Structural Abstraction

A Mechanism for Modular Program Construction

A Thesis
Presented to
The Academic Faculty

by

Shan Shan Huang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August, 2009

Structural Abstraction

A Mechanism for Modular Program Construction

Approved by:

Professor Yannis Smaragdakis,
Committee Chair
College of Computing
Georgia Institute of Technology

Professor Santosh Pande
College of Computing
Georgia Institute of Technology

Professor Richard LeBlanc
College of Computing
Georgia Institute of Technology

Dr. Spencer Rugaber
College of Computing
Georgia Institute of Technology

Professor Oege de Moor
Computing Laboratory
Oxford University

Date Approved: June 12, 2009

To Mom,

for teaching me to live life with courage and joy.

Acknowledgements

I wish I could say that careful planning and a tenacious drive toward a grand vision lead me to where I am today. But the truth is, this dissertation is the result of a series of serendipitous events in my life. I seem to have the good fortune of meeting the right people at the right time. For them, I am extremely grateful.

I had never written a single line of program when I entered MIT as a freshman. My choice of Computer Science as a major of study was the result of me being goaded by my peers, who claimed that the introductory computer science course (originally code-named 6.001) was extremely demanding, and my stubborn disbelief that there could be anything that is too demanding for me. Signing up for 6.001 was the beginning of the end. I thank my fellow freshmen for luring me into this course. I also thank the faculty of MIT for having cultivated the proper reputation for the course, and for having made computer science enchanting.

My choice of Yannis as an advisor bore initially out of a gut feeling that this was a guy who was interested in what I was interested in. Either due to extremely good fortune or an extremely omniscient gut, that choice led to the most educational and supportive mentorship I could have ever hoped for. I am honored to have worked with someone who possesses such a unique combination of vision, and an incredible attention to detail. Yannis has never failed to make me think harder, broader, and deeper all at the same time. Through our often intellectually challenging interactions, I learnt to distinguish interesting problems from nice engineering, and principled solutions from fancy hackery. After 6 years of training, I truly feel like a Jedi Knight. Yannis taught me how to learn, and I feel prepared to innovate in any discipline of computer science.

It must also be said that in addition to his fierce intellect, I always felt that Yannis had my best interest in mind, and wanted to make sure that I was not only being challenged, but also enthused, both about work and life. I could not have expected any more of him than I can from any mentor or friend.

People say that the two most important decisions in life is choosing one's thesis advisor, and

choosing the one's partner in marriage. In my case, my thesis advisor introduced me to the person I eventually married, Martin. I feel extremely fortunate that one random event in my life have lead to an even more random event: what are the chances, had it not been for the introduction arranged by Yannis and Martin's advisor, Eelco Visser, that a Dutchman and an American woman, living on separate continents, would fall in love and get married?

I am not sure if it is possible to sufficiently describe how lucky I feel to have Martin in my life. The intensity and joy with which he approaches his work inspires me daily; his insistence on doing all the "manly" things around the house (and my learning to finally let him) has helped greatly in the final years of my Ph.D. studies. But most of all, I am grateful that he accepts me with all my oddities and lunacies, and allows me to simply be me.

A power outage at the Portland Convention Center during OOPSLA 2006, and my being the student volunteer for a OOPSLA session during the outage, led to my fortuitous meeting with David Bacon. David chaired the session I volunteered for, and commissioned me to negotiate snacks for the attendees during the outage. David saw beyond my hunter-gatherer abilities, and offered me an internship with him at IBM T. J. Watson Research Center. My summer at Watson was one of the most inspiring experiences. It exposed me to research in parallel programming models and alternative hardware resources, topics I never thought I'd be involved in but now find extremely fascinating; it also introduced me to people who have become great friends. David has been an incredible mentor to me, even after the conclusion of my internship. His insights on life in industry versus academia, and the balance of work and life in general, have been incredibly valuable to me.

An unlikely email from Molham Aref, CEO of LogicBlox, to Yannis in 2004, an perhaps equally unlikely reply from Yannis, and a sub-prime mortgage crisis that led to the great financial meltdown of 2008—present, led to my post-Ph.D. position with LogicBlox. Had it not been for Molham and Yannis' relationship over the past few years, I would have never known about the exciting projects LogicBlox is embarking on; had it not been for the financial meltdown, I would have likely had an academic offer I was willing to accept, without even giving a thought to working in industry. In any case, I am grateful for Molham's confidence in me, for the chance to work for an exciting, growing company that can truly make a difference, and for the financial crisis for making me think hard about where I wanted to invest my Jedi Knight skill set.

Over the past 6 years, I have had the good fortune to meet and befriend many, many more people who have shaped and guided me. I am grateful for all my committee members for their time, their careful reading of the dissertation manuscript, and their insightful questions and comments. I am particularly grateful to have Oege de Moor as an advocate for my work, and an advisor in the great career decision between academia and industry. I am grateful to have Todd Millstein as a friend, advocate, and advisor, particularly through the job search process. I am grateful for Christoph Csallner, who was my officemate for 2 years and provided support and companionship when we were both transplanted from Georgia to Oregon. I am grateful to have Aaron VanDevender and his family in my life; they have served as great role models. I am grateful to Matt Might for sharing his experiences and wisdom in research, working for start-ups, marriage, and life in general. I am grateful to have wonderful in-laws, Hans, Ria, Marianne, Bart, baby Lara, and new baby Inge. They have supported me from afar, never failed to inquire about my progress or congratulate me on my milestones. And I am grateful for the roles they have played in Martin's life, for shaping him into the wonderful man he is today, and for sharing him with me.

I am grateful to Intel Corporation for giving me the Ph.D. Fellowship 2005—2006, the National Science Foundation for supporting me with the Graduate Research Fellowship from 2006—2009, and Eelco Visser and the good people of Delft University in the Netherlands for providing Subversion server space to host my dissertation.

Of course, the serendipitous event that started everything was my birth to the most loving, adventurous, good-natured, and good-humored parents I can imagine. Mom taught me through her living example how to never back away from adversity, to never loose sight of one's goals, and to always remember to laugh. I am fortunate to have inherited some of her tenacity for life, and her insatiable appetite for learning. Dad taught me to love with everything I have, to laugh heartily, and to never hold back. I wish circumstances allow me to spend more time with both of them.

Whether it is because I have been truly fortunate to have made the choices I did, or because I've been stubborn and resourceful enough to make bad choices into good ones, I feel blessed to be where I am today.

Table of Contents

List of Figures

Summary

Much of programming languages research has been directed at developing abstraction mechanisms to support the "separation of concerns": Orthogonal pieces of functionality should be developed separately; complex software can then be constructed through the composition of these pieces. The effectiveness of such mechanisms is derived from their support for *modularity and reusability*: The behavior of a piece of code should be reasoned about modularly—independently of the specific compositions it may participate in; the computation of a piece of code should allow specialization, so that it is reusable for different compositions. This dissertation introduces *structural abstraction*: a new abstraction mechanism that advances the state of the art in modularity and reusability by allowing the writing of highly reusable code—code whose structure can be specialized per composition, while maintaining a high level of support for modularity.

Structural abstraction addresses the reliance of mainstream abstraction mechanisms on fixed structures. Although abstraction mechanisms have been developed to allow code to be reused over ranges of values (e.g., procedural abstraction) and ranges of types (e.g., type genericity), their modularity guarantees crucially rely on separately developed pieces of code having fixed structures: A procedure is a structure with a fixed name and a fixed number of argument types, a type-generic procedure is a similar structure with a fixed number of type variables. This rigid notion of abstraction means a piece of code cannot be reused for clients with different structures, and thus often leads to the same piece of functionality having to be manually specialized for the (possibly infinite number of) different structures it may be composed with. Structural abstraction addresses this limitation by providing a disciplined way for code to inspect the structure of its clients in composition, and declare its own structure accordingly. The hallmark feature of structural abstraction is that, despite its emphasis on generality and greater reusability, it still allows modular type checking: A piece of structurally abstract code can be type-checked independently of its uses in compositions. Thus, errors in generic code are detected before composition time—an invaluable

feature for highly reusable components that will be statically composed by other programmers.

This dissertation introduces two specific techniques supporting structural abstraction: static type conditions, and morphing. Static type conditions allow code to be conditionally declared based on subtyping constraints. A client of such a piece of code can configure a desirable set of features by composing the code with types whose positions in a subtyping structure satisfy the appropriate typing conditions. Morphing allows code to be iteratively declared, by statically reflecting over the structural members of code that it would be composed with. A morphing piece of code can mimic the structure of its clients in composition, or change its shape according to its clients in a pattern-based manner. Thus, using either static type conditions or morphing, the structure of a piece of code is not statically determined, but can be automatically specialized to reflect the structure of its clients in composition.

We show how these structural abstraction techniques can be smoothly integrated with a modern Object-Oriented language, by implementing the languages cJ (for static type conditions) and MorphJ (for morphing), both of which are extensions of Java. We demonstrate the practical impact of structural abstraction by reimplementing a number of real world applications using cJ or MorphJ. We show that the reimplementations are more reusable, and, at the same time, modularly type-safe.

Chapter 1

My Thesis

Structural abstraction, the construction of programs that abstract over the structure of other programs, allows more concise and reusable capture of software functionality, while maintaining the ability to modularly reason about the behavior and correctness of such programs.

The inherent complexity of software is best managed through *separation of concerns* [28, 75]: orthogonal pieces of functionality should be developed separately; complex software can then be constructed through the composition of these pieces.

At the heart of the separation of concerns is the *modularity* and *reusability* of code. *Modularity* refers to the ability to reason about the behavior of a piece of code and guarantee its correctness,[1] *independently from its uses in compositions*. *Reusability* refers to the ability to use a piece of code in different compositions, where parts of its computation may be specialized differently per composition.

Abstraction is key in achieving both modularity and reusability. An abstract representation of a piece of code is a summary of its functionality that exposes the variable parts of its computation. The same piece of code can be composed with different instantiations of these variables, allowing its computation to be specialized for each composition. Varying degrees of requirements are placed on the values each variable may be instantiated to. The correctness of a piece of code can then be checked under the assumption that its variables are only instantiated with values that satisfy such requirements.

The more general the requirements are on its variables, the more reusable a piece of code is with different compositions. However, the more general the requirements, the less information there is to reason about the correctness of the code, thus making it harder to guarantee its

[1] There is a wide spectrum of what the correctness of a program can mean, from syntactic correctness, to type correctness, to the correctness of the program's runtime semantics. For the remainder of this dissertation, we concern ourselves with only the type-correctness of programs. Extending the notion of correctness beyond type-correctness is an orthogonal issue.

correctness independently of specific compositions. The development of abstraction mechanisms has thus been an exercise in raising the level of reuse while maintaining a high level of modularity.

Increasingly reusable, and equally modular, abstraction mechanisms have been developed: Procedural abstraction allows code to be specialized by the value of a variable; type genericity allows code to be specialized by the type as well as the value of a variable. The invariant among these mechanisms, however, is that the structure of a piece of code, and thus the structure of its abstraction, is always fixed and cannot be specialized by different compositions. For instance, the abstraction of a procedure is a fixed structure comprising its name, a fixed number of parameters and their types, and a return type. The abstraction of a type-generic procedure is a similarly fixed structure, with the addition of a fixed number of type parameters. Thus, for a piece of code to be used in different compositions, its programmer and clients in composition must agree on such a fixed structure; as the functionality of the code and the needs of its clients evolve, this structure must be continually redesigned, as well.

Indeed, this structurally rigid approach to abstraction has never served the separation of concerns very well. Its reliance on brittle, fixed interfaces had been mitigated by the fact that software used to be developed by teams closely working together. However, software is increasingly being constructed by composing together code developed by disparate teams. For example, a piece of software might use Hibernate [3] for communication with databases, Xerces for parsing XML [1], Guice [2] for handling the dependency injection between different components, etc. In this new development environment, the rigid approach to abstraction has become crippling: It is impossible for these teams to anticipate who their clients are, let alone agree on a fixed structure that suits the needs of every client.

Consequently, the modern programmer often is faced with the choice between the modularity and reusability of his code. He can choose to provide abstractions with fixed structures (and thus stay within the confines of modularity), by specializing his code for each client of a different structure. The reusability of his code invariably suffers. Each time a client with a new structure comes along, the programmer must provide another version of his code, specialized to the new structure, thus forcing a proliferation of declarations of highly similar functionality. Alternatively, the programmer could maximize the reusability of his code using meta-programming techniques,

such as reflection, code generation, or newer language mechanisms such as Aspect-Oriented Programming [61]. These techniques allow a piece of code to inspect the structure and computation of its client, *after composition*. Specialized code can then be generated to satisfy the needs of that client. While highly reusable, meta-programming techniques lack modularity: The code to be generated using these techniques depends on the control and data flow of the meta-program, both of which are not amenable to precise static analysis. Thus, it is impossible to make reasonably complete guarantees about the correctness of generated code indepndently of the compositions.

In any case, structurally rigid abstraction forces the entanglement of concerns: A programmer must either let the clients' structures dictate the way he develops his code; or he loses the ability to reason about and guarantee the correctness of his code.

Thus, the separation of concerns requires a fundamental shift away from the structurally rigid notion of abstraction. Instead, there needs to be a mechanism that supports the writing of code that can be composed with clients of different structures, and allows its own structure to be specialized accordingly.

This dissertation introduces *Structural Abstraction*, a new abstraction mechanism that allows the further separation of concerns by decoupling the rigid structural contract between a piece of code and its clients. A structurally abstract program does not need to be composed with clients of the same static structure. Instead, it allows its own structure, and thus its abstraction, to be shaped by the structure of its clients. It is thus a strictly more reusable mechanism, allowing code to be specialized not only by values and types, but also structures. By providing an expressive but controlled way for code to declare its own functionality based on the structure of its client, structurally abstract programs are still modular: The correctness of these programs can be guaranteed independently of their compositions with clients.

The remainder of this chapter first provides an overview of existing abstraction mechanisms (Section 1.1) and their limitations (Section 1.2). Section 1.3 then introduces structural abstraction techniques and briefly illustrates how they mitigate the problems presented by rigid abstractions. The remainder of this dissertation dives into the details of these structural abstraction techniques.

1.1 Existing Abstraction Mechanisms

Two abstraction mechanisms are pervasive in practice today: procedural abstraction and type genericity. Both are fully modular, though with varying degrees of support for reuse.[2] Additionally, there are various mechanisms (e.g., module systems, abstract data types, Object-Orientation) for the hierarchical organization and the reuse of groups of related functionality.

Procedural Abstraction Procedural abstraction is the earliest and most pervasive abstraction mechanism in modern programming languages. The key feature of procedural abstraction is that a piece of code can be declared with *value variables* (arguments). For example, the following procedure (in a Java [37] syntax) is declared with a variable `i`:

```
int plusOne(int i) { return i++; }
```

The abstraction of a procedure is a fixed structure comprising its name (`plusOne`), its argument types (`int`), and its return type (`int`). The arguments of a procedure allow its exact computation to vary based on the values these arguments are instantiated with; the argument types require these values to be within particular ranges. Procedural abstraction is modular: The implementation of a procedure can be checked against the value ranges dictated by the types in its abstraction, to guarantee that the code is always correct. For example, as long as `i` is an integer, the code `i++` is always correct (with respect to the static semantics of the language). Similarly, the procedure can be reused in composition with any concrete value of the specified type (e.g., all integers supported by the language).

Type Genericity The introduction of static typing inspired and necessitated a new abstraction mechanism: type genericity. Type-generic mechanisms allow code to be declared with *type variables*. In the following example, a type-generic procedure is declared with a type variable `T`:

```
<T> T identity(T t) { return t; }
```

[2] Certain implementations of type genericity are not modular, the most well-known one being C++ templates [49]. However, the techniques for how to implement a modularly type safe genericity mechanism is well-understood. Most modern type-generic languages (e.g., Java, C#, Haskell) do in fact support fully modular type-checking.

The abstraction of a type-generic procedure is a fixed structure comprising its type parameters, in addition to its name, argument and return types. Requirements on type variables (e.g., subtyping or structural constraints) allow sophisticated type genericity mechanisms, such as those supported by ML [71], Haskell [52], C# [39], and Java [37], to guarantee the correctness of type-generic code, regardless of what concrete types its type variables are instantiated to. In addition to specializing computation with specific values, the type variable allows the types used in the code to be specialized by each composition. Thus, compared to procedural abstraction, type genericity provides strictly more reusability for its code.

As its name suggests, "`identity`" is a polymorphic identity function that can take an argument of *any* type, and returns a value of that exact type.

Hierarchical Code Organization Mechanisms Code implementing related functionality is often hierarchically organized into modules (e.g., [64, 85, 94]), abstract data types (ADTs) [67], or classes, if in an Object-Oriented (OO) language.[3] This organization on top of individual procedures necessitates a new way to reuse a group of associated functionality. For instance, inheritance, mixins [15, 83], and traits [30]) are common techniques to facilitate group-reuse in OO languages. Nonetheless, the abstract representation of a module, an ADT, or a class still has a fixed structure, comprising the abstractions of its member procedures. This extra organizational structure in fact induces even more rigidity than individual procedures. Indeed, in the following section, we use examples from Java to demonstrate limitations caused by rigid structures imposed by classes.

1.2 A Need for Structurally Flexible Abstraction

The structurally rigid approach to abstraction assumes that the same functionality can be abstractly represented by the same static structure. In practice, however, there are two common ways that the same, or highly similar, functionality may need to vary the structure of its abstraction based on client need: conditionally, and iteratively. We show why existing mechanisms are ineffective in implementing these types of functionality, and illustrate the choice between modularity and reusability programmers must often make.

[3] ADTs and OO mechanisms are mostly seen as *data* abstraction mechanisms. In this dissertation, we are mostly concerned with their roles as *code* abstraction mechanisms, i.e., as hierarchical organizations of code.

1.2.1 Conditionally Exposed Functionality

Certain features of a piece of functionality are conditionally available based on the need of the client. For instance, code implementing an array list should offer the feature to add elements only if the client requires a variable-size array. Using existing mechanisms, a programmer can express this conditional feature by creating separate definitions of the array list, each representing a separate use case. For instance, in Java, we could declare the following two interfaces:

```
interface Array { ... }
interface VariableSizeArray extends Array {
  void add(Object o);
}
```

This approach maintains the modularity guarantees provided by existing mechanisms. Each interface is a specialization of array list, and the client can choose which one is most appropriate. However, the approach does not scale to support full reusability. To express an additional orthogonal condition, the number of abstract interfaces doubles. For instance, to represent array lists with mutable or immutable cells, we need to define two additional interfaces:

```
interface MutableArray extends Array { ... }
interface MutableVariableSizeArray extends VariableSizeArray { ... }
```

In general, to support n orthogonal conditional features, 2^n specialized interfaces are necessary. This is an unmaintainable approach from the programmer's point of view. It is equally unmanageable from the clients' point of view, since clients must keep track of 2^n different types and decide which one is appropriate.

An alternative approach that maximizes reusability is to provide all possible features of an array list through one interface, and allow the clients to configure appropriate features at composition time. Meta-programming techniques such as the C-pre-processing primitive `#ifdef` are often employed for this purpose. Conditional features are declared under an `#ifdef` *`CONDITION`* clause, for some meta-variable *`CONDITION`*. To include a particular feature, a client declares `#define` *`CONDITION`*.

Though flexible, the approach is not modular: The implementation of a piece of functionality can no longer be checked for correctness independently from client configurations. Inconsistencies between conditionally declared code using `#ifdef` and references to these declarations are

well-known issues in practice [33]. The only way to guarantee that a piece of code with n conditional declarations is always correct (compilable), regardless of client configurations, is to compile the code 2^n times! Even for moderately sized n, this approach to guaranteeing correctness is infeasible.

1.2.2 Iteratively Defined Functionality

Consider a piece of code that, given *any* class `X`, returns another class that contains the exact same methods as `X`, but logs each method's return value. That is, the code is a reusable representation of the functionality "logging", and abstracts over the exact methods it may be applied to. Such a capture of "logging" would need to apply its functionality iteratively to the methods of its client class, whatever that class may be. Note that this is a *compile-time iteration over the static structure of* `X`, and *not a runtime iteration* over dynamic collections.

Using existing mechanisms, to maintain the modularity of code, a programmer must specialize "logging" for each client class. For instance, to compose "logging" with classes `Foo`, `Bar`, and `Baz`, the programmer must provide classes `LoggedFoo`, `LoggedBar`, and `LoggedBaz`. These definitions, of course, are not at all reusable with any other client class.

To define a version of "logging" that is reusable with *any* client class, a programmers can resort to meta-programming techniques such as meta-object protocols (MOPs) [58], aspect-oriented programming (AOP) [61], or even lower-level techniques such as reflection and ad-hoc program generation using quote primitives, string templates, or bytecode engineering. For instance, using AspectJ [60], the flagship tool for AOP, one can implement "logging" as:

```
1 abstract aspect Logging<X> {
2   pointcut loggedMeth() : execution(* X.*(..));
3
4   after() returning(Object r) : loggedMeth() {
5     System.out.println("Returning " + r.toString());
6   }
7 }
```

For any type `X`, `pointcut loggedMeth()` identifies the execution of its methods. The "advice" on lines 4-6 specifies the logging code to be run after the execution of these methods, where the return values are caught by `Object r`. To compose "logging" with `Foo`, one can use the following declaration:

```
aspect LoggingFoo extends LoggingAspect<Foo> { }
```

The main problem with meta-programming techniques such as AOP is that they are not modular: There is no notion of separately reasoning about the definition of `LoggingAspect` and guaranteeing its correctness without knowing its specific compositions (e.g., with class `Foo`, `Bar`, `Baz`, etc.). Errors in `LoggingAspect` are not caught by its programmer, and may only reveal themselves for certain compositions. For instance, this particular declaration of `LoggingAspect` will result in ill-typed code if composed with a class that contains a method that returns `void`: the code on line 8 will attempt to invoke `toString()` on a `void` type. Yet the AspectJ compiler would not warn the programmer of this potential composition error.[4]

1.3 Structural Abstraction

Structural abstraction is a new approach to abstraction where the structure of code as well as its abstraction can be specialized, conditionally or iteratively, by its clients in composition. This dissertation presents two techniques to support structural abstraction: static type conditions and morphing. Static type conditions target the issue of code with conditional features, whereas morphing targets the issue where code structure needs to be iteratively shaped by the structure of its clients. Next, we show how static type conditions and morphing allow the concise, reusable, but, at the same time, modular capture of the types of functionality described in Section 1.2.

1.3.1 Static Type Conditions

The goal of static type conditions is to allow the conditional declaration of code without sacrificing modularity, i.e., the consistency in the declaration and use of conditional code is guaranteed independently of specific combinations of conditions configured by the clients. To this end, static type conditions use subtyping constraints as conditions, and code can be conditionally declared if certain subtyping conditions are satisfied.

We illustrate static type conditions in the language cJ, an extension of Java. cJ allows supertypes, fields, and methods of a class or interface to be provided only under some static subtyping

[4] This error can be viewed as not a limitation of the current AspectJ compiler, but a limitation of a more fundamental nature to the aspects event-based approach. For a detailed exposition on why, see Section 4.7.

condition. For instance, a cJ generic class, C<P>, may provide a member method m only when the type provided for parameter P is a subtype of a specific type Q. Clients of C<P> can configure the functionality necessary by parameterizing it in different ways: C<some-subtype-of-Q> would have access to method m, whereas C<some-type-not-a-subtype-of-Q> would not.

The problem illustrated in Section 1.2 can be easily solved in cJ with the following interface:

```
interface ArrayList<X> {
  <X extends VariableSize>?
  void add(Object o);
}
```

The method "add" is conditionally declared under the condition that X is some subtype of VariableSize, where VariableSize is a "marker" interface:

```
interface VariableSize { }
```

The declaration of the marker interface allows us to pull the condition of whether or not an array is being used as a variable-sized one into cJ's type system. The cJ compiler can now use this information to guarantee that "add" is always used under conditions that are at least as strong as the condition under which it is defined, i.e., when used, the instantiation of X is at least a subtype of VariableSize. Each additional orthogonal condition requires exactly one more marker interface. Multiple conditions can be specified using a conjunction of types, e.g., <X extends VariableSize & Mutable>. Thus, for every n conditions, we only need to declare n marker interfaces. There is no exponential number of types to maintain, for either the programmer or the clients.

1.3.2 Morphing

Morphing is a mechanism developed to allow a piece of functionality to be *iteratively composed* with the structure of its clients. We discuss morphing through the language MorphJ. MorphJ is an extension of Java, where the notion of type genericity in Java is extended so that the *structure* of a class or interface can vary based on the structure of type variables. For instance, the logging functionality discussed in Section 1.2.2 can be implemented as the following MorphJ generic class:

```
class Logging<class X> extends X {
  <R,Y*>[meth]for(public R meth (Y) : X.methods)
  public R meth (Y a) {
    R r = super.meth(a);
    System.out.println("Returned: " + r);
    return r;
  }
}
```

MorphJ allows class `Logging` to be declared as a subclass of its type parameter, `X`. The body of `Logging` is defined by static iteration (using the `for` statement) over all methods of `X` that match the pattern "`public R meth(Y)`". `Y`, `R`, `meth` are pattern-matching variables. `Y` and `R` can match any non-`void` type, and `meth` can match any name. Additionally, the `*` symbol following the declaration of `Y` indicates that `Y` matches any number of types (including zero). That is, the above pattern matches all `public` methods that return any non-`void` type. The pattern-matching variables are also used in the declaration of `Logging`'s methods: for each method of the type parameter `X`, `Logging` declares a method with the same name and type signature. (This does not have to be the case, as shown later.) Thus, the exact methods of class `Logging` are not determined until it is composed, or type-instantiated, with a concrete type. For instance, `Logging<java.lang.Object>` has methods `equals`, `hashCode`, and `toString`: these are the only public, non-`void`-returning methods of `java.lang.Object`.

A well-formed MorphJ generic class is guaranteed not to contain type errors no matter what the generic class is type-instantiated with. Note that the above class only iterates over the non-`void` methods, and statically avoids the error of invoking `toString()` on `void`. Had the pattern been different and allowed matching of `void`-returning methods, the MorphJ compiler would have reported the error *without having to compose `Logging` with any specific class `X`*.

1.4 Contributions

Structural abstraction represents a fundamental shift in the support for the separation of concerns. Instead of allowing code addressing separate concerns to be tied together through a brittle, rigid interface, structural abstraction allows code to adapt its structure to the needs of its compositions. The result is more reusable, more malleable software. The techniques presented in this dissertation, static type conditions and morphing, show that modularity does not have to suffer with this

increased level of reuse. By representing the structural relationships of code and its clients in a disciplined way, structurally abstract code can still be checked for its correctness independently of its eventual compositions.

Concretely, this dissertation illustrates the power of structural abstraction by combining static type conditions and morphing with OO mechanisms, and advances the state of the art in programming languages design with the following contributions:

- Both static type conditions and morphing, when combined with OO, allow expressing highly reusable generic classes concisely. Static type conditions, as implemented in cJ, allow a single generic class to express the functionality of an exponential number of regular Java classes. Morphing, as implemented in MorphJ, allows a generic class to express the functionality of an *infinite* number of regular Java classes.

- This dissertation shows that static type conditions and morphing can be smoothly integrated with a modern language: Java. No prior research has considered how type-safe conditional declarations can be implemented in a rich language setting (e.g., with subtyping), or how advanced language features, such as use-site variance, interact with conditional declarations. Furthermore, prior attempts at offering "reflective" declarations always required concepts separate from the base language, such as a "transformation". MorphJ shows how transformation can be a natural extension of the concept of a generic class.

- Both static type conditions and morphing offer full modular type safety. A cJ or MorphJ generic class is type-checked independently from its type-instantiations, and errors are detected if they can occur with *any* possible type parameter.

- This dissertation demonstrates the practical benefits of structural abstraction techniques with real-world applications.

  - cJ solves a well-known static type safety issue of the Java Collections Framework [5] (or, the "optional methods" problem). We are not aware of other language proposals that address this well-publicized need without sacrificing conciseness.

– MorphJ was used to re-engineer real-world applications. One of the more dramatic examples is a MorphJ reimplementation of Java Collections Framework [5], which replaces 314 lines of hardcoded proxy classes with 26 lines of highly reusable MorphJ code; Another such example is a MorphJ reimplementation of DSTM2 [41], a Java software transactional library that heavily uses reflection and bytecode engineering (with BCEL [9]) to transform sequential classes into transactional ones. The MorphJ reimplementation replaces 1,484 lines of Java reflection and BCEL library calls with 576 lines of MorphJ code.

1.5 Dissertation Roadmap

In the remainder of this dissertation, Chapter 2 presents static type conditions. The power of static type conditions is shown by solving the well-known *optional methods* problems with Java Collections Framework. A formal type system is presented, with detailed proofs of soundness in Appendix A.

Chapter 3 presents the features of morphing, as well as how it can be used to solve real-world programming problems. An informal introduction to the insights behind the separate type-checking for morphing is presented, along with a formalization of a core subset of MorphJ. Proofs of soundness of the formal type system are presented in Appendix B.

Finally, Chapter 4 discusses related work, and Chapter 5 concludes with a summary of the contributions of this work, along with future directions.

Chapter 2

Static Type Conditions: Modular and Reusable Declaration of Conditional Features

Static type conditions are an abstraction mechanism that allows code to be conditionally declared, based on subtyping conditions. A client of such a piece of code can then configure a desirable set of features, and thus the desired interface of the code, by composing the code with the types that satisfy the appropriate type conditions. Unlike conditional compilation techniques (e.g., the C/C++ "`#ifdef`" construct) static type conditions guarantee the modular type-safety of conditionally declared code: Regardless of specific client configurations, conditionally declared code is guaranteed to be well-typed, e.g., there will be no reference to undeclared code due to inconsistencies in the conditions used for declaration and reference.

We demonstrate the concept of static type conditions with the language cJ. cJ is an extension of Java that allows supertypes, fields, and methods of a class or interface to be provided only under some static subtyping condition. For instance, a cJ generic class, `C<P>`, may provide a member method `m` only when the type provided for parameter `P` is a subtype of a specific type `Q`.

cJ adds to generic Java classes and interfaces the ability to express case-specific code. As a specific application, cJ addresses the well-known shortcomings of the Java Collections Framework (JCF) [5]. JCF data structures often produce run-time errors when an "optional" method is called upon an object that does not support it. Within the constraints of standard Java, the authors of the JCF had to either sacrifice static type safety or suffer a combinatorial explosion of the number of types involved, each type specialized for a different client configuration. cJ avoids both problems, maintaining both static safety and conciseness (and thus reusability).

2.1 Introduction

Generic types increase the expressiveness and safety of a programming language. Since the introduction of Java and C#, researchers have worked on adding genericity mechanisms that were

subsequently integrated into the base languages themselves [16, 57, 96]. From a language design standpoint, modern genericity mechanisms offer a good trade-off between expressiveness and separate checkability. For instance, Java generics have limited expressiveness compared to undisciplined mechanisms, such as C++ template classes, yet offer the ability to detect static errors (e.g., type errors) without having to provide a specific type parameter that triggers the error.

This chapter introduces cJ: an extension of Java with support for static type conditions. cJ extends the expressiveness of modern genericity by allowing not only the types of declared methods and fields of a generic class to vary based on client composition, but also the *structure* of the class itself to vary. cJ provides this increased level of expressiveness without sacrificing any of Java's type-checking guarantees: a cJ generic class is separately type-checked to guarantee that it is always a well-typed class regardless of type-instantiations.

Specifically, cJ adds to Java the ability to place type-conditions on methods, fields, or supertypes. This is best illustrated with a small example. Consider the following generic cJ class:

```
class C<X> {
  X xRef;
  ...
  <X extends DataSource>?
  void store() { ... xRef.getConnection() ... }
}
```

In this example, the member method `store` is declared in a type-instantiation of generic class `C`, *only* when the type argument for `X` is a class (or interface) that implements (resp., extends) `DataSource`. The `<...>?` syntax is cJ's construct for a static type condition. One can read this syntax as "static-if", or just "if". The call `xRef.getConnection()` is well-typed only because type `X` is guaranteed to be a subtype of `DataSource` and, consequently, to provide the `getConnection` method.

cJ is translated by erasure, reducing to regular Java in a backward compatible manner. This allows us to solve a well-recognized problem in the Java Collections Framework (JCF) [5], the standard Java data structures library. Currently, JCF data structures support two main common interfaces (`Collection` and `Map`), regardless of optional behavior, such as whether the data structure is modifiable or not, and whether the data structure has variable size or not. Classes that do not support the corresponding operations throw `UnsupportedOperationExceptions` when the

operations are called at run-time. The design of the JCF is an instance of sacrificing static type safety in favor of conciseness. cJ solves this problem, maintaining both type safety and conciseness of expression.

Interestingly, cJ can be thought of as a language that allows cross-cutting [61] at the level of types. cJ type conditions are used to define many implicit types from a single class definition. In the above example, the single definition of class `C` can be thought of as defining the implicit types `C<subtype-of-DataSource>` and `C<not-DataSource>`. Thus, with cJ type-conditionals, one can add orthogonal "aspects" or "dimensions" to an existing type hierarchy. Our re-implementation of the JCF provides a vivid demonstration of this feature. Beginning from a simple subtyping hierarchy, we introduce variation based on whether a data structure supports content modification or size variability. The definitions of the various collections (e.g., the `List`, `Collection`, `Map`, and `Set` interfaces) are as simple as in plain Java, yet a much richer type hierarchy is produced by modifying each type with attributes chosen from a separate type hierarchy (with types such as `Modifiable`, `DeleteOnly`, and `Resizable`). Thus, whereas traditional Aspect-Oriented Programming (AOP) [61] tools allow cross-cutting code to be injected at the level of methods, using a dynamic, event-driven model (e.g., when methods of `Foo` are invoked, inject logging code), cJ supports separation of concerns at the type level, using a static model based on subtyping hierarchies. Type hierarchies can be specified separately to represent orthogonal concerns, and cJ allows their composition to form richer, derived hierarchies. Writing general code that exploits these derived hierarchies can be done through a natural extension of the Java variance/wildcards mechanism (e.g., "`? extends T`" clauses).

Although Java was chosen as the platform for our ideas, the cJ approach is far from Java-specific. The same programming and type-checking framework can be applied to other languages. Yet Java is a good representative of modern Object-Oriented (OO) languages and integrating with it demonstrates clearly both the benefits and the intricacies of our approach.

Static type conditions make the following concrete contributions to programming languages and software engineering research:

- cJ allows expressing highly reusable generic classes concisely. Compared to standard OO mechanisms, cJ allows a single generic class to express the functionality of an exponential

number of regular Java classes.

- cJ offers full modular type safety, analogous to that of the base Java language. A cJ generic class is checked separately from its uses. The type system ensures that the class is type-correct under any consistent combination of outcomes of the type-conditionals.

- Other research work [32, 66, 68, 74] has targeted the problem of type-safe conditional declarations. Nevertheless, many of the past mechanisms are in a simpler context (e.g., no subtyping) or do not allow some of the cJ features (e.g., conditional subtyping). None of the past research dealt with the integration of conditional members and subtypes with (use-site) variance, nor provided a backward compatible, erasure-based translation. Overall, cJ is distinguished by its power and its smooth integration into a modern language.

- cJ solves the static type safety issues of the JCF. We are not aware of other language proposals that address this well-publicized need without sacrificing conciseness.

The remainder of this chapter is organized as follows. Section 2.2 gives an informal introduction to the cJ language extensions. This serves as background for the motivating examples of Section 2.3 and the JCF case study. Section 2.4 presents interesting ways in which the cJ extensions interact with variance in Java. We then analyze the cJ implementation in Section 2.5. Section 2.6 formalizes cJ's type system, with detailed proofs of soundness presented in Appendix A.

2.2 cJ Language Introduction

We next give an informal overview of cJ's syntax and semantics, to prepare the ground for our motivating examples. A formal description of the language is laid out in Section 2.6.

2.2.1 cJ Basics and Examples

cJ is a conservative extension of Java—we assume Java 5 or above, with support for generics and variance ("wildcards") [16, 90] in the base language. cJ adds to Java the ability to change a type's structure depending on static type conditions. The language provides a static *type-conditional* (or just type-conditional) construct. The following is a simple example showing the use of a type-conditional:

```
class Foo<T> {
  <T extends Bar>?
  int i;
}
```

In the above example, `<T extends Bar>?` is a type-conditional. The declaration immediately following it, "`int i;`", exists only if `Foo` is parameterized by a subtype of `Bar`.

Type-conditionals can be used for the declaration of class- or interface-level methods and fields, as well as for the declaration of conditional supertypes. For instance, we can have:

```
class Foo<T> <T extends Serializable>? implements Serializable
{ ... }
```

The above class `Foo` implements interface `Serializable` only when it is parameterized by a type that also implements (or extends) `Serializable`.

Multiple declarations can exist in the same conditional block by surrounding them in `<...>`. For instance:

```
class Foo<T> {
  <T extends Bar>?
  <
    int i;
    void meth(T t) { }
  >
}
```

The above is equivalent to preceding each declaration individually with the type-conditional `<T extends Bar>?`.

There are two required components to each type-conditional block. The first is the type condition, defined inside "`<...>?`". Any syntax that is valid for defining the type parameters of a Java class is valid here as a type condition: the type conditions have the standard F-bounded polymorphism form [16], where a type parameter can be referenced by its own bound, e.g., "`T extends I<T>`". Note especially that "`extends`" is used to express all kinds of subtyping constraints (including interface conformance) and that the syntax admits conjunctions of subtyping bounds (e.g., "`T extends I<T> & J<T>`"), as well as bounding multiple parameters (e.g., "`S extends I<S>, T extends J<S>`"). Similarly to Java, it is *not* valid for a type parameter to appear by itself on the right hand side of `extends` (i.e. we cannot place lower bounds on a type parameter). Also note that only type parameters declared at the class/interface level are allowed in type conditions.

Polymorphic method type parameters are not allowed. The second required component, the consequent block, immediately follows the type condition. Declarations within this block exist for the enclosing type if and only if the type condition is true, after all type parameters are instantiated. A type-conditioned declaration is syntactically a declaration, hence, type-conditionals can nest.

cJ ensures that all uses of type-conditionals are statically safe. All code should be well-typed under its enclosing type conditions. Furthermore, all uses of class or interface members should be under equivalent or stronger conditions than those employed in the member's declaration. For instance, the following use is legal only if `J` is a subtype of `I`:

```
class Foo<T> {
  <T extends I>?
  int i;

  <T extends J>?
  void incI() { i++; } // legal iff J subtypes I
}
```

The following code is also legal, as the type conditions are strengthened by adding conjunctions:

```
class Foo<T,U> {
  <T extends I>?
  int i;

  <T extends I, U extends K>?
  void incI() { i++; } // legal: stronger condition

  <T extends I & J>?
  void decI() { i--; } // legal: stronger condition
}
```

2.2.2 Restrictions

There are some restrictions that cJ imposes on conditional declarations. These restrictions significantly simplify the translation and interfacing with existing Java code, as we will discuss in Section 2.5. The rule of thumb is that a cJ class (or interface) should be a legal Java class (interface) if all type conditions are removed.

A cJ class can have at most one `extends` clause, regardless of whether it is under a type-conditional. Of course, a cJ class can implement multiple interfaces and any of the `implements` clauses can be conditional.

Declarations that are conflicting per the standard Java rules are not allowed, even if their type-conditional conditions are exclusive. For instance, the following is illegal in cJ, even when neither Baz nor Bar are a subtype of the other:

```
interface IFoo<T> {
  <T extends Bar>?
  void foo(int i);

  <T extends Baz>?
  int foo(int i);  // duplicate definition
}
```

Furthermore, subtypes are required to define conditional methods under equivalent or weaker conditions than conflicting methods in their (possibly conditional) supertypes. For example:

```
interface ISuper<T,U> {
  <T extends Bar>?
  void meth1(int i);

  <T extends Bar>?
  void meth2(Object o);
}

interface ISub<T,U> extends ISuper<T,U> {
  <T extends Bar & Baz>?
  void meth1(int i); // illegal unless Bar subtypes Baz

  void meth2(Object o); // legal: weaker (no condition)
}
```

The above rules extend to the members of conditional supertypes. Their type conditions from the perspective of the subtype is the conjunction of the subtyping and the membership conditions. (E.g., a member under a static condition P in an interface implemented under condition Q should be thought of as being under a condition $P\&Q$ for the purposes of the above discussion.) Our formalism in Section 2.6 makes this definition precise.

2.3 cJ Benefits

Having introduced the cJ language, we can now examine some motivating examples. We first discuss a small example that demonstrates how a type-conditional avoids a combinatorial blowup of the number of classes required in a Java application. Then, we examine a specific case study: the Java Collections Framework and its well-known shortcomings with respect to static type safety.

2.3.1 The Argument for Safety and Conciseness

There are two ways to view the benefits of cJ over regular Java. In Java, when the contents of a class can vary with respect to multiple orthogonal concerns, the programmer can either choose to maintain static type safety and suffer a combinatorial explosion of the number of classes involved, or sacrifice static type safety in order to keep the number of classes manageable. cJ achieves both benefits simultaneously.

The conciseness benefits of cJ are relatively easy to see. When multiple conditionals capture different axes of variability, a cJ generic class corresponds to a hierarchy of many different regular classes. Consider a simple example class `C`:

```
class C<X> {
  <X extends Serializable>?
  public void store() { ... }
  ...
  <X extends Comparable<X>>?
  public X getMin() { ... }
}
```

That is, class `C` supports method `store` only when type parameter `X` is a serializable type. Similarly, `C` supports method `getMin` only when type parameter `X` is a comparable type.

To achieve the same effect with regular Java, the programmer needs to create separate classes that capture all possible combinations. One possibility would be the following class hierarchy:

```
class CommonC<X> {
  ... // the common parts of C
}

class CSer<X extends Serializable> extends CommonC<X> {
  public void store() { ... }
}

class CComp<X extends Comparable<X>> extends CommonC<X> {
  public X getMin() { ... }
}

class CCompSer<X extends Comparable<X> & Serializable> extends CSer<X> {
  public X getMin() { ... }
}
```

The result is four different classes, capturing the same content as the original cJ class. Method code is replicated: `CCompSer` cannot inherit `getMin` from `CComp` because it already has a superclass,

CSer. Furthermore, CCompSer is not a subtype of CComp, hence, a CCompSer object cannot be used where a CComp is expected, even though it supports the required methods of a CComp. Such code replication and subtyping problems can be alleviated by using delegation techniques and interfaces, but this may require significant code reorganization, weakening of encapsulation, and explicitly maintaining object identity. For instance, to minimize code length with delegation, the programmer often needs to enable access to members of another class, as well as manually ensure a one-to-one mapping among different sub-objects.

Note that this example deals with only two axes of variability: whether X is Comparable and whether X is Serializable. Still, the result is undesirable. In the general case, the number of Java classes required for a faithful emulation is exponential to the number of distinct type-conditionals in the cJ class, assuming a straightforward mapping. Overall code length will also be exponentially greater, unless delegation, with its aforementioned disadvantages, is used.

In practice, it is unlikely that Java developers would want to deal with this kind of combinatorial complexity. Instead, they will likely prefer to provide a single type that captures the union of all possible members. In that case, when an "unsupported" method is called, a run-time error can be signaled in the form of an exception. For instance, following Java conventions, our earlier example is likely to be written in standard Java as follows:

```
class C<X> {
  public void store() throws UnsupportedOperationException
  { ... }
  ...
  public X getMin() throws UnsupportedOperationException
  { ... }
}
```

This addresses the code size and number-of-types explosion problem at the expense of sacrificing static type safety. The type checker is no longer able to tell under what conditions the store and getMin operations would be illegal. A run-time type error is produced instead, when illegal operations get called.[1] Relative to plain Java, cJ combines the advantages of static type safety and code conciseness.

[1]The UnsupportedOperationException is a run-time exception (i.e., the compiler does not check that it is always caught or declared) and a member of the JCF. For the purposes of this paper, we use this exception type even for code outside the JCF. Any different exception could assume the same general role.

It is worth noting that the cJ compiler translates its input into plain Java by following an approach similar to that of the example above (i.e., a single class is produced, containing all possible members). Yet, the cJ type system statically ensures that no exceptions for unsupported methods are thrown at run-time. We describe the cJ implementation in Section 2.5.

