} \in (C_{concrete})^*.$
$\grave{c} <: \mathsf{argtypes}(s)\ \wedge$
$\exists\ m_1, m_2 \in \mathsf{relevant}(M, s).$
$\grave{c} \leq_{inh} \mathsf{specializers}(m_1)\ \wedge\ \grave{c} \leq_{inh} \mathsf{specializers}(m_2)\ \wedge$
$\neg\ \exists\ m \in \mathsf{relevant}(M, s).\ \grave{c} \leq_{inh} \mathsf{specializers}(m)\ \wedge\ m \leq_{meth} m_1\ \wedge\ m \leq_{meth} m_2.$

$\Longleftrightarrow$ ⟨by definition⟩
ImplementationIsConsistent

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Tech Report: TR94-03
Submission Date: March 2, 1994