Finally, an interesting question on the power of type-conditionals concerns their value under multiple inheritance. Multiple inheritance can address the problems of delegation, in that it allows composing a class modularly without violating object identity or encapsulation. If Java had multiple inheritance, in addition to its bounded generics, the above example could be expressed in the same amount of code as in cJ. Nevertheless, the main benefit of type-conditionals is not in minimizing the code length, but in minimizing the number of explicit types that users need to manage. Consider the case of a type hierarchy among types `I1`, `I2`, ..., `IN`. Type conditionals allow the programmer to create implicitly a virtual isomorphic hierarchy by using a single class `C<X>` with member and/or supertype declarations conditional on `X` extending `I1`, `I2`, etc. The language will automatically ensure that the two hierarchies have consistent structure. If, for instance, `I1` is a subtype of `I5`, all methods in `C` declared conditionally under `X extends I1` will be able to access methods declared conditionally under `X extends I5`. With traditional subtyping mechanisms, the user would need to create explicit types `C1`, `C2`, ..., `CN` with a subtyping hierarchy reflecting the one of `I1`, `I2`, ..., `IN`. Relieving the programmer from explicitly managing these types is the greatest advantage of type-conditionals in *any* language setting. As we discuss in the next section, the stated motivation of Java developers for choosing a type-unsafe solution for the JCF was not avoiding code size explosion but avoiding an explosion in the number of explicit types that users would need to deal with.

2.3.2 Case Study: Java Collections Framework

A striking demonstration of the problems presented above can be found in the Java Collections Framework [5]: the standard Java data structures library. The JCF supplies types such as `Collection`, `Set`, `Map`, and `List`. However, there are other cross-cutting concerns along which to organize these basic data structures. One such concern is that of "modifiability": is a data structure modifiable through its public interface or not? This concept is not captured via the Java type

system in the design of the JCF. Instead, any attempt to modify an "unmodifiable" collection results in the throwing of an `UnsupportedOperationException` at run-time. Another similar concern is that of size variability. Some data structures are modifiable, yet their size cannot change—arrays are a standard example. An array supports the operations of the `List` interface with the exception of `add` or `remove`, which throw `UnsupportedOperationException`. This is a case of circumventing the static type system in order to avoid a combinatorial explosion in the number of types specified in the library. In fact, six out of the fifteen methods of interface `Collection` in JDK 1.5 are optional and may result in run-time errors.

The above is a well-known issue. The very first "frequently asked question" in the Java Collections API Design FAQ [4] is:

> Why don't you support immutability directly in the core collection interfaces so that you can do away with optional operations (and `UnsupportedOperationException`)?

The design rationale reflected in the answer to this FAQ indirectly offers a compelling argument for cJ. The developers note:

> Clearly, static [compile time] type checking is highly desirable, and is the norm in Java. We would have supported it if we believed it were feasible. Unfortunately, attempts to achieve this goal cause an explosion in the size of the interface hierarchy ...

Subsequently, the Java Collections API developers proceed to give an illustration of the kinds of "explosion in size" problems that a type-safe design would encounter, if cross-cutting concerns such as "modifiable", "variable-size", "append-only", etc., are expressed in the type system. The Java Collections Design FAQ concludes:

> Now we're up to twenty or so interfaces and five iterators, and it is almost certain that there are still collections arising in practice that don't fit cleanly into any of the interfaces.

The above issue is not specific to the Java Collections Framework. Other developers of Java data structure libraries have identified the same shortcomings. Doug Lea (quoted in the JCF FAQ) authored a popular Java collections package and remarks:

> Much as it pains me to say it, strong static typing does not work for collection interfaces in Java.

(We invite the reader to consult online the informative FAQ answer, which we cannot reproduce here in its entirety.)

cJ addresses fully and cleanly the above problem with the JCF. Interfaces `Collection`, `List`, etc. are implemented modularly using type-conditionals. Specifically, there are three interesting properties that we capture: whether a collection is modifiable, whether it supports only deletions, and whether it supports both deletions and additions (i.e., all size change operations). These cross-cutting concerns are expressed using (marker) interfaces `Modifiable`, `DeleteOnly` and `Resizable`. The `Resizable` interface is a subtype of `DeleteOnly`—a resizable collection supports operations such as `clear` and `remove`, but also `add` and `addAll`. By combining these interfaces one can specify different flavors of each collection. This is done through a type parameter `M` passed to each collection generic class. For instance, interfaces `Collection` and `List` are implemented as shown in Figure 2.3.2.[2] (Note that the question-mark symbol is used both in our type-conditional syntax, and as a wildcard in order to specify variance in generic operations, per the standard Java syntax.)

Concrete classes that implement these interfaces (e.g., `ArrayList`) have similarly structured type-conditionals. This implementation is concise without sacrificing static type safety. The user of the `List` interface explicitly selects the desired flavor of the collection. For instance, a possible type instantiation of `List` is `List<Integer,Modifiable>`, signifying a modifiable (but not resizable) list of integers. Another possible instantiation is `List<Integer,Object>` (or any type that is not a subtype of `Modifiable` in place of `Object`) to signify a non-modifiable and non-resizable list. The programmer cannot accidentally call a `set` method on a collection that is statically specified to be unmodifiable. The need for an `UnsupportedOperationException` is eliminated.

The JCF case study serves well as a motivating example for the more powerful cJ features described in later sections. Specifically, the major question we have not yet addressed is how to write general code that abstracts over multiple cJ types. There are two ways to safely abstract over types in the Java type system. One way is to use interfaces—e.g., we may want to write code that works with all `Comparable` objects regardless of whether they are of type `Integer`, `String`, `Array`, etc. The other way is to use variance—e.g., we can write code that works with all `List<X>` objects, as long as the element type, `X`, is a subtype of a given type, say, `Number`. Both of these valuable

[2] Our re-implementation of the JCF can be found on the cJ website: http://www.cc.gatech.edu/∼ssh/cj.

```
interface Collection<E,M> extends Iterable<E,M> {
  <M extends Resizable>?
  <
  boolean add(E o);
  boolean addAll(Collection<? extends E, ?> c);
  >
  <M extends DeleteOnly>?
  <
  boolean removeAll(Collection<?, ?> c);
  void clear();
  ...
  >
  boolean contains(Object o);
  boolean isEmpty();
  ... // other methods common to all collections
}


interface List<E,M> extends Collection<E,M> {
  <M extends Resizable>?
  <
  void add(int index, E element);
  boolean addAll(int index, Collection<? extends E,?> c);
  >
  <M extends DeleteOnly>?
  <
  E remove(int index);
  ...
  >
  <M extends Modifiable>?
  E set(int index, E element);
  ... // other methods common to all lists
}
```

Figure 1: cJ imlementation of Collection and List from Java Collections Framework.

mechanisms are straightforwardly extended and enhanced in cJ.

cJ conditional supertypes enable abstraction using interfaces even for types that support the corresponding operations only conditionally. For instance, we can have definitions such as:

```
class ArrayList<X,M>
     <X extends Comparable<X>>? implements Comparable<List<X>> {
  ...
  <X extends Comparable<X>>?
  public int compareTo(List<X> that) { ... }
}
```

The above `ArrayList` class implements interface `Comparable` and provides the appropriate `compareTo` method only if its parameter type is also a `Comparable`.[3] Thus, we can use such `ArrayList` objects with code accepting any `Comparable` object—unlike the original Java `ArrayList` class.

The second kind of abstraction is quite interesting and practically valuable in the cJ setting. For instance, how can we write code that deals uniformly with `List` objects that support at least a `remove` operation, regardless of whether the objects are of type `ArrayList<E,DeleteOnly>` or `ArrayList<E,Resizable>` or any other compatible subtype and "flavor" combination? This is precisely the role of the question-mark wildcard types that appeared in our above Java Collections code—e.g., for method `addAll`. The general approach follows a natural extension of the standard Java variance mechanism. We discuss this topic in the next section.

2.4 Subtyping and Variance

cJ type-conditionals turn out to fit very well in the Java type checking framework. In particular, the relationships among different instantiations of the same generic cJ class fall out very simply from the standard rules for variance, with only a small addition. We next give a bird's eye view of wildcards and variance in the Java type system (readers familiar with variance can skip Section 2.4.1) and then discuss how these relate to cJ.

2.4.1 Variance and Wildcards

Here we only give a brief (and simplified) summary of Java wildcards as used to implement variance. A thorough treatment can be found in past literature [48, 90].

Java allows using generic types with a non-specific instantiation, through the wildcard syntax

[3] Our thanks to Phil Wadler and Martin Odersky for this motivating example.

"? extends T", "? super T" and "?". For a generic type C, the meaning of a C<? extends T> is "C instantiated with any subtype of T". For instance, the JCF Collection interface supports a method:

```
interface Collection<E> extends Iterable<E> {
  ...
  addAll(Collection<? extends E> c);
}
```

The wildcard means that if, for instance, we have an object of type Collection<Number>, we can pass as an argument to its addAll method an object of type Collection-of-some-subtype-of-Number. For instance (assuming Integer subtypes Number):

```
Collection<Number> c = new ArrayList<Number>();
Collection<Integer> ci = new ArrayList<Integer>();
... // populate ci
c.addAll(ci);
```

Similarly, the wildcard syntax "C<? super T>" means "C instantiated with any supertype of T", and the syntax "C<?>" means "C instantiated with anything".

Wildcards form an elegant way to write highly general code that can apply to multiple instantiations of generic types. Nevertheless, to statically ensure that the result is safe (i.e., that the object can indeed support all the operations that the code wants to perform on it) several restrictions need to be imposed.

- An object c of type C<? extends T> can only be used to call methods where the type parameter of C is in a *co-variant* position, i.e., appears only as the return type of a method, if at all. Also, fields of c typed as the type parameter of C can only be read from, not written to. For instance, given an object c of type Collection<? extends E>, we can never invoke a method such as "boolean add(E o)" on c, because this method is declared in interface Collection<E>, and the type parameter E appears as an argument type to add.

- Similarly, an object c of type C<? super T> can only be used to call methods with the type parameter of C in a *contra-variant* position, i.e., it appears only as an argument type to a method, if at all. Fields of c typed as the type parameter of C can be written to with values typed T, but only read as values of type Object.

- An object c of type C<?> can only be used to call methods where the type parameter of C

does not appear at all (*bi-variance*). Similarly, the fields of `c` typed as the type parameter of `C` can only be read as `Objects`, and not written to.

Next we discuss how a slight extension of the Java variance rules makes them apply transparently to cJ.

2.4.2 Variance and Type-Conditionals

We return to the original question regarding type-conditionals and subtyping. Consider a cJ class `C<X>`. Can we write code that is general enough to work type-safely with multiple instantiations of `C<X>` (i.e., for multiple values of `X`)? Consider the simple example from Section 2.3.1:

```
class C<X> {
  <X extends Serializable>?
  public void store() { ... }
  ...
  <X extends Comparable<X>>?
  public X getMin() { ... }
}
```

Intuitively, `X` is used in this example only in order to add more members to generic class `C`. Thus, a "stronger" `X` (i.e., one that will satisfy more "`extends`" type conditions) will only result in more members being added. In other words, if type `S` is a subtype of `T` then `C<S>` could safely be a subtype of `C<T>`—generic class `C` can be co-variant in its type parameter.

cJ, just like regular Java, does not automatically relate different instantiations of a generic class via subtyping. That is, in the Java and cJ type systems, an instantiation `C<A>` is never a subtype of `C<B>` for two distinct classes `A` and `B`, regardless of the contents of `C` or how `A` and `B` are related. However, if `A` is a subtype of `B`, then `C<A>` is a subtype of `C<? extends B>` and `C<B>` is a subtype of `C<? super A>`. The programmer can use such subtyping relations to write code that applies to multiple instantiations of a generic class and the language statically checks that the code is safe, based on the rules outlined earlier.

cJ enhances the variance rules to deal with type conditions. For instance, we can have the following method, accepting an argument of the above type `C`:

```
void export(C<? extends Serializable> c) {
 ... c.store(); ...
}
```

That is, the `export` method accepts objects of type `C`-of-some-subtype-of-`Serializable`. The

language ensures that the body of `export` uses its argument `c` correctly. In this case, the call to `store` is statically type safe, since for any subtype `X` of `Serializable`, type `C<X>` will support `store`.

The general rule for interactions between type parameters and variance is simple:

> An occurrence of type parameter `X` in an `<X extends ...>?` type-condition (on either a supertype declaration or a member declaration) constitutes a co-variant use.

Enhanced with the above rule, all other rules of the standard variance framework of Java apply and enable general type safety.

Consider, for instance, a `Queue` that supports averaging of its elements if they are `Numbers`. (This is an artificial example—the functionality is not part of the Java Collections Framework.):

```
interface Queue<X> {
  <X extends Number>?
  X average();
  ... // other methods
}
```

Both appearances of type parameter `X` are in co-variant positions: either in a type condition, or as a return type. In this case, a method can accept objects of type `Queue`-of-some-subtype-of-`Number` and call `average` on them safely. For instance, we can have:

```
void covariant(Queue<? extends Number> q) {
  Number a = q.average();
}
```

We already saw uses of variance in our Java Collections API case study. Consider the following excerpt from the definition of `Collection<E,M>` in Section 2.3.2:

```
interface Collection<E,M> extends Iterable<E,M> {
  ...
  boolean addAll(Collection<? extends E, ?> c);
  ...
  boolean removeAll(Collection<?, ?> c);
}
```

Methods `addAll` and `removeAll` in the above use arguments bi-variant with respect to the second type parameter of `Collection`. That is, these methods can accept any collection, regardless of whether it is modifiable or not, delete-only or not, etc. Note that the above type signatures statically prevent the implementation of methods `addAll` and `removeAll` from calling methods

such as `add`, `clear`, or `set` on their argument `c`: all these methods are declared conditionally and `c` may not support them. Intuitively, this reflects the intent of the interface for methods `addAll` and `removeAll`: they modify the object from which they are invoked, but not their argument object, from which they only read values to add or remove.

Overall, cJ type-conditionals are an excellent match for Java variance. Not only does variance offer a natural abstraction mechanism for conditional types, but also variance and type-conditionals offer the same kind of benefit in a programming language. Both mechanisms allow specifying a single class `C<X>` and having the type system automatically compute several useful derivative types. In the case of variance the derivative types are `C<? extends T>`, `C<? super T>` and `C<?>`, which contain only the co-variant, contra-variant, and bi-variant methods of the class, with respect to some type `T`. In the case of cJ the derivative types correspond to all possible outcomes of type-conditionals. For instance, `Modifiable-and-DeleteOnly-List` is an implicit type produced from the `List<E,M>` definition. Each cJ implicit type contains only the members that exist for this combination of conditions.

2.5 Implementation

The design of cJ was carefully planned to admit a simple *erasure-based* translation that is backward compatible with Java code. Each cJ generic class can be translated to a single Java generic class (which in turn can be translated to a single non-generic Java class, per the standard erasure translation of Java generics). This and other implementation topics are discussed next.

Erasure. The current cJ compiler is a source-to-source translator into Java. Nevertheless, exactly the same techniques could be used in a direct-to-bytecode translation. Indeed, the source-to-source translation has even more transparency requirements and demonstrates how well cJ fits the Java model.

cJ translates a class (or interface) with type-conditionals into a Java class (resp., interface) by removing all conditional statements. This enables a single class to play the role of all possible instantiations. Consider our earlier example:

```
class C<X> {
  <X extends Serializable>?
  public void store() { ... }
  ...
  <X extends Comparable<X>>?
  public X getMin() { ... }
}
```

cJ translates C into a class:

```
class C<X> {
  public void store() { ... }
  ...
  public X getMin() { ... }
}
```

Note that there is no need for a run-time exception. The cJ type system ensures statically that unsupported methods can never be called. (If client code is not compiled with the cJ compiler, there is no such guarantee. We later discuss how the user can explicitly request dynamic checks to ensure that these methods are not called.)

Erasure Intricacies. The cJ translation requires a few more steps than simply removing the type-conditionals. First, the cJ compiler translates the bodies of conditional methods using type casts that ensure the appropriate type conditions. Second, it supplies dummy method bodies to classes implementing (or extending) an interface (class) with unsupported methods. Lastly, it translates certain type instantiations into their "raw type" forms, and performs the same code generation that a plain Java compiler performs in translating generic code into non-generic code. We demonstrate these translations via examples.

When translating conditional code, the cJ compiler needs to maintain known type bounds for each expression. If this is different from the type the expression would have when conditionals are eliminated, then casts need to be output. Consider the example from the Introduction:

```
class C<X> {
  X xRef;
  ...
  <X extends DataSource>?
  void store() {
    ... xRef.getConnection() ...
  }
}
```

The call to `getConnection` is only valid because the type `xRef` is known to be a subtype of

DataSource. Thus, the compiler needs to emit a cast that will ensure this type constraint when the type-conditional is removed. The cast cannot fail at run-time, as the cJ static type checker ensures the store method is only called when X is indeed a subtype of DataSource. The translated code is:

```
class C<X> {
  X xRef;
  ...
  void store()  {
     ...((DataSource) xRef).getConnection()...
  }
}
```

In the case of interfaces (or abstract classes), our erasure translation means that classes implementing (extending) an interface (abstract class) may need to be automatically enhanced. Consider a conditional interface method. Erasure removes the type-conditional and the method will be declared for all instantiations of the interface. Yet, classes implementing some of these instantiations will not provide implementations of the method, as the method is undeclared for the given type parameters. For instance:

```
interface List<E,M> extends Collection<E,M> {
  <M extends DeleteOnly>?
  E remove(int index);
  ...
}

class FixedList<E> implements List<E,Object> {
  ... // no remove: Object is not subtype of DeleteOnly
}
```

The translation adds a dummy remove public method in FixedList. The translated version of the above example is as follows:

```
interface List<E,M> extends Collection<E,M> {
  E remove(int index);
  ...
}

class FixedList<E> implements List<E,Object> {
  ...
  public E remove(int index) throws UnsupportedOperationException {
    throw new UnsupportedOperationException();
  }
}
```

The same translation technique is used for safety: in a naive translation scheme, subclasses of a class that has conditional methods would inherit those methods because of the erasure translation in the superclass, allowing code not compiled with the cJ compiler to gain access to those methods. Instead, we ensure that the subclass overrides the method with a dummy implementation to avoid such accidental exposure of the superclass functionality. Note that this problem is similar to that faced by the designers of GJ [16], and the solution we adopt is also similar to theirs. For instance, consider the following class:

```
class Channel<T> {
  <T extends Trusted>?
  void disconnect() { ... }
}
```

Erasure will remove the type-conditional and, thus, expose the `disconnect` method. If the user wants to ensure security he/she can export only specialized subclasses that explicitly do not implement the `Trusted` interface:

```
class NonsecureChannel extends Channel<Object> { }
```

The cJ compiler will translate the latter into a class that is safe to use in an insecure environment, avoiding accidental exposure of the superclass method:

```
class NonsecureChannel extends Channel<Object> {
  void disconnect() throws UnsupportedOperationException {
    throw new UnsupportedOperationException();
  }
}
```

This translation technique does not help avoid the accidental exposure of fields, however. To protect a conditional field against unauthorized access, a programmer could designate the field `private`, and define getter/setter methods for it. The above translation technique for methods can then be used to protect the getter/setter methods from unauthorized uses.

In certain situations, a type instantiation considered legal by the cJ compiler might not be considered legal by a regular Java compiler. For example,

```
class C<X> {
  <X extends Enum<X>>?
  EnumSet<X> es = null;
}

class EnumSet<X extends Enum<X>> {
  ...
}
```

A simple erasure applied to class `C<X>` would erase the type condition `<X extends Enum<X>>?`. However, type instantiation `EnumSet<X>` is not compilable using a Java compiler, because `X` is nowhere declared to be a subtype of `Enum<X>`. In these situations, cJ translates type `EnumSet<X>` all the way down to its "raw type" form, `EnumSet`. Thus, the translation of `C<X>` would be:

```
class C<X> {
  EnumSet es = null;
}
```

The cJ compiler then needs to perform all the translations that a regular Java compiler does for expressions of type `EnumSet`, e.g., generating casts of return types of methods called on this type.

Translation and Backward Compatibility. The interesting aspect of the cJ translation, as described above, is that it is remarkably simple and fits very well the existing Java object model. The restrictions of the cJ language outlined in Section 2.2.2 are in place explicitly so that an elegant erasure-based translation can be supported. For instance, ensuring that methods do not conflict, even when they are under disjoint type conditions, means that we can employ the erasure-based translation without the need for method renamings. Similarly, ensuring that overriding methods (in a subclass) are declared under weaker type conditions than the overridden methods (in the superclass) enables a clean erasure by just removing the type-conditionals. It means that a subclass method does not "accidentally" override a valid superclass method when the subclass method should not really exist based on its type condition. Translating all cJ classes and methods one-to-one into Java classes and methods ensures good interfacing with client code, and even unsuspecting legacy (i.e., standard Java) code.

The cJ translation also includes some transparent special case handling purely for strong backward compatibility, even at the source level. This was motivated by our study of the Java Collections Framework. The special handling occurs when the cJ compiler is invoked in "compatibility mode"

and when a type parameter is used *only* in type-conditionals (and not, for instance, to declare references). In that case, the cJ compiler treats the parameter as optional. For instance, the cJ compiler can compile legacy Java code using the `Collection<E>` interface (and any of the classes implementing it) against the cJ library, which defines `Collection` as `Collection<E,M>`. (Either all optional parameters or none need to be omitted.) When a type parameter is omitted, the instantiation is assumed to satisfy all the type-conditionals, and any instantiation with full type parameters is a subtype of it, and vice versa. Our treatment is directly analogous to "raw types" in the translation of Java generics [16].

Furthermore, when a type parameter to a class is used only in type-conditionals (or transitively as a type argument to another class that uses this parameter only in type conditionals), then the cJ compiler removes it from the translated code. This means that the code generated from the cJ compiler can be used as a regular Java library, under plain Java compilers. This is best illustrated with an example. Consider the form of our standard `List` interface from the Java Collections API:

```
interface List<E,M> extends Collection<E,M> {
  <M extends Modifiable>?
  E set(int index, E element);

  <M extends Resizable>?
  void add(int index, E element);
  ...
}
```

`List` uses its type parameter `M` only in type conditions and to instantiate another type, `Collection`, where it is also used only in type conditions. `M` is never used as an argument or return type of a method. Therefore, the cJ compiler accepts code that refers to `List` with only one type parameter. At the same time, the translation of the above cJ interface into a plain Java interface eliminates the second type parameter:

```
interface List<E> extends Collection<E> {
  E set(int index, E element);

  void add(int index, E element);
  ...
}
```

In short, the cJ compiler compiles old-style Java code even against new-style (cJ) libraries that use extra type variables for type conditions. Furthermore, the cJ compiler translates new-style (type

conditional) library code into Java code that is source-compatible with existing Java client code, under standard Java compilers.

Clearly, our erasure translation has the same requirements as other erasure translations—e.g., that of GJ [16]—for the purpose of full integration in Java. For instance, the reflection mechanism needs to change to support cJ-translated code. This is not part of our current implementation.

2.6 Formalization

We present the formal syntax and typing rules for a subset of cJ. Our formalism is an extension of the formalism for Featherweight GJ (FGJ) with variance, by Igarashi and Viroli [48]. We call our calculus Featherweight cJ (FCJ). To aid readers in understanding, we highlight the FCJ-specific parts of the formalism in grey. FCJ captures a core subset of cJ functionality that allows us to explore the type-safety issues introduced by type-conditionals, with minimum extra baggage and duplication of work that has already been done for FGJ [47] and variance [48]. Our formalism requires that all type parameters declare upper bounds, which may be `Object`. Each class must declare a superclass, which may also be `Object`. Additionally, all superclass declarations must be guarded by type-conditionals, though unconditional superclasses can be expressed by having the type-conditional be `<X extends N>?`, where `N` is `X`'s declared upper bound. Similarly, all method declarations must be guarded by type-conditionals, as well. Conditional fields are not supported in the formalism, since the issues involving conditional member declarations are thoroughly represented by conditional methods. Interfaces are not part of either the original FGJ, or our formalism. Thus, we only support conditional superclasses. A class declaration includes a sequence of fields and method declarations. We assume an implicit constructor for each class, which takes as arguments expressions that can be used to initialize field values. The method body is simply an expression.

Note that the variance formalization by Igarashi and Viroli does not strictly model the wildcard implementation in Java. Some notable differences include the inability in the Igarashi and Viroli system to model wildcard capture, in which a wildcard is promoted to a fresh type variable. (Wildcard capture most commonly occurs when an expression typed using a wildcard is used as an argument in a method invocation. In typing that method invocation, the wildcard is converted to, or "captured" as, a fresh type variable.) There have been multiple attempts to formalize the

wildcard mechanism as it is implemented in Java (e.g., [22,89]). Of these, [89] does not have a proof of soundness. [22] does present the most complete model of Java wildcards as well as a proof of soundness. Unfortunately, [22] was published after the completion of the work in this chapter. Thus, we use the Igarashi and Viroli formalization here, as a solid basis for proving the soundness of our type system.

Notation. For readers unfamiliar with FGJ [47] and the variance formalism [48], we briefly introduce the notational conventions used. The meta-variables `C` and `D` range over class names; `X`, `Y`, and `Z` range over distinct type variables; `S`, `T`, `U`, `V`, and `W` range over types; `H`, `N`, `O`, `P`, `Q`, and `R` range over non-variable types (fully instantiated types); `f` and `g` range over field names; `m` ranges over method names; `x` ranges over parameter names; `d` and `e` range over expressions; and `M` ranges over method declarations. Meta-variable `v` represents variance annotations `o`, `+`, `-`, and `*`, for in-variant, co-variant, contra-variant, and bi-variant, respectively—e.g., `+T` corresponds to `? extends T` in the full Java syntax. Variance annotations can be placed in front of any non-variable type. In-variant is the assumed default, and thus, `C<oT>` is abbreviated to `C<T>`. A partial order $\leq$ on variance annotations can be defined as: $\mathtt{o} \leq \mathtt{+} \leq \mathtt{*}$, $\mathtt{o} \leq \mathtt{-} \leq \mathtt{*}$. $\mathtt{v}_1 \vee \mathtt{v}_2$ represents the least upper bound of $\mathtt{v}_1$ and $\mathtt{v}_2$.

In addition, we use a few shorthand conventions for conciseness. $\overline{\mathtt{X}}$ is a shorthand for $\mathtt{X}_1\mathtt{,}\ldots\mathtt{,}\mathtt{X}_n$, and similarly, $\overline{\mathtt{T}}\ \overline{\mathtt{x}}$ is a shorthand for $\mathtt{T}_1\mathtt{x}_1\mathtt{,}\ldots\mathtt{,}\mathtt{T}_n\mathtt{x}_n$. When this shorthand is applied to a type variable or a regular variable (i.e., fields, method arguments), it represents a sequence with no duplication. We use $\bullet$ to denote an empty sequence. The notation $\triangleleft$ is the shorthand for keyword `extends`, and $\uparrow$ is the shorthand for keyword `return` in method bodies.

We also assume a class table CT, which maps class names `C` to their declarations. A *program* is a pair $(CT, \mathtt{e})$ of a class table, and an expression.

2.6.1 Syntax

We present the FCJ syntax in Figure 2. The syntax follows closely the abstract syntax for FGJ with variance [48]. The main difference is the addition of a type-conditional construct in front of superclass and method declarations. The type-conditional construct, $\mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{R}}\mathtt{>?}$, evaluates to true if, after type parameter instantiation, the types for $\overline{\mathtt{X}}$ are subtypes of $\overline{\mathtt{R}}$. A fully instantiated class

has the declared superclass if and only if the type-conditional guarding the superclass declaration evaluates to true. Otherwise, it extends `Object`. Similarly, a method exists for a fully instantiated class if and only if its type condition evaluates to true. We inherit the assumption from FGJ that method declarations $\overline{\texttt{M}}$ have unique names—that is, we do not model method name overloading.

Syntax:

```
T      ::=  X | N
N      ::=  C<v̄T̄>
v      ::=  o | + | - | *
CL     ::=  class C<X̄◁N̄> S-if ◁D<S̄> {T̄ f̄; M̄}
M      ::=  S-if <Ȳ◁P̄> T m (T̄ x̄) {↑e;}
S-if   ::=  <X̄◁R̄>?
e      ::=  x
         |  e.f
         |  e.<T̄>m(ē)
         |  new C<T̄>(ē)
         |  (T)e
```

Figure 2: FCJ: Syntax

2.6.2 Type System

The main typing rules for FCJ are presented in Figure 3 and Figure 4. Δ and Γ are the two environments used in typing judgments. Δ is a type environment that ranges over subtyping assumptions of the form $\texttt{T}<:\texttt{S}$. Additionally, Δ also holds mappings from type variables introduced to represent variant types, to the corresponding variance annotations and bounds. For instance, $\Delta(\texttt{X})=(+,\texttt{T})$ indicates $\texttt{X}$ is introduced into Δ to represent a variant type with an upper-bound of $\texttt{T}$. Alternatively, $\Delta(\texttt{X})=(-,\texttt{T})$ indicates that $\texttt{X}$ represents a variant type with a lower-bound of $\texttt{T}$. When $\texttt{X}<:\texttt{N}\in\Delta$ and $\texttt{N}$ is a non-variable type, for all $\texttt{X}$, we say that Δ has non-variable bounds. Γ is a variable environment that maps a variable $\texttt{x}$ to its type $\texttt{T}$. We use $\texttt{x:T}$ to denote variable to type mapping.

To support the typing rules, we present some auxiliary definitions in Figure 5, and the definition of "open"($\Uparrow^{\Delta}$) and "close"($\Downarrow_{\Delta}$) of variant types in Figure 6. These rules and definitions follow closely the format of those in variance-based FGJ. We assume the reader has a certain familiarity with the FGJ formalization, though not necessarily with the Igarashi and Viroli variance formalism.

Expression typing:

$$\Delta;\Gamma \vdash \mathtt{x} \in \Gamma(\mathtt{x}) \qquad \text{(T-VAR)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \quad \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}_0) \Uparrow^{\Delta'} \mathtt{C<\overline{U}>} \quad \mathit{fields}(\ (\Delta,\Delta')\ ,\ \mathtt{C<\overline{U}>}) = \overline{\mathtt{S}}\ \overline{\mathtt{f}} \quad \mathtt{S}_i \Downarrow_{\Delta'} \mathtt{T}}{\Delta;\Gamma \vdash \mathtt{e}_0\mathtt{.f}_i \in \mathtt{T}} \qquad \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \quad \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}_0) \Uparrow^{\Delta'} \mathtt{C<\overline{T}>} \quad \mathit{mtype}(\ (\Delta,\Delta')\ ,\ \mathtt{m},\ \mathtt{C<\overline{T}>}) = \mathtt{<\overline{Y} \lhd \overline{P}>\overline{U} \to U}_0 \\ \{\overline{\mathtt{Y}}\} \cap \mathit{dom}(\Delta') = \emptyset \quad \Delta \vdash \overline{\mathtt{V}}\ \mathit{ok} \quad \Delta,\Delta' \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \quad \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \\ \Delta,\Delta' \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}} \qquad [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 \Downarrow_{\Delta'} \mathtt{T}\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0\mathtt{.<\overline{V}>m(\overline{e})} \in \mathtt{T}} \qquad \text{(T-INVK)}$$

$$\frac{\Delta \vdash \mathtt{C<\overline{T}>}\ \mathit{ok} \quad \mathit{fields}(\ \Delta\ ,\ \mathtt{C<\overline{T}>}) = \overline{\mathtt{U}}\ \overline{\mathtt{f}} \quad \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} <: \overline{\mathtt{U}}}{\Delta;\Gamma \vdash \mathtt{new\ C<\overline{T}>(\overline{e})} \in \mathtt{C<\overline{T}>}} \qquad \text{(T-NEW)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}\ \mathit{ok} \\ \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}_0) <: \mathit{bound}_\Delta(\mathtt{T}) \quad \text{or} \quad \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}) <: \mathit{bound}_\Delta(\mathtt{T}_0)\end{array}}{\Delta;\Gamma \vdash \mathtt{(T)e}_0 \in \mathtt{T}} \qquad \text{(T-CAST)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}\ \mathit{ok} \\ \Delta \not\vdash \mathit{bound}_\Delta(\mathtt{T}_0) <: \mathit{bound}_\Delta(\mathtt{T}) \quad \text{and} \quad \Delta \not\vdash \mathit{bound}_\Delta(\mathtt{T}) <: \mathit{bound}_\Delta(\mathtt{T}_0)\end{array}}{\Delta;\Gamma \vdash \mathtt{(T)e}_0 \in \mathtt{T}} \qquad \text{(T-SCAST)}$$

Method typing:

$$\frac{\begin{array}{c}\mathtt{\overline{X} <: \overline{N} \vdash \overline{R} <: \overline{N}} \qquad \Delta = \mathtt{\overline{X} <: \overline{R}, \overline{Y} <: \overline{P}} \qquad \Delta \vdash \mathtt{\overline{R}, \overline{P}, \overline{T}, T}_0\ \mathit{ok} \qquad \Delta;\mathtt{\overline{x}:\overline{T}, this:C<\overline{X}>} \vdash \mathtt{e}_0 \in \mathtt{S}_0 \\ \Delta \vdash \mathtt{S}_0 <: \mathtt{T}_0 \qquad CT(\mathtt{C}) = \mathtt{class\ C<\overline{X} \lhd \overline{N}>\ <\overline{X} \lhd \overline{V}>? \lhd D<\overline{S}>\ \{...\}} \\ \mathit{override}(\ \Delta\ ,\ \mathtt{m},\ \mathtt{<\overline{X} \lhd \overline{V}>?\ D<\overline{S}>},\ \mathtt{<\overline{X} \lhd \overline{R}>?\ <\overline{Y} \lhd \overline{P}>\overline{T} \to T}_0)\end{array}}{\mathtt{<\overline{X} \lhd \overline{R}>?\ <\overline{Y} \lhd \overline{P}>T}_0\ \mathtt{m\ (\overline{T}\ \overline{x})\ \{\ \uparrow e}_0\mathtt{;\ \}\ OK\ IN\ C<\overline{X} \lhd \overline{N}>}} \qquad \text{(T-METHOD)}$$

Class typing:

$$\frac{\begin{array}{c}\Delta_1 = \mathtt{\overline{X} <: \overline{N}} \quad \Delta_1 \vdash \mathtt{\overline{R} <: \overline{N}} \quad \Delta_1 \vdash \mathtt{\overline{N}}\ \mathit{ok} \quad \Delta_2 = \mathtt{\overline{X} <: \overline{R}} \\ \Delta_2 \vdash \mathtt{\overline{R}, D<\overline{S}>}\ \mathit{ok} \qquad \Delta_1 \vdash \mathtt{\overline{T}}\ \mathit{ok} \quad \mathtt{\overline{M}\ OK\ IN\ C<\overline{X} \lhd \overline{N}>}\end{array}}{\mathtt{class\ C<\overline{X} \lhd \overline{N}>\ <\overline{X} \lhd \overline{R}>? \lhd D<\overline{S}>\ \{\ \overline{T}\ \overline{f}\mathtt{;}\ \overline{M}\ \}\ OK}} \qquad \text{(T-CLASS)}$$

Well-formed types:

$$\Delta \vdash \mathtt{Object}\ \mathit{ok} \qquad \text{(WF-OBJECT)}$$

$$\frac{\mathtt{X} \in \mathit{dom}(\Delta)}{\Delta \vdash \mathtt{X}\ \mathit{ok}} \qquad \text{(WF-VAR)}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<\overline{X} \lhd \overline{N}>\ <\overline{X} \lhd \overline{R}>? \lhd D<\overline{S}>\ \{...\}} \quad \Delta \vdash \mathtt{\overline{T}}\ \mathit{ok} \quad \Delta \vdash \mathtt{\overline{T} <: [\overline{T}/\overline{X}]\overline{N}}}{\Delta \vdash \mathtt{C<\overline{vT}>}\ \mathit{ok}} \qquad \text{(WF-CLASS)}$$

Figure 3: FCJ: Typing Rules

Subtyping:

$$\frac{}{\Delta \vdash \mathtt{T} <: \mathtt{T}} \quad \text{(S-REFL)}$$

$$\frac{\Delta \vdash \mathtt{S} <: \mathtt{T} \quad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{S} <: \mathtt{U}} \quad \text{(S-TRANS)}$$

$$\frac{\Delta(\mathtt{X}) = (+, \mathtt{T})}{\Delta \vdash \mathtt{X} <: \mathtt{T}} \quad \text{(S-UBOUND)}$$

$$\frac{\Delta(\mathtt{X}) = (-, \mathtt{T})}{\Delta \vdash \mathtt{T} <: \mathtt{X}} \quad \text{(S-LBOUND)}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C}\langle \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} \rangle\ \langle \overline{\mathtt{X}} \lhd \overline{\mathtt{R}} \rangle ? \lhd \mathtt{D}\langle \overline{\mathtt{S}} \rangle\ \{\ldots\} \qquad \Delta \vdash \mathtt{C}\langle \overline{\mathtt{T}} \rangle \Uparrow^{\Delta'} \mathtt{C}\langle \overline{\mathtt{U}} \rangle \qquad \Delta, \Delta' \vdash \overline{\mathtt{U}} <: [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{R}} \qquad ([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{D}\langle \overline{\mathtt{S}} \rangle) \Downarrow_{\Delta'} \mathtt{T}}{\Delta \vdash \mathtt{C}\langle \overline{\mathtt{T}} \rangle <: \mathtt{T}} \quad \text{(S-CLASS)}$$

$$\frac{\overline{\mathtt{v}} \leq \overline{\mathtt{w}} \quad \text{if } \mathtt{w}_i \leq \mathtt{-}, \text{ then } \Delta \vdash \mathtt{T}_i <: \mathtt{S}_i \quad \text{if } \mathtt{w}_i \leq \mathtt{+}, \text{ then } \Delta \vdash \mathtt{S}_i <: \mathtt{T}_i}{\Delta \vdash \mathtt{C}\langle \overline{\mathtt{v}}\overline{\mathtt{S}} \rangle <: \mathtt{C}\langle \overline{\mathtt{w}}\overline{\mathtt{T}} \rangle} \quad \text{(S-VAR)}$$

Figure 4: FCJ: Subtyping Rules

To enhance the understanding of our type system, we first highlight some important additions to FGJ made by Igarashi and Viroli regarding variance. We then delve into the rules and definitions specifically changed for the inclusion of type-conditionals in cJ.

Background on Variance Formalism. The two most important additions of the Igarashi and Viroli system over FGJ are the concepts of "open" and "close" (Figure 6). Before any type is used (i.e., for field or method invocation, or in a subtyping judgment), it must be "opened" first. Opening a type means that we introduce a fresh type variable for each co- or contra-variantly defined type. For example, before we can check the validity of invoking method `m` in type `C<+T>`, we must open this type by introducing a fresh type variable `X` into Δ, where $\Delta \vdash \mathtt{X} <: \mathtt{T}$. To look for method `m` in `C<+T>` now means to look for `m` in `C<X>`. If `T` occurs anywhere in `m`'s type, it is replaced by `X`, as well.

This "opening" conveniently disallows illegal accesses of methods or fields that we informally described in Section 2.4. For example, suppose that in the definition of class `C<X>`, we have method `D m (X x) { ... }`. The type parameter `X` appears in a contra-variant position—as method `m`'s argument type. This means that any co-variantly instantiated type `C<+T>` should not be able to invoke method `m`. This is indeed the case in this formalism: we first open `C<+T>` to `C<Y>`, where

Method type lookup:

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C}\langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\rangle\ \langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{Q}}\rangle?\ \triangleleft\mathtt{D}\langle\overline{\mathtt{S}}\rangle\{\ldots\ \overline{\mathtt{M}}\} \quad \langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{R}}\rangle?\ \langle\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\rangle\mathtt{U}_0\ \mathtt{m}\ (\overline{\mathtt{U}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e};\}\in\overline{\mathtt{M}} \quad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{R}}}{mtype(\Delta,\ \mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\langle\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\rangle\overline{\mathtt{U}}\to\mathtt{U}_0)} \quad \text{(MT-CLASS)}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C}\langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\rangle\ \langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{Q}}\rangle?\ \triangleleft\mathtt{D}\langle\overline{\mathtt{S}}\rangle\{\ldots\ \overline{\mathtt{M}}\} \quad \mathtt{m}\notin\overline{\mathtt{M}} \quad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}}}{mtype(\Delta,\ \mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = mtype(\Delta,\ \mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D}\langle\overline{\mathtt{S}}\rangle)} \quad \text{(MT-SUPER)}$$

Method body lookup:

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C}\langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\rangle\ \langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{Q}}\rangle?\ \triangleleft\mathtt{D}\langle\overline{\mathtt{S}}\rangle\{\ldots\ \overline{\mathtt{M}}\} \quad \langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{R}}\rangle?\ \langle\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\rangle\mathtt{U}_0\ \mathtt{m}\ (\overline{\mathtt{U}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e};\}\in\overline{\mathtt{M}} \quad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{R}}}{mbody(\Delta,\ \mathtt{m}\langle\overline{\mathtt{W}}\rangle,\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = (\overline{\mathtt{x}}, [\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e})} \quad \text{(MB-CLASS)}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C}\langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\rangle\ \langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{Q}}\rangle?\ \triangleleft\mathtt{D}\langle\overline{\mathtt{S}}\rangle\{\ldots\ \overline{\mathtt{M}}\} \quad \mathtt{m}\notin\overline{\mathtt{M}} \quad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}}}{mbody(\Delta,\ \mathtt{m}\langle\overline{\mathtt{W}}\rangle,\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = mbody(\Delta,\ \mathtt{m}\langle\overline{\mathtt{W}}\rangle,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D}\langle\overline{\mathtt{S}}\rangle)} \quad \text{(MB-SUPER)}$$

Field lookup:

$$fields(\Delta, \mathtt{Object}) = \bullet \quad \text{(F-OBJECT)}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C}\langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\rangle\ \langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{R}}\rangle?\ \triangleleft\mathtt{D}\langle\overline{\mathtt{U}}\rangle\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \} \quad fields(\Delta,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D}\langle\overline{\mathtt{U}}\rangle) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} \quad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{R}}}{fields(\Delta,\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = \overline{\mathtt{D}}\ \overline{\mathtt{g}},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\ \overline{\mathtt{f}}} \quad \text{(F-CLASS-S)}$$

$$\frac{\Delta \not\vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{R}} \quad CT(\mathtt{C}) = \mathtt{class\ C}\langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\rangle\ \langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{R}}\rangle?\triangleleft\mathtt{D}\langle\overline{\mathtt{U}}\rangle\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \}}{fields(\Delta,\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\ \overline{\mathtt{f}}} \quad \text{(F-CLASS)}$$

Unconditional method type lookup:

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C}\langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\rangle\ \langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{Q}}\rangle?\triangleleft\mathtt{D}\langle\overline{\mathtt{S}}\rangle\{\ldots\ \overline{\mathtt{M}}\} \quad \langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{R}}\rangle?\langle\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\rangle\mathtt{U}_0\ \mathtt{m}\ (\overline{\mathtt{U}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e};\}\in\overline{\mathtt{M}} \quad \Delta' = [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{X}} <: \overline{\mathtt{R}})}{mtype_{uc}(\mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = (\Delta',\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\langle\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\rangle\overline{\mathtt{U}}\to\mathtt{U}_0))} \quad (\mathrm{MT}_{\mathrm{UC}}\text{-CLASS})$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C}\langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\rangle\ \langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{Q}}\rangle?\triangleleft\mathtt{D}\langle\overline{\mathtt{S}}\rangle\{\ldots\ \overline{\mathtt{M}}\} \quad \mathtt{m}\notin\overline{\mathtt{M}} \quad mtype_{uc}(\mathtt{m},\ \mathtt{D}\langle\overline{\mathtt{S}}\rangle) = (\Delta, \langle\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\rangle\overline{\mathtt{U}}\to\mathtt{U}_0) \quad \Delta' = \Delta, [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{X}} <: \overline{\mathtt{Q}})}{mtype_{uc}(\mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = (\Delta',\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\langle\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\rangle\overline{\mathtt{U}}\to\mathtt{U}_0))} \quad (\mathrm{MT}_{\mathrm{UC}}\text{-SUPER})$$

Valid method overriding:

$$\frac{mtype_{uc}(\mathtt{m},\ \mathtt{D}\langle\overline{\mathtt{S}}\rangle) = (\Delta',\ \langle\overline{\mathtt{Z}}\triangleleft\overline{\mathtt{Q}}\rangle\overline{\mathtt{T}}\to\mathtt{T}_0)\ \text{ implies } [\overline{\mathtt{Y}}/\overline{\mathtt{Z}}](\overline{\mathtt{T}}, \mathtt{T}_0, \overline{\mathtt{Q}}) = (\overline{\mathtt{U}}, \mathtt{U}_0, \overline{\mathtt{P}}) \ \text{ and } \ \Delta, \Delta', \overline{\mathtt{X}} <: \overline{\mathtt{R}} \vdash \overline{\mathtt{X}} <: \overline{\mathtt{H}}}{override(\Delta,\ \mathtt{m},\ \langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{R}}\rangle?\ \mathtt{D}\langle\overline{\mathtt{S}}\rangle,\ \langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{H}}\rangle?\langle\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\rangle\overline{\mathtt{U}}\to\mathtt{U}_0)}$$

Bound of type:

$$bound_\Delta(\mathtt{N}) = \mathtt{N} \qquad \frac{\Delta(\mathtt{X}) = (+, \mathtt{S})}{bound_\Delta(\mathtt{X}) = bound_\Delta(\mathtt{S})}$$

Figure 5: FCJ: Auxiliary definitions

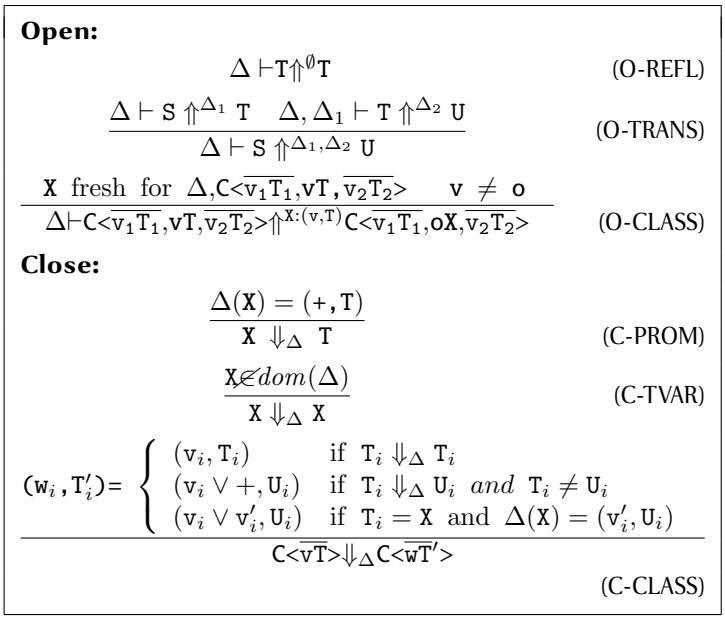

Figure 6: FCJ: Open and Close Rules

$\Delta \vdash \mathtt{Y} <: \mathtt{T}$. We then check that any invocation of `m` passes in an argument of some subtype of `Y`. However, `Y` is simply a type variable with an *upper* bound of `T`. According to the subtyping rules in Figure 4, no type can be deemed a subtype of `Y` (except `Y` itself, which is not available before the opening, and thus cannot be the type of any argument passed to `m`). Thus, no invocation of `m` on an expression of type `C<+T>` can be well-typed. Similarly, had the type parameter `X` appeared in a co-variant position in `C<X>`, e.g., as the return type of a method, an expression with the contra-variantly instantiated type `C<-T>` would not have been able to invoke that method.

Since "open" introduces new type variables into the type environment, it is always paired with a "close" operation, where the newly introduced type variable is closed down to its bound and removed from the type environment. Closing also re-introduces variance annotations, using a conservative combination of the variance annotations of the current use context (i.e., the type being closed) and the surrounding definition context used for the preceding "open".

Auxiliary Definitions. Function $mtype(\Delta,\ \mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)$ returns the signature of method `m`, in type $\mathtt{C}\langle\overline{\mathtt{T}}\rangle$, in the form of $\langle\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\rangle\overline{\mathtt{U}} \rightarrow \mathtt{U}_0$. $mtype(\Delta,\ \mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)$ is defined under two rules:

- MT-CLASS says that if method `m` is declared in class $\mathtt{C}\langle\overline{\mathtt{X}}\rangle$ with type-conditional $\langle\overline{\mathtt{X}} \triangleleft \overline{\mathtt{R}}\rangle$?,

and the type-conditional is satisfied by substituting types $\overline{\mathtt{T}}$ for type parameters $\overline{\mathtt{X}}$, i.e., $\Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{R}}$, then $mtype(\Delta,\ \mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)$ is defined to be the types of $\mathtt{m}$ in $\mathtt{C}\langle\overline{\mathtt{X}}\rangle$.

- MT-SUPER covers the condition when method $\mathtt{m}$ is not declared in class $\mathtt{C}\langle\overline{\mathtt{X}}\rangle$ at all. In this case, if the type-conditional for superclass $\mathtt{D}\langle\overline{\mathtt{S}}\rangle$ is satisfied by the substitution $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]$, then $mtype(\Delta,\ \mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)$ is defined as $mtype(\Delta,\ \mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{D}\langle\overline{\mathtt{S}}\rangle))$.

Note that we do not need a case for when $\mathtt{m}$ is declared in $\mathtt{C}\langle\overline{\mathtt{X}}\rangle$, but the type-conditional guarding it is not satisfied by the substitution $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]$. As explained in Section 2.2.2, the type conditions on a subclass method must be weaker than the type conditions guarding the method it overrides in the superclass (this restriction is formalized in the *override* rule, which we explain later in this section). Thus, if method $\mathtt{m}$'s type conditions in $\mathtt{C}\langle\overline{\mathtt{X}}\rangle$ cannot be satisfied by the assumptions in Δ, then its type conditions in the superclass of $\mathtt{C}\langle\overline{\mathtt{X}}\rangle$ cannot possibly be satisfied. There is no need to invoke MT-SUPER in this case.

$mbody(\Delta,\ \mathtt{m}\langle\overline{\mathtt{W}}\rangle,\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)$ returns a pair, $(\overline{\mathtt{x}},\ \mathtt{e})$. $\overline{\mathtt{x}}$ are the parameters of $\mathtt{m}$, and $\mathtt{e}$ is $\mathtt{m}$'s body. $\overline{\mathtt{W}}$ are the actual types inferred for a polymorphic method $\mathtt{m}$. Note $mbody$ is similarly defined under the same two conditions that $mtype$ is.

$fields(\Delta,\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)$ returns a sequence of fields in class $\mathtt{C}\langle\overline{\mathtt{T}}\rangle$. `Object` has no fields. For all other types $\mathtt{C}\langle\overline{\mathtt{T}}\rangle$, $fields(\Delta,\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)$ returns the sequence of fields declared in $\mathtt{C}\langle\overline{\mathtt{T}}\rangle$, and, if the type-conditional guarding $\mathtt{C}\langle\overline{\mathtt{T}}\rangle$'s superclass, $\mathtt{D}\langle\overline{\mathtt{S}}\rangle$, is satisfied by the substitution $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]$, the value of $fields(\Delta,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{D}\langle\overline{\mathtt{S}}\rangle))$ is returned, as well.

The predicate $override(\Delta,\ \mathtt{m},\ \langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{R}}\rangle?\mathtt{D}\langle\overline{\mathtt{S}}\rangle,\ \langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{H}}\rangle?\langle\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\rangle\overline{\mathtt{U}}\rightarrow\mathtt{U}_0)$ judges if a method $\mathtt{m}$, with signature $\langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{H}}\rangle?\langle\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\rangle\overline{\mathtt{U}}\rightarrow\mathtt{U}_0$ may be defined in a class that has a conditional superclass $\mathtt{D}\langle\overline{\mathtt{S}}\rangle$, guarded by condition $\langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{R}}\rangle$. The extra complication in this rule reflects the restrictions described in Section 2.2.2. There are two aspects of our translation to consider. Firstly, recall that our translation scheme erases all type conditionals. After translation, a class $\mathtt{C}\langle\overline{\mathtt{X}}\rangle$ extends its superclass $\mathtt{D}\langle\overline{\mathtt{S}}\rangle$ unconditionally. Consequently, even if a method $\mathtt{m}$ in $\mathtt{C}\langle\overline{\mathtt{X}}\rangle$ is declared under a type-conditional that precludes the condition for the superclass, the type of $\mathtt{m}$ in $\mathtt{C}\langle\overline{\mathtt{X}}\rangle$ still cannot conflict with the type of $\mathtt{m}$ in $\mathtt{D}\langle\overline{\mathtt{S}}\rangle$. Secondly, also recall that if a method in a subclass has the same type signature as a method in a superclass, we require the subclass method to *always*

override the superclass method. This means the type-conditional on the subclass method must be implied by the type-conditional on the superclass method. This requirement ensures that we do not have to dynamically decide whether a class's own method implementation should be invoked or it should call `super.m(...)`.

To reflect these restrictions, $override$ uses the function $mtype_{uc}$ to *unconditionally* recurse up the chain of superclasses to find a method's signature. $mtype_{uc}$(m, C<$\overline{\texttt{T}}$>) returns a pair, (Δ',<$\overline{\texttt{Y}}$◁$\overline{\texttt{P}}$>$\overline{\texttt{U}}$→$\texttt{U}_0$). Δ' contains subtyping assumptions that must be satisfied for method m to have type <$\overline{\texttt{Y}}$◁$\overline{\texttt{P}}$>$\overline{\texttt{U}}$→$\texttt{U}_0$. The second part of the pair is m's signature as defined in the closest superclass up the unconditional chain of inheritance.

The $override$ rule uses the *implies* notation (as in FGJ [47]) to indicate that the restrictions represented by the consequent of the *implies* need to be satisfied only if the antecedent is true. In this case, the antecedent is: $mtype_{uc}$(m, D<$\overline{\texttt{S}}$>)=(Δ',<$\overline{\texttt{Z}}$◁$\overline{\texttt{Q}}$>$\overline{\texttt{T}}$→$\texttt{T}_0$). This means that method m is defined in either D<$\overline{\texttt{S}}$> or some conditional superclass of D<$\overline{\texttt{S}}$>. If this antecedent is true, then the parameter, return types, and bounds on the inferred types must be the same in the subclass as they are in the conditional superclass. It must also be true that given all conditions guarding the chain of conditional superclasses $mtype_{uc}$ recursed through to find m, and the condition guarding m itself, the condition guarding the subclass method is true, as well. This is checked by augmenting Δ with Δ' and $\overline{\texttt{X}}$<:$\overline{\texttt{R}}$, and requiring that these are sufficient to show $\overline{\texttt{X}}$<:$\overline{\texttt{H}}$.

Note that the definition of $override$ is dependent on the subtyping rules defined in Figure 4. Depending on the specific algorithm implementing our declarative subtyping rules, it is possible that the subtyping condition guarding the overriding method, $\overline{\texttt{X}}$<:$\overline{\texttt{H}}$, cannot be shown to be true using the assumptions in Δ, Δ', and $\overline{\texttt{X}}$<:$\overline{\texttt{R}}$. Consequently, certain valid overriding methods cannot be proven so. To see this concretely, let Δ=Foo<X><:Baz, where Foo is defined as: `class Foo<X> <X◁Bar>?◁Baz {...}`. If we want to show that $\Delta \vdash$X<:Bar, we need to deconstruct type Foo<X>, and infer from Foo<X><:Baz that X<:Bar must be true, as well. Our current implementation does not deconstruct types to do such inference. Subtyping assumptions thrown into Δ' by $mtype_{\mathrm{uc}}$ are only effective if the type variables $\overline{\texttt{X}}$ are not buried inside of constructed types, such as Foo<X>. We are currently working on a decidable algorithm for deconstructing types to get more precise subtyping assumptions. Note that this is a standard point of trade-off: a

too powerful reasoning procedure may well end up being undecidable. A conservative algorithm, on the other hand, will reject some programs because of its inability to establish the conditions for their soundness. The latter is typically preferable in practice, since, in this setting, troublesome programs tend to be highly contrived.

The two rules for unconditional field lookup are only used in proving the correctness of erasure, using the erasure rules detailed in the accompanying technical report [45]. They are included for completeness of the auxiliary functions.

Type Rules. Most of the rules presented in Figure 3 are the same as their variance-based FGJ counterparts. We now go through the ones particular to the type-conditional extensions in FCJ.

T-FIELD and T-INVK: these rules define when a field reference or a method invocation, respectively, is well-typed. Even though they look identical to their variance-based FGJ counterparts, they use functions *fields* and *mtype*, which fully encapsulate the lookup of conditional supertypes and conditional methods, as previously explained.

T-METHOD: The interesting change to the T-METHOD rule from its counterpart in variance-based FGJ is that the environment Δ (under which the return and parameter types, as well as the body of the method, expression e_0, must be well-typed) is augmented with the type bound $\overline{\mathtt{X}}<:\overline{\mathtt{R}}$, which is the type-conditional under which the method is declared. Intuitively, this says that if a method is declared under type-conditional $<\overline{\mathtt{X}}\lhd\overline{\mathtt{R}}>?$, then in the scope of the body of the method it can be assumed that the type environment Δ supports this bound.

T-CLASS: Note that the conditional superclass $\mathtt{D}<\overline{\mathtt{S}}>$ needs to be proved well-typed under Δ augmented with the type-conditional condition guarding it.

2.7 Proof of Soundness

We prove the soundness of our type system by proving subject reduction and progress properties [95]. We state the reductions rules (Figure 7) and theorems here. Interested readers can find the full version of the proofs in Appendix A.

Theorem 1 [Subject Reduction]: If $\Delta;\Gamma\vdash\mathsf{e}\in\mathtt{T}$ and $\mathsf{e}\longrightarrow\mathsf{e}'$, where Δ has non-variable bounds, then $\Delta;\Gamma\vdash\mathsf{e}'\in\mathtt{S}$ and $\Delta\vdash\mathtt{S}<:\mathtt{T}$ for some $\mathtt{S}$.

Reduction Rules:

$$\frac{\mathit{fields}(\emptyset,\ \texttt{C<}\overline{\texttt{T}}\texttt{>})=\overline{\texttt{U}}\ \overline{\texttt{f}}}{\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).f}_i \longrightarrow \texttt{e}_i} \quad \text{(R-FIELD)}$$

$$\frac{\mathit{mbody}(\emptyset,\ \texttt{m<}\overline{\texttt{V}}\texttt{>},\ \texttt{C<}\overline{\texttt{T}}\texttt{>})=(\overline{\texttt{x}},\texttt{e}_0)}{\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).<}\overline{\texttt{V}}\texttt{>m(}\overline{\texttt{d}}\texttt{)} \longrightarrow [\overline{\texttt{d}}/\overline{\texttt{x}},\ \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)/this]e}_0} \quad \text{(R-INVK)}$$

$$\frac{\emptyset \vdash \texttt{C<}\overline{\texttt{T}}\texttt{><:T}}{\texttt{(T)new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \longrightarrow \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}} \quad \text{(R-CAST)}$$

$$\frac{\texttt{e}_0 \longrightarrow \texttt{e}_0'}{\texttt{e}_0\texttt{.f} \longrightarrow \texttt{e}_0'\texttt{.f}} \quad \text{(RC-FIELD)}$$

$$\frac{\texttt{e}_0 \longrightarrow \texttt{e}_0'}{\texttt{e}_0\texttt{.<}\overline{\texttt{V}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} \longrightarrow \texttt{e}_0'\texttt{.<}\overline{\texttt{V}}\texttt{>m(}\overline{\texttt{e}}\texttt{)}} \quad \text{(RC-INV-RECV)}$$

$$\frac{\texttt{e}_i \longrightarrow \texttt{e}_i'}{\texttt{e}_0\texttt{.<}\overline{\texttt{V}}\texttt{>m(...,e}_i\texttt{,...)} \longrightarrow \texttt{e}_0\texttt{.<}\overline{\texttt{V}}\texttt{>m(...,e}_i'\texttt{,...)}} \quad \text{(RC-INV-ARG)}$$

$$\frac{\texttt{e}_i \longrightarrow \texttt{e}_i'}{\texttt{new C<}\overline{\texttt{T}}\texttt{>(...,e}_i\texttt{,...)} \longrightarrow \texttt{new C<}\overline{\texttt{T}}\texttt{>(...,e}_i'\texttt{,...)}} \quad \text{(RC-NEW-ARG)}$$

$$\frac{\texttt{e}_0 \longrightarrow \texttt{e}_0'}{\texttt{(T)e}_0 \longrightarrow \texttt{(T)e}_0'} \quad \text{(RC-CAST)}$$

Figure 7: FCJ: Reduction Rules

Theorem 2 [Progress]: Let e be a well-typed expression.

1. If e has $\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).f}$ as a subexpression, then $\mathit{fields}(\emptyset,\ \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{U}}\ \overline{\texttt{f}}$, and $\texttt{f}=\texttt{f}_i$.

2. If e has $\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).m(}\overline{\texttt{d}}\texttt{)}$ as a subexpression, then $\mathit{mbody}(\emptyset,\ \texttt{m},\ \texttt{C<}\overline{\texttt{T}}\texttt{>}) = (\overline{\texttt{x}}, \texttt{e}_0)$ and $|\overline{\texttt{x}}| = |\overline{\texttt{d}}|$.

Theorem 3 [Type Soundness]: If $\emptyset;\emptyset \vdash \texttt{e} \in \texttt{T}$ and $\texttt{e} \longrightarrow^* \texttt{e}'$ being a normal form, then $\texttt{e}'$ is either a value v such that $\emptyset;\emptyset \vdash v \in \texttt{S}$ and $\emptyset \vdash \texttt{S<:T}$ for some S, or an expression that includes $\texttt{(T)new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}$ where $\emptyset \not\vdash \texttt{C<}\overline{\texttt{T}}\texttt{><:T}$.

Chapter 3

Morphing: Structurally Shaping Code in the Image of Other Code

This chapter presents a powerful abstraction mechanism called *morphing*. Morphing allows code to be declared by statically iterating over the structural members of code that it would be composed with. As such, the structure of morphing code is not statically determined, but automatically specialized to reflect the structure of its client in composition. We discuss morphing through MorphJ—a language that extends an Object-Oriented base language, Java, with a morphing mechanism. MorphJ extends the notion of type-genericity in Java so that the structure of a generic class, i.e., its methods and fields, can vary according to the structure of the types it is composed with.

Morphing allows programmers to express common programming patterns in a highly reusable way that is otherwise not supported by conventional techniques. For instance, morphing allows one to implement generic proxies (i.e., classes that can be parameterized with another class and export the same public methods as that class); default implementations (e.g., a generic type with default behavior for every method, configurable for any interface); semantic extensions (e.g., specialized behavior for methods that declare a certain annotation); and more.

Morphing's hallmark feature is that, despite its emphasis on generality and greater reusability, it still allows modular type checking: A morphing class can be type-checked independently of its uses in compositions. Thus, errors in generic code are detected before composition time—an invaluable feature for highly reusable components that will be statically composed by other programmers.

3.1 Introduction

Morphing targets functionality that needs to iteratively adapt its implementation to the structure of its client in composition. For instance, consider the following task: how would you write a piece of code that, given *any* class `X`, returns another class that contains the exact same methods as `X`, but logs each method's return value? That is, the code is a reusable representation of the functionality

"logging", and abstracts over the exact structure of the class (i.e., its declared methods) it may be applied to.

Capturing this level of abstraction has traditionally been only possible with techniques such as meta-object protocols (MOPs) [58], aspect-oriented programming (AOP) [61], or various forms of meta-programming (e.g., reflection and ad-hoc program generation using quote primitives, string templates, or bytecode engineering). While just about any programmer can write a method that logs its return value, the techniques listed above can require steep learning curves or suffer from poor integration with the base language. More importantly, these techniques do not offer any modular safety guarantee: There is no guarantee that a piece of code would always be well-typed regardless of its uses in specific compositions.

Morphing offers the power to express structurally abstract code, while maintaining modular safety guarantees. We demonstrate the power of morphing through MorphJ—a reference language that demonstrates what we consider the desired expressiveness and safety features of an advanced morphing language. MorphJ morphing can express highly reusable object-oriented components (i.e., generic classes) whose exact members (e.g., fields and methods) are not known until the component is parameterized with concrete types. For instance, the following MorphJ class implements the "logging" extension described above:

```
1  class Logging<class X> extends X {
2    <R,Y*>[meth]for(public R meth (Y) : X.methods)
3    public R meth (Y a) {
4      R r = super.meth(a);
5      System.out.println("Returned: " + r);
6      return r;
7    }
8  }
```

MorphJ allows class `Logging` to be declared as a subclass of its type parameter, `X`. The body of `Logging` is defined by static iteration (using the `for` statement, on line 2) over all methods of `X` that match the pattern "`public R meth(Y)`". `Y`, `R`, and `meth` are pattern-matching variables. `Y` and `R` can match any non-`void` type, and `meth` can match any identifier. Additionally, the `*` symbol following the declaration of `Y` indicates that `Y` can match a vector of any number of types (including zero). That is, the above pattern matches all `public` methods that return any non-`void` type. The pattern-matching variables are also used in the declaration of `Logging`'s methods: for

each method of the type parameter `X` matched, `Logging` declares a method with the same name and type signature. (This does not have to be the case, as shown later.) Thus, the exact methods of class `Logging` are not determined until it is composed, or type-instantiated, with a concrete type. For instance, `Logging<java.lang.Object>` has methods `equals`, `hashCode`, and `toString`: these are the only `public`, non-`void`-returning methods of `java.lang.Object`.

The above example illustrates the basic feature of MorphJ: code can be declared by iterating over members of a class or interface matching a pattern. MorphJ also allows even more expressive iteration conditions using *nested patterns*. For instance, the following code iterates over the `public`, non-`void`-returning methods of `X`, *such that there is also some method in* `X` *with the same name, taking no argument*:

```
<R,Y*>[meth]for ( public R meth(Y) : X.methods; some R meth() : X.methods ) ...
```

The *positive nested pattern*, "`some R meth() : X.methods`", places a positive existential condition on the outer, primary pattern. A negative existential condition can be similarly imposed on the primary pattern by annotating the nested pattern with the keyword `no`, instead of `some`, making it a *negative nested pattern*.

"Reflective" program pattern matching and transformation, as in the "logging" example, are not new. Several pattern matching languages have been proposed in prior literature (e.g., [11–13, 91]) and most of them specify transformations based on some intermediate program representation (e.g., abstract syntax trees) although the patterns resemble regular program syntax. Compared to such work, MorphJ makes the following contributions:

- MorphJ makes reflective transformation functionality a natural extension of Java generics. For instance, our above example class `Logging` appears to the programmer as a regular class, rather than as a separate kind of entity, such as a "transformation". Using a generic class is a matter of simple type-instantiation, which produces a regular Java class, such as `Logging<java.lang.Object>`.

- MorphJ's support for nested patterns significantly enhances the expressiveness of pattern-based reflective declarations. Without nested patterns, pattern-based reflective languages are

mostly limited to expressing wrapper-like patterns [35], such as the logging class `Logging<X>`.

- Nested patterns also form the basis for adding more features to the MorphJ language, such as static `if` and static `errorif` statements. The static `if` allows code to exist or not depending on the existence or absence of other members. The static `errorif` acts as a type-cast: it allows the type system to assume the existence or absence of a member, and, if the assumption is violated, a composition-time error occurs. This enables MorphJ to express conditional features like those in Chapter 2, in addition to iterative features.

- The power of morphing is illustrated with real-world applications. We present four smaller examples: a MorphJ reimplementation of Java Collections Framework [5] that replaces 314 lines of hardcoded proxies with 26 lines of highly reusable MorphJ code; two variants of MorphJ generic classes that provide default implementations for methods—either to provide default behavior to all methods of any interface (for testing purposes, for instance), or to provide implementations for unimplemented interface methods, for a combination of any class and any interface; and a generic list implementation that can sort its elements by any `Comparable` field of its elements. We also present a large-scale case study where MorphJ was used to reimplement DSTM2 [41], a Java software transactional library that heavily uses reflection and bytecode engineering (with BCEL [9]) to transform sequential classes into transactional ones. The MorphJ reimplementation replaces 1,484 lines of Java reflection and BCEL library calls with 576 lines of MorphJ code.

- MorphJ generic classes support separate type checking—a generic class is type-checked independently of its type-instantiations, and errors are detected if they can occur with *any* possible type parameter. This is an invaluable property for generic code: It prevents errors that only appear for some type parameters, which the author of the generic class may not have predicted. This problem has been the target of some prior work, such as type-safe reflection [29], compile-time reflection (CTR) [34], and safe program generation [44]. Yet none of these mechanisms offer MorphJ's modular type checking guarantees. For instance, the Genoupe [29] approach has been shown unsafe, as the reasoning depends on properties that can change at runtime; CTR [34] only captures undefined variable and type incompatibility

errors, does not offer a formal system or proof of soundness, and has limited expressiveness compared to MorphJ (especially with respect to nested patterns and the ability to match arbitrary numbers of method arguments); SafeGen [44] has no soundness proof and relies on the capabilities of an automatic theorem prover—an unpredictable and unfriendly process from the programmer's perspective. We formalize our type system and prove it sound. We offer a decidable algorithm for type-checking.

An Illustration of Separate Type-Checking For an example of separate type checking, consider a "buggy" generic class:

```
class CallWithMax<class X> extends X {
  <A>[m]for(public int m (A) : X.methods)
  int meth(A a1, A a2) {
    if (a1.compareTo(a2) > 0)
      return super.meth(a1);
    else
      return super.meth(a2);
  }
}
```

The intent is that class `CallWithMax<C>`, for some `C`, imitates the interface of `C` for all single-argument methods that return `int`, yet adds an extra formal parameter to each method. The corresponding method of `C` is then called with the greater of the two arguments passed to the method in `CallWithMax<C>`. It is easy to define, use, and deploy such a generic transformation without realizing that it is not always well-typed: not all types `A` will support the `compareTo` method. MorphJ detects such errors when compiling the above code, independently of specific type-instantiations. In this case, the fix is to strengthen the pattern with the constraint that `A` must be a subtype of `Comparable<A>`:

```
<A extends Comparable<A>>[m]for(public int m (A) : X.methods)
```

Additionally, the above code has an even more insidious error. The generated methods in `CallWithMax<C>` are not guaranteed to correctly override the methods in its superclass, `C`. For instance, if `C` contains two methods, `int foo(int)` and `String foo(int,int)`, then the latter will be improperly overridden by the generated method `int foo(int,int)` in `CallWithMax<C>` (which has the same argument types but an incompatible return type). MorphJ statically catches this

error. This is an instance of the complexity of MorphJ's modular type checking when dealing with unknown entities.

In the remainder of the chapter, Section 3.2 introduces basic MorphJ language features; Section 3.3 gives real-world examples in which basic MorphJ features increases the modularity and reusability of software components; Section 3.4 introduces advanced MorphJ features such as nested patterns; Section 3.5 provide two real-world examples using nested patterns. Section 3.6 gives a gentle and casual overview of MorphJ's type system, while Section 3.7 formalizes a core subset of MorphJ and presents the type rules. The proofs of soundness for the formal type system are presented in Appendix B. We briefly discuss the implementation of MorphJ in Section 3.8.

3.2 Basic MorphJ Features

MorphJ adds to Java the ability to include *reflective iteration blocks* inside a class or interface declaration. The purpose of a reflective iteration block is to *statically iterate* over a certain subset of a type's methods or fields, and produce a declaration or statement for each element in the iterator. Static iteration means that no runtime reflection exists in compiled MorphJ programs. All declarations or statements within a reflective block are "generated" at compile-time.

A reflective iteration block (or reflective block) has similar syntax to the existing `for` iterator construct in Java. There are two main components to a reflective block: the iterator definition, and the code block for each iteration. The following is a MorphJ class declaration with a very simple reflective block:

```
class C<T> {
  for ( static int foo () : T.methods ) {|
    public String foo () { return String.valueOf(T.foo()); }
  |}
}
```

We overload the keyword `for` for static iteration. The iterator definition immediately follows `for`, delimited by parentheses. This defines the set of elements for iteration, which we call the *reflective range* (or just *range*) of the iterator. The iterator definition has the basic format "*pattern* : *reflection set*". The *reflection set* is defined by applying the `.methods` or `.fields` keywords to a type, designating all methods or fields of that type. The *pattern* is either a method or field signature pattern, used to filter out elements from the reflection set. Only elements that match

the pattern belong in the reflective range. In the example above, the reflective range contains only `static` methods of type `T`, with name `foo`, no argument, and return type `int`.

The second component of a reflective block is delimited by `{|...|}`, and contains either method or field declarations or a block of statements. The reflective block is itself syntactically a declaration or block of statements, but we prevent reflective blocks from nesting. In case of a single declaration (as in most examples in this paper), the delimiters can be dropped. The declarations or statements are "generated", once for each element in the reflective range of the block. In the example above, a method `public String foo() { ... }` is declared for each element in the reflective range. Thus, if `T` has a method `foo` matching the pattern `static int foo()`, a method `public String foo()` exists for class `C<T>`, as well.

The reflective block in the previous example is rather boring. Its reflective range contains at most one method, and we know statically the type and name of that method. For more flexible patterns, we can introduce type and name variables for pattern matching. Pattern matching type and name variables are defined right before the `for` keyword. They are only visible within that reflective block, and can be used as regular types and names. For example:

```
class C<T> {
  T t;
  C(T t) { this.t = t; }

  <A>[m] for (int m (A) : T.methods )
  int m (A a) { return t.m(a); }
}
```

The above pattern matches methods of *any* name that take one argument of *any* type and return `int`. The matching of multiple names and types is done by introducing a type variable, `A`, and a name variable, `m`. Name variables match *any* identifier and are introduced by enclosing them in `[...]`. The syntax for introducing pattern matching type variables extends that for declaring type parameters for generic Java classes: new type variables are enclosed in `<...>`. We can give type variable `A` one or more bounds (e.g., `<A extends Foo & Bar>`), and the bounds can contain `A` itself (e.g., `<A extends Comparable<A>>`). Multiple type variables can be introduced, as well: `<A extends Foo,B extends Bar>`. In addition to the Java generics syntax, we can annotate a type parameter with keywords `class` or `interface`. For instance `<interface A>` declares a type

parameter `A` that can only match an interface type. (This extension also applies to non-pattern-matching type parameters, in which case `A` can only be instantiated with an interface.) A semantic difference between pattern matching type parameters and type parameters in Java generics is that a pattern matching type parameter is not required to be a non-primitive type. In fact, without any declared bounds or `class`/`interface` keyword, `A` can match any type that is not `void`—this includes primitive types such as `int`, `boolean`, etc. To declare a type variable that only matches non-primitive types, one can write `<A extends Object>`.

The type and name variables declared for the reflective block can be used as regular types and names inside the block. In the example above, a method is declared for each method in the reflective range, and each declaration has the same name and argument types as the method that is the current element in the iteration. The body of the method calls method `m` on a variable of type `T`—whatever the value of `m` is for that iteration, this is the method being invoked.

Often, a user does not care (or know) how many arguments a method takes. It is only important to be able to faithfully replicate argument types inside the reflective block. We provide a special syntax for matching *any* number of types: a `*` suffix on the pattern matching type variable definition. For instance, if a pattern matching type variable is declared as `<A*>`, then `String m (A)` is a method pattern that matches any method returning `String`, no matter how many arguments it takes (including zero arguments), and no matter what the argument types are. Even though `A*` is technically a vector of types, it can only be used as a single entity inside of the reflective block. MorphJ provides no facility for iterating over the vector of types matching `A`. This relieves us from having to deal with issues of order or length.

MorphJ also offers the ability to construct *new* names from a name variable, by prefixing the variable with a constant. MorphJ provides the construct `#` for this purpose. To prefix a name variable `f` with the static name `get`, the user writes `get#f`. Note that `get` cannot be another name variable. Creating names out of name variables can cause possible naming conflicts. In later sections, we discuss in detail how the MorphJ type system ensures that the resulting identifiers are unique. MorphJ also offers the ability to create a string out of a name variable (i.e., to use the name of the method or field that the variable currently matches as a string) via the syntax *`var`*`.name`. The example below demonstrates these features:

```
class C<T> {
  T t;
  C(T t) { this.t = t; }

  <R,A*>[m] for (public R m (A) : T.methods )
  R delegate#m (A a) {
    System.out.println("Calling method "+ m.name + " on "+ t.toString());
    return t.m(a);
  }
}
```

The above example shows a simple proxy class that declares methods that mimic the (non-`void`-returning) `public` methods of its type parameter. Declared method names are the original method names prefixed by the constant name `delegate`. Declared methods call the corresponding original methods after logging the call.

In addition to the above features, MorphJ also allows matching arbitrary modifiers (e.g., `final`, `synchronized` or `transient`), exception clauses, and Java annotations. MorphJ has a set of conventions to handle modifier, exception, and annotation matching so that patterns are not burdened with unnecessary detail—e.g., for most modifiers, a pattern that does not explicitly mention them matches regardless of their presence. We do not elaborate further on these aspects of the language, as they represent merely engineering conveniences and are orthogonal to the main MorphJ insights: the morphing language features, combined with a modular type-checking approach.

3.3 Applications Using Basic Morphing Features

Even with the basic features introduced in the previous section, MorphJ opens the door for expressing a large number of useful idioms in a general, reusable way. We next show three applications using MorphJ that demonstrate the power of its structural abstraction mechanism.

3.3.1 Generic Synchronization Proxy

The Java Collections Framework (JCF) [5] defines a number of synchronization proxy classes for its main data structure interfaces. For instance, `SynchronizedCollection<E>`, whose definition is shown in Figure 8(a), is a synchronization proxy for `Collection<E>`. For each method of `Collection<E>`, `SynchronizedCollection<E>` defines a method with the same signature.

Within each method body, a mutex is first acquired, and the call is delegated to the underlying Collection<E> object.

The definition of SynchronizedCollection<E> exhibits two levels of structural redundancy. At the method level, every method shares the exact same structure, with variations only in the method a call is delegated to, and the arguments used in the delegation. Another level of redundancy exists at the class level. The code for SynchronizedCollection<E> represents a fixed composition of the "synchronization" behavior with the data structure Collection<E>. If the synchronization behavior is desired for a different data structure, a new class would need to be defined. For instance, Figure 8(b) is the definition of SynchronizedList<E>, the fixed composition of "synchronization" with List<E>. The definition of SynchronizedList<E> shares the exact same structural pattern as SynchronizedCollection<E>: A method is declared for each method of List<E>, and each method acquires a mutex before delegating the call.

```
class SynchronizedCollection<E>
                  implements Collection<E> {
  Collection<E> c;
  Object mutex;
  ... // constructors that initialize
      // c and mutex

  public int size() {
    synchronized(mutex) {
      return c.size();
    }
  }
  public boolean remove(E e) {
    synchronized(mutex) {
      return c.remove(e);
    }
  }
  ... // repeat for all methods in
      // Collection.
}
```

(a) Synchronization proxy for Collection.

```
class SynchronizedList<E>
            implements List<E> {
  List<E> l;
  Object mutex;
  ... // constructors that initialize
      // l and mutex

  public int size() {
    synchronized(mutex) {
      return l.size();
    }
  }
  public int indexOf (E e) {
    synchronized(mutex) {
      return l.indexOf(o);
    }
  }
  ... // repeat for all methods in
      // List.
}
```

(b) Synchronization proxy for List.

Figure 8: Definition of synchronization proxies in JCF.

In JCF, four more such synchronization proxies are defined: SynchronizedRandomAccessList,

SynchronizedSet, SynchronizedSortedSet, SynchronizedMap, and SynchronizedSortedMap. Using MorphJ, we can implement a highly generic Synchronized<X> class (Figure 9) to replace all Synchronized* classes in JCF. The MorphJ class can also be reused with many more data structures that may desire the synchronization behavior.

The MorphJ class Synchronized is a subtype of its type parameter, X. Additionally, X is required to be an interface, not a class. The body of Synchronized contains two reflective iteration blocks: lines 7-12, and lines 14-19. The block on lines 7-12 iterates over all non-void returning method of the type parameter X . For each matching method in X, a method with the same name and type signature is declared in Synchronized<X>. The body of the method synchronizes on a mutex first, and then delegates the method call to the underlying X object. The reflective iteration block on lines 14-19 declares similar methods for the void returning methods of X. Thus, Synchronized<Collection<E>> is the functional equivalent of the hardcoded proxy SynchronizedCollection<E> we saw in Figure 8.

```
1  public class Synchronized<interface X> implements X {
2    X      me;
3    Object mutex;
4
5   // ... constructor declarations
6
7    <R,A*>[m] for (public R m (A) : X.methods)
8    R m (A args) {
9      synchronized (mutex) {
10       return me.m(args);
11     }
12   }
13
14   <A*>[m] for (public void m (A) : X.methods)
15   void m (A args) {
16     synchronized (mutex) {
17      me.m(args);
18     }
19   }
20 }
```

Figure 9: A highly generic Synchronized<X> class in MorphJ

The full version of the above MorphJ class[1] consists of less than 26 lines of code, replacing more than 314 lines of `Sychronized*` class definitions in the JCF. Compared to classes such as `SynchronizedCollection<E>`, `Synchronized<X>` is more general: It can be composed with any interface needing the "synchronization" behavior. (A similar generic class can be defined to add the behavior to classes.) Furthermore, `Synchronized<X>` is also immune to interface changes in the interface of `X`. When a method is added to or deleted from `X`, or when a method signature changes, the structure of `Synchronized<X>` automatically adapts.

3.3.2 Default Class

Consider a general "default implementation" class that adapts its contents to any interface used as a type parameter. The class implements all methods in the interface, with each method implementation returning a default value. This functionality is particularly useful for testing purposes—e.g., in the context of an application framework (where parts of the hierarchy will be implemented only by the end user), in uses of the Strategy pattern [35] with "neutral" strategies, etc. Figure 10 shows such a MorphJ generic class. (Note that the keyword `throws` in the pattern does not prevent methods with no exceptions from being matched, since `E` is declared to match a possibly-zero length vector of types.)

One can easily think of ways to enrich this example with more complex default behavior, e.g., returning random values or calling constructor methods, instead of using statically determined default values. The essence of the technique, however, is in the iteration over existing methods and special handling of each case of return type. This is only possible because of MorphJ's morphing capabilities. In practice, random testing systems often implement very similar functionality (e.g., [25]) using unsafe run-time reflection. Errors in the reflective or code generating logic are thus not caught until they are triggered by the right combination of inputs, unlike in the MorphJ case.

3.3.3 Sort-by

A common scenario in data structure libraries is that of supporting sorting according to different fields of a type. Although one can use a generic sorting routine that accepts a comparison function,

[1]The full version of the MorphJ class declares constructors that allows the caller of a constructor to set the mutex to either `this`, or an arbitrary object.

```
class DefaultClass<interface I> implements I {
  // For each method returning a non-primitive type, make it return null
  <R extends Object,A*,E*>[m] for( R m (A) throws E : T.methods )
  public R m ( A a ) throws E { return null; }

  // For each method returning a primitive type, return a default value
  <A*,E*>[m]for( int m (A) throws E : T.methods )
  public int m (A a ) throws E { return 0; }

  ... // repeat the above for each primitive return type.

  // For each method returning void, simply do nothing.
  <A*,E*>[m] for ( void m (A) throws E : T.methods )
  public void m (A a) throws E { }
}
```

Figure 10: A generic MorphJ class providing default implementations for every method of its type parameter interface.

the comparison function needs to be custom-written for each field of a type of interest. Instead, a simpler solution is to morph comparison functions based on the fields of a type.

Consider the implementation of an `ArrayList` in Figure 11, modeled after the `ArrayList` class in the Java Collections Framework. `ArrayList<E>` supports a method `sortBy#f` for every field `f` of type `E`. The power of the above code does not have to do with comparing elements of a certain *type* (this can be done with existing Java generics facilities), but with calling the comparison code on the exact fields that need it. For instance, a crucial part that is not expressible with conventional techniques is the code `e1.f.compareTo(e2.f)` (line 11), for any field `f`.

The examples in this section illustrate the power of MorphJ's morphing features: a generic class or interface can be shaped by the properties of the members of the type it is composed with. The morphing approach is similar to reflection, yet all reasoning is performed statically, there is syntax support for easily creating new fields and methods, and type safety is statically guaranteed.

Even more examples from the static reflection or generic aspects literature [29, 34, 44, 60] can be viewed as instances of morphing and can be expressed in MorphJ. For instance, the CTR work [34] allows the user to express a "transform" that iterates over methods of a class that have a `@UnitTestEntry` annotation and generate code to call all such methods while logging the unit

```
1  public class ArrayList<E> extends AbstractList<E>
2    implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
3    ...// ArrayList fields and methods.
4
5    // For each Comparable field of E, declare a sortBy method
6    <F extends Comparable<F>>[f]for(public F f : E.fields)
7    public void sortBy#f () {
8      Collections.sort(this,
9                       new Comparator<E> () {
10                        public int compare(E e1, E e2) {
11                          return e1.f.compareTo(e2.f);
12                        }
13                      });
14   }
15 }
```

Figure 11: An ArrayList implementation providing sorting methods by each comparable field of its element type.

test results. The same example can be expressed in MorphJ, with several advantages over CTR: MorphJ is better integrated in the language, using generic classes instead of a "transform" concept; MorphJ is a more expressive language, e.g., allowing matching methods with an arbitrary number and types of arguments; MorphJ offers much stronger guarantees of modular type safety, as its type system detects the possibility of conflicting definitions (CTR only concentrates on preventing references to undefined entities) and we offer a proof of type soundness (Appendix B).

3.4 Advanced Morphing with Nested Patterns

Though we've shown a number of applications of reflective declaration using simple patterns, there are many tasks that are otherwise perfectly suited for the morphing mechanism, that cannot be expressed using simple patterns. MorphJ thus introduces *nested patterns*. A nested pattern has the same syntactic form as a primary pattern, but is preceded by the keywords "some" (for a *positive* nested pattern) or "no" (for a *negative* nested pattern). Like primary patterns, nested patterns can only reflect over concrete types, or type variables of the generic class. A nested pattern places a condition (nested condition) on each element matched by the primary pattern. An element must be matched by the primary pattern *and* satisfy all nested conditions to be a part of a reflective block's iteration range. We illustrate the need for and uses of nested patterns with the following

examples.

3.4.1 Negative Nested Pattern

A negative nested pattern exerts a condition that is only satisfied if there is *nothing* in the range of elements matched by the pattern.

To see the necessity for negative nested patterns, consider the pesky problem of defining "getter" methods for fields in a class. Currently, programmers deal with this by repeating the same boiler-plate code for each field. This seems to be a task perfectly suited for pattern-based reflective declaration. However, we cannot implement this in a type-safe way using the basic pattern-based feature of MorphJ. Consider the following attempt:

```
class AddGetter<class X> extends X {
  <F>[f] for ( F f : X.fields )
  F get#f () { return super.f; }
}
```

`AddGetter<X>` defines a method `get#f()` for each field `f` in type variable `X`. `get#f` denotes an identifier that begins with the string "`get`", followed by the identifier matched by `f`.[2] However, `AddGetter<X>` is not modularly type safe—we cannot guarantee that no matter what `X` is instantiated with, `AddGetter<X>` is always well-typed! Suppose we have class `C`:

```
class C {
  Meal lunch;  ... // other methods.
  boolean getlunch() { return isNoon() ? true : false; }
}
```

`AddGetter<C>` contains method "`Meal getlunch()`", which incorrectly overrides method "`boolean getlunch()`" in its superclass, `C`. For this reason, the definition of `AddGetter<X>` does not pass MorphJ's type-checking.

This is an error of under-specified requirements. The definition of `AddGetter<X>` should clearly specify that it can only declare method `get#f` for those fields `f` where a conflictingly defined `get#f` *does not already exist* in `X`. What we need is to place a *negative existential condition* on each field matched by the pattern: for all fields `f` of `X` such that method `get#f()` does not already exist in `X`, declare the method `get#f()`.

[2] This MorphJ class only defines getter methods for non-`private` fields: the semantics of pattern "`F f`" without modifier specification is that it matches all non-`private` fields. This is a limitation with the subclassing-as-extension based approach that MorphJ adopts.

Negative nested patterns give us precisely the ability to specify such negative existential conditions. The following is a modularly type safe implementation of `AddGetter<X>`:

```
1  class AddGetter<X> extends X {
2    <F>[f]for( F f : X.fields ; no get#f() : X.methods )
3    F get#f() { return super.f; }
4  }
```

The nested pattern condition on line 2 is only satisfied by those fields `f` of `X` for which there is no method `get#f()` in `X`. (The missing return type in the nested pattern is a MorphJ shorthand for matching both `void` and non-`void` return types.) Observe that this `AddGetter<X>` class will not introduce ill-typed code for *any* `X`. Potentially conflicting method declarations are prevented by the negative nested pattern. A field `f` for which a method `get#f()` already exists in `X` does not satisfy the nested condition, and thus is not in the iteration range of the reflective block.

3.4.2 Positive Nested Pattern

A positive nested pattern exerts a condition that is only satisfied if there is *some* (i.e., at least one) element in the range matched by the pattern.

To see when positive nested patterns may be useful, consider how one would define a class `Pair<X,Y>`, which is a container for objects `x` and `y` of types `X` and `Y`, respectively. For every non-`void` method that `X` and `Y` have in common (i.e., same method name and argument types), `Pair<X,Y>` should declare a method with the same name and argument types, but a return type that is another `Pair`, constructed from the return types of that method in `X` and `Y`.

In order to express this type of functionality, we need a *positive existential condition*: for all methods in `X`, such that another method with the same name and argument types exists in `Y`, declare a method that invokes both and returns a `Pair` of their values.

With a positive nested pattern, we can define the `Pair<X,Y>` class as shown in Figure 12. Methods of `Pair<X,Y>` are defined using the reflective block on lines 5-10. The primary pattern on line 6 matches all non-`void` and non-primitive methods of `X`. For each such method, the positive nested condition on line 7 is only satisfied if a method with the same name and argument types also exists in `Y`. Thus, the primary and nested patterns in this class find precisely all methods that `X` and `Y` share in name and argument types. For each such method, `Pair<X,Y>` declares a method with the same name and argument types, and a body that invokes the corresponding method of

X and Y. The return type is another Pair, constructed from the return values of the invocations. Following the same pattern, the class can be enhanced to also handle methods returning primitive types or void.

```
1  public class Pair<X,Y> {
2     X x; Y y;
3     public Pair(X x, Y y) { this.x = x; this.y = y; }
4
5    <RX extends Object, RY extends Object, A*>[m]
6    for(   public RX m(A) : X.methods ;
7      some public RY m(A) : Y.methods )
8    public Pair<RX,RY> m(A args) {
9       return new Pair<RX,RY>(x.m(args), y.m(args));
10   }
11 }
```

Figure 12: A Pair container class using positive nested patterns.

3.4.3 More Features: if, errorif

Nested patterns enable other powerful language features. The reflective declarations we have seen so far have been iteration-based: a piece of code is declared for each element in the iteration range. MorphJ also supports condition-based reflective declarations and statements. (Section 3.4.4 explains precisely how nested patterns enable condition-based reflective declarations.) An example (from an application discussed in detail in Section 3.5) illustrates the usage of pure conditions in reflective declarations:

```
<R> if ( no public R restore() : X.methods )
public void restore() { ... }
```

The above reflective declaration block consists of a statically exerted condition, specified by the pattern following the if keyword. If the pattern condition is satisfied, the code following the condition is declared. Thus, method void restore() is only declared if a method restore(), with any non-void return type, does not already exist in X.

Another useful feature is introduced by the errorif keyword. errorif acts as a type-assertion, allowing the programmer to express facts that he/she knows to be true about a type parameter. For instance, the programmer can express a mixin class that is only applicable to non-conflicting

classes:

```
class SizeMixin<X> extends X {
  <F> errorif ( some F size : X.fields )
  int size = 0;
}
```

In this case, the programmer wants to assert that, if the parameter type `C` already contains a field named `size`, this is not an error in the definition of `SizeMixin` but in the instantiation `SizeMixin<C>`. Thus, the `errorif` construct serves as a typical type-cast: it is both an assumption that the type system can use (i.e., when checking `SizeMixin<X>` it can be assumed that `X` has no `size` field) and at the same time a type obligation for a later type-checking stage. Unlike, however, traditional type-casts that turn a static type check into a run-time type check, an `errorif` turns a modular type check into a non-modular (type-instantiation time), but still static, type check.

A Note on the Similarity of MorphJ and cJ The `if` capability of MorphJ allows the declaration of conditional code, similar to that allowed by static type conditions supported by cJ (Section 2). Static type conditions allow code to be conditionally declared based on subtyping conditions, where the subtyping hierarchy is nominal (at least in the implementation context of cJ); MorphJ allows code to be conditionally declared based on the presence (or absence) of methods or fields matching certain patterns, which are structural conditions. One can achieve similar conciseness and safety benefits of cJ by remodeling the subtyping conditions into the structural conditions supported by MorphJ. However, since MorphJ does not integrate with Java's use-site variance annotation (i.e., the wildcard), one cannot create a type in MorphJ that cut across multiple, orthogonal conditions (e.g., `List<? extends VariableSize & Mutable>`). The MorphJ `if` construct, however, is more expressive in allowing code to be declared under negative conditions, i.e., the absence of certain methods or fields. There is no way to express negative conditions using static type conditions. A more detailed comparison between MorphJ and cJ is carried out in Section 4.1.

3.4.4 Semantics of Nested Patterns

Similarly to primary patterns, a nested pattern introduces an iteration. However, nested patterns are only used to return a true/false decision. For instance, in class `Pair<X,Y>`, the nested pattern

iterates over all the methods in `Y` matching the pattern, but the iteration only serves to verify whether a matching method exists, not to produce different code for each matching method. Furthermore, multiple nested patterns are all nested at the same level, forming a conjunction of their conditions.

Nested patterns may use pattern variables that are not bound by any primary pattern. However, there are restrictions as to how variables bound only by nested patterns can be used in code introduced by the reflective block (i.e., the reflective declaration). Pattern variables bound by only negative nested patterns cannot be used in the reflective declaration at all. For instance, the pattern "`no public R restore()`" above bound type variable `R`. However, `R` only appears in a negative nested pattern, and thus cannot be used in the declaration of `restore()`. Intuitively, a variable in a negative nested pattern is never bound to any concrete type or identifier—no match can exist for the negative nested condition to be satisfied. Clearly, an unbound variable cannot be used in declarations.

Pattern variables that are bound by positive nested patterns, however, can be used in the reflective declaration, if we can determine that exactly one element can be matched by the nested pattern. This is the case only if all uniquely identifying parts of the nested pattern use either constants, or pattern variables bound by the primary pattern. The uniquely identifying parts of a method pattern are its name and argument types, and the uniquely identifying part of a field pattern is its name. For example, in `Pair<X,Y>`, the positive nested pattern "`some RY m(A) : Y.methods`" uses `m` and `A` in its uniquely identifying parts. Both pattern variables are bound by the primary pattern. Thus, we can use `RY` in the reflective declaration, even though it only appears in the nested pattern.

It may not be immediately obvious how `if` and `errorif` relate to nested patterns. However, the type system machinery that enables `if` and `errorif` is precisely that of nested patterns. The patterns used as `if` and `errorif` conditions are regular nested patterns (with `some` and `no`) with the same semantics and conditions (e.g., limitations on when bound variables can appear in a reflective declaration). Indeed, even our actual implementation of the static `if` statement translates it intermediately into a static `for` loop with a special "unit" value for the primary pattern condition.

We do not allow the nesting of primary patterns—i.e., it is not legal to have nested static

`for` loops. However, `if` and `errorif` declarations and statements can be freely nested within the scopes of one another, or within the scope of a static `for` loop.

3.5 Applications using Nested Patterns

We next show two real world applications, re-implemented concisely and safely with nested patterns.

3.5.1 DSTM2

DSTM2 [41] is a Java library implementing object-based software transactional memory. It provides a number of "transactional factories" that take as input a sequential class, and generate a transactional class. Each factory supports a different transactional policy. The strength of DSTM2 is in its flexibility. Users can mix and match policies for objects, or define new "factories" implementing their own transactional policies.

In order to automatically generate transactional classes, DSTM2 factory classes use a combination of Java reflection, bytecode engineering with BCEL [9], and anonymous class definitions. For instance, for any input Java class, DSTM2 uses the Java reflection API to retrieve all fields annotated with `@atomic`, and generates appropriate "getter" and "setter" methods for these fields by injecting the bytecode representation of these methods into the input class file. However, the information needed for these generations is purely static and structural. The authors of DSTM2 had to employ low-level run-time techniques because the Java language does not offer enough support for compile-time transformation of classes. MorphJ, however, is a good fit for this task.

In our re-implementation of DSTM2's factories and supporting classes, 1,484 (non-commented, non-blank) lines of Java code are replaced with 576 lines of MorphJ code. For example, we replaced DSTM2's `factory.shadow.RecoverableFactory<X>` and `factory.shadow.Adaptor<X>` with the MorphJ class `Recoverable<X>` in Figure 13.

For each field of `X`, `Recoverable<X>` creates a shadow field, as well as getter and setter methods that acquire a lock from a transactional manager first, perform the read or write, and then resolve conflicts before returning. Furthermore, it creates `backup()` and `restore()` methods to backup and restore fields to and from their shadow fields.[3]

[3]The MorphJ implementation shown in this document replicates the predecessor of DSTM2, SXM [40], by iterating

```
1  @atomic public class Recoverable<class X> extends X {
2    // for each atomic field of X, declare a shadow field.
3    <F>[f]for(@atomic F f: X.fields; no shadow#f: X.fields)
4    F shadow#f;
5
6    // for each field of X, declare a getter.
7    <F>[f]for(@atomic F f: X.fields; no get#f(): X.methods)
8    public F get#f () {
9      Transaction me  = Thread.getTransaction();
10     Transaction other = null;
11     while (true) {
12       synchronized (this) {
13         other = openRead(me);
14         if (other == null) { return f; }
15       }
16       manager.resolveConflict(me, other);
17     }
18   }
19
20   // for each field of X, declare a setter
21   <F>[f]for(@atomic F f : X.fields;
22             no set#f(F) : X.methods)
23   public void set#f ( F val ) {
24     ... // code to open transaction.
25     f = val;
26     ... // code resolving conflict.
27   }
28
29   // create backup method
30   <R>if ( no R backup() : X.methods )
31   public void backup() {
32     <F>[f] for (@atomic F f : X.fields)
33     shadow#f = f;
34   }
35
36   // create recover method
37   <R>if ( no R recover() : X.methods )
38   public void recover() {
39     // restore field values from shadow fields.
40     <F>[f] for ( @atomic F f : X.fields )
41     f = shadow#f;
42   }
43 }
```

Figure 13: A recoverable transactional class in MorphJ.

The advantage of the MorphJ implementation is two-fold. First, `Recoverable<X>` is guaranteed to never declare conflicting declarations. For example, `shadow#f` is only declared if this field does not already exist in `X`, and `backup()` is only declared if such a method does not already exist in `X`. Implementations using reflection and bytecode engineering enjoy no such guarantees, and must instead rely on thorough testing to discover potential bugs.

Secondly, class `Recoverable<X>` is easier to write and understand. For example, the code for generating a `backup()` method in DSTM2's `RecoverableFactory<X>` is illustrated in Figure 14. We invite the reader to compare the `backup()` method declaration in Figure 13 (lines 29-33) to the code in Figure 14.

Interestingly, the predecessor of DSTM2 is a C# software transactional memory library called SXM [40]. It was re-implemented by Fähndrich, Carbin and Larus as the "quintessential example" of Compile-Time Reflection (CTR) [34]. However, CTR's safety guarantees only concern the validity of references, and not declaration conflicts. We give a more detailed exposition of CTR in Chapter 4.

3.5.2 Default Implementations for Interface Methods

Java ensures that a class cannot be declared to "implement" an interface unless it provides implementations for all of the interface's methods. This often results in very tedious code. For instance, it is common in code dealing with the Swing graphics library to implement an event-listener interface, yet provide empty implementations for most of the interface methods because the application does not care about the corresponding events. In response to this need, there have been mechanisms proposed [43, 72] that allow the programmer to specify that he/she wants just a default implementation for all members of an interface that are not already implemented by a class. These past solutions introduced new keywords (or Java annotations) for this specific feature. They either have no guarantee for the well-typedness of generated code [43], or require extensions to the Java type system [72]. These changes to the underlying language are required to support just this *one feature*. In contrast, we can express these language extensions as a MorphJ generic class that is guaranteed to always produce well-typed code. Figure 15 shows a slightly simplified version

over the `@atomic` fields of classes, and generating backup fields as needed. The new version, DSTM2, uses factories to generate transactional classes from interface definitions. It parses the interface's method names that begin with "`get`" and "`set`" to determine the field names to generate. There is a corresponding MorphJ implementation that operates on input interfaces. We discuss that particular version in Section 4.7.3.

```
public class RecoverableFactory<X> extends BaseFactory<X> {
   ...
  public void createBackup() {
    InstructionList il = new
    InstructionList(); MethodGen method =
      new MethodGen(ACC_PUBLIC, Type.VOID, Type.NO_ARGS,
                    new String[] { }, "backup",
                    className, il, _cp);

    for (Property p : properties) {
      InstructionHandle ih_0 =
        il.append(_factory.createLoad(Type.OBJECT, 0));
      il.append(_factory.createLoad(Type.OBJECT, 0));
      il.append(_factory
                .createFieldAccess(className, p.name,
                                   p.type,
                                   Constants.GETFIELD));
      il.append(_factory.
                .createFieldAccess(className,
                                    p.name + "$",
                                    p.type,
                                    Constants.PUTFIELD));
    }

    InstructionHandle ih_24 =
            il.append(_factory.createReturn(Type.VOID));
    method.setMaxStack();
    method.setMaxLocals();
    _cg.addMethod(method.getMethod());
    il.dispose();
  }
}
```

Figure 14: DSTM2 code for creating a method `backup()`.

```
class DefaultImplementation<X,interface I> implements I {
  X x;
  DefaultImplementation(X x) { this.x = x; }

  // for all methods in I, if the same method does
  // not appear in X, provide default implementation.
  <R extends Object,A*>[m]for( R m (A) : I.methods ;
                               no R m (A) : X.methods )
  R m (A a) { return null; }

  // for all methods in X that *do* correctly override
  // methods in I, we need to copy them.
  <R,A*>[m]for(   R m (A) : I.methods ;
              some R m (A) : X.methods )
  R m (A a) { return x.m(a); }

  // for all methods in X, such that there is no method
  // in I with the same name and arguments, copy method.
  <R,A*>[m]for( R m (A) : X.methods;
              no m (A) : I.methods)
  R m (A a) { return x.m(a); }
}
```

Figure 15: A MorphJ generic class providing default implementations of methods in any interface `I`.

of the MorphJ solution to this problem. (For conciseness, we elide the declarations dealing with `void`- or primitive-type-returning methods, which roughly double the code.)

Class `DefaultImplementation<X,I>` copies all methods of type `X` that either correctly implement methods in interface `I`, or are guaranteed to not conflict with methods in `I`. For methods in `I` that have no counterpart in `X`, a default implementation is provided. Methods in `X` that conflict with methods in `I` (same argument types, different return types) are ignored. The code for `DefaultImplementation<X,I>` demonstrates the power of nested patterns, both in terms of expressiveness, and in terms of type safety. The application naturally calls for different handling of methods in a type, based on the existence of methods in another type. Furthermore, these declarations are guaranteed to be unique, and their uniqueness is crucially based on nested patterns.

Note that the difference between `DefaultImplementation<X,I>`, shown here, and `DefaultClass<I>` shown in Figure 10, is that `DefaultImplementation<X,I>` preserves the behavior of those methods of `I` that are already defined in `X`. `DefaultClass<I>`, on the other hand, is only

parameterized by an interface type variable, and provides default implementations for *all* methods of that interface.

3.6 Type-Checking MorphJ: A Casual Discussion

Higher variability always introduces complexity in type systems. For instance, polymorphic types require more sophisticated type systems than monomorphic types, because polymorphic types can reference type "variables", whose exact values are unknown at the definition site of the polymorphic code. In MorphJ, in addition to type variables, there are also *name* variables—declarations and references can use names reflectively retrieved from type variables. Thus, the exact values of these names are not known when writing a generic class. Yet, the author of the generic class needs to have some confidence that his/her code will work correctly with any parameterization in its intended domain. The job of MorphJ's type system is to ensure that generic code does not introduce static errors, for *any* type parameter that satisfies the author's stated assumptions.

There are two main challenges in type-checking a MorphJ program: 1) how do we determine that declarations made with name variables are unique, i.e., there are no naming conflicts, when we do not know the exact identifiers these name variables could represent, and 2) how do we determine that references always refer to declared members and are well-typed, when we know neither the exact names of the members referenced, or the exact names of the members declared. The key insight is to treat a declaration as a *range* of declared elements. (A declaration made without pattern variables has a one-element range.) Determining the uniqueness of two declarations then reduces to determining whether their ranges are *disjoint*. Similarly, a reference is also a range. Determining whether a reference is valid then reduces to determining *containment*: Are all entities in the reference range contained, i.e., have corresponding entities, within the declaration range?

In this section, we present through examples the main problems and insights for checking reference validity and declaration uniqueness in MorphJ programs. We focus on declarations and references made by reflecting over type variables: Reflecting over non-variable types is simply syntactic sugar for manually inlining the declarations. We further focus on the rules for type-checking methods—rules for fields are a trivial adaptation of those for methods.

3.6.1 Uniqueness of Declarations

We use range disjointness to check whether declarations are unique. In the case of method declarations, uniqueness means two methods within the same class (including inherited methods) cannot have the same name and argument types.[4]

3.6.1.1 Internally Well-defined Range

A simple property to establish is the uniqueness of declarations introduced by the same reflective iteration block.

Simple case: Consider a simple MorphJ class:

```
class CopyMethods<X> {
  <R,A*>[m] for( R m (A) : X.methods ; nestedConditions )
  R m (A a) { ... }
}
```

`CopyMethods<X>`'s methods are declared within one reflective block, which iterates over all the methods of type parameter `X`. For each method returning a non-`void` type that also satisfies *nestedConditions*, a method with the same signature is declared for `CopyMethods<X>`.

How do we guarantee that, given any `X`, `CopyMethods<X>` has unique method declarations (i.e., each method is uniquely identified by its ⟨name, argument types⟩ tuple)? Observe that `X` can only be instantiated with another well-formed type (the base case being `Object`), and all well-formed types have unique method declarations. Thus, if a type merely copies the method signatures of another well-formed type, as `CopyMethods<X>` does, it is guaranteed to have unique method signatures, as well. The same principle also applies to reflective field declarations. Nested conditions only serve to remove more methods from the outer iteration range. Thus, regardless of the specific *nestedConditions*, the above declaration is always legal.

It is important to make sure that reflective declarations copy *all* the uniquely identifying parts of a method or field. For example, the uniquely identifying parts of a method are its name *together with* its argument types. Thus, a reflective method declaration that only copies either name or

[4] In Java, methods in a subclass are allowed to override their counterparts in the superclass with co-variant return types. This involves a relaxation of the rules we describe in this section: Return types in the subclass are allowed to be subtypes of their counterparts in the superclass.

argument types would not be well-typed. For example:

```
class CopyMethodsWrong<X> {
  <R,A*>[m] for( R m (A) : X.methods )
  R m () { ... }
}
```

The reflective declaration in `CopyMethodsWrong<X>` only copies the return type and the name of the methods of a well-formed type. This would cause an error if instantiated with a type with an overloaded method:

```
class Overloaded {
  int bar (int a);
  int bar (String s);
}
```

`CopyMethodsWrong<Overloaded>` would have two methods, both named `bar`, taking no arguments.

There is no way to express nested conditions that would filter out methods defined using overloaded method names. Such a nested condition would need to explicitly say that there are no methods in `X` with the same name, but *different* arguments than the methods matched by the primary pattern. MorphJ explicitly does not allow such inequality conditions.

Beyond Copy and Paste: Morphing of classes and interfaces is not restricted to copying the members of other types. Matched type and name variables can be used freely in reflective declarations and statements. For example:

```
class ChangeArgType<X> {
  <R,A extends Object>[m] for ( R m (A) : X.methods ; nestedConditions )
  R m ( List<A> a ) { /* do for all elements */ ... }
}
```

In `ChangeArgType<X>`, for each method of `X` that takes one non-primitive type argument `A` and returns a non-`void` type `R`, a method with the same name and return type is declared. However, instead of taking the same argument type, this method takes a `List` instantiated with the original argument type. Even though `ChangeArgType<X>` does not copy `X`'s method signatures exactly, we can still guarantee that all methods of `ChangeArgType<X>` have unique signatures, no matter what `X` is. The key is that a reflective declaration can manipulate the uniquely identifying parts of a method, (i.e., name and argument types), by using them in type (or name) compositions, as long

as these parts remain in the uniquely identifying parts of the new declaration. The following is an example of an *illegal* manipulation of types:

```
class IllegalChange<X> {
  <R,A>[m] for ( R m (A) : X.methods ; nestedConditions )
  A m ( R a ) { ... }
}
```

In the above example, the uniquely identifying part of `X`'s method is no longer the uniquely identifying part of `IllegalChange<X>`'s method: the argument types of `X`'s method is no longer part of the argument types of `IllegalChange<X>`'s method. Using class `Overloaded` defined above, `IllegalChange<Overloaded>` will cause an error in the generated code. Again, no nested conditions here can prevent this type of declaration conflicts.

3.6.1.2 Uniqueness Across Ranges

When there are multiple reflective blocks in the same type declaration, we need to guarantee that the declarations in one block do not conflict with the declarations in another block. For two reflective method declarations, their uniqueness means that the *range* of their ⟨name, argument types⟩ tuples cannot overlap. This can be determined by a *two-way unification* of the two declarations. In a two-way unification, pattern variables from *both* reflective blocks are unification variables.

Let us start with a simple example. Consider the following class:

```
1  class DisjointDecs<X> {
2    <R>[m] for(R m (int) : X.methods; nestedConditions1 )
3    R m (int i) { ... }
4
5    <S>[n] for(S n (int) : X.methods; nestedConditions2 )
6    S n (int i, String s) { ... }
7  }
```

It is easy to see that the declarations on lines 3 and 6 cannot overlap for any `X`. There is no unification to make the two signatures have the same ⟨name, argument types⟩ tuple, because there is simply no way to unify {`int`} with {`int`,`String`}.

When two method signatures do unify, there may be overlap in the declarations. However, if we can prove that conditions causing the overlap are infeasible, then the declarations are still unique. An overlap is infeasible if the unification mapping producing the overlap, when applied to

the primary and nested patterns, produces mutually exclusive conditions. Note that a non-empty primary pattern range states a condition, as well—it is a positive condition that says *some* element exists in this range.

Consider the following class:

```
1  class StillUnique<X> {
2    <A1>[m]for( String m (A1) : X.methods ; nestedConditions1 )
3    void m (A1 a) { ... }
4
5    <A2>[n]for( int n (A2)    : X.methods ; nestedConditions2 )
6    void n (A2 a) { ... }
7  }
```

The declared signatures on lines 3 and 6 unify with the mapping {m↦n, A1↦A2}. Applying this mapping to the primary patterns on lines 2 and 5, we get "`String n (A2) : X.methods`", and "`int n (A2) : X.methods`". Methods matched by these patterns can cause conflicting declarations. However, having at least one method in both of these ranges means that there need to be two methods in `X` with the same name and argument types, but different return types. This directly contradicts the fact that `X` is a well-formed type. Thus, this unification mapping produces mutually exclusive conditions between the two primary pattern conditions, and there are no elements that would make the mapping possible. These declarations are thus still disjoint.

To generalize the rules for determining range disjointness, let $\langle P_n,T_n\rangle$ denote the range of pattern P_n matching over the methods of type T_n, let $+$ prefix a positive pattern condition, and $-$ prefix a negative condition. There are two conditions for determining disjointness of ranges:

- $+\langle P_n,T_n\rangle$ and $+\langle P'_n,S_n\rangle$ are mutually exclusive if T_n is a subtype of S_n, and P_n, P'_n have unifying method name and argument types, but different return types.
- $+\langle P_n,T_n\rangle$ and $-\langle P'_n,S_n\rangle$ are mutually exclusive if $\langle P'_n,S_n\rangle$ subsumes $\langle P_n,T_n\rangle$

We apply these rules on all pairs of conditions. A single mutual exclusive pair guarantees the disjointness of ranges. We applied the first rule to prove that `StillUnique<X>` contains unique method declarations. The following example demonstrates an application of the second rule:

```
1  public class UnionOfStatic<X,Y> {
2    <A*>[m] for( static void m (A) : X.methods; nestedConditions)
3    public static void m(A args) { X.m(args); }
4
5    <B*>[n] for( static void n (B)      : Y.methods ;
6              no static void n (int, B) : X.methods )
7    public static int n(int count, B args) {
8      for (int i = 0; i < count; i++) Y.n(args);
9      return count;
10   }
11 }
```

The two method declarations on lines 3 and 7 have signatures that can be unified with the mapping { A↦{int,B}, m↦n}. Applying this substitution to the primary pattern on line 2 yields "`static void n(int,B) : X.methods`". Having a method in the range of this pattern directly contradicts the condition of the negative nested pattern on line 6, which states there should be no methods in the range of "`static void n(int,B) : X.methods`". Thus, the two method declarations are unique for all `X` and `Y`.

Reflective and Regular Methods Together: Declaration conflicts can also occur when a class has both regular and reflectively declared members. For example, in the following class declaration, we cannot guarantee that the methods declared in the reflective block do not conflict with method `int foo()`.

```
class Foo<X> {
  int foo () { ... }

  <R,A*>[m]for ( R m (A) : X.methods ; nestedConditions )
  R m (A a) { ... }
}
```

Just as in the case of multiple iterators, the main issue is establishing the disjointness of declaration ranges, with the regular methods acting as a constant declaration range. In this case, a simple way is to use a nested condition to ensure that there is no method named `foo` in `X`: `no foo(S) : X.methods`, where `S` is declared as pattern type variable `S*`. This nested condition is mutually exclusive with the primary pattern condition—no method in the range of the primary pattern can possibly satisfy the nested condition, and thus no method will be declared at all through this reflective block.

Using Static Prefixes: In general, we can guarantee the uniqueness of declarations across reflective blocks by proving either type signature or name uniqueness. A general way to establish the uniqueness of declarations is by using unique static prefixes on names. (For static prefixes to be uniquely identifying, they must not be prefixes of each other.) For instance, our earlier example can be rewritten correctly as:

```
class Manipulation<X> {
  <R>[m] for ( R m (List<X>) : X.methods )
  R list#m (List<X> a) { ... }

  <R>[m] for ( R m (X) : X.methods )
  R nolist#m (List<X> a) { ... }
}
```

Proper Method Overriding and Mixins: Proper overriding means that a subtype should not declare a method with the same name and arguments as a method in a supertype, but a non-covariant return type. Ensuring proper method overriding is a special case of declaration range disjointness: if two methods' ⟨name, argument types⟩ tuples are not unique, they are still well-typed declarations if we can establish that the overriding method's return type is a subtype of the overriden return type.

One case that deserves some discussion is that of a type variable used as a supertype. (In case the type is a class, it is implicitly assumed to be non-final.) This is sometimes called a *mixin* pattern [15, 82]. Since the supertype could potentially be any type, we have no way of knowing its declarations. For instance, the following class is unsafe and will trigger a type error, as there is no guarantee that the superclass does not already contain an incompatible method `foo`.

```
class C<class T> extends T {
  int foo () { ... }
}
```

Static prefixes are similarly insufficient to guarantee that subtype methods do not conflict with supertype methods. As a result, there are two legal ways to declare mixins in MorphJ.

First, the subclass may contain *no* members other than reflective iterators over its supertype that declare overriding versions for (some subset of) the supertype's methods. For instance, the following is a legal MorphJ class:

```
class C<class T> extends T {
  <R,A>[m] for (R m (A) : T.methods)
  R m (A a) { ... }
}
```

The class correctly overrides some of its superclass's methods (those accepting and returning one argument).

Alternatively, the subclass may use proper nested conditions to eliminate possibilities of illegal overriding. For instance:

```
class C<class T> extends T {
  if ( no foo () : T.methods )
  int foo () { ... }
}
```

3.6.2 Validity of References

Another challenge of modular type checking for a morphing language is to ensure the validity of references. We use the term "validity" to refer to the property that a referenced entity has a definition, and its use is well-typed.

3.6.2.1 Reference within the Same Range

Let us take another look at class `Logging<X>` from Section 3.1:

```
1  class Logging<class X> extends X {
2    <R,Y*>[meth]for(public R meth (Y) : X.methods)
3    public R meth (Y a) {
4      R r = super.meth(a);
5      System.out.println("Returned: " + r);
6      return r;
7    }
8  }
```

How do we know that the method invocation "`super.meth(a)`" (line 4) is valid? Notice that the range of `meth` (i.e., all the identifiers it could expand to) is exactly the names of methods matched by the primary pattern on line 2: all non-`void` methods of `X`. This range is certainly contained by the range of all methods declared for `X`. Thus, we know method `meth` exists, no matter what `X` is. Furthermore, how do we know we are invoking `meth` with the right arguments? The type of `a` is `Y`: exactly the argument type `m` of `X` is expecting.

3.6.2.2 Reference Across Ranges

Things get a bit more complex when a name variable bound in one reflective block references a method declared in a different reflective block. Consider the following class, which logs the arguments of methods accepting strings, before calling `Logging` to log the return value.

```
1  class LogStringArg<class Y> {
2    Logging<Y> loggedY;
3
4     <T>[n] for ( public T n(String) : Y.methods )
5     public T n (String s) {
6         System.out.println("arg: " + s);
7         return loggedY.n(s);
8     }
9  }
```

How do we know that `loggedY.n(s)` (line 7) is a valid reference, when the methods of `loggedY` are defined in a different class and a different reflective block? The key is to determine that the range of `n` is contained by the range of method names in `Logging<Y>`. This is to say that the range of `n`'s enclosing reflective block should be contained by the range of `Logging<Y>`'s declaration reflective block. Observe that the declaration block of `Logging<Y>` is defined over methods of `Y` (after substituting `Y` for `X`), as is the reflective block enclosing `n`. Secondly, the pattern for the declaration block of `Logging<Y>` is more general than the pattern for the reflective block enclosing `n`: the former matches all non-`void` methods, and the latter matches all non-`void` methods taking exactly one `String` argument. Thus, any method that is matched by the reference reflective block's pattern is matched by the declaration reflective block's pattern, regardless of what `Y` is. Consequently, there is always a method `n` in `Logging<Y>`.

Whether a pattern is more general than another can be systematically determined by finding a *one-way unification* from the more general pattern to the more restricted one. In a one-way unification, only pattern variables declared for the more general pattern are used as unification variables. All other pattern variables are considered constants. In this example, we can unify "`public R m(A)`" to "`public T n(String)`" using the mapping $\{\texttt{R}\mapsto\texttt{T}, \texttt{m}\mapsto\texttt{n}, \texttt{A}\mapsto\{\texttt{String}\}\}$.

We also use this unification mapping in determining whether `n` is invoked with the right argument types. We apply the mapping to the method declaration in `Logging<Y>`, and get the declared signature "`public T n(String)`". Since `s` has the type `String`, the invocation is clearly

correct. Furthermore, we can check that the result of the invocation is of type `T`, which is precisely the expected return type of the method enclosing "`loggedY.n(s)`".

For a case with nested patterns, consider the following class:

```
1  class VoidPair<X,Y> {
2    X x; Y y; ...// constructor to initialize x and y.
3
4    <A*>[m]for (  public void m(A) : X.methods ;
5              some public void m(A) : Y.methods )
6    public void m (A a) {   x.m(a);  y.m(a);   }
7  }
```

`VoidPair<X,Y>` declares a method for every `void` method that `X` and `Y` share in name and argument types, and invokes that method on `x` and `y`. Using the reference rules described previously, we know that `x.m(a)` is a valid reference. Furthermore, because the pattern variables used in the positive nested pattern on line 5 are all bound by the primary pattern, we know that if the nested condition is satisfied, there is exactly one element in the range of the nested pattern, so the call `y.m(a)` is unambiguous. Since the types also match, `y.m(a)` is a valid reference, as well.

Let us now consider the general case of a reference made in one reflective block, to declarations made in another reflective block, when both blocks have nested patterns. Let R_d and R_r denote the ranges for the reflective blocks of the declaration and the reference, respectively. There are two sufficient conditions to determine that R_r is subsumed by R_d. First, the primary range of R_r must be subsumed by the primary range of R_d. Second, for all methods that are in the primary range of R_r (and thus also in the primary range of R_d), if the method satisfies the nested conditions of R_r, it should also satisfy the nested conditions of R_d. That is to say, the nested conditions of R_r should be stronger, and imply the nested conditions of R_d.

Determining that one nested condition implies another can be reduced to single range containment. Using the same notation as before, we have two ways of determining that one condition implies another:

- $+\langle N_r,T_r\rangle$ implies $+\langle N_d,T_d\rangle$ if $\langle N_d,T_d\rangle$ contains $\langle N_r,T_r\rangle$.
- $-\langle N_r,T_r\rangle$ implies $-\langle N_d,T_d\rangle$ if $\langle N_r,T_r\rangle$ contains $\langle N_d,T_d\rangle$.

Intuitively, $+\langle N_r,T_r\rangle$ is satisfied if there is at least one element in $\langle N_r,T_r\rangle$. Then there is certainly at least one element in a larger range, as well. Thus, $+\langle N_d,T_d\rangle$ should be satisfied.

Similar reasoning applies for the implication between two negative conditions.

To be more concrete, consider the following class:

```
 8  class CallVoidsWithString<T,S> {
 9    VoidPair<T,S> voidPair;
10    ... // constructor to initialize voidPair
11    [n]for (  public void n(String) : T.methods ;
12         some public void n(String) : S.methods )
13    public void n (String s) { voidPair.n(s); }
14  }
```

For every `void` method taking one `String` argument that `T` and `S` have in common, `CallVoidsWithString<T,S>` declares a method with the same signature, and invokes a method with the same name on `voidPair`, of type `VoidPair<T,S>`. This reference is valid if the range of the reflective block on lines 11-12 is contained by the range of the declaration reflective block (lines 4-5 in the definition of `VoidPair`).

The range of the primary pattern on line 11 is contained by the range of the declaration's primary pattern (line 4), by the one-way unification mapping {`m`↦`n`, `A`↦{`String`}}.

To check whether the nested pattern on line 5 contains the nested pattern on line 12, note that we first apply the unification mappings obtained from unifying the primary patterns—we only want to determine this containment relationship for those methods that lie in the range of both primary patterns. In our example, after applying the unification mapping to the positive nested pattern on line 5 (and also substituting `S` for `Y`), we have "`public void n(String) :  S.methods`". This clearly contains "`public void n(String) :  S.methods`" on line 12.

These two conditions guarantee us that reference `voidPair.n(s)` is always a valid one. It is easy to check that this is indeed the case.

The above approach generalizes to an arbitrary number of nested conditions: Each nested condition in the declaration range must be implied by at least one nested condition in the reference range. A range with no nested patterns is equivalent to a range with a positive nested pattern that contains everything, or a negative nested pattern that is contained by everything. The case where there are only nested patterns (i.e., `if` and `errorif` statements) can be reduced to a range with a special primary pattern value that contains only itself and is contained only by itself.

3.7 Formalization

We present a formalization, FMJ, which is based on FGJ [47], with differences (other than the simple addition of our extra environment, Λ) highlighted in grey. Figures in which all rules are new to FMJ (Figures 18,20,21) are not highlighted at all, for better readability.

3.7.1 Syntax

The syntax of FMJ is presented in Figure 16. We adopt many of the notational conventions of FGJ: C,D denote constant class names; X,Y,Z denote type variables; N,P,Q,R denote non-variable types (which may be constructed from type variables); S,T,U,V,W denote types; f denotes field names; m denotes non-variable method names; x,y denote argument names. Notations new to FMJ are: η denotes a variable method name; n denotes either variable or non-variable names; o denotes a nested condition operator (either + or -, for the keywords some or no, respectively).

T	::=	X | N
N	::=	C<$\overline{\text{T}}$>
CL	::=	class C<$\overline{\text{X}}$◁$\overline{\text{N}}$>◁ N {$\overline{\text{T}}$ $\overline{\text{f}}$; $\overline{\text{M}}$}
	|	class C<$\overline{\text{X}}$◁$\overline{\text{N}}$>◁ T {$\overline{\text{T}}$ $\overline{\text{f}}$; $\overline{\mathfrak{M}}$}
M	::=	T m ($\overline{\text{T}}$ $\overline{\text{x}}$) {↑e;}
$\mathfrak{M}$	::=	<$\overline{\text{Y}}$◁$\overline{\text{P}}$>for($\mathbb{M}_p$;$o\mathbb{M}_n$) U η ($\overline{\text{U}}$ $\overline{\text{x}}$) {↑e;}
o	::=	+ | -
$\mathbb{M}$	::=	V η ($\overline{\text{V}}$):X.methods
e	::=	x | e.f | e.n($\overline{\text{e}}$) | new C<$\overline{\text{T}}$>($\overline{\text{e}}$)
n	::=	m | η

Figure 16: FMJ: Syntax

We use the shorthand $\overline{\text{T}}$ for a sequence of types $\text{T}_0,\text{T}_1,\ldots,\text{T}_n$, and $\overline{\text{x}}$ for a sequence of unique variables $\text{x}_0,\text{x}_1,\ldots,\text{x}_n$. We use : for sequence concatenation, e.g. $\overline{\text{S}}:\overline{\text{T}}$ is a sequence that begins with $\overline{\text{S}}$, followed by $\overline{\text{T}}$. We use $\bullet$ to denote an empty sequence. We use $\in$ to mean "is a member of a sequence" (in addition to set membership). We use ... for values of no particular significance to a rule. ◁ and ↑ are shorthands for the keywords `extends` and `return`, respectively. Note that all classes must declare a superclass, which can be `Object`.

FMJ formalizes some core MorphJ features that are representative of our approach. One simplification is that we do not allow a nested pattern to use any pattern type or name variables

not bound by its primary pattern. We also do not formalize reflecting over a statically known type, or using a constant name in reflective patterns. These are decidedly less interesting cases from a typing perspective. The zero or more length type vectors `T*` are also not formalized. These type vectors are a mere matching convenience. Thus, safety issues regarding their use are covered by non-vector types. We do not formalize reflectively declared fields—their type checking is a strict adaptation of the techniques for checking methods. Lastly, static name prefixes, casting expressions and polymorphic methods are not formalized.

FGJ does not support method overloading, and FMJ inherits this restriction. Thus, a method name alone uniquely identifies a method definition. Furthermore, since we allow no fresh name variables in nested patterns, there can be only one name variable in a pattern, and we use the keyword η for name variables, instead of allowing arbitrary identifiers. A reflective definition must also use this same name (since static prefixes are not allowed and constant names would be illegal due to conflicts). This results in a small simplification over the informal rules, but leaves their essence intact.

A program in FMJ is an $(\mathtt{e}, CT)$ pair, where `e` is an FMJ expression, and CT is the class table. We place some conditions on CT: Every class declaration `class C...` has an entry in CT; `Object` is not in CT. The subtyping relation derived from CT must be acyclic, and the sequence of ancestors of every instantiation type is finite. (The last two properties can be checked with the algorithm of [8] in the presence of mixins.)

3.7.2 Typing Judgments

The main typing rules of FMJ are presented in Figure 17, with auxiliary definitions presented in Figures 18 and 19. *We recommend reading our text description and referring to the rules as needed in the flow of the text. Alternatively, the PDF version of this document contains hyperlinks in type rules, from operators to their text description.*

There are three environments used in our typing judgments:

- Δ: Type environment. Maps type variables to their upper bounds.
- Γ: Variable environment. Maps variables (e.g., `x`) to their types.

Expression typing:

$$\Delta;\Lambda;\Gamma\vdash \mathtt{x} \in \Gamma(\mathtt{x}) \qquad \text{(T-VAR)}$$

$$\frac{\Delta;\Lambda;\Gamma\vdash \mathtt{e}_0\in \mathtt{T}_0 \qquad \Delta\vdash \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0))=\overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Delta;\Lambda;\Gamma\vdash \mathtt{e}_0.\mathtt{f}_i\in \mathtt{T}_i} \qquad \text{(T-FIELD)}$$

$$\frac{\Delta;\Lambda;\Gamma\vdash \mathtt{e}_0\in \mathtt{T}_0 \qquad \Delta;\Lambda;\Gamma\vdash \overline{\mathtt{e}}\in\overline{\mathtt{S}} \qquad \Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathtt{T}_0)=\overline{\mathtt{T}}\to\mathtt{T} \qquad \Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{T}}}{\Delta;\Lambda;\Gamma\vdash \mathtt{e}_0.\mathtt{n}(\overline{\mathtt{e}})\in\mathtt{T}} \qquad \text{(T-INVK)}$$

$$\frac{\Delta\vdash \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}\ \mathit{ok} \quad \Delta\vdash \mathit{fields}(\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>})=\overline{\mathtt{U}}\ \overline{\mathtt{f}} \qquad \Delta;\Lambda;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}} \qquad \Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{U}}}{\Delta;\Lambda;\Gamma\vdash \texttt{new C<}\overline{\mathtt{T}}\texttt{>(}\overline{\mathtt{e}}\texttt{)}\in\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}} \qquad \text{(T-NEW)}$$

Method typing:

$$\frac{\begin{array}{c}\Delta=\overline{\mathtt{X}}<:\overline{\mathtt{N}} \quad \Gamma=\overline{\mathtt{x}}\mapsto\overline{\mathtt{T}},\mathtt{this}\mapsto\mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>} \quad \Lambda=\emptyset \quad \Delta\vdash\overline{\mathtt{T}},\mathtt{T}_0\ \mathit{ok}\\ \Delta;\Lambda;\Gamma\vdash\mathtt{e}_0\in\mathtt{S}_0 \quad \Delta\vdash\mathtt{S}_0<:\mathtt{T}_0\\ \mathit{CT}(\mathtt{C})=\texttt{class C<}\overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}\texttt{><:N \{...\}} \quad \Delta;\Lambda\vdash\mathit{override}(\mathtt{m},\ \mathtt{N},\ \overline{\mathtt{T}}\to\mathtt{T}_0)\end{array}}{\mathtt{T}_0\ \mathtt{m}\ \texttt{(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\texttt{) \{ }\uparrow\mathtt{e}_0\texttt{; \} OK IN C<}\overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}\texttt{>}} \qquad \text{(T-METH-S)}$$

$$\frac{\begin{array}{c}\Delta=\overline{\mathtt{X}}<:\overline{\mathtt{N}},\overline{\mathtt{Y}}<:\overline{\mathtt{P}} \quad \Gamma=\overline{\mathtt{x}}\mapsto\overline{\mathtt{T}},\mathtt{this}\mapsto\mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>} \quad \Delta\vdash\overline{\mathtt{P}},\overline{\mathtt{T}},\mathtt{T}_0,\overline{\mathtt{N}}\ \mathit{ok} \quad \Delta\vdash\mathbb{M}_p,\mathbb{M}_f\ \mathit{ok}\\ R_p=\mathit{range}(\mathbb{M}_p,\texttt{<}\overline{\mathtt{Y}}\texttt{<:}\overline{\mathtt{P}}\texttt{>}) \quad R_n=\mathit{range}(\mathbb{M}_f,\bullet) \quad \Lambda=\langle R_p,oR_n\rangle\\ \Delta;\Lambda;\Gamma\vdash\mathtt{e}\in\mathtt{S}_0 \quad \Delta\vdash\mathtt{S}_0<:\mathtt{T}_0\\ \mathit{CT}(\mathtt{C})=\texttt{class C<}\overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}\texttt{><:T \{ ... \}} \quad \Delta;\Lambda\vdash\mathit{override}(\eta,\ \mathtt{T},\ \overline{\mathtt{T}}\to\mathtt{T}_0)\end{array}}{\texttt{<}\overline{\mathtt{Y}}\texttt{<:}\overline{\mathtt{P}}\texttt{>for(}\mathbb{M}_p\texttt{;}o\mathbb{M}_f\texttt{) }\mathtt{T}_0\ \eta\ \texttt{(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\texttt{) \{}\uparrow\texttt{e;\} OK IN C<}\overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}\texttt{>}} \qquad \text{(T-METH-R)}$$

Class typing:

$$\frac{\Delta=\overline{\mathtt{X}}<:\overline{\mathtt{N}} \quad \Delta\vdash\overline{\mathtt{N}},\mathtt{N},\overline{\mathtt{T}}\ \mathit{ok} \quad \overline{\mathtt{M}}\ \texttt{OK IN C<}\overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}\texttt{>}}{\texttt{class C<}\overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}\texttt{><:N \{ }\overline{\mathtt{T}}\ \overline{\mathtt{f}}\texttt{; }\overline{\mathtt{M}}\texttt{\} OK}} \qquad \text{(T-CLASS-S)}$$

$$\frac{\begin{array}{c}\Delta=\overline{\mathtt{X}}<:\overline{\mathtt{N}} \quad \Delta\vdash\overline{\mathtt{N}},\mathtt{T},\overline{\mathtt{T}}\ \mathit{ok} \quad \text{for all } \mathfrak{M}_i\in\overline{\mathfrak{M}},\ \mathfrak{M}_i\ \texttt{OK IN C<}\overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}\texttt{>}\\ \text{for all } \mathfrak{M}_i,\mathfrak{M}_j\in\overline{\mathfrak{M}}, \quad \Lambda_i=\mathit{reflectiveEnv}(\mathfrak{M}_i) \quad \Lambda_j=\mathit{reflectiveEnv}(\mathfrak{M}_j)\\ \Delta\vdash\mathit{disjoint}(\Lambda_i,\ \Lambda_j)\end{array}}{\texttt{class C<}\overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}\texttt{><:T \{ }\overline{\mathtt{T}}\ \overline{\mathtt{f}}\texttt{; }\overline{\mathfrak{M}}\texttt{\} OK}} \qquad \text{(T-CLASS-R)}$$

Well-formed types:

$$\Delta\vdash\texttt{Object}\ \mathit{ok} \qquad \text{(WF-OBJECT)}$$

$$\frac{\mathtt{X}\in\mathit{dom}(\Delta)}{\Delta\vdash\mathtt{X}\ \mathit{ok}} \qquad \text{(WF-VAR)}$$

$$\frac{\mathit{CT}(\mathtt{C})=\texttt{class C<}\overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}\texttt{><:T \{ ... \}} \qquad \Delta\vdash\overline{\mathtt{T}}\ \mathit{ok} \qquad \Delta\vdash\overline{\mathtt{T}}<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}}{\Delta\vdash\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}\ \mathit{ok}} \qquad \text{(WF-CLASS)}$$

Well-formed Patterns:

$$\frac{\mathbb{M}=\mathtt{U}_0\ \eta\ \texttt{(}\overline{\mathtt{U}}\texttt{) :T.methods} \qquad \Delta\vdash\mathtt{U}_0,\overline{\mathtt{U}},\mathtt{T}\ \mathit{ok}}{\Delta\vdash\mathbb{M}\ \mathit{ok}} \qquad \text{(WF-PAT)}$$

Figure 17: FMJ: Typing Rules

Range Construction Functions:

$$\frac{\mathbb{M}{=}\mathtt{U}_0\ \eta\ (\overline{\mathtt{U}})\ :\ \mathtt{X.methods}}{range(\mathbb{M},{<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>})\ {=}(\mathtt{X},\ {<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>}\overline{\mathtt{U}}{\to}\mathtt{U}_0)}$$

$$\frac{\begin{array}{c}\mathfrak{M}{=}{<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>}\mathtt{for}\ (\mathbb{M}_p;o\mathbb{M}_f)\ \mathtt{U}_0\ \eta\ (\overline{\mathtt{U}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e};\}\\ R_p{=}range(\mathbb{M}_p,{<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>})\qquad R_n{=}range(\mathbb{M}_f,\bullet)\qquad \Lambda{=}\langle R_p,R_n\rangle\end{array}}{reflectiveEnv(\mathfrak{M})\ {=}\Lambda}$$

Specializing reflective environment:

$$\frac{\begin{array}{c}\Lambda_d{=}\langle R_p,oR_n\rangle\qquad R_p{=}(\mathtt{T}_i,\ {<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>}\overline{\mathtt{U}}{\to}\mathtt{U})\qquad R_n{=}(\mathtt{T}_j,\ \overline{\mathtt{V}}{\to}\mathtt{V})\\ \Delta;\Lambda{\vdash}mtype(\mathtt{m},\ \mathtt{T}_i){=}\overline{\mathtt{U}}'{\to}\mathtt{U}'\qquad R'_p{=}(\mathtt{T}_i,\ \overline{\mathtt{U}}'{\to}\mathtt{U}')\\ R'_n{=}(\mathtt{T}_j,\ \overline{\mathtt{V}}'{\to}\mathtt{V}'_0)\qquad \Delta;\Lambda{\vdash}mtype(\mathtt{m},\ \mathtt{T}_j){=}\overline{\mathtt{V}}'{\to}\mathtt{V}'_0\end{array}}{\Delta;\Lambda{\vdash}specialize(\mathtt{m},\ \Lambda_d){=}\langle R'_p,{+}R'_n\rangle}\ \text{(SP-+)}$$

$$\frac{\begin{array}{c}\Lambda_d{=}\langle R_p,oR_n\rangle\qquad R_p{=}(\mathtt{T}_i,\ {<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>}\overline{\mathtt{U}}{\to}\mathtt{U})\qquad R_n{=}(\mathtt{T}_j,\ \overline{\mathtt{V}}{\to}\mathtt{V})\\ \Delta;\Lambda{\vdash}mtype(\mathtt{m},\ \mathtt{T}_i){=}\overline{\mathtt{U}}'{\to}\mathtt{U}'\qquad R'_p{=}(\mathtt{T}_i,\ \overline{\mathtt{U}}'{\to}\mathtt{U}')\\ \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]{\vdash}R'_p{\sqsubseteq}_R R_p\quad R'_n{=}[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R_n\\ \left\{\begin{array}{l}\Delta;\Lambda\vdash mtype(\mathtt{m},\mathtt{T}_j)\ \text{is undefined}\\ \Delta;\Lambda\vdash mtype(\mathtt{m},\mathtt{T}_j)=\overline{\mathtt{V}}'\to\mathtt{V}'_0\ \text{and}\ [\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\mathtt{V}:\overline{\mathtt{V}})\neq(\mathtt{V}'_0:\overline{\mathtt{V}}')\end{array}\right.\ \text{or}\end{array}}{\Delta;\Lambda{\vdash}specialize(\mathtt{m},\ \Lambda_d){=}\langle R'_p,{-}R'_n\rangle}\ \text{(SP--)}$$

Subtype range validity:

$$\Delta\vdash validRange(\emptyset,\ \mathtt{T})\ \text{(VR-NOREFL)}$$

$$\frac{R_p{=}(\mathtt{X},\ \ldots)}{\Delta\vdash validRange(\langle R_p,{-}R_n\rangle,\mathtt{X})}\ \text{(VR-VAR)}$$

$$\frac{\begin{array}{c}CT(\mathrm{C}){=}\mathrm{class\ C}{<}\overline{\mathtt{X}}{\lhd}\overline{\mathtt{N}}{>}{\lhd}\mathrm{S}\ \{\ \ldots\ \overline{\mathfrak{M}}\}\qquad \Delta\vdash validRange(\Lambda,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathrm{S})\\ \text{for all}\quad \mathfrak{M}_i{\in}\overline{\mathfrak{M}}\qquad \mathfrak{M}_i{=}{<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>}\mathtt{for}(\mathbb{M}_p;o\mathbb{M}_f)\ldots\qquad \Lambda'{=}reflectiveEnv(\mathfrak{M}_i)\\ \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]{\vdash}\Lambda{\sqsubseteq}_\Lambda\Lambda'\ \text{ or }\ \Delta{\vdash}disjoint(\Lambda',\ \Lambda)\end{array}}{\Delta\vdash validRange(\Lambda,\ \mathrm{C}{<}\overline{\mathtt{T}}{>})}\ \text{(VR-CLASS)}$$

Figure 18: FMJ: Auxiliary definitions.

Method type lookup:

$$\frac{\Lambda{=}\langle R_p, oR_n\rangle \qquad R_p{=}(\mathtt{X},\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0)}{\Delta;\Lambda\vdash \mathit{mtype}(\eta,\ \mathtt{X}){=}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0} \quad \text{(MT-VAR-RI)}$$

$$\frac{\Lambda{=}\langle R_p, {+}R_n\rangle \qquad R_n{=}(\mathtt{X},\ \overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0)}{\Delta;\Lambda\vdash \mathit{mtype}(\eta,\ \mathtt{X}){=}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0} \quad \text{(MT-VAR-R2)}$$

$$\frac{\Lambda{=}\langle R_p, oR_n\rangle \qquad R_p{=}(\mathtt{T},\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{V}}\mathtt{\to}\mathtt{V}_0) \qquad \Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathit{bound}_\Delta(\mathtt{X})){=}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0}{\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathtt{X}){=}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0} \quad \text{(MT-VAR-S)}$$

$$\frac{CT(\mathtt{C}){=}\mathtt{class\ C}\mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{>}\mathtt{\lhd N\ \{...\ \ }\overline{\mathtt{M}}\mathtt{\}\quad U_0\ m\ (}\overline{\mathtt{U}}\ \overline{\mathtt{x}}\mathtt{)\ \{\uparrow e;\}}\in\overline{\mathtt{M}}}{\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}){=}[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0)} \quad \text{(MT-CLASS-S)}$$

$$\frac{CT(\mathtt{C}){=}\mathtt{class\ C}\mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{>\lhd N\ \{...\ \ }\overline{\mathtt{M}}\mathtt{\}\quad m}\notin\overline{\mathtt{M}}}{\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}){=}\mathit{mtype}(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N})} \quad \text{(MT-SUPER-S)}$$

$$\frac{\begin{array}{c} CT(\mathtt{C}){=}\mathtt{class\ C}\mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{>\lhd T\ \{...\ \ }\overline{\overline{\mathfrak{M}}}\mathtt{\}} \\ \mathfrak{M}_i{=}\mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>for(}\mathbb{M}_p\mathtt{;}o\mathbb{M}_f\mathtt{)\ S_0\ }\eta\mathtt{\ (}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\ \{\uparrow e;\}}\quad \mathfrak{M}_i\in\overline{\overline{\mathfrak{M}}} \\ \Lambda_d{=}[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathit{reflectiveEnv}(\mathfrak{M}_i)) \\ \begin{cases} \Delta;\Lambda\vdash \mathit{specialize}(\mathtt{m},\ \Lambda_d)=\Lambda_r & \text{if } \mathtt{n}=\mathtt{m} \\ \Lambda_r=\Lambda & \text{if } \mathtt{n}=\eta \end{cases} \qquad \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda\Lambda_d \end{array}}{\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}){=}[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{S}}\mathtt{\to}\mathtt{S}_0)} \quad \text{(MT-CLASS-R)}$$

$$\frac{\begin{array}{ll} & CT(\mathtt{C}){=}\mathtt{class\ C}\mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{>\lhd T\ \{...\ \ }\overline{\overline{\mathfrak{M}}}\mathtt{\}} \\ \text{for all} & \mathfrak{M}_i\in\overline{\overline{\mathfrak{M}}}\qquad \Lambda_d{=}[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathit{reflectiveEnv}(\mathfrak{M}_i)) \\ & \begin{cases} \Delta;\Lambda\vdash \mathit{specialize}(\mathtt{m},\ \Lambda_d)=\Lambda_r & \text{if } \mathtt{n}=\mathtt{m} \\ \Lambda_r=\Lambda & \text{if } \mathtt{n}=\eta \end{cases} \\ \text{implies} & \Delta\vdash \mathit{disjoint}(\Lambda_r,\ \Lambda_d) \end{array}}{\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}){=}\mathit{mtype}(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T})} \quad \text{(MT-SUPER-R)}$$

Field lookup:

$$\Delta\vdash \mathit{fields}(\mathtt{Object})\ {=}\ \bullet \quad \text{(FD-OBJ)}$$

$$\frac{CT(\mathtt{C}){=}\mathtt{class\ C}\mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{>\lhd\ T\ \{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;\ ...\}} \qquad \Delta\vdash \mathit{fields}(\mathit{bound}_\Delta([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T})){=}\overline{\mathtt{D}}\ \overline{\mathtt{g}}}{\Delta\vdash \mathit{fields}(\mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}){=}\overline{\mathtt{D}}\ \overline{\mathtt{g}},[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\ \overline{\mathtt{f}}} \quad \text{(FD-CLASS)}$$

Valid method overriding:

$$\frac{\Delta\vdash \mathit{validRange}(\Lambda,\ \mathtt{T}) \qquad \Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathtt{T}){=}\overline{\mathtt{V}}\mathtt{\to}\mathtt{V}_0\ \text{ implies }\ (\overline{\mathtt{V}}{=}\overline{\mathtt{U}}\quad\text{and}\quad \mathtt{V}_0{=}\mathtt{U}_0\ )}{\Delta;\Lambda\vdash \mathit{override}(\mathtt{n},\ \mathtt{T},\ \overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0)}$$

Figure 19: FMJ: Method type lookup, overriding and field lookup.

- Λ: Reflective iteration environment. Λ has the form $\langle R_p, oR_n \rangle$, where R_p is the primary pattern, and oR_n the nested pattern. Recall that o can be + or -.

  - R_p has the form $(\mathtt{T}_1, <\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}} > \overline{\mathtt{U}} \rightarrow \mathtt{U}_0)$. $\mathtt{T}_1$ is the reflective type, over whose methods R_p iterates. $\overline{\mathtt{Y}}$ are pattern type variables, bounded by $\overline{\mathtt{P}}$, and $\overline{\mathtt{U}} \rightarrow \mathtt{U}_0$ the method pattern.
  - R_n has a similar form: $(\mathtt{T}_2, \overline{\mathtt{V}} \rightarrow \mathtt{V}_0)$. However, note the lack of pattern type variables, due to the (formalism-only) simplification that the nested pattern not use pattern type variables not already bound in the primary pattern.

  There is no nesting of reflective loops. Thus, Λ contains at most one $\langle R_p, oR_n \rangle$ tuple.

We define two functions (Figure 18) to help us construct the reflective environment as well as the two ranges corresponding to the primary and nested pattern easily:

- $\mathit{reflectiveEnv}(\mathfrak{M})$ constructs the Λ corresponding to the reflective declaration $\mathfrak{M}$.
- $\mathit{range}(\mathbb{M}, <\overline{\mathtt{Y}} \lhd \overline{\mathtt{Q}}>)$ constructs the R corresponding to the pattern $\mathbb{M}$, where $\overline{\mathtt{Y}}$ are the pattern type variables, and bounded by $\overline{\mathtt{Q}}$.

We use the $\mapsto$ symbol for mappings in the environments. For example, $\Delta = \mathtt{X} \mapsto \mathtt{C}<\overline{\mathtt{T}}>$ means that $\Delta(\mathtt{X}) = \mathtt{C}<\overline{\mathtt{T}}>$. Every type variable must be bounded by a non-variable type. The function $\mathit{bound}_\Delta(\mathtt{T})$ returns the upper bound of type $\mathtt{T}$ in Δ. $\mathit{bound}_\Delta(\mathtt{N}) = \mathtt{N}$, if $\mathtt{N}$ is not a type variable. And $\mathit{bound}_\Delta(\mathtt{X}) = \mathit{bound}_\Delta(\mathtt{S})$, where $\Delta(\mathtt{X}) = \mathtt{S}$.

In order to keep our type rules manageable, we make two simplifying assumptions. To avoid burdening our rules with renamings, we assume that pattern type variables have globally unique names (i.e., are distinct from pattern type variables in other reflective environments, as well as from non-pattern type variables). We also assume that all pattern type variables introduced by a reflective block are bound (i.e., used) in the corresponding primary pattern. Checking this property is easy and purely syntactic.

The core of this type system is in determining reflective range containment and disjointness. Thus, we begin our discussion with a detailed explanation of the rules for containment and disjointness.

Reflective range containment:

$$\frac{\Lambda{=}\langle R_p, oR_n\rangle \qquad \Lambda'{=}\langle R'_p, o'R'_n\rangle \qquad R'_p{=}(\mathtt{T}'_p,\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{V}}\mathtt{\to}\mathtt{V}_0) \qquad \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_p \sqsubseteq_R R'_p \qquad \begin{cases} \Delta;\bullet \vdash R_n \sqsubseteq_R [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R'_n & \text{if } o = o' = + \\ \Delta;\bullet \vdash [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R'_n \sqsubseteq_R R_n & \text{if } o = o' = - \end{cases}}{\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash \Lambda \sqsubseteq_\Lambda \Lambda'} \quad \text{(SB-}\Lambda\text{)}$$

Single range containment:

$$\frac{R_1{=}(\mathtt{T}_1,\ \mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0) \quad R_2{=}(\mathtt{T}_2,\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{V}}\mathtt{\to}\mathtt{V}_0) \quad \Delta\vdash\overline{\mathtt{P}}\mathtt{,}\overline{\mathtt{Q}}\ \mathtt{OK} \quad \Delta\vdash \mathtt{T}_2\mathtt{<:}\mathtt{T}_1 \quad \Delta'{=}\Delta,\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{Q}},\overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{P}} \quad \Delta';[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash \mathit{unify}(\mathtt{U}_0{:}\overline{\mathtt{U}},\ \mathtt{V}_0{:}\overline{\mathtt{V}})}{\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_1 \sqsubseteq_R R_2} \quad \text{(SB-}R\text{)}$$

Reflective range disjointness:

$$\frac{\Lambda{=}\langle R_p, oR_n\rangle \qquad \Lambda'{=}\langle R'_p, o'R'_n\rangle \qquad \Delta\vdash {+}R_p \otimes {+}R'_p \ \text{ or } \ \Delta\vdash {+}R_p \otimes o'R'_n \ \text{ or } \ \Delta\vdash {+}R'_p \otimes oR_n \ \text{ or } \ \Delta\vdash oR_n \otimes o'R'_n}{\Delta\vdash \mathit{disjoint}(\Lambda,\ \Lambda')} \quad \text{(DS-}\Lambda\text{)}$$

Mutually exclusion of range conditions:

$$\frac{R_1{=}(\mathtt{T},\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0) \quad R_2{=}(\mathtt{S},\ \mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{V}}\mathtt{\to}\mathtt{V}_0) \quad \Delta\vdash \mathtt{T}\mathtt{<:}\mathtt{S} \ \text{ or } \ \mathtt{S}\mathtt{<:}\mathtt{T} \quad \Delta'{=}\Delta,\overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{P}},\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{Q}} \quad \overline{\mathtt{Z}}{=}\overline{\mathtt{X}}\mathtt{:}\overline{\mathtt{Y}} \quad \text{for all } \overline{\mathtt{W}},\ \ \Delta';[\overline{\mathtt{W}}/\overline{\mathtt{Z}}]\vdash \mathit{unify}(\overline{\mathtt{U}},\ \overline{\mathtt{V}}) \quad \text{implies} \quad [\overline{\mathtt{W}}/\overline{\mathtt{Z}}]\mathtt{U}_0 \neq [\overline{\mathtt{W}}/\overline{\mathtt{Z}}]\mathtt{V}_0}{\Delta\vdash {+}R_1 \ \otimes \ {+}R_2} \quad \text{(ME-1)}$$

$$\frac{R_1{=}(\mathtt{T},\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0) \quad R_2{=}(\mathtt{S},\ \mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{V}}\mathtt{\to}\mathtt{V}_0) \quad \Delta'{=}\Delta,\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}},\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{Q}} \quad \Delta';[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash R_1 \sqsubseteq_R R_2}{\Delta\vdash {+}R_1 \ \otimes \ {-}R_2} \quad \text{(ME-2)}$$

Figure 20: FMJ: Containment and disjointness rules

3.7.2.1 Containment and Disjointness

The range of a reflective environment, $\langle R_p, oR_n \rangle$, comprises methods in the primary range R_p, that also satisfy the nested pattern oR_n. The nested pattern +R_n (or -R_n) is satisfied if there is at least one method (or no method, resp.) in the range of R_n. We call ranges of R_p and R_n *single ranges*. In this section, we explain the rules for determining the following three relations:

- $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}] \vdash \Lambda \sqsubseteq_{\Lambda} \Lambda'$. Range of Λ is contained within the range of Λ', under the assumptions of type environment Δ and the unifying type substitutions of $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]$.
- $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}] \vdash R_1 \sqsubseteq_{R} R_2$. Single range R_1 is contained within single range R_2, under the assumptions of Δ and the unifying type substitutions of $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]$.
- $\Delta \vdash \mathit{disjoint}(\Lambda, \Lambda')$. The range of Λ and Λ' are disjoint under the assumptions of Δ.

Single range containment. In determining the containment between two reflective environments, we must first see how containment is determined between two single ranges. Rule SB-R (Figure 20) defines the two conditions for single range containment. First, the reflective type of the larger range, R_2, should be a subtype of R_1's reflective type. It is only meaningful to talk about containment if the reflective set of R_2 (i.e., all methods of the reflective type of R_2, regardless of whether they can be matched by the pattern) can be mapped *onto* the reflective set of R_1. We determine this relation using subtyping: A subtype's methods can be mapped onto its supertype's methods. Secondly, R_2's pattern should be more general than R_1's pattern. This means that a *one-way* unification exists from the pattern of R_2 to the pattern of R_1, where only the pattern type variables in R_2 are considered variables in the unification process. $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]$ are the substitutions that satisfy such one-way unification: $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}] \vdash \mathit{unify}(\mathtt{U}_0{:}\overline{\mathtt{U}},\ \mathtt{V}_0{:}\overline{\mathtt{V}})$.

Rule UNI (Figure 21) describes a standard unification condition with a twist: unifying substitutions (for pattern type variables) must respect the subtyping bounds of the type variables. For example, the substitution [`Y`/`Object`], where $\Delta \vdash$ `Y`<:`Number`, does *not* unify `Y` and `Object`, because the bound of `Y` is tighter than `Object`. Depending on whether the unification variables $\overline{\mathtt{Z}}$ appear in $\overline{\mathtt{T}}$, $\overline{\mathtt{S}}$, or both, the ability to unify $\overline{\mathtt{T}}$ and $\overline{\mathtt{S}}$ may mean all types matched by $\overline{\mathtt{S}}$ can be matched by $\overline{\mathtt{T}}$ (one-way unification), vice versa, or that there is some intersection in the types

matched by $\overline{\mathtt{T}}$ and $\overline{\mathtt{S}}$ (two-way unification).

Figure 21 also lists a number of rules defining when type T is a valid substitution for S: $\Delta \vdash \mathtt{T} \prec: \overline{\mathtt{Z}} \mathtt{S}$. This is to say that T and S can match at least some of the same types, using $\overline{\mathtt{Z}}$ as pattern type variables. The most complex case in the pattern matching rules is PM-PVARS, which defines when there is an intersection in the types matched by two different pattern type variables, $\mathtt{Z}_i$ and $\mathtt{Z}_j$. Recall that if $\mathtt{Z}_i$ is bounded by type $\mathtt{T}_i$, it can match any subtype of $\mathtt{T}_i$; similarly, $\mathtt{Z}_j$ bounded by type $\mathtt{T}_j$ can match any subtype of $\mathtt{T}_j$. Technically, for there to be an intersection in the types $\mathtt{Z}_i$ and $\mathtt{Z}_j$ can match, there must be a type that is a subtype of both $\mathtt{T}_i$ and $\mathtt{T}_j$. Since we are modeling a core subset of Java without multiple inheritance (of interfaces), this type must be a subtype of either $\mathtt{T}_i$ or $\mathtt{T}_j$. This is to say, either $\mathtt{Z}_i$ has a greater upper bound than $\mathtt{Z}_j$ and is capable of matching every type matched by $\mathtt{Z}_j$, or vice versa. PM-PVARS describes this either-or condition by taking one of the type variables up to its bound, and invoking the pattern matching rules on that bound and the remaining type variable.

PM-VAR says that for a pattern matching type variable Z to match a regular type T, the upper bound of Z must be able to match the upper bound of T. If the bounds are the exact same types (PM-REFL), we are done. If they are types constructed from the same generic type, we recursively invoke the pattern matching rules on their respective type parameters (PM-CL). If T's declared upper bound is a strict subtype of the upper bound of Z, rule PM-CL-S traces up the declared superclass of T's upper bound, until either PM-REFL or PM-CL can be invoked, or the class `Object` is reached and no more recursion can occur.

Reflective (nested) range containment. SB-Λ (Figure 20) defines the conditions for the range of reflective environment $\Lambda = \langle R_p, oR_n \rangle$ to be contained within the range of $\Lambda' = \langle R'_p, o'R'_n \rangle$. These conditions reflect precisely the informal rules we presented in the previous section. First, regardless of nested patterns, the *primary* range of Λ should at least be contained within the primary range of Λ'. Second, for every method in R_p that satisfies the nested pattern oR_n, the corresponding method in R'_p should satisfy the nested pattern $o'R'_n$. There are a couple of ways to guarantee oR_n *implies* $o'R'_n$. If $+R_n$ is true, and R_n is contained within R'_n, then $+R'_n$ is also true (i.e., if there is at least one method in R_n, then there is at least one method in a larger

range, R'_n). This condition is expressed by $\Delta;\bullet \vdash R_n \sqsubseteq_R [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R'_n$, if $o{=}o'{=}+$. We apply the unifying type substitutions for the primary ranges to the nested range R'_n: In order to properly compare the ranges of R_n and R'_n, we need to restrict R'_n to what it can be for the methods that are matched by both R_p and R'_p. Note that we are using an empty sequence of type substitutions ($\bullet$) in determining that R_n is contained within $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R'_n$. This is because nested patterns do not have pattern type variables of their own, and pattern type variables from the primary pattern are treated as constants in the nested patterns. Similarly, if -R_n is true, and R_n contains R'_n, -R'_n is also true.

Reflective range disjointness. Disjointness of reflective ranges is defined by rule DS-Λ (Figure 20). DS-Λ says that for two reflective ranges to be disjoint, we must be able to find one pair of mutually exclusive ($\otimes$) pattern conditions—this includes the implicit (+) conditions stated by the primary patterns. There are two rules for pattern condition mutual exclusion: ME-1 and ME-2, defined in Figure 20.

ME-1 says that two + pattern conditions are mutually exclusive if the patterns reflect over types with an inheritance relationship, and the patterns' argument types unify, whereas their return types do not. This means the two patterns are matching over methods of the same argument types, but different return types. For both of these pattern conditions to be satisfied, i.e. at least one method in each range, there need to be at least two methods with the same argument types and different return types, declared by the same class (or one is declared by a subclass). This violates the correct method overriding rule of well-formed types. Since these patterns only reflect over well-formed types, these conditions can never be both satisfied.

ME-2 says that +R_1 and -R_2 are mutually exclusive, if R_1 is contained within R_2. If there is no method in the larger range, then certainly there cannot be at least one method in a smaller range. And vice versa. Thus, the two conditions can never be both satisfied.

These rules reflect very closely the informal rules we presented previously, modulo the small differences in the formalism mentioned in Section 3.7.1: We do not need to distinguish between generated/declared patterns and primary patterns in the formalism, as the uniqueness of entities in the primary pattern implies (through name uniqueness, since there is no overloading) the

uniqueness of declared entities.

Additionally, DS-Λ makes two overly conservative (and sound) simplifications over our implementation. First, it is possible for the + condition stated by a primary pattern to be mutually exclusive from the nested condition of its own nested pattern. This results in a reflective declaration that can never be expanded, and thus, a range that is disjoint from every other range. We do not check for this in our formalism. Secondly, when two pattern conditions are shown to be mutually exclusive, a one- or two-way unification exists between either the argument types (ME-1), or the argument and return types (ME-2, via SB-R). This unification represents the methods that lie in the intersection of the two ranges defined by the patterns. In practice, we only need to check whether these methods can in fact exist, by checking whether this particular mapping causes any of the remaining patterns to be mutually exclusive. However, in this formalism, we do not apply this unification, and thus check for general mutual exclusion.

Type Unification:

$$\frac{[\overline{\mathtt{U}}/\overline{\mathtt{Z}}]\overline{\mathtt{T}}=[\overline{\mathtt{U}}/\overline{\mathtt{Z}}]\overline{\mathtt{S}} \quad \text{for all } \mathtt{Z}_i{\in}\overline{\mathtt{Z}},\ \Delta{\vdash}\mathtt{U}_i{\prec:_{\overline{\mathtt{Z}}}}\mathtt{Z}_i}{\Delta;[\overline{\mathtt{U}}/\overline{\mathtt{Z}}]{\vdash}\mathit{unify}(\overline{\mathtt{T}},\ \overline{\mathtt{S}})} \quad \text{(UNI)}$$

Pattern matching rules:

$$\Delta{\vdash}\mathtt{T}{\prec:_{\overline{\mathtt{Z}}}}\mathtt{T} \quad \text{(PM-REFL)}$$

$$\frac{\Delta{\vdash}\overline{\mathtt{T}}{\prec:_{\overline{\mathtt{Z}}}}\overline{\mathtt{S}}}{\Delta{\vdash}\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}{\prec:_{\overline{\mathtt{Z}}}}\mathtt{C}\texttt{<}\overline{\mathtt{S}}\texttt{>}} \quad \text{(PM-CL)}$$

$$\frac{CT(\mathtt{C})=\texttt{class C<}\overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}\texttt{>}\triangleleft\mathtt{T}\ \texttt{\{...\}} \qquad \Delta{\vdash}[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}{\prec:_{\overline{\mathtt{Z}}}}\mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}}{\Delta{\vdash}\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}{\prec:_{\overline{\mathtt{Z}}}}\mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}} \quad \text{(PM-CL-S)}$$

$$\frac{\mathtt{Z}{\in}\overline{\mathtt{Z}} \quad \mathtt{T}{\notin}\overline{\mathtt{Z}} \quad \mathit{bound}_\Delta(\mathtt{T})=\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>} \quad \Delta{\vdash}\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}{\prec:_{\overline{\mathtt{Z}}}}[\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}/\mathtt{Z}]\mathit{bound}_\Delta(\mathtt{Z})}{\Delta{\vdash}\mathtt{T}{\prec:_{\overline{\mathtt{Z}}}}\mathtt{Z}} \quad \text{(PM-VAR)}$$

$$\frac{\mathtt{Z}_i{\in}\overline{\mathtt{Z}} \quad \mathtt{Z}_j{\in}\overline{\mathtt{Z}} \quad \begin{cases} \Delta \vdash [\mathtt{Z}_i/\mathtt{Z}_j]\mathit{bound}_\Delta(\mathtt{Z}_j) \prec:_{\overline{\mathtt{Z}}} \mathtt{Z}_i & \mathit{or} \\ \Delta \vdash [\mathtt{Z}_j/\mathtt{Z}_i]\mathit{bound}_\Delta(\mathtt{Z}_i) \prec:_{\overline{\mathtt{Z}}} \mathtt{Z}_j \end{cases}}{\Delta{\vdash}\mathtt{Z}_i{\prec:_{\overline{\mathtt{Z}}}}\mathtt{Z}_j} \quad \text{(PM-PVARS)}$$

Figure 21: FMJ: Unification and pattern matching functions.

3.7.2.2 Valid Method Invocation

The rest of the typing rules add the machinery to standard FGJ type checking to express checks using range containment and disjointness. For instance, to determine valid method invocation is

to determine that the reflective environment of the invocation is contained within the reflective environment of the declaration. T-INVK (Figure 17) specifies conditions for a well-typed method invocation. It relies on $\Delta;\Lambda \vdash mtype(\mathtt{n},\ \mathtt{T})$ (Figure 19) to handle the complexities in determining the type of method $\mathtt{n}$ in $\mathtt{T}$, where $\mathtt{n}$ can either be a constant or variable name. We highlight the interesting *mtype* rules.

MT-VAR-R1 and MT-VAR-R2 say that the type of method with a variable name η in a type $\mathtt{X}$, where $\mathtt{X}$ is either the reflective type for the primary pattern or the reflective type of a *positive* nested pattern, is exactly the type specified by the primary (or nested, respectively) pattern. Otherwise, if $\mathtt{T}$ is a type variable, then we must look for the method type in its bound (MT-VAR-S).

MT-CLASS-R lists conditions for retrieving the type of $\mathtt{n}$ in $\mathtt{C}\langle\overline{\mathtt{T}}\rangle$, where $\mathtt{C}\langle\overline{\mathtt{X}}\rangle$ has reflectively declared methods. If $\mathtt{n}$ is a name variable, this is simply checking that the range of reference, which is the current reflective environment, is contained within the declaration reflective environment. When $\mathtt{n}$ is a constant name $\mathtt{m}$, however, we need to check whether $\mathtt{m}$ is within the range of method names in the declaration reflective range. We do so by *specializing* the declaration reflective environment using $\mathtt{m}$. Specializing a range based on a constant name is just a way to package the information for uniform use by our containment and disjointness checks (which apply to entire ranges with patterns and not single methods). The main property of $specialize(\mathtt{m},\ \Lambda_d){=}\Lambda_r$ (Figure 5) is that $\mathtt{m}$ is the name of a method in the declaration range, Λ_d, (i.e., a declared method), if and only if the specialized range, Λ_r, is subsumed by Λ_d.

The result of *mtype* is the declared types, with the substitutions of $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]$, and the type substitutions for unifying the declaration range and the reference range, $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]$.

Subtyping rules:

$$\Delta \vdash \mathtt{T}<:\mathtt{T} \qquad \text{(S-REFL)}$$

$$\Delta \vdash \mathtt{X} <:\ \Delta(\mathtt{X}) \qquad \text{(S-VAR)}$$

$$\frac{\Delta \vdash \mathtt{S}<:\mathtt{T} \quad \Delta \vdash \mathtt{T}<:\mathtt{U}}{\Delta \vdash \mathtt{S}<:\mathtt{U}} \qquad \text{(S-TRANS)}$$

$$\frac{CT(\mathtt{C}){=}\mathtt{class\ C}\langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\rangle\triangleleft\ \mathtt{T}\ \ \{...\}}{\Delta \vdash \mathtt{C}\langle\overline{\mathtt{T}}\rangle\ <:\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}} \qquad \text{(S-CLASS)}$$

Figure 22: FMJ: Subtyping rules.

3.7.2.3 Uniqueness of Definitions

T-METH-R (Figure 17) ensures that methods declared within one reflective block do not conflict with methods in the superclass (i.e., we have proper overriding). The condition is enforced using *override* (Figure 19). $override(\mathtt{n},\ \mathtt{T},\ \overline{\mathtt{U}}{\to}\mathtt{U}_0)$ determines whether method `n`, defined in some *subclass* of `T` with type signature $\overline{\mathtt{U}}{\to}\mathtt{U}_0$, properly overrides method `n` in `T`. If method `n` exists in `T`, it must have the exact same argument and return types as `n` in the subclass. (We made a simplification over FGJ: FGJ allows a covariant return type for overriding methods, whereas we disallow it to simplify the pattern matching rules in Figure 21.) Additionally, the reflective range of `n` in the subclass must be either completely contained within one of `T`'s reflective ranges, or disjoint from all the reflective ranges of `T` (and, transitively, `T`'s superclasses). This condition is enforced using $\Delta{\vdash}validRange(\Lambda,\ \mathtt{T})$ (Figure 18).

T-CLASS-R ensures that the reflective blocks within a well-typed class have no declarations that conflict with each other, by requiring ranges of reflective blocks in a class to be disjoint pairwise. Since each block has unique names within itself, the pairwise disjointenss guarantees names across all blocks are unique, as well.

3.7.3 Soundness

We prove the soundness of FMJ by proving Subject Reduction and Progress for an expression `e`. Figure 23 defines the operational semantics of FMJ, and Figure 24 defines the method body lookup rules necessary for the operational semantics. Note that method body lookup rules are only defined for lookups under a constant name `m`. A name variable is only meaningful under a reflective environment Λ. But reduction rules, and thus method body lookup, are done under empty environments, where $\Lambda{=}\emptyset$. Thus it is not meaningful to define method body lookup for name variable η.

Next, we show the main theorems, important supporting lemmas, and their proof sketches. It should not surprised the reader that the highlighted lemmas are generally those supporting method lookup or invocation, which are the most complex parts of our language and formalism, and also the parts that deviate most from the original FGJ. Detailed proofs can be found in Appendix B.

Reduction Rules:

$$\frac{\emptyset \vdash \mathit{fields}(\mathtt{C<\overline{T}>}) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}}{\mathtt{new\ C<\overline{T}>(\overline{e}).f}_i \longrightarrow \mathtt{e}_i} \quad \text{(R-FIELD)}$$

$$\frac{\mathit{mbody}(\mathtt{m},\ \mathtt{C<\overline{T}>}) = (\overline{\mathtt{x}}, \mathtt{e}_0)}{\mathtt{new\ C<\overline{T}>(\overline{e}).m(\overline{d}) \longrightarrow [\overline{d}/\overline{x},\ new\ C<\overline{T}>(\overline{e})/this]e_0}} \quad \text{(R-INVK)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}'_0}{\mathtt{e}_0\mathtt{.f} \longrightarrow \mathtt{e}'_0\mathtt{.f}} \quad \text{(RC-FIELD)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}'_0}{\mathtt{e}_0\mathtt{.m(\overline{e})} \longrightarrow \mathtt{e}'_0\mathtt{.m(\overline{e})}} \quad \text{(RC-INV-RECV)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}'_i}{\mathtt{e}_0\mathtt{.m(...,e}_i\mathtt{,...)} \longrightarrow \mathtt{e}_0\mathtt{.m(...,e}'_i\mathtt{,...)}} \quad \text{(RC-INV-ARG)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}'_i}{\mathtt{new\ C<\overline{T}>(...,e}_i\mathtt{,...)} \longrightarrow \mathtt{new\ C<\overline{T}>(...,e}'_i\mathtt{,...)}} \quad \text{(RC-NEW-ARG)}$$

Figure 23: FMJ: Reduction Rules

Method body lookup:

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<\overline{X}\lhd\overline{N}>\lhd N\ \{...\ \ \overline{M}\}} \quad \mathtt{U}_0\ \mathtt{m}\ \mathtt{(\overline{U}\ \overline{x})}\ \mathtt{\{\uparrow e;\}} \in \mathtt{\overline{M}}}{\mathit{mbody}(\mathtt{m},\ \mathtt{C<\overline{T}>}) = [\mathtt{\overline{T}/\overline{X}}](\mathtt{\overline{x}}, \mathtt{e})} \quad \text{(MB-CLASS-S)}$$

$$\frac{\begin{array}{c} CT(\mathtt{C}) = \mathtt{class\ C<\overline{X}\lhd\overline{N}>\lhd T\ \{...\ \ \overline{\mathfrak{M}}\}} \\ \mathfrak{M}_i \in \overline{\mathfrak{M}} \quad \mathfrak{M}_i = \mathtt{<\overline{Y}\lhd\overline{P}>for(}\mathbb{M}_p\mathtt{;}o\mathbb{M}_f\mathtt{)\ S}_0\ \eta\ \mathtt{(\overline{S}\ \overline{x})\ \{\uparrow e;\}} \\ \Delta = \mathtt{\overline{X}<:\overline{N},\overline{Y}<:\overline{P}} \qquad \Lambda_d = [\mathtt{\overline{T}/\overline{X}}](\mathit{reflectiveEnv}(\mathfrak{M}_i)) \\ \Delta;\emptyset \vdash \mathit{specialize}(\mathtt{m},\ \Lambda_d) = \Lambda_r \qquad \Delta;[\mathtt{\overline{W}/\overline{Y}}] \vdash \Lambda_r \sqsubseteq_\Lambda \Lambda_d \end{array}}{\mathit{mbody}(\mathtt{m},\ \mathtt{C<\overline{T}>}) = [\mathtt{\overline{T}/\overline{X}}][\mathtt{\overline{W}/\overline{Y}}](\mathtt{\overline{x}}, [\mathtt{m}/\eta]\mathtt{e})} \quad \text{(MB-CLASS-R)}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<\overline{X}\lhd\overline{N}>\lhd N\ \{...\ \ \overline{M}\}} \quad \mathtt{m} \notin \mathtt{\overline{M}}}{\mathit{mbody}(\mathtt{m},\ \mathtt{C<\overline{T}>}) = \mathit{mbody}(\mathtt{m},\ [\mathtt{\overline{T}/\overline{X}}]\mathtt{N})} \quad \text{(MB-SUPER-S)}$$

$$\frac{\begin{array}{ll} & CT(\mathtt{C}) = \mathtt{class\ C<\overline{X}\lhd\overline{N}>\lhd T\ \{...\ \ \overline{\mathfrak{M}}\}} \\ \text{for all} & \mathfrak{M}_i \in \overline{\mathfrak{M}}\ \ \mathfrak{M}_i = \mathtt{<\overline{Y}\lhd\overline{P}>for(}\mathbb{M}_p\mathtt{;}o\mathbb{M}_f\mathtt{)\ S}_0\ \eta\ \mathtt{(\overline{S}\ \overline{x})\ \{\uparrow e;\}} \\ & \Delta = \mathtt{\overline{X}<:\overline{N},\overline{Y}<:\overline{P}} \qquad \Lambda_d = [\mathtt{\overline{T}/\overline{X}}](\mathit{reflectiveEnv}(\mathfrak{M}_i)) \\ & \Delta;\emptyset \vdash \mathit{specialize}(\mathtt{m},\ \Lambda_d) = \Lambda_r \\ \text{implies} & \Delta \vdash \mathit{disjoint}(\Lambda_r,\ \Lambda_d) \end{array}}{\mathit{mbody}(\mathtt{m},\ \mathtt{C<\overline{T}>}) = \mathit{mbody}(\mathtt{m},\ [\mathtt{\overline{T}/\overline{X}}]\mathtt{T})} \quad \text{(MB-SUPER-R)}$$

Figure 24: FMJ: Method body lookup rules.

Theorem 4 [Subject Reduction]: If $\Delta;\Lambda;\Gamma \vdash \mathtt{e} \in \mathtt{T}$, and $\mathtt{e} \rightarrow \mathtt{e}'$, then $\Delta;\Lambda;\Gamma \vdash \mathtt{e}' \in \mathtt{S}$ and $\Delta \vdash \mathtt{S} <: \mathtt{T}$ for some S.

Proof Sketch: We prove by structural induction on the reduction rules in Figure 23. The most interesting case is R-INVK, where $\mathtt{e}=\mathtt{new\ C<\overline{T}>(\overline{e}).m(\overline{d})}$, $\mathtt{e}'=\mathtt{[\overline{d}/\overline{x},new\ C<\overline{T}>/this]}\mathtt{e}_0$.

It is easy to see from R-INVK, T-NEW, and T-INVK that,

$mbody(\mathtt{m},\ \mathtt{C<\overline{T}>})=(\overline{\mathtt{x}},\mathtt{e}_0)$ $\quad \Delta \vdash \mathtt{new\ C<\overline{T}>(\overline{e})} \in \mathtt{C<\overline{T}>}$ $\quad \Delta \vdash \mathtt{C<\overline{T}>}\ ok$

$\Delta;\Lambda \vdash mtype(\mathtt{m},\ \mathtt{C<\overline{T}>})=\overline{\mathtt{T}}' \rightarrow \mathtt{T}$ $\quad \Delta;\Gamma;\Lambda \vdash \overline{\mathtt{d}} \in \overline{\mathtt{S}}$ $\quad \Delta \vdash \overline{\mathtt{S}} <: \overline{\mathtt{T}}'$

The conclusion follows from the following: 1) the expression $\mathtt{e}_0$, obtained via *mbody*, is of type $\mathtt{S}'$ where $\Delta \vdash \mathtt{S}' <: \mathtt{T}$: that is, the body of the method is a subtype of its defined return type (Lemma 16); 2) the expression after term substitution, $\mathtt{e}'=\mathtt{[\overline{d}/\overline{x},new\ C<\overline{T}>/this]}\mathtt{e}_0$, is of type S where $\Delta \vdash \mathtt{S} <: \mathtt{S}'$ (Lemma 19).

Lemma 16: If $\Delta;\Lambda \vdash mtype(\mathtt{m},\ \mathtt{C<\overline{T}>})=\overline{\mathtt{S}} \rightarrow \mathtt{S}$, $mbody(\mathtt{m},\ \mathtt{C<\overline{T}>})=(\overline{\mathtt{x}},\mathtt{e}_0)$, where $\Delta \vdash \mathtt{C<\overline{T}>}\ ok$, then there exists a type $\mathtt{S}'$ such that $\Delta \vdash \mathtt{S}' <: \mathtt{S}$, $\Delta \vdash \mathtt{S}'\ ok$, $\Delta;\Lambda;\overline{\mathtt{x}} \mapsto \overline{\mathtt{S}},\mathtt{this} \mapsto \mathtt{C<\overline{T}>} \vdash \mathtt{e}_0 \in \mathtt{S}'$.

Proof Sketch: We prove by induction on the rules of *mbody* (Figure 24). Case MB-CLASS-S does not involve any reflectively declared methods, and thus has a similar proof to that used in FGJ proofs. Cases MB-SUPER-S and MB-SUPER-R can be easily proven through the induction hypothesis. The most interesting case, thus, is MB-CLASS-R, where method body is retrieved from a reflectively declared method $\mathfrak{M}_i$. The corresponding *mtype* is retrieved using rule MT-CLASS-R.

First, notice that there is a bit of ambiguity in MT-CLASS-R (as well as MB-CLASS-R), where $\mathfrak{M}_i$ is defined to be one of $\overline{\mathfrak{M}}$. We first prove using Lemma 17 that there is indeed only one such $\mathfrak{M}_i$ that satisfies the other conditions for the rule to hold.

It is obvious from the correspondence between MT-CLASS-R and MB-CLASS-R, then,

$\Lambda_d=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](reflectiveEnv(\mathfrak{M}_i))$ $\quad \Delta;\Lambda \vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda_r$ $\quad \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}] \vdash \Lambda_r \sqsubseteq_\Lambda \Lambda_d$

$\Delta;\Lambda \vdash mtype(\mathtt{m},\ \mathtt{C<\overline{T}>})=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{S}}'' \rightarrow \mathtt{S}'')$ $\quad mbody(\mathtt{m},\ \mathtt{C<\overline{T}>})=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{x}},[\mathtt{m}/\eta]\mathtt{e})$

Through T-METH-R, we have:

$\Delta''=\overline{\mathtt{X}} <: \overline{\mathtt{N}},\overline{\mathtt{Y}} <: \overline{\mathtt{P}}$ $\quad \Lambda''=reflectiveEnv(\mathfrak{M}_i)$ $\quad \Gamma''=\overline{\mathtt{x}} \mapsto \overline{\mathtt{S}}'',\mathtt{this} \mapsto \mathtt{C<\overline{X}>}$

$\Delta'';\Lambda'';\Gamma'' \vdash \mathtt{e} \in \mathtt{V}$ $\quad \Delta'' \vdash \mathtt{V} <: \mathtt{S}''$

Thus, to prove that $\Delta;\Lambda;\overline{\mathtt{x}} \mapsto \overline{\mathtt{S}},\mathtt{this} \mapsto \mathtt{C<\overline{T}>} \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta]\mathtt{e} \in [\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{S}''$, we need to show that 1) type substitutions $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]$, which are obtained through type-instantiation of a generic class

(e.g., instantiating C<$\overline{\mathtt{X}}$> with $\overline{\mathtt{T}}$) preserve expression typing as well as subtyping (Lemma 23 and Lemma 21, respectively, which are similarly used in other FGJ-based formalisms), 2) type substitutions $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]$ and name substitution $[\mathtt{m}/\eta]$, when $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]$ and $\mathtt{m}$ are results of a *specialize* operation, also preserves typing (Lemma 35).

Lemma 19 (Term Substitution Preserves Typing). If $\Delta;\Lambda;\Gamma,\overline{\mathtt{x}}\mapsto\overline{\mathtt{T}}\vdash\mathtt{e}\in\mathtt{T}$, η does not appear in $\mathtt{e}$, $\Delta;\Lambda;\Gamma\vdash\overline{\mathtt{d}}\in\overline{\mathtt{S}}$, $\Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{T}}$, then $\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}\in\mathtt{T}'$, for some $\mathtt{T}'$ where $\Delta;\Lambda;\Gamma\vdash\mathtt{T}'<:\mathtt{T}$.

Proof Sketch: We prove by induction on expression typing rules T-* (Figure 17). The most interesting case is T-INVK, where $\mathtt{e}=\mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)}$. By induction hypothesis,

$$\Delta;\Lambda;\Gamma\vdash\mathtt{e}_0\in\mathtt{T}_0 \quad \Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0\in\mathtt{T'}_0 \quad \Delta\vdash\mathtt{T'}_0<:\mathtt{T}_0$$

We need to show that $\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{T}_0)=\overline{\mathtt{T}}\rightarrow\mathtt{T}$ implies $\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{T'}_0)=\overline{\mathtt{T}}\rightarrow\mathtt{T}$. That is to say, the method $\mathtt{m}$ is defined for a superclass, then it is defined in the subclass, as well, with the same type signature. Lemma 34, which is a standard lemma for FGJ-based formalisms, proves this point. However, in our proofs of Lemma 34, we needed to prove two additional, interesting lemmas regarding the properties of range containment. Lemma 31 shows that type substitutions preserve single range containment; Lemma 38 shows that single range containment is transitive. The preservation of reflective range (i.e., primary and nested) containment and transitivity then follows easily from these lemmas regarding single ranges.

Lemma 31 (Substitution Preserves Single Range Containment): If $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_1\sqsubseteq_R R_2$, $\Delta_1\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$, where $\Delta_1\vdash\overline{\mathtt{U}}$ ok, and none of $\overline{\mathtt{X}}$ appears in Δ_1, none of $\overline{\mathtt{X}}$ appears on $\overline{\mathtt{Y}}$, then $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash R_1\sqsubseteq_R R_2$, where $\overline{\mathtt{W}}'=[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{W}}$.

Lemma 38 (Single Range Containment is Transitive): If $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_1\sqsubseteq_R R_2$, $\Delta;[\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\vdash R_2\sqsubseteq_R R_3$, then $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\vdash R_1\sqsubseteq_R R_3$,

Proof Sketch: Both proofs follow from analysis of rule SB-R.

Theorem 5 [Progress]: Let $\mathtt{e}$ be a well-typed expression. 1. If $\mathtt{e}$ has $\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{).f}$ as a subexpression, then $\emptyset\vdash fields(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=\overline{\mathtt{U}}\ \overline{\mathtt{f}}$, and $\mathtt{f}=\mathtt{f}_i$. 2. If $\mathtt{e}$ has $\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{).m(}\overline{\mathtt{d}}\mathtt{)}$ as a subexpression, then $mbody(\mathtt{m},\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=(\overline{\mathtt{x}},\mathtt{e}_0)$ and $|\overline{\mathtt{x}}|=|\overline{\mathtt{d}}|$.

Proof Sketch: Proof follows from T-FIELD and T-INVK respectively, and from the well-typedness

of subexpressions.

Theorem 6 [Type Soundness]: If $\emptyset;\emptyset;\emptyset \vdash e \in T$ and $e \longrightarrow^* e'$, then e' is a value v such that $\emptyset;\emptyset;\emptyset \vdash v \in S$ and $\emptyset \vdash S <: T$ for some type S.

Proof: Follows from Theorems 4 and 5.

3.8 MorphJ Implementation

It should not be a surprise to readers that MorphJ cannot be implemented using an erasure-based approach, as is adopted in the implementation of Java generics. In an erasure-based implementation, all type-instantiations of a generic share the same copy of bytecode. The type variables of a generic are "erased" to their upper bounds in that shared bytecode. Appropriate casts are inserted at the use-sites of type-instantiations. This particular approach works for Java generics because the *structure* of a Java generic does not vary with its type-instantiations—a generic always declares the same methods, fields, and superclasses regardless of its type arguments. A MorphJ generic class, however, may declare different methods and fields, as well as different supertypes, depending on its type arguments. Thus, it is impossible for different type-instantiations of a MorphJ generic to share the same copy of bytecode. MorphJ is thus implemented via expansion: Every type-instantiation of a MorphJ generic is expanded to its own bytecode representation.

Unlike the expansion-based approach taken by C++ templates, a MorphJ generic is still separately compiled. After type-checking, a MorphJ generic is compiled to an annotated bytecode file. The bytecode sequences corresponding to reflectively declared code are annotated with information pertaining to the associated reflective iterator: patterns, nested patterns, and pattern-matching variables. This copy of bytecode is then used as a template for expansion. Upon type-instantiation, the annotated bytecode sequences are expanded using information in the annotations. For instance, a block of bytecode for a reflectively declared method may be expanded to bytecode for multiple methods, by replacing type and name variables in the original with concrete types and identifiers. Alternatively, a block of bytecode may be removed entirely if the range of its associated reflective iterator is empty. Type-checking of the expanded bytecode is not necessary, since MorphJ's source-level type-checking guarantees that any type-instantiation of a generic is always well-typed. The bytecode of a MorphJ generic class, assuming it has reflective declarations, is not

considered valid by the Java bytecode verifier. Its expanded versions, however, are.

MorphJ is implemented using the JastAdd extensible compiler framework for Java [31], and ASM Java bytecode library [19]. MorphJ is available via *http://code.google.com/p/morphing/*.

Chapter 4

Related Work

The limitations stemming from rigidity in code structure have been the focus of much previous research, starting as early as the design of Lisp macros in the 1960's. The structural abstraction techniques presented in this dissertation, static type conditions and morphing, can be seen as the latest steps in this line of research. We begin this chapter by discussing how static type conditions and morphing relate to each other (Section 4.1). We then compare these two techniques to traditional, low-level meta-programming mechanisms in Section 4.2. We then draw comparisons between structural abstraction and the more recent efforts in providing higher-level language support (i.e., with either more emphasis on safety, or on higher level language constructs) for meta-programming. Specifically, Section 4.3 compares structural abstraction to safe program generation and transformation efforts, and Section 4.4 compares structural abstraction to Aspect-Oriented Programming (AOP). Section 4.5 offers a discussion on language techniques that offer support for only conditional declarations, and Section 4.6 discussions those that only support iterative declarations. There is a striking similarity between the kinds of problems that can be addressed with AOP, and those addressable by morphing. This is evident in our use of logging and synchronization as use cases for morphing, which are also examples commonly used in AOP literature. To further study the differences in the two techniques, Section 4.7 concludes the chapter with a detailed case study, in which we attempt to re-engineer two applications with both morphing and AOP, and delineate the pros and cons of either approach.

4.1 Static Type Conditions vs. Morphing

cJ and MorphJ both add to Java a reflective "if", though the mechanisms for expressing these conditions and the kind of conditions expressible differ. Additionally, MorphJ adds a reflective "for", as well as the ability to create declarations with non-constant names. Thus, MorphJ is a more ambitious language with significantly more complexity, which is reflected in the different design

focus and implementation decisions for cJ and MorphJ.

The cJ reflective "if" is based on subtyping conditions in a nominal subtyping system, whereas the MorphJ reflective "if" is based on structural constraints, e.g., whether a type has a method or field matching a certain pattern. Though this difference in mechanism seems inconsequential, the MorphJ-style "if" constraints can be easily integrated with a structural subtyping system. But more importantly, the cJ "if" conditions are restricted to straightforward subtyping and conjunctions of multiple subtyping conditions. MorphJ, however, allows the expression of negative conditions, using "`if ( no` *`PATTERN`* `)`" clauses. MorphJ uses negative conditions extensively to ensure the absence of conflicting declarations. Supporting negative conditions in cJ would have had limited effectiveness due to the open nature of Java's type hierarchies. Recall that the main reasoning in cJ's type system is whether one condition can be implied by another. A negative condition bounded by an interface type can never be safely implied by any condition other than itself. Negative conditions bounded by class types are amenable to more reasoning. Most of our use cases for static type conditions, however, utilize interface types in type conditions. Thus, we did not see an imminent need to support negative conditions in cJ.

cJ is designed with backward compatibility in mind, enabling an erasure-based translation. cJ language constructs can be "erased", producing regular Java code in a one-to-one correspondence between cJ generic classes and Java generic classes. Additionally, cJ interacts smoothly with advanced features in the Java type system, such as variance [48, 90] and polymorphic methods. In contrast, MorphJ takes a more radical approach, favoring feature-richness and integration of ideas over backward compatibility and implementation integration. This difference is most evident in MorphJ's implementation, which employs an expansion-based translation. MorphJ generic classes produce one regular non-generic Java class per type-instantiation. This implementation approach is harder to support in conjunction with some of Java's features (e.g., dynamic loading) but yields more power—e.g., the ability to express mixins as generic subclasses. Furthermore, we have not concerned ourselves with supporting features such as variance and polymorphic methods. Studying the interaction of these features with MorphJ is left for future work.

Overall, we do not view MorphJ as a language extension that can be easily integrated in standard Java. (After all, integrating with standard Java seems a near-hopeless proposition even for

more modest research proposals, as the Java language has matured and the rate of change has decreased dramatically.) Instead, we view MorphJ as a more radical idea, intended to demonstrate the principles of morphing and to influence future language designers. Our goal with MorphJ is to show the first morphing language with a sound modular type checking system, and a smooth integration of concepts in an Object-Oriented framework.

4.2 Comparison to Traditional Meta-programming Techniques

Meta-programming techniques offer the ability for programs to generate other programs, allowing the structure of generated programs to change based an arbitrary conditions. Instances of such techniques are macro facilities such as Lisp macros, reflection, meta-object protocols [27, 58], or pattern-based program generation and transformation [11–13, 91]. The goal of structural abstraction is to promote meta-programming capabilities offered in these low-level mechanisms to high-level language features, with support for full modular type-safety. None of the above mechanisms offer such safety guarantees: A macro or meta-class cannot be type-checked independently from their compositions, in a way that guarantees it is well-typed for all its possible compositions.

An interesting instance of meta-programming is the C++ template mechanism [7, 49]. C++ templates provide a powerful (Turing-complete) way to statically configure code. Conditional declaration can be easily accomplished through template specialization. Even though there is no inherent support for reflection in C++, it has been shown that reflection can be emulated using template meta-programming [10], even though it is not applicable to unsuspecting classes. Though powerful, C++ templates are unsafe: There is little static checking capability beyond the checking of templates after instantiation. Furthermore, there is no way to guarantee that a template computation will even terminate. The C++ community has developed ideas on statically validating the input to a template [69, 81] and the general idea of *concepts* has emerged and even developed as a language-independent notion [51]. Nevertheless, concept-based techniques concentrate on validating the type parameters of a generic class, rather than configuring it under static conditions. cJ and MorphJ offer the ability to configure classes based on structural information, without sacrificing static type safety and with a smooth integration in the base language.

An interesting special case of program generation is *staging languages* such as MetaML [87]

and MetaOCaml [21]. These languages offer modular type safety: The generated code is guaranteed correct for any input, if the generator type-checks. Nevertheless, MetaML and MetaOCaml do not allow generating identifiers (e.g., names of variables) or types that are not constant (as is supported in MorphJ). Generally, staging languages target program specialization rather than full program generation: The program must remain valid even when staging annotations are removed. Thus, staging programs do not abstract over the structure of programs—they are quite explicit in the structure of the generated program, in fact. It is interesting that even recent meta-programming tools, such as Template Haskell [80] are explicitly not modularly type safe—its authors acknowledge that they sacrifice the MetaML guarantees for expressiveness.

4.3 Comparison to Efforts in Safe Program Generation/Transformation

Recent mechanisms such as Genoupe [29], SafeGen [44], and compile-time reflection (CTR) [34] attempt to add safety guarantees to meta-programming, while maintaining expressiveness. Nevertheless, these approaches either fail to achieve full safety, or reject programs in a way that is not transparent to the programmer. For instance, the Genoupe approach has been shown unsafe, as the reasoning depends on properties that can change at runtime; SafeGen has no soundness proof and relies on the capabilities of an automatic theorem prover—an unpredictable and unfriendly process from the programmer's perspective.

MorphJ's closest relative is CTR [34]. CTR is an extension to C# that pioneered the use of patterns for reflective iteration and was one of the first systems to aim for modular type safety. Nevertheless, its modular guarantees concern only validity of references and not the absence of declaration conflicts. A unique aspect of CTR (compared to MorphJ) is that it transforms classes in-place, which enables some interesting applications. MorphJ, on the other hand, only allows enhancement of classes through subtyping or delegation. MorphJ improves over CTR, however, by adding more expressiveness through nested patterns, while keeping or strengthening the typing guarantees. For instance, CTR does not allow matching multiple method argument types, or existential conditions on iteration ranges.

In addition to lacking full type safety, neither of these mechanisms integrate seamlessly with a programming language, as cJ and MorphJ do. All these approaches require an concept outside

the base language, such as a "generator" or a "transform". In cJ and MorphJ, the concept of code generation is incorporated with the concept of generic classes. Additionally, these mechanisms use complex syntax for retrieving reflective members, whereas MorphJ utilizes patterns very similar to method and field signatures.

4.4 Comparison to AOP Tools

4.4.1 Static Type Conditions

cJ can be viewed as an instance of the aspect-oriented programming paradigm [61], because of its ability to allow a class to be configured based on the structure of a different type hierarchy (representing a cross-cutting concern). As we demonstrated with the cJ implementation of the JCF, the cross-cutting concern "modifiability" is separated in the type system from the intrinsic form of a data structure (e.g., whether it is a list, a set, or a map). The type system does maintain concepts such as "modifiable list", "unmodifiable map", etc., but these are derived from their component types. In fact, the cJ reimplementation of the JCF can be compared to previous work that uses AOP to enforce consistency in data structure and behavior [62, 76]—in the cJ reimplementation of JCF, consistency in the usage of data structures along cross-cutting dimensions is enforced by the cJ type system.

Nevertheless, cJ differs from common aspect-oriented languages like AspectJ [59] in that separation of concerns in cJ is confined to static conditions, expressed using types—AspectJ allows cross-cutting concerns to be specified using dynamic conditions such as control flows. Another difference is that code for a single concern in cJ is interspersed throughout traditional Java language components (i.e., classes) and not collected in a single entity. For instance, methods addressing the concern "modifiable" are interspersed in all classes and interfaces cross-cut by the concern, e.g., `ArrayList`, `Collection`, etc. In AspectJ, code of a certain concern is generally grouped together into an "aspect". It is worth mentioning that previous study has shown that often times, it is most convenient to intersperse concern code, rather than grouping them into AspectJ-style aspects [55]. A rather superficial difference between cJ and AOP tools is that the current cJ implementation does not allow type conditions to be used at the statement or expression level. This is an implementation detail, due to the fact that we wanted to maintain the erasability of cJ programs into regular

Java programs, without employing runtime reflection.

4.4.2 Morphing

Morphing addresses a small but central part of AOP functionality: aspect advice of structural program features, such as method before-, after-, and around-advice. Particularly, the logging and synchronization examples shown in Chapter 3 are frequent use cases for AOP languages. Thus, it is worth delineating the similarities and distinct differences between morphing and AOP. We next compare MorphJ to AspectJ [60], a representative AOP tool for Java. A more elaborate case study drawing detailed differences between morphing and AOP is presented in Section 4.7.

4.4.2.1 How Functionality is Added

Both MorphJ and AspectJ allow functionality that cross-cut multiple class definitions to be defined in a modular way. For example, the method logging functionality can be defined in one MorphJ class, `Logged` (Section 3.1). However, the way such functionalities are added into a base class definition is one of the main differences between MorphJ and AspectJ. With MorphJ, cross-cutting functionality is added "into" a base class through explicit parameterization of the morphing class. The new functionality only exists in the parameterized morphing class, whereas the definition of the base class itself does not change. For example, the parameterized morphing class `Logged<java.lange.Object>` has the functionality that all non-`void`-returning methods are logged. However, the definition of `java.lang.Object` itself remains unchanged. In AspectJ, an aspect definition states the classes a functionality should be added to.[1] In this way, the new functionality is weaved with the code of the original class. The program cannot simultaneously use the separate notions of "original class" and "class with the cross-cutting functionality". One way to view the semantics of AspectJ is as changing the original class's definition. For instance, given an AspectJ aspect that adds logging code to each non-`void`-returning method of `java.lang.Object`, the class `java.lang.Object` itself can be thought of as changed after aspect application.[2] Indeed this also happens to be the way current AspectJ compilers implement the semantics of weaving.

[1] In the case of generic aspects in AspectJ 5, the affected classes can be specified through parameterization of the aspect.

[2] Changing the semantics of `java.lang.Object` is potentially dangerous, since it is the root of all Java classes. For this reason, AspectJ does not allow advice to be declared for methods and fields of `java.lang.Object`, or for any type in the core Java library, i.e., those in package `java.lang`.

We view explicit parameterization in MorphJ as an important feature for two reasons. First, the ability to leave the original class definition untouched is an important one. For example, a programmer should be able to use both synchronized and unsynchronized versions of a data structure in the same program, depending on his/her needs. This is indeed the case with the MorphJ class `Synchronized`, shown in Figure 9. With AspectJ, however, the programmer must choose one or the other. AOP purists may hold the view that cross-cutting functionality enhancements, by definition, should be applied to all classes that need them. But as shown through the synchronization example, this is a very rigid requirement. Furthermore, if indeed all instantiations of a class should have a particular cross-cutting functionality, it should be possible to extend MorphJ with a global search-and-replace tool, replacing all instances of a class with the explicitly parameterized version of a morphing class. This is part of future work, however. Our current research focuses on the fundamental core of morphing, and we expect that usability enhancements will come later.

Secondly, explicit parameterization provides a way to clearly document and control the semantics of a program. This is particularly true when multiple, separately-defined functionality enhancements need to be added to a class. One of the much researched topics in AOP is aspect interaction. When one defines an aspect in AspectJ, there is no good way to specify the order of its application relative to all other aspects, some of which may be unknown to the aspect developer. Furthermore, an addition of another aspect unknown to the developer can change the program semantics in unexpected ways. This is an undesirable characteristic in terms of modularity. MorphJ, on the other hand, allows explicit control of functionality addition through instantiation order. The type-instantiation order gives a clear meaning as to where and how functionality is added.

4.4.2.2 Modular Type Safety and Trade-offs in Expressiveness

The other main difference between MorphJ and AspectJ is MorphJ's guarantee of modular type safety. In order to make such a guarantee, we limited our attention to some specific features instead of adding maximum expressiveness to the language. Pattern matching in MorphJ is simple and high level by design. A programmer can only inspect classes at the level of method and field signatures: MorphJ pattern matching applies to reflection-level structural elements of a type. In contrast, AspectJ allows a programmer to match on a program's dynamic execution characteristics,

using keywords such as `cflow` (for control flow) and `cflowbelow` in pointcuts.

Though MorphJ limits its pattern matching to the type signature level, it does allow matching using subtype-based semantic conditions, in contrast to the purely syntactic matching of signatures AspectJ offers. For instance, using pattern-matching type variables, MorphJ allows one to express a pattern that matches all methods that return *some* subtype of `java.lang.Comparable`. This is a pattern not expressible through AspectJ. The combination of pattern matching and the static `for` construct in MorphJ provides a controlled but useful kind of programmability in defining where a certain functionality should be introduced.

Although we make comparisions only to AspectJ, the arguments in this section generalize to other AOP tools, as well. AspectJ is representative in the way it applies aspects, and is perhaps the most expressive aspect language today.

4.5 Work Related Specifically to Static Type Conditions

Clearly the idea of a type-conditional is closely related to conditional compilation, as with the C/C++ preprocessor "`#ifdef`" construct. Although `#ifdef` is valuable for configuring large projects, it addresses very different needs from cJ. Conditional compilation gives low-level manual control for software configuration. In the context of a portable language, like Java, an `#ifdef` statement becomes less useful. At the same time, conditional compilation suffers from the lack of any form of safety control. The use of conditional flags may be inconsistent, resulting in invalid configurations that are not detected until one attempts to select them. There has been work on adding some safety to conditional compilation by analyzing all configurations of a C program, and there is evidence that such a heuristic approach may work in several contexts—especially for refactoring [36]. Nevertheless, cJ offers full static safety guarantees, eliminating the problem altogether. Furthermore, the type-conditions of cJ are structured, richer than mere propositions, and well-integrated with the Java type system.

Conditional methods have been explored in OO languages in work at least as early as CLU [66]. Nevertheless, CLU does not support subtyping, so the language context of this work is notably different. It is, thus, difficult to compare CLU to cJ, where our main goal is to maintain static type safety, yet, at the same time, maintain a clean subtyping hierarchy used for abstraction. Past work

on optional methods in Java was also presented by Myers et al. [74]. This was in the context of a proposal for adding genericity to Java, and it includes the feature of attaching `where` clauses to individual methods. The conditions on the `where` clauses, however, are "structural" constraints—i.e., does type parameter `T` provide method `void foo();`. This mechanism does not support conditional subtyping—e.g., it is not possible to express that a `Collection` is `Comparable`, if the elements it holds are `Comparable`. Even more importantly, the work by Myers et al. does not support type-safe abstraction over classes with conditional methods, as in the interaction of cJ with variance.

More recently, Emir et al. presented an extension to C# to support generalized type constraints on methods [32]. This extension allows both upper and lower bound type conditions on methods. This is similar to cJ in that methods exist conditionally based on the instantiation of parametric types. Nevertheless, there are significant differences, and in future work we plan to pursue combining the two approaches. cJ currently does not allow using the type parameter of a polymorphic method inside a type conditional—a crucial feature in Emir et al.'s work. At the same time, cJ has several features not found in the generalized type constraints approach. First, cJ supports conditional definitions of fields, as well as conditional subtyping. Furthermore, the cJ (and Java) form of variance we examined earlier is a "use-site variance" mechanism as opposed to the "definition-site variance" supported in Emir et al.'s work. In addition to being part of standard Java, we believe that use-site variance is a mechanism better suited for imperative programming languages in general. In this setting, a single class definition is unlikely to produce types that are purely co-variant, purely contra-variant, or purely bi-variant. Instead defining a class will likely implicitly yield a co-variant part, a contra-variant part, etc. In use-site variance, these subsets of the class functionality are derived automatically from a single definition. In definition-site variance, they have to be explicitly separated out into distinct interfaces by the programmer. Thus, we believe use-site variance to be a more user-friendly system and a natural fit for Java.

It is not surprising that conditional declarations have been a focus of software product lines (SPL) research. Recently, Kästner et al. presented a SPL tool CIDE, which allows the conditional declaration of code that is guaranteed to be type-safe [54]. Conditions in CIDE are not expressed in terms of subtyping, and CIDE allows disjunction (e.g., *`COND1 or COND2`*), as well as negation.

cJ does not support either: disjunctive subtyping conditions present problems when we need to determine a most general type to downcast a variable to (i.e., when the variable is typed by the type variable used in the conditional); negation is difficult to support in a language with the open-world assumption, as we explained in Section 4.1. Even though CIDE allows more expressive conditions, it does not allow multiple configurations to exist in the same application. A piece of code can only be configured once by a client, and used in that manner. For instance, in CIDE, a client would not have the ability to use `Collection<Modifiable>` and `Collection<VariableSize>` in the same program. Thus, this mechanism is used mostly for configuration management.

cJ's support for conditionally declared supertypes can be emulated to a certain extent using "generalized interfaces" [92] for Java. JavaGI, the Java extension proposed by Wehr et al., allows a method declaration to be provided for a generic class, only if the type parameters of the generic class obey certain subtyping conditions. However, JavaGI does *not* allow such conditional methods to be provided for generic interfaces. For instance, a *compareTo* method can be declared for `ArrayList<X>` where `X extends Comparable<X>`. However, a *compareTo* method cannot be conditionally declared for the interface `List<X>`. In cJ, conditional declarations are supported for both classes and interfaces. Of course, the main goal of JavaGI is not to support conditional declarations, but to incorporate core features of Haskell type-classes [53] into Java. As a result, JavaGI supports many useful features that are not supported by cJ, e.g., retroactive interface implementation (where a class can be declared to implement an interface separately from its declaration, and provide implementations for interface methods separately, as well), binary methods, bounded existential types, etc. Along the same line, one can achieve similar effect to conditional declarations in Haskell using type-classes.

In languages with (multi-)methods outside classes, the work on constraint-based polymorphism in Cecil [68] is related to cJ. Cecil provides users the ability to add constraints to both methods and supertypes. The constraints can be subtyping constraints, as well as structural constraints, requiring a type to provide a particular method. This is a very different context from that of our work, however. Furthermore, the Cecil type system does not have an analogue of our variance approach to abstracting over all objects with or without some of the conditionally defined members.

It is tempting to find parallels between cJ and advanced OO modularization mechanisms such

as traits [30, 79], mixins [15], or mixin layers [82]. These approaches vary in expressiveness and several of them are insufficient for solving the combinatorial explosion problems identified in Section 2.3. For instance, C++-based mixins or mixin layers would still require a large number of compositions, with explicit subtyping links added among them, in order to express the required functionality of the Java Collections Framework. It is possible that a traits-based mechanism could serve to alleviate many of the problems in the Java Collections Framework (albeit with a complete rewrite). Nevertheless, there is no such mechanism currently for Java that would tie well with the rest of the language's type system (e.g., variance) and execution model. Furthermore, no mixin or traits mechanism offers capabilities similar to those shown in Section 2.3.1, i.e., the ability to add extra members only when a type parameter that is already used for other purposes has a certain subtyping property.

Type-conditionals in cJ can be viewed as being similar to type-safe variant records work—e.g., [77]. Nevertheless, variant records mechanisms typically try to address the problem of *run-time* variability with static type-safety. cJ is not concerned with changes to the type of a variable during run-time. Instead, cJ focuses on the static configurability of components. The techniques used to ensure static type safety in the case of variant records and in the case of cJ show this difference clearly: Statically safe variant records typically require the programmer to specify what code will get executed for any possible type. Indeed, this is the best one can hope when the object can indeed vary at run-time. In contrast, cJ statically ensures that the legal operations on an object are fully known.

4.6 Work Related Specifically to Morphing

An extension of traits [78] offers pattern-based reflection by allowing a trait to use name variables for declarations. However, [78] does not offer static iteration over the members of classes—a name-generic trait must be mixed in once for each name instance.

There has been a line of work focused on providing statically type-safe generic traversal of data structures [50, 63]. For instance, the "scrap your boilerplate" [63] line of work offers extensions of Haskell that allow code to abstract over the exact structure of the data types it acts on, and to have the appropriate functions invoked when their expected data types are encountered during

traversals. Abstracting over the structures of data types in functional languages is similar to abstracting over the fields and methods of classes in object-oriented languages. [93] offers such generic traversal capabilities for Java. However, whereas [50, 63, 93] focus on offering structurally-generic *traversal*, MorphJ focuses on structurally-generic *declarations*. Neither of [50, 63, 93] allow more functions to be declared using the names or types retrieved from a non-specific data type. Thus, these techniques fall short of MorphJ (and static reflection work in general [29, 34, 44, 46]) in this respect. On the other hand, MorphJ is not well-suited for writing generic traversal code. Traversing data structures and invoking methods on objects encountered is largely based on the dynamic types of these objects. MorphJ's reflective declarations are based purely on the static types of fields and methods.

4.7 A Comparative Study of AOP and Morphing

Aspect-Oriented Programming and morphing can be seen as alternative and complementary approaches to addressing cross-cutting concerns. AOP is execution-based: points in program execution are identified, cross-cutting code is woven around these points; Morphing is declaration-based: structural patterns of a program are identified, code is declared to mimic these structural patterns, while defining additional functionality. In this section, we compare these approaches through the re-engineering of two applications: Java Collections Framework, the standard data structures library for Java, and DSTM2, a software transactional memory framework. We capture cross-cutting concerns in these applications using AspectJ, the flagship tool for AOP, and MorphJ, a Java extension supporting morphing. We evaluate the suitability of AspectJ and MorphJ in terms of the reusability and modularity of the resulting software. We identify characteristics of concerns that are structural in nature, thus favoring a declaration-based approach for expressiveness and safety.

4.7.1 Introduction and Background

Cross-cutting concerns are aspects of a program that cross-cut the program's dominant organizational structure. Aspect-Oriented Programming (AOP) [61] has emerged as the most visible approach in modularly addressing cross-cutting concerns. Section 3 presented an alternative approach, morphing, which allows the declaring of code that mimics the *structure* of other code, but with modified behavior. The result is similar to weaving new functionality across the structure

of existing code, through new declarations. In this section, we present case studies on the re-engineering of two real-world Java applications, using both AOP and code morphing. Our studies show that whereas AOP is well-suited for concerns with definite patterns in program *execution*, code morphing is well-suited for concerns with patterns in a program's static *structure*.

As a simple exposition, we use the prototypical cross-cutting concern, "logging", to demonstrate the principle similarities between AOP and morphing.

Logging with Plain OO: We illustrate the plain Object-Oriented (OO) solution using Java. One way to add logging to a Java class is by refining it through subclassing:

```
1  class LoggedFoo extends Foo {
2    public Bar meth1 ( String s ) {
3      Bar r = super.meth1(s);
4      Logger.log("meth1 returning " + r.toString());
5      return r;
6    }
7    public int meth2 () {
8      int i = super.meth2();
9      Logger.log("meth2 returning " + i);
10     return i;
11   }
12   ... // similarly log other methods of Foo.
13 }
```

`LoggedFoo` refines its superclass `Foo` by adding logging code to each method. The logging code is replicated over multiple methods. With some imagination, one can also envisage the code being replicated over multiple classes (e.g., `LoggedBar`, `LoggedBaz`, etc.). This is to say that logging code cross-cuts the dominant organizational structure of Java programs—their class/interface hierarchies.

Ideally, we want to define logging once, in a reusable way, and apply it to whichever method or class requires it.

Logging with AOP: We illustrate the AOP solution to this problem using AspectJ [60], AOP's flagship tool. We can encapsulate all logging code in the following generic aspect (a similar example was also shown in Section 1.2.2):

```
1  abstract aspect LoggingAspect<X> {
2    pointcut loggedMeth() : execution(* X.*(..));
3
4    after() returning(Object r) : loggedMeth() {
5      Logger.log(
6            thisJoinPoint.getSignature().getName() +
7            " returning " + r.toString());
8    }
9  }
```

For any type `X`, pointcut `loggedMeth()` identifies the execution of its methods. The advice on lines 4-8 specifies the logging code to be run after the execution of these methods. To apply "logging" to `Foo`, one would use the following declaration:

```
aspect LoggingFoo extends LoggingAspect<Foo> { }
```

Logging with Morphing: We have shown in both Section 1.2.2 and 3.1 a generic MorphJ "logging" class that logged the return values of all non-`void`-returning methods. Here, we show a fully fleshed out version that records the execution of both `void`- and non-`void`-returning methods:

```
1  class LoggedClass<class X> extends X {
2    <R,A*>[m] for ( R m (A) : X.methods )
3    R m (A args) {
4      R r = super.m(args);
5      Logger.log(m.name + " returning " + r);
6      return retval;
7    }
8
9    <A*>[m] for ( void m (A) : X.methods )
10   void m (A args) {
11     super.m(args);
12     Logger.log(m.name + " returning void");
13   }
14 }
```

`LoggedClass` extends its own type parameter `X`. The reflective iteration block on lines 2-7 defines a static iteration over all methods of `X` that take any number of arguments, and have non-`void` returns. For each such method, it declares a method with the same signature, and inserts proper logging code. Lines 9-13 constitute a similar block for the `void`-returning methods of `X`.

Instantiating `LoggedClass` with any type has a similar effect to "weaving" logging code into that type. For example, `LoggedClass<Foo>` declares the exact same methods as `Foo`, with every

method's return value logged.

Execution vs. Declaration The AOP approach is a deliberately dynamic, *execution*-based one. Pointcuts identify "well-defined points in the *execution* of a program" [60]: e.g., the execution of a method, the read or write of a field, etc. Advice code can be applied before, after, or around these points in execution. Even though we use AspectJ as the representative AOP language, virtually all AOP implementations (e.g., [20, 42, 70]) are built on the philosophy of identifying and advising points in program execution.

In contrast, morphing offers a static, *declaration*-based approach. Programmers use patterns to capture static, *structural* elements of a program (e.g., methods of a class). New code is then declared using information captured by the patterns (e.g., a method's name, argument and return types), while expressing additional functionality. Morphing is the latest in a long line of research in component composition and reuse. It is related to feature-oriented programming [14, 56], mixins and mixin layers [15, 83], traits [78, 79], Hyper/J [88], object synthesis [26], etc. However, morphing is the first of its kind to offer high level, *separately type-checkable* language support for reflecting over the structure of code, and *declaring* new code using information obtained from reflection.

A cursory look at the logging example might suggest that AOP and code morphing are moral equivalents. However, differences emerge for more substantial applications.

In this section, we study two real-world Java applications that deal with cross-cutting concerns in different, but representative ways. Java Collections Framework (JCF) [5] is the standard data structures library for Java. It handles cross-cutting concerns such as the "synchronization" of data structure access by creating proxy classes, and replicating synchronization code for each method of a proxy, across all proxies. DSTM2 [41] is a software transactional memory framework. It handles cross-cutting concerns such as "transaction management" using a much lower level mechanism: code generation through the Java reflection API [37] and bytecode engineering library (BCEL) [9].

We attempt to capture the cross-cutting concerns in these applications using both AspectJ and MorphJ. The results show that while AspectJ has extremely powerful constructs for identifying patterns in program execution and advising points captured by these patterns, it is not well-suited for concerns that identify strongly with a program's static structure. We identify these concerns

as *structurally cross-cutting concerns*. MorphJ, on the other hand, has more expressive constructs for capturing the *structure* of programs, and allowing code declarations to mimic the captured structure. It is thus very well-suited for addressing structural concerns.

4.7.2 Case Study I: Java Collections Framework

Java Collections Framework (JCF) [5] is the standard data structures library for Java. In the version of JCF included with Java 5.0, there are 14 main interfaces (e.g. `Collection`, `List`, etc.), and 46 classes implementing these interfaces. To provide these data structures with features that are orthogonal to the representation of data itself, JCF uses proxy classes. For example, to offer the "synchronization" feature, JCF defines a number of synchronization proxies: `SynchronizedCollection`, `SynchronizedList`, etc. To obtain a synchronized version of a data structure, programmers call static methods in the utility class `java.util.Collections`. For example, `Collections.synchronizedCollection(c)` returns a synchronized version of a `Collection` object `c`.

```
class SynchronizedCollection
                  implements Collection {
  Collection c;
  Object mutex;
  ... // constructors that initialize
      // c and mutex

  public int size() {
    synchronized(mutex) {
      return c.size();
    }
  }
  public boolean remove(Object o) {
    synchronized(mutex) {
      return c.remove(o);
    }
  }
  ... // repeat for all methods in
      // Collection.
}
```

(a) Synchronization proxy for `Collection`.

```
class SynchronizedList
            implements List {
  List l;
  Object mutex;
  ... // constructors that initialize
      // l and mutex

  public int size() {
    synchronized(mutex) {
      return l.size();
    }
  }
  public int indexOf(Object o) {
    synchronized(mutex) {
      return l.indexOf(o);
    }
  }
  ... // repeat for all methods in
      // List.
}
```

(b) Synchronization proxy for `List`.

Figure 25: Comparison of the JCF synchronization proxy classes.

Anatomy of the Proxies: We take a closer look at two reperesentative JCF proxies, first shown in Figure 8, but shown here again in Figure 25 for ease of reference. Closer inspection shows that these proxies are the result of synchronization code being specialized at both the method and class levels.

At the method level, the code for synchronization is clearly the block `synchronized(mutex) {...}`. However, this block is specialized for the structure of each method of the type being proxied—e.g., the method's name and arguments. For example, in `SynchronizedCollection`, the block in `size()` and that in `remove(Object o)` differ only in the method name and arguments used for their delegation calls.

At the class level, the code for each proxy is specialized for the *structure* of the type being proxied—e.g., that type's name and methods. The only difference between `SynchronizedCollection` and `SynchronizedList` are due to the *structural* differences between `Collection` and `List`. Each proxy declares a field holding the object that method calls are delegated to. Such fields for `SynchronizedCollection` and `SynchronizedList` differ only in their types (a structural element of the field): `Collection` vs. `List`. Furthermore, the two proxies declare the same methods modulo the differences in the declared methods (again, structural elements) of `Collection` vs. the declared methods of `List`.

Thus, the structural specialization of the synchronization code is the root cause for the replication of synchronization code across methods and classes. Each time synchronization is needed for a different type, with a different structure, the synchronization code must be specialized to the structure of that type, through the declaration of a new proxy class.

Advantages and Disadvantages: The one obvious advantage of the proxy solution is that programmers have the flexibility to use either synchronized or unsynchronized versions of the same data structure as they see fit.

Yet another advantage that is easily taken for granted is that a pure Java solution stays within the bounds of the Java type system, and enjoys the separate type-checking guarantees provided by Java—a synchronization proxy can be type-checked *separately* from the code that may use it, and guaranteed that it is well-typed wherever it is used.

These advantages, however, come at a heavy cost. The biggest of these is the tight coupling in structure between each synchronization proxy and the type it proxies for. For instance, the structure of `SynchronizedList` is tightly coupled with that of `List`. If the structure of `List` changes, e.g., a method is added, or a method signature changes, the structure of `SynchronizedList` must also change.

Furthermore, programmers are limited to the 7 synchronization proxies JCF provides. To obtained a synchronized version of a type with no JCF-provided proxy, a programmer has to either choose a more general supertype *with* a JCF-provided proxy, or implement his/her own.

Next, we explore how we can approach this problem using AOP and code morphing, mitigating the disadvantages of the current solution, while maintaining the advantages.

4.7.2.1 AOP with AspectJ

At first glance, it may seem that this task is a good fit for AspectJ. However, the devil is in the details.

Attempt #1: To advise the methods of a type (and its subtypes) with synchronization code, we have two choices: We can either advise the points when methods are *executed*, (i.e., advice code is injected to where methods are defined), or advise the points when methods are *called* (i.e., advice code is injected to where methods are invoked). However, AspectJ disallows the execution-site advisement of methods declared by classes in the `java.*` packages. This, unfortunately, includes the JCF classes. Thus, we explore the call-site advisement option with the following generic aspect:

```
1  abstract aspect SynchCall<X> pertarget(callMeths()) {
2    pointcut callMeths() : call(* X+.*(..));
3    Object mutex;
4    Object around() : callMeths() {
5      synchronized(mutex) { return proceed(); }
6    }
7  }
```

Given any type `X`, `SynchCall<X>` declares a pointcut `callMeths`, which captures the call to any method of `X` and its subtypes. An `around` advice (lines 4-6) injects a synchronization construct around calls captured by `callMeths`. AspectJ's `proceed()` keyword allows the advice to turn control back to the intercepted code. Additionally, the `pertarget` construct in the declaration creates one

instance of the aspect—and thus one instance of `mutex`—per object of type `X` whose method invocations are intercepted.

To apply the synchronization advice to `Collection`, we declare the following concrete aspect:

```
aspect SynchCollectionAspect extends SynchCall<Collection> { }
```

Evaluation: Compared to the plain Java solution, the `SynchCall` aspect provides a very modular definition of the synchronization concern. By instantiating `SynchCall` with different concrete types, we can apply the synchronization functionality to any type—whether it be data structures in the core JCF, or those defined by programmers themselves.

However, this solution offers programmers less control over when the synchronization code is applied, as well as less language-level support for type safety.

Once the `SynchCall` aspect has been applied to a data structure, all calls to that data structure's methods are advised. The unsynchronized version of that data structure is no longer available. In order to have more fine-grained advisement, programmers need to be able to enhance the pointcut `callMeths` with additional program execution patterns. Capturing arbitrarily precise points in program execution with pointcuts is a difficult issue recognized by many researchers (e.g., [55, 73]). One technique often employed is hook methods: empty methods that serve no purpose other than to punctuate a point in program execution that could be further advised by aspects. However, such technique is widely considered unnatural for programmers [73].

The `SynchCall` aspect also suffers from AspectJ's lack of support for *separate type-checking*. An aspect cannot be type-checked separately from the programs it may advise, to guarantee that its advisement will not introduce type errors. As an example, `SynchCall` is compiled by the AspectJ compiler without issues. However, there is actually an insidious type error in this aspect: The `around` advice naively declares its return type to be `Object`, even though the pointcut `callMeths` identifies the call to all methods of `X`, including the `void`-returning ones! The body of the advice then returns the result of the method by calling `return proceed();`.

This is an error by the aspect implementer, and should be caught by a type-checker before the aspect is distributed to users. However, in AspectJ, the error is only discovered if the call of a

void-returning method is matched by the pointcut, and advised. Even then, the AspectJ compiler does not complain about the inappropriateness of returning a value from a void-method. Instead, a NullPointerException ensues. A type error has been turned into a runtime exception!

It is important to note that had the JCF classes *not* been part of the java.* packages, an execution-site advisement solution would look very much like the call-site solution we just presented, and poses the same advantages and problems.

Attempt #2: Since the lack of separate type-checking is a fundamental limitation of AspectJ, we attempt to make an AspectJ solution that at least affords programmers more control over the application of the synchronization code. Given the difficulty in refining pointcuts, we attempt a more explicit approach.

For each data structure, we declare a *new*, empty subclass. We then use an aspect to advise this new class instead. Programmers may use this new class when synchronization is wanted. For example, for a synchronized version of Collection, we declare interface SynchCollection that extends Collection. We can then use the following pointcut to identify calls to methods of SynchCollection:

```
pointcut callSynchCollection(SynchCollection c) :
   target(c) && (call(* Collection+.*(..)));
```

The above pointcut matches calls to all methods defined by Collection and its subtypes. Additionally, the target(c) construct forces the receiver of these calls to be an object of type SynchCollection. A method call invoked from an object of (the unsynchronized type) Collection will thus not be matched by this pointcut. We can now advise callSynchCollection with synchronization code (similar to the code in our previous attempt), without affecting the objects of Collection. This refinement allows a programmer to use SynchCollection wherever a synchronized version of Collection is required. This more explicit approach is very much in the spirit of the original JCF, where a programmer explicitly obtains a synchronized version of Collection c by invoking Collections.synchronizedCollection(c).

Evaluation: Though this second attempt provides us with more precise control over the application of concern code, it loses a bit of modularity. For every data structure in JCF, we need to declare a synchronized subtype for advisement. What is more tedious than the declarations of types, though, is that these new types must be related through a type hierarchy that mimics the type hierarchy of their corresponding originals. For instance, since interface `List` extends `Collection`, we must declare `SynchList` to extend `SynchCollection`. And for every class implementing `Collection`, we must declare their corresponding synchronized subclasses to implement `SynchCollection`. This isomorphic type hierarchy is another form of structural specialization. If the structure of the JCF type hierarchy changes, this synchronization hierarchy must change, as well.

Observant readers may notice that the original type hierarchy cannot be faithfully replicated for *classes*. For example, class `AttributeList` extends `ArrayList`. Thus, `SynchAttributeList` must extend `SynchArrayList`. However, `SynchAttributeList` already extends `AttributeList`! Without support for multiple inheritance, we simply cannot accomplish this task.

Another issue affecting the usability of this solution is that the dynamic types of objects are not taken into account in pointcut matching. When a `SynchArrayList` object is assigned to a variable of type `ArrayList`, method calls invoked from this variable are no longer matched by `callSynchCollection`! It is possible, however, that AspectJ could be enhanced to eliminate this issue.

4.7.2.2 Morphing with MorphJ

The MorphJ solution is a MorphJ generic class that, when instantiated with a concrete type, results in a class that is equivalent to the manually declared proxy classes. The code below was first introduced in Chapter 3, Figure 9. We replicate it here for ease of reference:

```
1  public class Synchronized<interface X> implements X {
2    X      me;
3    Object mutex;
4
5   // ... constructor declarations
6
7    <R,A*>[m] for (public R m (A) : X.methods)
8    R m (A args) {
9      synchronized (mutex) {
10       return me.m(args);
11     }
12   }
13
14   <A*>[m] for (public void m (A) : X.methods)
15   void m (A args) {
16     synchronized (mutex) {
17      me.m(args);
18     }
19   }
20 }
```

The MorphJ generic class `Synchronized` can be instantiated by any Java interface, and implements that interface. The core synchronization functionality is expressed in two reflective declaration blocks, on lines 7-12, and 14-19. The first reflective declaration block statically iterates over every method of `X` matched by the pattern, "`public R m (A)`". Given that `R`, `A`, and `m` are pattern-matching variables, and `A` matches a sequence of types of any length, this pattern matches all non-`void`-returning methods of `X`. For each such method, this block declares a method with the exact same signature, which first synchronizes on a mutex, and then delegates its call to the underlying object of type `X`. Similarly, the second reflective declaration block declares the `void`-returning methods of `X`.

To obtain a synchronized version of an interface, e.g., `Collection`, one may instantiate `Synchronized` with that type, e.g., `Synchronized<Collection>`. A similar MorphJ generic class can be defined for use with classes.

Evaluation Similar to the AspectJ solution, the MorphJ generic class `Synchronized` offers a modular encapsulation of the synchronization functionality. In addition, the MorphJ solution offers programmers precise control over where the synchronization functionality is applied, as well as language support for separate type-checking. However, the MorphJ solution does share some of

the AspectJ solution's issues with respect to emulating the JCF type hierarchy.

The modularity of the MorphJ solution comes from the expressiveness of the reflective declaration blocks. These blocks allow the declaration of code that abstracts over the structure of unknown types. E.g., it is not necessary to know what methods a type has to declare methods with the same signatures, or to invoke these methods. `Synchronized` can be applied to any interface, in or outside of JCF, to create synchronized versions.

MorphJ supports an *explicit* model of concern application: To obtain a synchronized version of a data structure, a programmer must explicitly instantiate `Synchronized` with that data structure. This affords programmers precise control over the application of concern code.

Furthermore, a MorphJ generic class can be type-checked separately from all the (possibly infinite number of) types it can be instantiated with. If the MorphJ type-checker puts its stamp of approval on the well-typedness of a MorphJ generic class, the class is guaranteed to not introduce type errors no matter where and how it is used. Note that the type error we had in `SynchCall` simply cannot exist in MorphJ: Attempting to add the `return` keyword on line 17 would cause the MorphJ type-checker to complain that one cannot return from a `void`-returning method.

In the MorphJ solution, there is no need to manually declare synchronized subtypes of the original JCF classes—this subtyping happens automatically through the instantiation of `Synchronized`. However, the MorphJ solution does share a similar problem as the AspectJ solution with respect to type hierarchies. For reasons of type safety, instantiations of generic classes have an *invariant* subtyping relationship. For example, even though `List` is a subtype of `Collection`, `Synchronized<List>` is *not* an automatic subtype of `Synchronized<Collection>`. Obviously, we would like to be able to assign an object of type `Synchronized<List>` to a variable of type `Synchronized<Collection>`. One possible solution is to use the Java wildcard notation for variant parametric types [16, 89]. For example, `Synchronized<List>` is safely and automatically a subtype of `Synchronized<? extends Collection>`. This solution, however, must rely on the programmers to put more thought into declaring the most general types possible for variables. Furthermore, for type safety issues, certain methods cannot be invoked from `Synchronized<? extends Collection>`, whereas they could be from `Synchronized<Collection>`. Lastly but certainly not least, MorphJ does not currently support the use of wildcards in morphing generic

classes. Supporting this advanced feature is still future work.

4.7.3 Case Study II: DSTM2

DSTM2 [41] is a software transactional memory framework written in Java. It provides a number of "factory" classes supporting different transaction management policies. In Section 3.5, we saw the MorphJ implementation for a version of DSTM2 which accepts annotated Java classes as inputs. Here, we describe the latest version of DSTM2, which accepts Java interfaces as inputs. Given a Java interface (conforming to a certain format), a DSTM2 factory can be used to create objects implementing that interface, and protected by the factory's transaction policy. For example, consider the following interface:

```
interface INode {
  int   getValue();
  void  setValue(int   i);
  INode getNext ();
  void  setNext (INode n);
}
```

All interface inputs to DSTM2 must have `get`*f*/`set`*f* method pairs, for some set of properties *f*'s. The following code invokes `RecoverableFactory` to create an object implementing `INode`, managed by the "recoverable" policy:

```
RecoverableFactory rfactory = new RecoverableFactory(INode.class);
INode recoverableNode = rfactory.create();
```

Under the hood, `RecoverableFactory` performs magic by *generating* a class implementing `INode` as follows:

- For each pair of `get`*f*/`set`*f* methods, create a field *f*, as well as a "shadow" field `shadow`*f*.
- Provide implementations for `get`*f*/`set`*f* methods (field accessors) for the reading and writing of *f*, with code for transaction management. For instance, each method must first back up the values of each field into its corresponding shadow field. After reading/writing, it must reconcile the transaction with the global state, and possibly restore the values of each field from its shadow field.

The cross-cutting concern of interest in DSTM2 is "transaction management". The code for transaction management is almost the same for each field accessor, modulo the differences in the

name and type of the field being accessed, and the names and types of fields declared in the enclosing class. DSTM2 is the perfect example where replicating and specializing the concern code for each method and class is insufficient. DSTM2 cannot anticipate what interfaces it may receive as inputs, and thus cannot create specialized transactional implementations of these interfaces statically. Thus, DSTM2 employs a combination of Java reflection and bytecode engineering with BCEL [9].

Advantages and Disadvantages: One clear advantage of using reflection and BCEL is that it is indeed reusable. All the code for transaction management is defined in one place—or rather, all the code for transaction management is *generated*, and the generator code is defined in one place. This generator code can be reused with any input Java interface.

Another advantage is that it allows the mixing and matching of transaction policies. Objects of the same interface can be created using factories supporting different policies, but can interact.

However, reflection and BCEL are extremely low level techniques. Code written using these techniques is not only hard to understand and maintain, it also completely bypasses the Java type system. The type safety of generated classes is not at all guaranteed by the type-checker, and framework developers can only rely on testing, an incomplete technique, to find bugs that may introduce *type* errors into the generated code. (We point out a bug we discovered in the DSTM2 implementation later in this section.)

We next explore how to address "transaction management" with AOP and morphing, maintaining the modularity and flexibility of the original design, while providing high-level implementations with separate type-checking.

4.7.3.1 AOP with AspectJ

One thing we definitely cannot do with AspectJ, is generating new classes implementing arbitrary interfaces. Thus, given an interface such as `INode`, an AspectJ solution would require the users to provide a very basic implementation of `INode` before any aspect could advise it. This is a bit tedious, but not difficult. The following is an outline of such a basic implementation:

```
class SimpleNode implements INode {
  int value;  // field for getValue/setValue methods
  INode next; // field for getNext/setNext methods

  // basic, non-transactional implementations.
  int setValue(int i) { value = i; }
  ... // implement getValue, setNext, getNext
}
```

If we accept this as a prerequisite of an AspectJ solution, we can move on to advising the `get`*f* and `set`*f* methods of any type `X`. As with the previous case study, we have the choice of advising the execution sites of these methods, or the call sites. As call site advisement is a more flexible and less invasive approach (leaving the original definition untouched for further advisement), we use the following call-site pointcuts to identify the calls to the `get`*f* and `set`*f* methods:

```
pointcut getMeths(X x) : target(x) && call(* X.get*());
pointcut setMeths(X x) : target(x) && call(void X.set*(..));
```

For each field, we need some way of storing its value in case a transaction is rolled back. The DSTM2-generated classes accomplish this by declaring a shadow field for each field. In AspectJ, we can achieve a similar effect by holding old values of fields in a hash map, keyed by field names.[3]

The following generic aspect shows a skeleton solution, with the difficult parts (lines 7 and 9) purposely elided:

```
1  abstract aspect RecoverableAspect<X>
2      pertarget(getMeths(X) || setMeths(X)) {
3    ... // getMeths() and setMeths() definitions
4
5    HashMap shadowValues = new HashMap();
6    void around(X x) : setMeths(x) {
7      ... // backup all fields and open transaction
8      proceed(x);
9      ... // resolve conflict, and possibly roll back.
10   }
11   Object around(X x) : getMeths(x) { ... }
12 }
```

We again use `around` advice to inject transaction management code around the calls to `get`*f* and `set`*f* methods. The real challenge, however, is how we can backup the field values before the transaction begins (elided code on line 7), and restore them if the transaction needs to be rolled

[3]Intertype declarations in AspectJ do not work because we are declaring a *generic* aspect for any type `X`, and we do not know exactly what fields are in `X` until the aspect becomes instantiated.

back (elided code on line 9).

In order to backup the value of each field, we must first retrieve its current value, and store it in `shadowValues`, using the field's name as the key. However, the advice code is generic to the type it is operating on. We do not know the exact fields of `X` until it is instantiated. Thus, we must resort to using reflection. The following code retrieves all fields from the type of `x`, the target of a `set`*f*/`get`*f* call, and stores each field's value in `shadowValues` map. We have underlined calls to the reflection API:

```
Field[] fields = x.getClass().getFields();
for ( Field f : fields )
  shadowValues.put(f.getName(), f.get(x));
```

We must invoke similar code to restore the values of fields in case of a rollback.

Now we can apply the recoverable policy to any type by declaring a concrete aspect extending the instantiated version of `RecoverableAspect`. For example, to apply the policy to `SimpleNode`, we declare:

```
aspect RecoverableSimpleNodeAspect extends RecoverableAspect<SimpleNode> { }
```

Evaluation: The AspectJ solution is reusable: transaction management code is completely encapsulated within an aspect, and can be applied to any type.

This solution is also reasonably flexible. Recall that DSTM2 allows programmers to mix and match different transaction policies for objects of the *same* type. We can certainly define other policies using more aspects. For example, we can define `ObstructionFreeAspect`, which supports the "obstruction free" transaction policy. However, applying `RecoverableAspect` to `SimpleNode` universally advises the calls to the `get`*f* and `set`*f* methods of `SimpleNode` with recoverable transaction policy. Thus, in order to create objects managed by a different policy, we must declare another class implementing `INode`, e.g., `SimpleNode2`, and apply the aspect for that policy to `SimpleNode2` instead. Thus, for each policy, users must create a different class to be weaved by this policy's aspect. This is essentially the same technique we employed in the JCF case study.

This solution is also arguably more high level than the original DSTM2 implementation. There is no need to understand the Java bytecode standard to understand and maintain this code. However,

we did have to resort to the reflection API to retrieve and store values from and to fields. Not only does this affect performance—an important characteristic in transactional memory systems, it also, again, bypasses the Java type system. There is no guarantee that the code using reflection does not cause typing errors, e.g., storing the object of a wrong type into a field.

4.7.3.2 Morphing with MorphJ

The MorphJ solution essentially reproduces what would have been generated by the original DSTM2, but with a MorphJ generic class. We highlight the parts that contrast most with the AOP solution.

The following MorphJ generic class is a transactional implementation of any type `X`, using the recoverable policy:

```
1  class Recoverable<interface X> implements X {
2    <F>[f]for(   F   get#f()  : X.methods;
3           some void set#f(F) : X.methods ) {|
4    F f;
5    F shadow#f;
6    |}
7    ... // implementations of get#f and set#f methods.
8  }
```

The outer and nested patterns on lines 2 and 3 pick out exactly the `get`*f*/`set`*f* method pairs of any type `X`, for any name *f*. For each such pair, two fields are declared: `f`, and `shadow#f`—the field and its corresponding shadow field. Note that the AspectJ solution cannot enforce this format on the input interface or class.

The primary challenge in the AspectJ solution is how to backup and restore field values. This task is quite simple in the MorphJ solution. The following code backs up each field in `Recoverable<X>`:

```
<F>[f]for(F get#f() : X.methods;
          some void set#f(F) : X.methods; )
shadow#f = f;
```

The above reflective declaration block declares an assignment statement per iteration. The iterator definition picks out exactly the same `get`*f*/`set`*f* method pairs of `X`, and stores the value of each `f` into its corresponding `shadow#f`. Similar code can be used for restoring old values from shadow fields back into the originals.

Evaluation: `Recoverable<X>` is a modular definition of the recoverable transaction policy, and

can be applied to any interface `X`. Additionally, programmers have explicit control over where (and which) policy is applied. In contrast to the AOP solution, there is no need for declaring placeholder implementations just for concern weaving.

Additionally, the MorphJ solution is high level, using neither reflection nor bytecode engineering. The reflective declaration block of MorphJ lifts reflecting over the *static structure* of types to the language level. But the key difference between MorphJ and other languages offering pattern-based matching over structure (including AspectJ's own method/field pattern syntax), is that not only can programmers reflect over the structure of other types, they can *declare* code using the information captured by reflection. The ability to write code using these reflectively captured names and types is crucial to the implementation of backup and restore code without resorting to reflection.

High level support for reflection, both at the introspection *and* the declaration level, not only allows more expressive code, but also benefits the type safety of the language. As we mentioned previously, MorphJ provides separate type-checking. This means `Recoverable<X>` can be type-checked without knowing all the possible concrete types that `X` could be instantiated with, and guarantee that no matter what `X` is replaced with, the code is always well-typed.

To punctuate this point, our presentation of `Recoverable<X>` replicates a bug in the original DSTM2. Certain instantiations of `X` can cause type errors in the presented `Recoverable<X>`. `TrickyNode` is such a class:

```
interface TrickyNode {
  int   getvalue();
  void  setvalue(int i);
  float getshadowvalue();
  void  setshadowvalue(float f);
}
```

`Recoverable<TrickyNode>` contains fields `int value` and `int shadowvalue` from method pair `getvalue/setvalue`, and `float shadowvalue` and `float shadowshadowvalue` from method pair `getshadowvalue/setshadowvalue`. Clearly, the declarations `int shadowvalue` and `float shadowvalue` conflict.

The MorphJ compiler complains about these possible conflicting declarations when compiling `Recoverable`—without us having to provide the interface that induces this error. The benefit of a separately type-checkable language is exactly that programmers are warned of errors they did

not anticipate. The correct implementation of the field declaration block should have the following iterator definition:

```
<F>[f] for(  F get#f() : X.methods;
      some void set#f(F) : X.methods;
      no getshadow#f() : X.methods) ...
```

The second nested pattern blocks out any method `get#f` for which there is already a `getshadow#f` method. This means method `getvalue()` in `TrickyNode` would not be part of the iteration, and thus, `int value` and `int shadowvalue` are not declared in `Recoverable<TrickyNode>`.[4]

`TrickyNode` may seem contrived, but it illustrates the point that when declarations are done by reflecting over the structure of another type, using unknown *names*, errors of duplicate or conflicting declarations are easy to make. In fact, this exact bug exists in the DSTM2 implementation, and we only realized it after the MorphJ compiler complained about our buggy `Recoverable` class.

4.7.4 Discussion

We were able to produce reusable definitions of cross-cutting concerns using either AOP or morphing. However, the solutions always differed in 1) how easy and precise it is for programmers to control where concern code is applied, and 2) how high level the solution code is, in terms of both syntax and type checking support. These differences are consequences of the structural nature of the concerns.

Structurally Cross-Cutting Concerns: A Definition: We define a structurally cross-cutting concern as follows:

- The applicability of the concern is associated with static, *structural* patterns of programs, instead of runtime, *execution* patterns.
- The applicability of the concern does not change over the lifetime of an object, or depend on complex run-time state of the program. (The code for the concern itself could have runtime state, of course.)

[4] One could argue that this definition could result in unexpected instantiations. Programmers could employ MorphJ's `error` pattern to produce a compile-time error when the pattern is blocked.

It is easy to see why synchronization and transaction management are structural concerns. First, the position of application of synchronization or transaction management can be entirely captured by patterns on the structure of the programs. The code for synchronization needs to be applied to every method of a data structure type; the code for transaction management needs to be applied to every field accessor of a type, and the backup and restore code needs to be applied to every field of a type. Both methods and fields are part of a type's structure. They are not, however, part of a program's execution pattern.

Secondly, it is most commonly the case that a programmer chooses to either have a synchronized version of a data structure or not, by creating a data structure object with synchronization protection (or not). Once a data structure object is created, it is unlikely that programmers will add or remove synchronization from the object during a program's runtime, conditioned by the program's runtime state. The same can be said for transaction management.

The structural nature of these concerns can help us explain why they can be better addressed with one approach, and cause problems with the other.

Precise Control of Concern Application: The main issue with the AOP solutions of our case studies is the inability to precisely control where concern code is applied. In the JCF case study, the AOP solution does not easily allow programmers to use both synchronized and unsynchronized versions of the same data structure; in the DSTM2 case study, the AOP solution does not easily allow the mixing and matching of transaction management policies by controlling exactly which policies are applied to which objects.

The issue lies with AOP's focus on capturing and advising program execution patterns. AspectJ provides powerful pointcut patterns for capturing these execution patterns. However, there is generally no pattern powerful enough to precisely describe any arbitrary point in the lifetime of one object and that object exactly. Thus, if an object's usage does not follow an execution pattern—as is generally the case—it cannot be easily captured by AspectJ, either.

In contrast, morphing focuses on providing expressive constructs for capturing a program's static, structural patterns, and allowing the declaration of new code using the information captured by the patterns. A MorphJ generic class allows the declaration of code that is automatically

"weaved" into the structure of another program when the generic class is instantiated. This "weaving", however, happens through the creation of a new type, leaving the original type untouched and free to be "weaved" again. This deliberately explicit approach provides programmers with fine grained control over where a concern is applied.

At the same time, however, the explicit approach also forces programmers to be very aware of where a concern is needed. It could be very onerous if indeed a concern is to be weaved everywhere in a program. Furthermore, adding a concern that was not originally thought of involves whole program modification. AspectJ, on the other hand, makes applying concerns universally or obliviously very easy.[5]

Morphing also offers no constructs for capturing program execution patterns. For instance, there is no easy, pattern-based way to pick out constructor invocations on singleton types, nor to determine a recursive call of a method. Both patterns are easily identified using AspectJ. Thus, if a concern's application pattern relies on program execution patterns, capturing where the concern should apply in morphing can only be done programmatically.

High-Level Definitions: Regardless of the model and pattern of concern application, our case studies showed that in both AOP and morphing solutions, some code needs to be declared in a way that abstracts over the structure of other code. Storing the values of all fields into the backup fields (or into a hash map), while the declaring type of these fields is unknown, is one such example. The code for doing so must abstract over the differences in the declared fields of different types. AspectJ does not have support for declaring such structurally abstract code. In contrast, the entire morphing approach is based on declaring structurally abstract code, through reflective declaration blocks. MorphJ's patterns capture both types *and names*. The ability to declare code using these names is crucial to MorphJ's success in capturing DSTM2 functionality without using either reflection or bytecode engineering.

The ability to declare structurally abstract code is not at all at odds with AspectJ's execution-based advisement model. It is possible to incorporate this feature into AspectJ's intertype declarations, resulting in a more flexible, execution-based approach.

[5] Although, the effect of obliviousness on modularity is actively debated in research [42].

High level language support means not only convenient syntax, but also the checking of static program semantics. One of the crucial properties of modular software development is separate type checking—a piece of code should be type-checked separately from the (possibly infinite number of) programs it could be used with. It should be guaranteed that well-typed code would not introduce type errors to the programs it applies to.

AspectJ's execution-based nature makes providing separate type checking a major challenge. In order to reason about the typing properties of its advice, the AspectJ type checker must reason about the *execution* of programs described by the pointcuts. Reasoning about dynamic properties with a static system is in general undecidable.

The static nature of MorphJ, however, affords the MorphJ type checker plenty of static, structural information. MorphJ thus provides strong separate type checking guarantees. As we demonstrated in our case studies, this property is very useful in catching subtle but serious bugs.

4.7.5 A Summary of the Comparative Study

We compared AOP and morphing's relative suitability in addressing cross-cutting concerns by re-engineering two real-world Java applications. Even though reusable definitions are possible through either approach, AOP solutions were consistently less flexible in concern application, and lack modularity: There is no guarantees for the type-safety of the resulting code. The *structural* nature of the concerns studied is the cause for this difference. AOP is well-suited for capturing program *execution* patterns, and advising points captured by those patterns. However, it provides no support for capturing structural patterns. Morphing's strength is exactly the opposite, making it well-suited for addressing structural concerns, while providing strong typing guarantees.

Chapter 5

Conclusion

Structural abstraction represents a radical departure from the prevailing notion of abstraction, where reusability and modularity rely on the rigidity in code structure. Structurally abstract code can be declared so that its own structure can be automatically specialized through composition. As such, structural abstraction achieves a higher level of reuse compared to dominant abstraction mechanisms. At the same time, structural abstraction maintains a high level of modularity guarantee: The correctness of a piece of structurally abstract code is still checked independently from its eventual compositions, and errors are caught in a modular fashion.

5.1 A Broader Lesson

A broader idea that emerges from this work is that of *principled reflection*. Reflection has always been an important mechanism for introspecting and abstracting over code structure. However, existing reflection implementations (e.g., in languages such as Java, C#, or in more powerful systems like Meta-Object Protocols or Lisp macros) confuse compile-time entities, e.g., types and names, with runtime values, e.g., strings. For instance, in the Java reflection API, the class of an object is represented as another object—a runtime entity—of type `Class`; the name of a class is represented as a `String` object. Furthermore, existing reflection mechanisms allow powerful Turing-complete constructs to dictate the control and data flow of reflective code. For example, in Java, reflective code can be embedded in iteration and conditional constructs that employ Turing-complete conditions; data retrieved through reflection can be arbitrarily manipulated (e.g., stored in variables that can be reassigned or mutated). The combination of compile/runtime confusion and unrestrained power of Turing-complete control structures severely limits the modularity of the reflective approach: A compiler has very limited power to reason about what is being reflected and how reflected entities are manipulated in the declaration of new code.

Structural abstraction represents a principled approach to reflection. In both static type conditions and morphing, compile-time entities such as types and names stay as such to the compiler, through the use of type and name variables. Furthermore, code reflects over other code through patterns and subtyping condition, which offer limited forms of iteration and conditionals that are amenable to static analysis. This approach affords the compiler a great deal of power in reasoning about what is being reflected and how the entities retrieved via reflection are used. The combination of a clear separation between compile- and run-time entities, along with limited control constructs, is key in obtaining modular safety guarantees for structurally abstract code.

Principled reflection also gives rise to a new model of code as abstract sets: sets whose exact elements are unknown, but all elements are described by known properties. Code as abstract sets is a generalization of the traditional concept, in which a piece of code is a concrete, compile-time entity. In this new model, a piece of code with statically resolvable structure (e.g., exact name and argument types) is simply an abstract set with a cardinality of one. In a highly generic language like MorphJ, however, a reflectively declared method (or field, statement, etc.) is treated as an abstract set of methods (fields or statements, respectively) whose properties are determined by the reflective iterator it is defined under. The notion of abstract sets affords an easy way to model common type-safety properties such as declaration uniqueness and referential integrity. Declaration uniqueness can be checked through the disjointness of abstract sets, and referential integrity can be checked through set containment.

The challenge in reasoning about highly generic code is to construct a type system where *most type-correct programs can be proven so automatically*. The more variable the code, the more difficult it is to find the balance between the soundness and the completeness of the approach. The treatment of typing properties as set disjointness and containment occupies a sweet spot in the spectrum of code variability and completeness of type systems.

Of course, there is much more that can be reflected from code than just its static structure [17]. The approach of this work may serve as a basis for the design of a principled reflection of runtime properties.

5.2 A New Paradigm of Programming

Structural abstraction forms a sound foundation for a new paradigm of programming, where code is no longer considered an entity with a rigid, fixed structure, but instead, as data structures that can be inspected by and used to shape the structure of other code. This new paradigm opens the door for rethinking many of the existing mechanisms for code reuse and language interaction.

5.2.1 Rethinking Inheritance vs. Delegation

The mechanism for code reuse in many Object-Oriented languages is inheritance: class `A extends B`, and thus `A` inherits (and reuses) all the code in `B`. The criticisms of inheritance as a mechanism for reuse are well-documented in the research literature [24, 30]. Inheritance confuses the role of a class as a model for object behavior, and its role as an organizational unit of code. When `A` inherits code from `B`, `A` is automatically treated as a subtype, or a refinement of the model, of `B`. This may not be the desirable behavior—`A` may simply want to reuse some code. Inheritance is also a coarse-grained way for code reuse: Often times a subclass only needs to reuse a small subset of its superclass's methods, but is forced to inherit all of them. Various alternative reuse mechanisms have been proposed, such as mixins, traits, etc. [15, 23, 30]. These techniques either have trouble providing modular safety guarantees [15, 23], or they have limitations with respect to encapsulating and sharing state [30].

An alternative model of code reuse is delegation [65, 84]. Instead of inheriting code from class `B`, class `A` defines methods that model its own behavior, and delegates method calls to the appropriate methods in `B` if the behavior is shared. In the delegation approach, `A` is not treated as a subtype of `B`. `A` can also reuse as much or as little of `B`'s code as it sees fit. The synchronization proxies in Java Collections Framework described in Section 3.3 can be seen as code reuse through delegation. However, as is also evident through the synchronization proxies, delegation involves an enormous amount of boiler plate delegation method declarations. Furthermore, if `A` needs to reuse code from multiple classes through delegation, keeping track of state shared between different delegates is cumbersome and error-prone. Additionally, there is some performance penalty involved in each delegation.

Structural abstraction presents a fresh perspective to the inheritance vs. delegation debate.

Using morphing, class A can easily share code with class B by defining its methods (or fields) through reflective iteration over methods (or fields) of B. The body of these methods may delegate calls to the methods of B, which emulates the classic delegation approach. Code reuse does not confuse itself with the concept of modeling: Class A does not have to be a subtype of B to reuse code from B. Yet boiler-plate code is replaced with one reflective iteration block.

Alternatively, syntax could be introduced to indicate to the compiler that the code from B should actually be copied into the body of A, which more closely emulates inheritance. State variables of B can be explicitly kept inside of A. The overhead of delegation calls can also be removed, since the code of B is now inlined in svA, instead of being delegated to.

Furthermore, structural abstraction allows code reuse at a granularity that is defined by the programmer. Class A can choose to inherit all the methods in B, or only a certain subset of methods in B by refining the patterns used in reflective iteration over B. A may also choose to inherit methods of B, as well as methods from C, D, etc., using multiple reflective iteration blocks. This provides an emulation of "multiple inheritance". But, in the structurally abstract approach, programmers have explicit control over how methods from multiple classes are incorporated and interact. For instance, programmers can explicitly state that methods with the same signature from B, C, and D should first invoke the code of B, then D, then C (or any arbitrary order, or even leave out invocation of code of C). Similarly, state variables from multiple classes can be kept separate, or combined if they have the same name and type. In fact, this new form of inheritance can be generally defined as a MorphJ generic class A<X>, where A can inherit from any type X. We can similarly define generic classes A<X,Y>, or A<X,Y,Z>, etc. for generic "multiple inheritance".

Thus, structural abstraction is capable of supporting a hybrid approach between inheritance and delegation, where code reuse is kept a separate concept from modeling, granularity of code reuse can be explicitly controlled by the programmers, semantics of shared code and state can be explicitly managed, and the compiler has the opportunity to optimize away delegation.

5.2.2 Cross-language Structural Abstraction

Though the work presented in this dissertation focuses on using structural abstraction to support the composition of pieces of code written in the same, general-purpose language, structural

abstraction can be a powerful technique for facilitating the composition of pieces of code written in different, possibly domain-specific languages.

It is increasingly common that code written in one language must shape its own structure by reflecting over code written in a "foreign" language. The most notable example is the interaction between various general-purpose languages (e.g., Java, Ruby), and domain-specific languages for database schema description (e.g., XML) or querying (e.g., SQL). For instance, Java programs routinely create classes that reflect the structure of underlying database tables, so that data can be represented natively as objects. Programmers also often generate code in a foreign language to reflect the structure of code in the host language, e.g., constructing SQL queries from within Java, by reflecting over the structure of a Java class.

This type of structural correspondence across languages is commonly accomplished using ad-hoc techniques such as reflection APIs and string manipulation. Recently, frameworks such as Hibernate [3] and Ruby on Rails [6] have emerged to hide the complexity of such techniques from end programmers. Regardless, the underlying mechanisms are undisciplined: Code written using them is verbose, difficult to understand and maintain, and most importantly, carries no guarantees of correctness independently of specific instantiations. One of the most visible consequences of such undisciplined practices is exposure of code to security issues such as query injection attacks. Various program analysis [38,86] and language-based [18] approaches have been developed to detect such attacks. But these techniques focus on ensuring the syntactic correctness of query commands generated from the host language, not the semantic correctness of code generated, e.g., whether the database indeed contains the fields referenced by a generated SQL command. Furthermore, program analyses are necessarily best-effort techniques that tend to be either overly conservative, or unsound.

Cross-language structural abstraction offers a promising approach to allow tighter, semantically-aware integration of languages. A cross-language structural abstraction mechanism would allow principled reflection over code of a foreign language, with controlled iteration and conditionals over the structure of such code (e.g., a pattern-based static iteration over the columns of a database table description). It can expose the right kinds of code-level variables (e.g., variables representing column types and names) for reflecting and retrieving code from the foreign language. Lastly, a

principled cross-language reflection would provide a mapping from the static semantics of the foreign language to properties of abstract sets (e.g., all column names of a table are unique), so that these properties can be used in the type checking of code generated in either the foreign or the host language.

As software development moves toward a proliferation of domain-specific languages, composition of code written in different domain specific languages is going to be the key challenge in modular software construction. Cross-language structural abstraction can play an important role in the reusability and modularity of code.

To conclude, structural abstraction represents a significant trend in the evolution of programming languages. Most major advances in programming languages are modularity or reusability enhancements. The first step was taken with *procedural abstraction* in the 50s and 60s, which culminated in structured programming languages. Procedural abstraction captured algorithmic logic in a form that could be multiply reused both in the same program and across programs, over different data objects. The next major abstraction step was arguably *type abstraction* or *polymorphism*, which allowed the same abstract logic to be applied to multiple types of data, although the low-level code for each type would end up being substantially different. Structural abstraction may very well be the next big step in language evolution, where code can abstract over the structure of other program elements. The inclusion of such constructs in mainstream languages will be a topic of major importance for decades to come, and this dissertation represents a big step forward in this direction.

Appendix A

Featherweight cJ (FCJ): Proof of Soundness

We provide here proofs for the theorems presented in Section 2.6, as well as their supporting lemmas. Readers familiar with the proofs presented in [47] and [48] will find these proofs similar.

Theorem 1 (Subject Reduction)**.** *If* $\Delta;\Gamma\vdash \mathtt{e}\in\mathtt{T}$ *and* $\mathtt{e}\longrightarrow\mathtt{e}'$*, where* Δ *has non-variable bounds, then* $\Delta;\Gamma\vdash\mathtt{e}'\in\mathtt{S}$ *and* $\Delta\vdash\mathtt{S}<:\mathtt{T}$ *for some* S*.*

Proof. We prove using induction on the derivation of $\mathtt{e}\longrightarrow\mathtt{e}'$.

Case R-FIELD:

$\mathtt{e}=\mathtt{new\ C}\langle\overline{\mathtt{T}}\rangle(\overline{\mathtt{e}}).\mathtt{f}_i \quad \mathtt{e}'=\mathtt{e}_i$

By T-FIELD, T-NEW, and O-REFL:

$\Delta;\Gamma\vdash\mathtt{new\ C}\langle\overline{\mathtt{T}}\rangle(\overline{\mathtt{e}})\in\mathtt{C}\langle\overline{\mathtt{T}}\rangle \quad \Delta\vdash \mathit{bound}_\Delta(\mathtt{C}\langle\overline{\mathtt{T}}\rangle)\Uparrow^{\emptyset}\mathtt{C}\langle\overline{\mathtt{T}}\rangle \quad \mathit{fields}(\Delta,\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)=\overline{\mathtt{U}}\ \overline{\mathtt{f}}$

$\mathtt{U}_i\Downarrow_{\emptyset}\mathtt{U}_i$, where $\mathtt{U}_i=\mathtt{T} \quad \Delta;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}} \quad \Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{U}}$

Let $\mathtt{S}=\mathtt{S}_i$

Case R-INVK:

$\mathtt{e}=\mathtt{new\ C}\langle\overline{\mathtt{T}}\rangle(\overline{\mathtt{e}}).\langle\overline{\mathtt{V}}\rangle\mathtt{m}(\overline{\mathtt{d}}) \quad \mathtt{e}'=[\overline{\mathtt{d}}/\overline{\mathtt{x}},\ \mathtt{new\ C}\langle\overline{\mathtt{T}}\rangle(\overline{\mathtt{e}})/\mathtt{this}]\mathtt{e}_0$

$\mathit{mbody}(\mathtt{m}\langle\overline{\mathtt{V}}\rangle,\mathtt{C}\langle\overline{\mathtt{T}}\rangle)=(\overline{\mathtt{x}},\mathtt{e}_0)$

By T-INVK, T-NEW, O-REFL:

$\Delta;\Gamma\vdash\mathtt{new\ C}\langle\overline{\mathtt{T}}\rangle(\overline{\mathtt{e}})\in\mathtt{C}\langle\overline{\mathtt{T}}\rangle \quad \Delta\vdash \mathit{bound}_\Delta(\mathtt{C}\langle\overline{\mathtt{T}}\rangle)\Uparrow^{\emptyset}\mathtt{C}\langle\overline{\mathtt{T}}\rangle$

$\mathit{mtype}(\Delta,\ \mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)=\langle\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}\rangle\overline{\mathtt{U}}\to\mathtt{U}_0 \quad \Delta\vdash\mathtt{C}\langle\overline{\mathtt{T}}\rangle\ ok \quad \Delta\vdash\overline{\mathtt{V}}\ ok$

$\Delta\vdash\overline{\mathtt{V}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \quad \Delta;\Gamma\vdash\overline{\mathtt{d}}\in\overline{\mathtt{S}} \quad \Delta\vdash\overline{\mathtt{S}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}} \quad [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0\Downarrow_{\emptyset}\mathtt{T}$, where $\mathtt{T}=[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0$.

By Lemma 1, for some N, $\mathtt{S}_0$,

$\Delta\vdash\mathtt{C}\langle\overline{\mathtt{T}}\rangle<:\mathtt{N} \quad \Delta\vdash\mathtt{N}\ ok \quad \Delta\vdash\mathtt{S}_0<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 \quad \Delta;\overline{\mathtt{x}}:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}},\mathtt{this}:\mathtt{N}\vdash\mathtt{e}_0\in\mathtt{S}_0$

By Lemma 4, there is some $\mathtt{S}_0'$ such that:

$\Delta;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}},\ \mathtt{new\ C}\langle\overline{\mathtt{T}}\rangle/\mathtt{this}]\mathtt{e}_0\in\mathtt{S}_0' \quad \Delta\vdash\mathtt{S}_0'<:\mathtt{S}_0$

By S-TRANS, $\Delta \vdash \mathtt{S}'_0 <: \mathtt{T}$. Let $\mathtt{S} = \mathtt{S}'_0$.

Case R-CAST:

$\mathtt{e} = \mathtt{(T)new\ C<\overline{T}>(\overline{e})} \quad \mathtt{e}' = \mathtt{new\ C<\overline{T}>(\overline{e})} \quad \emptyset \vdash \mathtt{C<\overline{T}> <: T}$

By T-CAST, T-NEW: $\Delta \vdash \mathtt{new\ C<\overline{T}>(\overline{e})} \in \mathtt{C<\overline{T}>}$

Let $\mathtt{S} = \mathtt{C<\overline{T}>}$.

Case RC-FIELD:

$\mathtt{e} = \mathtt{e}_0\mathtt{.f} \quad \mathtt{e}' = \mathtt{e}'_0\mathtt{.f} \quad \mathtt{e}_0 \longrightarrow \mathtt{e}'_0$

By T-FIELD:

$\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \quad \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}_0) \Uparrow^{\Delta'} \mathtt{C<\overline{U}>} \quad \mathit{fields}((\Delta,\Delta'),\ \mathtt{C<\overline{U}>}) = \overline{\mathtt{S}}\ \overline{\mathtt{f}} \quad \mathtt{S}_i \Downarrow_{\Delta'} \mathtt{T}$

By induction hypothesis, there exists $\mathtt{S}_0$ such that:

$\Delta;\Gamma \vdash \mathtt{e}'_0 \in \mathtt{S}_0 \quad \Delta \vdash \mathtt{S}_0 <: \mathtt{T}_0$

By Lemma 2, for some Δ'', $\mathtt{D<\overline{S}>}$, $\overline{\mathtt{V}}$, $\mathtt{V}'_0$,

$\Delta \vdash \mathit{bound}_\Delta(\mathtt{S}_0) \Uparrow^{\Delta''} \mathtt{D<\overline{S}>} \quad \mathit{fields}(\Delta,\Delta'',\ \mathtt{D<\overline{S}>}) = \mathtt{...,}\overline{\mathtt{V}}\ \overline{\mathtt{f}} \quad \mathtt{V}_i \Downarrow_{\Delta''} \mathtt{V}'_0 \quad \Delta \vdash \mathtt{V}'_0 <: \mathtt{T}$

By T-FIELD, $\Delta;\Gamma \vdash \mathtt{e}'_0\mathtt{.f} \in \mathtt{V}'_0$.

Let $\mathtt{S} = \mathtt{V}'_0$.

Case RC-INV-RECV:

$\mathtt{e}_0\mathtt{.<\overline{V}>m(\overline{e})} \longrightarrow \mathtt{e}'_0\mathtt{.<\overline{V}>m(\overline{e})} \quad \mathtt{e}_0 \longrightarrow \mathtt{e}'_0$

By the induction hypothesis, for some $\mathtt{T}'_0$:

$\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \quad \Delta;\Gamma \vdash \mathtt{e}'_0 \in \mathtt{T}'_0 \quad \Delta \vdash \mathtt{T}_0, \mathtt{T}'_0\ \mathit{ok} \quad \Delta \vdash \mathtt{T}'_0 <: \mathtt{T}_0$

By T-INVK,

$\Delta \vdash \mathit{bound}_\Delta(\mathtt{T}_0) \Uparrow^{\Delta'} \mathtt{C<\overline{T}>} \quad \mathit{mtype}((\Delta,\Delta'),\ \mathtt{m},\ \mathtt{C<\overline{T}>}) = \mathtt{<\overline{Y}<\overline{P}>\overline{U} \to U_0}$

$\{\overline{\mathtt{Y}}\} \cap \Delta' = \emptyset \qquad \Delta \vdash \overline{\mathtt{V}}\ \mathit{ok} \qquad \Delta,\Delta' \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$

$\Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \Delta,\Delta' \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}} \quad [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 \Downarrow_{\Delta'} \mathtt{T}$

By Lemma 3, for some Δ'', $\mathtt{D<\overline{S}'>}$, $\overline{\mathtt{P}}'$, $\overline{\mathtt{U}}'$, and $\mathtt{U}'_0$,

$\Delta \vdash bound_\Delta(\mathtt{T}'_0) \Uparrow^{\Delta''} \mathtt{D}\langle\overline{\mathtt{S}}'\rangle$ $\quad mtype((\Delta,\Delta''),\ \mathtt{m},\ \mathtt{D}\langle\overline{\mathtt{S}}'\rangle) = \langle\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}'\rangle \overline{\mathtt{U}}' \to \mathtt{U}'_0$

$\Delta, \Delta'' \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}'$ $\quad \Delta, \Delta'' \vdash \overline{\mathtt{W}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}'$

$[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}'_0 \Downarrow_{\Delta''} \mathtt{V}'_0$ $\quad \Delta \vdash \mathtt{V}'_0 <: \mathtt{T}$

By T-INVK, $\Delta;\Gamma \vdash \mathtt{e}'_0.\langle\overline{\mathtt{V}}\rangle\mathtt{m}(\overline{\mathtt{e}}) \in \mathtt{V}'_0$.

Case RC-INV-ARG:

$\mathtt{e}_0.\langle\overline{\mathtt{V}}\rangle\mathtt{m}(\ldots,\mathtt{e}_i,\ldots) \longrightarrow \mathtt{e}_0.\langle\overline{\mathtt{V}}\rangle\mathtt{m}(\ldots,\mathtt{e}'_i,\ldots)$ $\quad \mathtt{e}_i \longrightarrow \mathtt{e}'_i$

By T-INVK:

$\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0$ $\quad \Delta \vdash bound_\Delta(\mathtt{T}_0) \Uparrow^{\Delta'} \mathtt{C}\langle\overline{\mathtt{T}}\rangle$

$mtype((\Delta,\Delta'),\ \mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = \langle\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\rangle \overline{\mathtt{U}} \to \mathtt{U}_0$ $\quad \{\overline{\mathtt{Y}}\} \cap \Delta' = \emptyset$

$\Delta \vdash \overline{\mathtt{V}}$ $\ ok$ $\quad \Delta, \Delta' \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$

$\Delta; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}}$ $\quad \Delta, \Delta' \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}$

$[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 \Downarrow_{\Delta'} \mathtt{T}$

$\Delta;\Gamma \vdash \mathtt{e}_i \in \mathtt{S}_i$. By induction hypothesis, for some $\mathtt{S}'_i$,

$\Delta;\Gamma \vdash \mathtt{e}_i' \in \mathtt{S}'_i$ $\quad \Delta \vdash \mathtt{S}'_i <: \mathtt{S}_i$

By S-TRANS and Lemma 5, $\Delta,\Delta' \vdash \mathtt{S}'_i <: \mathtt{U}_i$

By T-INVK, $\Delta;\Gamma \vdash \mathtt{e}_0.\langle\overline{\mathtt{V}}\rangle\mathtt{m}(\ldots,\mathtt{e}'_i,\ldots) \in \mathtt{T}$

Case RC-NEW-ARG:

$\mathtt{new}\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle(\ldots,\mathtt{e}_i,\ldots) \longrightarrow \mathtt{new}\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle(\ldots,\mathtt{e}'_i,\ldots)$ $\quad \mathtt{e}_i \longrightarrow \mathtt{e}'_i$

By T-NEW: $\mathtt{T} = \mathtt{C}\langle\overline{\mathtt{T}}\rangle$

$\Delta \vdash \mathtt{C}\langle\overline{\mathtt{T}}\rangle$ $\ ok$ $\quad fields(\Delta,\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}$ $\quad \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}}$ $\quad \Delta \vdash \overline{\mathtt{S}} <: \overline{\mathtt{U}}$ $\quad \Delta;\Gamma \vdash \mathtt{e}_i \in \mathtt{S}_i$

By induction hypothesis, for some $\mathtt{S}'_i$, $\quad \Delta;\Gamma \vdash \mathtt{e}'_i \in \mathtt{S}'_i$ $\quad \Delta \vdash \mathtt{S}'_i <: \mathtt{S}_i$

By S-TRANS and T-NEW, $\Delta;\Gamma \vdash \mathtt{new}\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle(\ldots,\mathtt{e}'_i,\ldots) \in \mathtt{C}\langle\overline{\mathtt{T}}\rangle$

Case RC-CAST:

$(\mathtt{T})\mathtt{e}_0 \longrightarrow (\mathtt{T})\mathtt{e}'_0$ $\quad \mathtt{e}_0 \longrightarrow \mathtt{e}'_0$

There are two cases when $\Delta; \Gamma \vdash (\mathtt{T})\mathtt{e}_0 \in \mathtt{T}$:

1. T-CAST:

   $\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \quad \Delta \vdash \mathtt{T}\ ok \quad \Delta \vdash \mathtt{T}_0 <: bound_\Delta(\mathtt{T})$ or $\Delta \vdash bound_\Delta(\mathtt{T}) <: \mathtt{T}_0$

   By induction and S-TRANS, there is a $\mathtt{T}'_0$ such that

   $\Delta;\Gamma \vdash \mathtt{e}'_0 \in \mathtt{T}'_0 \quad \Delta \vdash \mathtt{T}'_0 <: \mathtt{T}_0 \quad \Delta \vdash \mathtt{T}'_0 <: bound_\Delta(\mathtt{T})$ or $\Delta \vdash bound_\Delta(\mathtt{T}) <: \mathtt{T}'_0$

   By T-CAST, $\Delta;\Gamma \vdash \mathtt{(T)e}'_0 \in \mathtt{T}$.

2. T-SCAST: similar to the case by T-CAST.

□

Theorem 2 (Progress)**.** *Let* e *be a well-typed expression.*

1. If e *has* $\mathtt{new\ C<\overline{T}>(\overline{e}).f}$ *as a subexpression, then* $fields(\emptyset,\ \mathtt{C<\overline{T}>}) = \mathtt{\overline{U}\ \overline{f}}$*, and* $\mathtt{f} = \mathtt{f}_i$*.*

2. If e *has* $\mathtt{new\ C<\overline{T}>(\overline{e}).m(\overline{d})}$ *as a subexpression, then* $mbody(\emptyset,\ \mathtt{m},\ \mathtt{C<\overline{T}>}) = (\overline{\mathtt{x}}, \mathtt{e}_0)$ *and* $|\overline{\mathtt{x}}| = |\overline{\mathtt{d}}|$.

Proof. 1. Immediate from T-FIELD, T-NEW. 2. Immediate from T-INVK, T-NEW, and the definition of $mbody$. □

Theorem 3 (Type Soundness)**.** *If* $\emptyset;\emptyset \vdash \mathtt{e} \in \mathtt{T}$ *and* $\mathtt{e} \longrightarrow^* \mathtt{e}'$ *being a normal form, then* e′ *is either a value* v *such that* $\emptyset;\emptyset \vdash v \in \mathtt{S}$ *and* $\emptyset \vdash \mathtt{S} <: \mathtt{T}$ *for some* S*, or an expression that includes* $\mathtt{(T)new\ C<\overline{T}>(\overline{e})}$ *where* $\emptyset \not\vdash \mathtt{C<\overline{T}> <: T}$.

Proof. Proof is immediate from Theorem 1 and Theorem 2. □

Lemma 1. *If* $mtype(\Delta,\ \mathtt{m},\ \mathtt{C<\overline{T}>}) = \mathtt{<\overline{Y} \lhd \overline{P}> \overline{U} \to U}_0$, $mbody(\Delta,\ \mathtt{m<\overline{V}>},\ \mathtt{C<\overline{T}>}) = (\overline{\mathtt{x}}, \mathtt{e}_0)$, $\Delta \vdash \mathtt{C<\overline{T}>}\ ok$*,* $\Delta \vdash \mathtt{\overline{V}}\ ok$*, and* $\Delta \vdash \mathtt{\overline{V} <: [\overline{V}/\overline{Y}]\overline{P}}$*, then there exist some* N *and* S *such that* $\Delta \vdash \mathtt{C<\overline{T}> <: N}$ *and* $\Delta \vdash \mathtt{N}\ ok$ *and* $\Delta \vdash \mathtt{S} <: \mathtt{[\overline{V}/\overline{Y}]U}_0$ *and* $\Delta;\overline{\mathtt{x}}:\mathtt{[\overline{V}/\overline{Y}]\overline{U}}, \mathtt{this:N} \vdash \mathtt{e}_0 \in \mathtt{S}$.

Proof. Prove by induction on the derivation of $mtype$.

Case MT-CLASS:

$CT(\mathtt{C}) = \mathtt{class\ C<\overline{X} \lhd \overline{N}>\ <\overline{X} \lhd \overline{H}> ? \lhd D<\overline{S}>\ \{...\overline{M}\}}$

$\mathtt{<\overline{X} \lhd \overline{R}> ? <\overline{Y} \lhd \overline{P}'> U}'_0\ \mathtt{m\ (\overline{U}'\ \overline{x})\ \{\ \uparrow e}'_0\mathtt{;\}} \in \mathtt{\overline{M}} \qquad \Delta \vdash \mathtt{\overline{T} <: [\overline{T}/\overline{X}]\overline{H}}$

$\mathtt{\overline{P} = [\overline{T}/\overline{X}]\overline{P}'} \quad \mathtt{\overline{U} = [\overline{T}/\overline{X}]\overline{U}'} \quad \mathtt{U}_0 = \mathtt{[\overline{T}/\overline{X}]U}'_0$

By MB-CLASS:

$mbody(\Delta,\ \mathtt{m}\langle\overline{\mathtt{V}}\rangle,\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)=(\overline{\mathtt{x}},[\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e}'_0)\quad \mathtt{e}_0=[\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e}'_0$

By T-METHOD:

$\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\overline{\mathtt{X}}<:\overline{\mathtt{R}}\qquad \Delta'=\overline{\mathtt{X}}<:\overline{\mathtt{H}},\overline{\mathtt{Y}}<:\overline{\mathtt{P}}'\qquad \Delta'\vdash\overline{\mathtt{P}}',\overline{\mathtt{H}},\overline{\mathtt{U}}',\mathtt{U}'_0\ ok$

$\Delta';\overline{\mathtt{x}}:\overline{\mathtt{U}}',\mathtt{this}:\mathtt{C}\langle\overline{\mathtt{X}}\rangle\vdash\mathtt{e}'_0\in\mathtt{S}_0\quad \Delta'\vdash\mathtt{S}_0<:\mathtt{U}'_0$

$CT(\mathtt{C})=\mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle\ \langle\overline{\mathtt{X}}\lhd\overline{\mathtt{H}}\rangle?\lhd\mathtt{D}\langle\overline{\mathtt{S}}\rangle\ \{\ldots\overline{\mathtt{M}}\}$

$override(\Delta',\ \mathtt{m},\ \langle\overline{\mathtt{X}}\lhd\overline{\mathtt{H}}\rangle?\mathtt{D}\langle\overline{\mathtt{S}}\rangle,\ \langle\overline{\mathtt{X}}\lhd\overline{\mathtt{R}}\rangle?\overline{\mathtt{U}}'\rightarrow\mathtt{U}'_0)$

By WF-CLASS: $\Delta\vdash\overline{\mathtt{T}}\ ok\quad \Delta\vdash\overline{\mathtt{T}}<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$

By Lemma 5, 6, 7, 10,

$\Delta\vdash[\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S}_0<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}'_0\quad \Delta;\overline{\mathtt{x}}:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}},\mathtt{this}:\mathtt{C}\langle\overline{\mathtt{T}}\rangle\vdash\mathtt{e}'_0\in[\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S}_0$

Let $\mathtt{N}=\mathtt{C}\langle\overline{\mathtt{T}}\rangle$, $\mathtt{S}=[\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S}_0$.

Case MT-SUPER:

$mtype(\Delta,\ \mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)=mtype(\Delta,\ \mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D}\langle\overline{\mathtt{S}}\rangle)$

$CT(\mathtt{C})=\mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle\ \langle\overline{\mathtt{X}}\lhd\overline{\mathtt{H}}\rangle?\lhd\mathtt{D}\langle\overline{\mathtt{S}}\rangle\ \{\ldots\overline{\mathtt{M}}\}$

$\mathtt{m}\notin\overline{\mathtt{T}}\quad \Delta\vdash\overline{\mathtt{T}}<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{H}}$

By MB-SUPER:

$mbody(\Delta,\ \mathtt{m}\langle\overline{\mathtt{V}}\rangle,\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)=mbody(\Delta,\ \mathtt{m}\langle\overline{\mathtt{V}}\rangle,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D}\langle\overline{\mathtt{S}}\rangle)$

By WF-CLASS and $\Delta\vdash\mathtt{C}\langle\overline{\mathtt{T}}\rangle\ ok$, $\Delta\vdash\overline{\mathtt{T}}\ ok\quad \Delta\vdash\overline{\mathtt{T}}<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$

By T-CLASS, $\overline{\mathtt{X}}<:\overline{\mathtt{H}}\vdash\mathtt{D}\langle\overline{\mathtt{S}}\rangle\ ok$

By Lemma 9, $\Delta\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D}\langle\overline{\mathtt{S}}\rangle\ ok$

By induction hypothesis, there is $\mathtt{N}'$, $\mathtt{S}'$such that

$\Delta\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D}\langle\overline{\mathtt{S}}\rangle<:\mathtt{N}\quad \Delta\vdash\mathtt{N}'ok\quad \Delta\vdash\mathtt{S}'<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0\quad \Delta;\overline{\mathtt{x}}:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}},\mathtt{this}:\mathtt{N}\vdash\mathtt{e}_0\in\mathtt{S}'$

By S-CLASS, $\Delta\vdash\mathtt{C}\langle\overline{\mathtt{T}}\rangle<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D}\langle\overline{\mathtt{S}}\rangle$

By S-TRANS, $\Delta\vdash\mathtt{C}\langle\overline{\mathtt{T}}\rangle<:\mathtt{N}'$.

Let $\mathtt{N}=\mathtt{N}'$.

By Lemma 7, there exists S such that $\Delta;\overline{\mathtt{x}}:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}],\mathtt{this}:\mathtt{C}\langle\overline{\mathtt{T}}\rangle\vdash\mathtt{e}_0\in\mathtt{S}$, $\Delta\vdash\mathtt{S}<:\mathtt{S}'$.

By S-TRANS, $\Delta \vdash \mathtt{S} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0$.

□

Lemma 2. *If Δ has non-variable bounds and $\Delta \vdash bound_\Delta(\mathtt{T}) \Uparrow^{\Delta_1} \mathtt{C}\langle\overline{\mathtt{T}}\rangle$ and $fields((\Delta,\Delta_1),\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}$ and $\mathtt{U}_i \Downarrow_{\Delta_1} \mathtt{U}'$, then for any $\mathtt{S}$ such that $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and $\Delta \vdash \mathtt{S}\ ok$, it holds that $\Delta \vdash bound_\Delta(\mathtt{S}) \Uparrow^{\Delta_2} \mathtt{D}\langle\overline{\mathtt{S}}\rangle$ and $fields((\Delta,\Delta_2),\ \mathtt{D}\langle\overline{\mathtt{S}}\rangle) = \ldots, \overline{\mathtt{V}}\ \overline{\mathtt{f}}$ and $\mathtt{V}_i \Downarrow_{\Delta_2} \mathtt{V}'$ and $\Delta \vdash \mathtt{V}' <: \mathtt{U}'$ for some Δ_2, $\mathtt{D}\langle\overline{\mathtt{S}}\rangle$, $\overline{\mathtt{V}}$, and $\mathtt{V}'$.*

Proof. We prove by induction on the derivation of $\Delta \vdash \mathtt{S} <: \mathtt{T}$.

Case S-REFL:

Let $\Delta_2 = \Delta_1$, $\mathtt{D}\langle\overline{\mathtt{S}}\rangle = \mathtt{C}\langle\overline{\mathtt{T}}\rangle$, $\overline{\mathtt{V}} = \overline{\mathtt{U}}$, and $\mathtt{V}' = \mathtt{U}'$.

Case S-TRANS:

Let there be a type $\mathtt{U}$ such that $\Delta \vdash \mathtt{S} <: \mathtt{U}$, $\Delta \vdash \mathtt{U} <: \mathtt{T}$, and $\Delta \vdash \mathtt{U}\ ok$.

By induction hypothesis, there exists Δ_u, $\overline{\mathtt{f}}_u$, $\mathtt{D}_u\langle\overline{\mathtt{S}}_u\rangle$, $\overline{\mathtt{V}}_u$, and $\mathtt{V}'_u$ such that:

$\Delta \vdash bound_\Delta(\mathtt{U}) \Uparrow^{\Delta_u} \mathtt{D}_u\langle\overline{\mathtt{S}}_u\rangle$ $\quad fields((\Delta,\Delta_u),\ \mathtt{D}_u\langle\overline{\mathtt{S}}_u\rangle) = \ldots, \overline{\mathtt{V}}_u\ \overline{\mathtt{f}}$

$\mathtt{V}_{u_i} \Downarrow_{\Delta_u} \mathtt{V}'_u$ $\quad \Delta \vdash \mathtt{V}'_u <: \mathtt{U}'$

Also by induction hypothesis, there exists Δ_s, $\overline{\mathtt{f}}_s$, $\mathtt{D}_s\langle\overline{\mathtt{S}}_s\rangle$, $\overline{\mathtt{V}}_s$, and $\mathtt{V}'_s$ such that:

$\Delta \vdash bound_\Delta(\mathtt{S}) \Uparrow^{\Delta_s} \mathtt{D}_s\langle\overline{\mathtt{S}}_s\rangle$ $\quad fields((\Delta,\Delta_s),\ \mathtt{D}_s\langle\overline{\mathtt{S}}_s\rangle) = \ldots, \overline{\mathtt{V}}_s\ \overline{\mathtt{f}}$

$\mathtt{V}_{s_i} \Downarrow_{\Delta_s} \mathtt{V}'_s$ $\quad \Delta \vdash \mathtt{V}'_s <: \mathtt{V}'_u$

Let $\Delta_2 = \Delta_s$, $\mathtt{D}\langle\overline{\mathtt{S}}\rangle = \mathtt{D}_s\langle\overline{\mathtt{S}}_s\rangle$, $\overline{\mathtt{V}} = \overline{\mathtt{V}}_s$, and $\mathtt{V}' = \mathtt{V}'_s$.

Case S-UBOUND:

$bound_\Delta(\mathtt{S}) = bound_\Delta(\mathtt{T})$.

Let $\Delta_1 = \Delta_2$, $\mathtt{D}\langle\overline{\mathtt{S}}\rangle = \mathtt{C}\langle\overline{\mathtt{T}}\rangle$, $\overline{\mathtt{V}} = \overline{\mathtt{U}}$, and $\mathtt{V}' = \mathtt{U}'$.

Case S-LBOUND:

$\mathtt{X}$ is a bound, and thus the precondition of the lemma, that Δ has non-variable bounds, is not satisfied.

Case S-CLASS: $\mathtt{S} = \mathtt{D}\langle\overline{\mathtt{W}}\rangle$

$CT(\mathtt{D})$=class D<$\overline{\mathtt{X}}$◁$\overline{\mathtt{N}}$> <$\overline{\mathtt{X}}$◁$\overline{\mathtt{R}}$>?◁C<$\overline{\mathtt{S}}'$> { $\overline{\mathtt{H}}$ $\overline{\mathtt{g}}$; $\overline{\mathtt{M}}$ }

$\Delta\vdash\mathtt{D}\langle\overline{\mathtt{W}}\rangle\Uparrow^{\Delta_2}\mathtt{D}\langle\overline{\mathtt{S}}\rangle$ $\quad\Delta,\Delta_2\vdash\overline{\mathtt{S}}<:[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\overline{\mathtt{R}}$ $\quad[\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{C}\langle\overline{\mathtt{S}}'\rangle\Downarrow_{\Delta_2}\mathtt{T}$

There are three rules defining *fields*. By analyze them case by case.

Subcase F-OBJECT: Trivial

Subcase F-CLASS-S:

$\mathit{fields}((\Delta,\Delta_2),\ \mathtt{D}\langle\overline{\mathtt{S}}\rangle)=\overline{\mathtt{V}}\ \overline{\mathtt{f}},[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\overline{\mathtt{H}}\ \overline{\mathtt{g}}$ where $\mathit{fields}((\Delta,\Delta_2),\ [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{C}\langle\overline{\mathtt{S}}'\rangle)=\overline{\mathtt{V}}\ \overline{\mathtt{f}}$

By Lemma 13, $\mathtt{V}_i\Downarrow_{\Delta_2}\mathtt{U}_i$.

Subcase F-CLASS:

T must be Object, since condition for superclass is not satisfied. Conclusion follows trivially.

Case S-VAR: $\mathtt{S}=\mathtt{C}\langle\overline{\mathtt{v}}\overline{\mathtt{S}}\rangle$ $\quad\mathtt{T}=\mathtt{C}\langle\overline{\mathtt{w}}\overline{\mathtt{T}}\rangle$ $\quad\overline{\mathtt{v}}\leq\overline{\mathtt{w}}$

We analyze the four possible values of $\mathtt{w}_j$ separately. Without loss of generality, we assume the rest of $\overline{\mathtt{w}}$ remain invariant, e.g. $\mathtt{w}_i$=o, $j\neq i$.

Subcase $\mathtt{w}_i$=o:

$\mathtt{v}_j$=o, and $\mathtt{S}_j=\mathtt{T}_j$. Thus, S=T. Conclusion follows trivially.

Subcase $\mathtt{w}_j$=+:

$\mathtt{v}_j$=o or +, $\Delta\vdash\mathtt{S}_j<:\mathtt{T}_j$.

If $\mathtt{v}_j$=+, let $\Delta_1=\mathtt{T}'_j:(+,\mathtt{T}_j)$, $\Delta_2=\mathtt{S}'_j:(+,\mathtt{S}_j)$,

$\mathit{bound}_\Delta(\mathtt{C}\langle\overline{\mathtt{T}}_1,+\mathtt{T}_j,\overline{\mathtt{T}}_2\rangle)\Uparrow^{\Delta_1}\mathtt{C}\langle\overline{\mathtt{T}}_1,\mathtt{T}'_j,\overline{\mathtt{T}}_2\rangle$ $\quad\mathit{fields}((\Delta,\Delta_1),\ \mathtt{C}\langle\overline{\mathtt{T}}_1,\mathtt{T}'_j,\overline{\mathtt{T}}_2\rangle)=\overline{\mathtt{U}}\ \overline{\mathtt{f}}$ $\quad\mathtt{U}_i\Downarrow_{\Delta_1}\mathtt{U}'$

$\mathit{bound}_\Delta(\mathtt{C}\langle\overline{\mathtt{S}}_1,+\mathtt{S}_j,\overline{\mathtt{S}}_2\rangle)\Uparrow^{\Delta_2}\mathtt{C}\langle\overline{\mathtt{S}}_1,\mathtt{S}'_j,\overline{\mathtt{S}}_2\rangle$ $\quad\mathit{fields}((\Delta,\Delta_2),\ \mathtt{C}\langle\overline{\mathtt{S}}_1,\mathtt{S}'_j,\overline{\mathtt{S}}_2\rangle)=\overline{\mathtt{V}}\ \overline{\mathtt{f}}$ $\quad\mathtt{V}_i\Downarrow_{\Delta_2}\mathtt{V}'$

By Lemma 14(2), $\Delta\vdash\mathtt{U}_i\Downarrow_{\Delta_2}\mathtt{U}'$, and $\Delta\vdash\mathtt{V}'<:\mathtt{U}'$.

If $\mathtt{v}_j$=o, let $\Delta_1=\mathtt{T}'_j<:\mathtt{T}_j$, $\Delta_2=\emptyset$. Similarly to the above case, by Lemma 14(1), $[\mathtt{T}_j/\mathtt{T}'_j]\mathtt{U}_i\Downarrow_\emptyset\mathtt{V}'$, $\Delta\vdash\mathtt{V}'<:\mathtt{U}'$.

Subcase $\mathtt{w}_i=$ -:

The proof is similar to the above cases.

□

Lemma 3. *If* Δ *has non-variable bounds and* $\Delta\vdash \mathtt{T}$ *ok, and* $\Delta\vdash bound_\Delta(\mathtt{T})\Uparrow^{\Delta_1}\mathtt{C}<\overline{\mathtt{T}}>$ *and* $mtype((\Delta,\Delta_1),\ \mathtt{m},\ \mathtt{C}<\overline{\mathtt{T}}>)=<\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}>\overline{\mathtt{U}}\to\mathtt{U}_0$ *and* $\Delta\vdash\overline{\mathtt{V}},\overline{\mathtt{W}}$ *ok and* $\Delta,\Delta_1\vdash\overline{\mathtt{V}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$ *and* $\Delta,\Delta_1\vdash\overline{\mathtt{W}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}$ *and* $[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0\Downarrow_{\Delta_1}\mathtt{V}_0$*, then for any* S *such that* $\Delta\vdash\mathtt{S}<:\mathtt{T}$ *and* $\Delta\vdash\mathtt{S}$ *ok, there exists* Δ_2, $\mathtt{D}<\overline{\mathtt{S}}>$, $\overline{\mathtt{P}}'$, $\overline{\mathtt{U}}'$, *and* $\mathtt{U}'_0$ *such that* $\Delta\vdash bound_\Delta(\mathtt{S})\Uparrow^{\Delta_2}\mathtt{D}<\overline{\mathtt{S}}>$ *and* $mtype((\Delta,\Delta_2),\ \mathtt{m},\ \mathtt{D}<\overline{\mathtt{S}}>)=<\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}'>\overline{\mathtt{U}}'\to\mathtt{U}'_0$ *and* $\Delta,\Delta_2\vdash\overline{\mathtt{V}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{P}'$*, and* $\Delta,\Delta_2\vdash\overline{\mathtt{W}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}'$*, and* $[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0\Downarrow_{\Delta_2}\mathtt{V}'_0$*, and* $\Delta\vdash\mathtt{V}'_0<:\mathtt{V}_0$.

Proof. Prove by induction on the derivation of $\Delta\vdash\mathtt{S}<:\mathtt{T}$.

Case S-REFL: Trivial.

Case S-TRANS:

Let U be such that $\Delta\vdash\mathtt{S}<:\mathtt{U}$, $\Delta\vdash\mathtt{U}<:\mathtt{T}$.

By induction:

$\Delta\vdash bound_\Delta(\mathtt{U})\Uparrow^{\Delta_u}\mathtt{D}_u<\overline{\mathtt{S}}_u>$ $\quad mtype((\Delta,\Delta_u),\ \mathtt{m},\ \mathtt{D}_u<\overline{\mathtt{S}}_u>)=<\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}'_u>\overline{\mathtt{U}}'_u\to\mathtt{U}'_{u_0}$

$\Delta,\Delta_u\vdash\overline{\mathtt{V}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}'_u$ $\quad\Delta,\Delta_u\vdash\overline{\mathtt{W}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}'_u$ $\quad[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0\Downarrow_{\Delta_u}\mathtt{V}'_{u_0}$ $\quad\Delta\vdash\mathtt{V}'_{u_0}<:\mathtt{V}_0$

Also by induction hypothesis:

$\Delta\vdash bound_\Delta(\mathtt{S})\Uparrow^{\Delta_s}\mathtt{D}_s<\overline{\mathtt{S}}_s>$ $\quad mtype((\Delta,\Delta_s),\ \mathtt{m},\ \mathtt{D}_s<\overline{\mathtt{S}}_s>)=<\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}'_s>\overline{\mathtt{U}}'_s\to\mathtt{U}'_{s_0}$

$\Delta,\Delta_s\vdash\overline{\mathtt{V}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}'_s$ $\quad\Delta,\Delta_s\vdash\overline{\mathtt{W}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}'_s$ $\quad[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_{u_0}\Downarrow_{\Delta_s}\mathtt{V}'_{s_0}$ $\quad\Delta\vdash\mathtt{V}'_{s_0}<:\mathtt{V}_{u_0}$

By S-TRANS: $\Delta\vdash\mathtt{V}'_{s_0}<:\mathtt{V}_0$

Let $\Delta_2=\Delta_s$, $\mathtt{D}<\overline{\mathtt{S}}>=\mathtt{D}_s<\overline{\mathtt{S}}_s>$, $\overline{\mathtt{P}}'=\overline{\mathtt{P}}'_s$, $\overline{\mathtt{U}}'=\overline{\mathtt{U}}'_s$, $\mathtt{U}'_0=\mathtt{U}'_{s_0}$.

Case S-UBOUND: easy, since $bound_\Delta(\mathtt{S})=bound_\Delta(\mathtt{T})$.

Case S-LBOUND: precondition of the lemma that Δ has nonvariable bounds is not satisfied.

Case S-CLASS: Let $\mathtt{S}=\mathtt{D}<\overline{\mathtt{W}}>$

$CT(\mathtt{D})=\mathtt{class}\ \mathtt{D}<\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}>\ <\overline{\mathtt{X}}\lhd\overline{\mathtt{R}}>?\lhd\mathtt{C}<\overline{\mathtt{S}}'>\ \{\ldots\}$

$\Delta\vdash\mathtt{D}<\overline{\mathtt{W}}>\Uparrow^{\Delta_2}\mathtt{D}<\overline{\mathtt{W}}'>$ $\quad\Delta,\Delta_2\vdash\overline{\mathtt{W}}'<:[\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\overline{\mathtt{R}}$ $\quad[\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\mathtt{C}<\overline{\mathtt{S}}'>\Downarrow_{\Delta_2}\mathtt{T}$

There are two rules for the value of $mtype((\Delta,\Delta_2),\ \mathtt{m},\ \mathtt{D}<\overline{\mathtt{W}}'>)$. We analyze them case by case:

Subcase MT-CLASS:

$mtype((\Delta,\Delta_2),\ \mathtt{m},\ \mathtt{D}\langle\overline{\mathtt{W}}'\rangle)=[\overline{\mathtt{W}}'/\overline{\mathtt{X}}](\langle\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}''\rangle\overline{\mathtt{U}}''\to\mathtt{U}_0'')$

$\overline{\mathtt{P}}'=[\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\overline{\mathtt{P}}''\quad\overline{\mathtt{U}}'=[\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\overline{\mathtt{U}}''\quad\mathtt{U}_0'=[\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\mathtt{U}_0''$

$mtype((\Delta,\Delta_1),\ \mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\langle\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}''\rangle\overline{\mathtt{U}}''\to\mathtt{U}_0'')$

$\overline{\mathtt{P}}=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{P}}''\quad\overline{\mathtt{U}}=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}}''\quad\mathtt{U}_0=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0''$

By Lemma 13(2), $\Delta,\Delta_2\vdash\overline{\mathtt{V}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\overline{\mathtt{P}}''\quad\Delta,\Delta_2\vdash\overline{\mathtt{W}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\overline{\mathtt{U}}$

By substitution, $\Delta,\Delta_2\vdash\overline{\mathtt{V}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}'\quad\Delta,\Delta_2\vdash\overline{\mathtt{W}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}'$

By Lemma 13(1), $[\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\mathtt{U}_0''\Downarrow_{\Delta_2}\mathtt{V}_0'$

It follows from substitution that: $[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0'\Downarrow_{\Delta_2}\mathtt{V}_0'$

By Lemma 14, $\Delta\vdash\mathtt{V}_0'<:\mathtt{V}_0$

Subcase MT-SUPER:

The proof is obvious for the cases where $mtype(\Delta,\ \mathtt{m},\ \mathtt{D}\langle\overline{\mathtt{W}}'\rangle)=mtype(\Delta,\ \mathtt{m},\ [\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\mathtt{C}\langle\overline{\mathtt{S}}'\rangle)$.

Case S-VAR:

We analyze the four possible values of $\mathtt{w}_j$ separately. Without loss of generality, we assume the rest of $\overline{\mathtt{w}}$ remain invariant, e.g. $\mathtt{w}_i=\mathtt{o}$, $j\neq i$.

Subcase $\mathtt{w}_j=\mathtt{o}$: $\mathtt{S}_j=\mathtt{T}_j$. Thus, $\mathtt{S}=\mathtt{T}$.

Subcase $\mathtt{w}_j=+$: $\mathtt{S}_j<:\mathtt{T}_j$

$\Delta\vdash\mathtt{C}\langle\overline{\mathtt{T}}_1,+\mathtt{T}_j,\overline{\mathtt{T}}_2\rangle\Uparrow^{\Delta_1}\mathtt{C}\langle\overline{\mathtt{T}}_1,\mathtt{X}',\overline{\mathtt{T}}_2\rangle\quad\Delta_1(\mathtt{X}')=(+,\mathtt{T}_j)$

$mtype((\Delta,\Delta_1),\ \mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}_1,\mathtt{X}',\overline{\mathtt{T}}_2\rangle)=[\mathtt{X}'/\mathtt{T}_j](\langle\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}''\rangle\overline{\mathtt{U}}''\to\mathtt{U}_0'')$

$\overline{\mathtt{P}}=[\mathtt{X}'/\mathtt{T}_j]\overline{\mathtt{P}}''\quad\overline{\mathtt{U}}=[\mathtt{X}'/\mathtt{T}_j]\overline{\mathtt{U}}''\qquad\mathtt{U}_0=[\mathtt{X}'/\mathtt{T}_j]\mathtt{U}_0''$

$\mathtt{v}_j$ could be either + or o.

If $\mathtt{v}_j=+$,

$\Delta\vdash\mathtt{C}\langle\overline{\mathtt{S}}_1,+\mathtt{S}_j,\overline{\mathtt{S}}_2\rangle\Uparrow^{\Delta_2}\mathtt{C}\langle\overline{\mathtt{S}}_1,,\mathtt{X}',\overline{\mathtt{S}}_2\rangle\quad\Delta_2(\mathtt{X}')=(+,\mathtt{S}_j)$

$mtype((\Delta,\Delta_2),\ \mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{S}}_1,\mathtt{X}',\overline{\mathtt{S}}_2\rangle)=[\mathtt{X}'/\mathtt{S}_j](\langle\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}''\rangle\overline{\mathtt{U}}''\to\mathtt{U}_0'')$

$\overline{\mathtt{P}}'=[\mathtt{X}'/\mathtt{S}_j]\overline{\mathtt{P}}''\quad\overline{\mathtt{U}}'=[\mathtt{X}'/\mathtt{S}_j]\overline{\mathtt{U}}''\qquad\mathtt{U}_0'=[\mathtt{X}'/\mathtt{S}_j]\mathtt{U}_0''$

By Lemma 13(2), $\Delta,\Delta_2\vdash\overline{\mathtt{V}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}'\quad\Delta,\Delta_2\vdash\overline{\mathtt{W}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}'$

By Lemma 14(2), $[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0\Downarrow_{\Delta_2}\mathtt{V}_0'\quad\Delta\vdash\mathtt{V}_0'<:\mathtt{V}_0$

If $\mathtt{v}_j{=}\mathtt{o}$,

$\Delta{\vdash}\mathtt{C}{<}\overline{\mathtt{S}}_1,\mathtt{S}_j,\overline{\mathtt{S}}_2{>}{\Uparrow}^{\Delta_2}\mathtt{C}{<}\overline{\mathtt{S}}_1,\mathtt{S}_j,\overline{\mathtt{S}}_2{>}\quad\Delta_2{=}\emptyset$

$mtype((\Delta,\Delta_2),\ \mathtt{m},\ \mathtt{C}{<}\overline{\mathtt{S}}_1,\mathtt{S}_j,\overline{\mathtt{S}}_2{>}){=}{<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}'{>}\overline{\mathtt{U}}'{\to}\mathtt{U}'_0$

By Lemma 7, $\Delta,\Delta_2{\vdash}\overline{\mathtt{V}}{<:}[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}'\quad\Delta,\Delta_2{\vdash}\overline{\mathtt{W}}{<:}[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}'$

By Lemma 14(1), $[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0{\Downarrow}_{\Delta_2}\mathtt{V}'_0$, $\Delta{\vdash}\mathtt{V}'_0{<:}\mathtt{V}_0$

Subcase $\mathtt{w}_j{=}$- or *: Similar to above.

□

Lemma 4 (Term Substitution Preserves Typing)**.** *For any* Δ *that has non-variable bounds, if* $\Delta;\Gamma,\overline{\mathtt{x}}{:}\overline{\mathtt{T}}{\vdash}\mathtt{e}{\in}\mathtt{T}_0$ *and* $\Delta;\Gamma{\vdash}\overline{\mathtt{d}}{\in}\overline{\mathtt{S}}$ *where* $\Delta{\vdash}\overline{\mathtt{S}}{<:}\overline{\mathtt{T}}$*, then* $\Delta;\Gamma{\vdash}[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}{\in}\mathtt{S}_0$ *for some* $\mathtt{S}_0$ *such that* $\Delta{\vdash}\mathtt{S}_0{<:}\mathtt{T}_0$.

Proof. Prove by induction on the derivaion of $\Delta;\Gamma,\overline{\mathtt{x}}{:}\overline{\mathtt{T}}{\vdash}\mathtt{e}{\in}\mathtt{T}_0$.

Case T-VAR:

If $\mathtt{x}{\notin}\overline{\mathtt{x}}$, then the conclusion is trivially true.

If $\mathtt{x}{\in}\overline{\mathtt{x}}$, let $\mathtt{x}{=}\mathtt{x}_i$, $\Delta;\Gamma,\overline{\mathtt{x}}{:}\overline{\mathtt{T}}{\vdash}\mathtt{x}{\in}\mathtt{T}_i$.

$\Delta;\Gamma{\vdash}[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{x}{\in}\mathtt{S}_i\quad\Delta;\Gamma{\vdash}\mathtt{S}_i{<:}\mathtt{T}_i$

Case T-FIELD: $\mathtt{e}{=}\mathtt{e}_0.\mathtt{f}_i$

$\Delta;\Gamma{\vdash}\mathtt{e}_0{\in}\mathtt{T}_0\quad\Delta{\vdash}\mathit{bound}_\Delta(\mathtt{T}_0){\Uparrow}^{\Delta'}\mathtt{C}{<}\overline{\mathtt{U}}{>}\quad\mathit{fields}((\Delta,\Delta'),\ \mathtt{C}{<}\overline{\mathtt{U}}{>}){=}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\quad\mathtt{S}_i{\Downarrow}_{\Delta'}\mathtt{T}$

By induction hypothesis, there exists an $\mathtt{S}_0$ such that $\Delta;\Gamma{\vdash}[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0{\in}\mathtt{S}_0$, $\Delta{\vdash}\mathtt{S}_0{<:}\mathtt{T}_0$.

By Lemma 2,

$\Delta{\vdash}\mathit{bound}_\Delta(\mathtt{S}_0){\Uparrow}^{\Delta''}\mathtt{D}{<}\overline{\mathtt{V}}{>}\quad\mathit{fields}((\Delta,\Delta''),\ \mathtt{D}{<}\overline{\mathtt{V}}{>}){=}\ldots\overline{\mathtt{W}}\ \overline{\mathtt{f}}\quad\mathtt{W}_i{\Downarrow}_{\Delta''}\mathtt{W}'_i\quad\Delta{\vdash}\mathtt{W}'_i{<:}\mathtt{T}$

By T-FIELD, $\Delta;\Gamma{\vdash}[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0.\mathtt{f}_i{\in}\mathtt{W}'_i$.

Case T-INVK: $\mathtt{e}{=}\mathtt{e}_0.{<}\overline{\mathtt{V}}{>}\mathtt{m}(\overline{\mathtt{e}})$

$\Delta;\Gamma\vdash \mathtt{e}_0\in \mathtt{T}_0 \qquad \Delta\vdash bound_\Delta(\mathtt{T}_0)\Uparrow^{\Delta'}\mathtt{C}<\overline{\mathtt{T}}>$

$mtype((\Delta,\Delta'),\ \mathtt{m},\ \mathtt{C}<\overline{\mathtt{T}}>)=<\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}>\overline{\mathtt{U}}\to\mathtt{U}_0$

$\overline{\mathtt{Y}}\cap dom(\Delta')=\emptyset \quad \Delta\vdash\overline{\mathtt{V}}$ ok

$\Delta,\Delta'\vdash\overline{\mathtt{V}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \quad \Delta,\Delta'\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}}$

$\Delta,\Delta'\vdash\overline{\mathtt{S}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}} \quad [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0\Downarrow_{\Delta'}\mathtt{T}$

By the induction hypothesis, there exists a $\mathtt{T}'_0$, $\overline{\mathtt{S}}'$ such that

$\Delta;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0\in\mathtt{T}'_0 \quad \Delta\vdash\mathtt{T}'_0<:\mathtt{T}_0 \quad \Delta;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}}\in\overline{\mathtt{S}}' \quad \Delta\vdash\overline{\mathtt{S}}'<:\overline{\mathtt{S}}$

By Lemma 3, there exists $\mathtt{D}<\overline{\mathtt{T}}'>$ such that,

$\Delta\vdash bound_\Delta(\mathtt{T}'_0)\Uparrow^{\Delta_2}\mathtt{D}<\overline{\mathtt{T}}'> \quad mtype((\Delta,\Delta_2),\ \mathtt{m},\ \mathtt{D}<\overline{\mathtt{T}}'>)=<\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}'>\overline{\mathtt{U}}'\to\mathtt{U}'_0$

$\Delta,\Delta_2\vdash\overline{\mathtt{V}}<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}' \qquad \Delta,\Delta_2\vdash\overline{\mathtt{S}}'<:[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}' \quad [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}'_0\Downarrow_{\Delta_2}\mathtt{T}' \quad \Delta\vdash\mathtt{T}'<:\mathtt{T}$

By T-METHOD, $\Delta;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}\in\mathtt{T}'$.

Case T-NEW: $\mathtt{e}=\mathtt{new}\ \mathtt{C}<\overline{\mathtt{T}}>(\overline{\mathtt{e}}),\ \mathtt{T}=\mathtt{C}<\overline{\mathtt{T}}>$.

$\Delta\vdash\mathtt{C}<\overline{\mathtt{T}}>$ ok $\quad fields(\Delta,\ \mathtt{C}<\overline{\mathtt{T}}>)=\overline{\mathtt{U}}\ \overline{\mathtt{f}} \quad \Delta;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}} \quad \Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{U}}$

By induction hypothesis, there exists $\overline{\mathtt{S}}'$such that

$\Delta;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}}\in\overline{\mathtt{S}}' \quad \Delta\vdash\overline{\mathtt{S}}'<:\overline{\mathtt{S}}$

By S-TRANS and T-NEW, $\Delta;\Gamma\vdash\mathtt{new}\ \mathtt{C}<\overline{\mathtt{T}}>([\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}})\in\mathtt{C}<\overline{\mathtt{T}}>$.

Case T-CAST and T-SCAST: Easy using induction hypothesis.

□

Lemma 5 (Weakening)**.** *Suppose* $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\overline{\mathtt{N}}$ *ok and* $\Delta\vdash\mathtt{U}$ *ok.*

1) If $\Delta\vdash\mathtt{S}<:\mathtt{T}$*, then* $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\mathtt{S}<:\mathtt{T}$.

2) If $\Delta\vdash\mathtt{S}$ *ok, then* $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\mathtt{S}$ *ok.*

3) If $\Delta;\Gamma\vdash\mathtt{e}\in\mathtt{T}$*, then* $\Delta;\Gamma,\mathtt{x}:\mathtt{U}\vdash\mathtt{e}\in\mathtt{T}$ *and* $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}};\Gamma\vdash\mathtt{e}\in\mathtt{T}$.

4) If $fields(\Delta,\ \mathtt{T})=\overline{\mathtt{U}}\ \overline{\mathtt{f}}$*, then* $fields((\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}),\ \mathtt{T})=\overline{\mathtt{U}}\ \overline{\mathtt{f}}$.

5) If $mtype(\Delta,\ \mathtt{m},\ \mathtt{C}<\overline{\mathtt{T}}>)=<\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}>\overline{\mathtt{U}}\to\mathtt{U}_0$*, then* $mtype((\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}),\ \mathtt{m},\ \mathtt{C}<\overline{\mathtt{T}}>)=<\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}>\overline{\mathtt{U}}\to\mathtt{U}_0$.

Proof. 1)Prove by induction on the derivation of $\Delta\vdash\mathtt{S}<:\mathtt{T}$

Case S-REFL, S-TRANS, S-UBOUND, S-LBOUND: Trivial.

Case S-CLASS: $\mathtt{S}{=}\mathtt{C}\langle\overline{\mathtt{T}}\rangle$

$\mathtt{class\ C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle\ \langle\overline{\mathtt{X}}\lhd\overline{\mathtt{R}}\rangle?\lhd\mathtt{D}\langle\overline{\mathtt{S}}\rangle\ \{\ \ldots\ \ \}$

$\Delta\vdash\mathtt{C}\langle\overline{\mathtt{T}}\rangle\Uparrow^{\Delta'}\mathtt{C}\langle\overline{\mathtt{U}}\rangle\quad\Delta,\Delta'\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{R}}\quad[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{D}\langle\overline{\mathtt{S}}\rangle\Downarrow_{\Delta'}\mathtt{T}$

By induction hypothesis, $\Delta,\Delta',\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{R}}$

By Lemma 15,

$\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\mathtt{C}\langle\overline{\mathtt{T}}\rangle\Uparrow^{\Delta'}\mathtt{C}\langle\overline{\mathtt{U}}\rangle\quad[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{D}\langle\overline{\mathtt{S}}\rangle\Downarrow_{\Delta'}\mathtt{T}$

Thus, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\mathtt{C}\langle\overline{\mathtt{T}}\rangle<:\mathtt{T}$.

Case S-VAR: $\mathtt{S}{=}\mathtt{C}\langle\overline{\mathtt{S}}\rangle$, $\mathtt{T}{=}\mathtt{C}\langle\overline{\mathtt{T}}\rangle$.

By induction hypothesis, all conditions for S-VAR still hold when Δ becomes $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}$. Thus, we have $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\mathtt{C}\langle\overline{\mathtt{S}}\rangle<:\mathtt{C}\langle\overline{\mathtt{T}}\rangle$.

2) Prove by induction on the derivation of well-formed type rules.

Case WF-OBJECT: Trivial.

Case WF-VAR: $\mathtt{S}\in dom(\Delta)$ implies that $\mathtt{S}\in dom(\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}})$. Done.

Case WF-CLASS: $\mathtt{S}{=}\mathtt{C}\langle\overline{\mathtt{T}}\rangle$

$CT(\mathtt{C}){=}\mathtt{class\ C}\langle\overline{\mathtt{Y}}\lhd\overline{\mathtt{Q}}\rangle\ \langle\overline{\mathtt{Y}}\lhd\overline{\mathtt{R}}\rangle?\lhd\mathtt{D}\langle\overline{\mathtt{S}}\rangle\ \{\ldots\}\quad\Delta\vdash\overline{\mathtt{T}}\ \mathrm{ok}\quad\Delta\vdash\overline{\mathtt{T}}<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}}$

By induction hypothesis, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\overline{\mathtt{T}}\ \mathrm{ok}\quad\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\overline{\mathtt{T}}<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}}$

Thus, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\mathtt{C}\langle\overline{\mathtt{T}}\rangle\ ok$.

3) Prove by induction on the derivation of $\Delta;\Gamma\vdash\mathtt{e}\in\mathtt{T}$ rules.

Case T-VAR: Trivial.

Case T-FIELD: $\mathtt{e}{=}\mathtt{e}_0\mathtt{.f}_i$, $\Delta;\Gamma\vdash\mathtt{e}_0\mathtt{.f}_i\in\mathtt{T}$.

$\Delta;\Gamma\vdash\mathtt{e}_0\in\mathtt{T}_0\quad\Delta\vdash bound_\Delta(\mathtt{T}_0)\Uparrow^{\Delta'}\mathtt{C}\langle\overline{\mathtt{U}}\rangle\quad fields((\Delta,\Delta'),\ \mathtt{C}\langle\overline{\mathtt{U}}\rangle){=}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\quad\mathtt{S}_i\Downarrow_{\Delta'}\mathtt{T}$

By Lemma 15, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash bound_\Delta(\mathtt{T}_0)\Uparrow^{\Delta'}\mathtt{C}\langle\overline{\mathtt{U}}\rangle$

By induction hypothesis, $\Delta,\overline{X}<:\overline{N}\vdash e_0\in T_0$, $\Delta;\Gamma,x:U\vdash e_0\in T_0$.

By definition of *bound*, $bound_\Delta(C<\overline{U}>)=bound_{\Delta,\overline{X}<:\overline{N}}(C<\overline{U}>)$.

The conclusion follows from the above conditions and T-FIELD.

Case T-INVK: $e=e_0.<\overline{V}>m(\overline{e})\in T$.

$\Delta;\Gamma\vdash e_0\in T_0$ $\quad \Delta\vdash bound_\Delta(T_0)\Uparrow^{\Delta'}C<\overline{T}>$ $\quad mtype((\Delta,\Delta'),\ m,\ C<\overline{T}>)=<\overline{Y}\lhd\overline{P}>\overline{U}\to U_0$

$\{\overline{Y}\}\cap dom(\Delta')=\emptyset$ $\quad \Delta\vdash\overline{V}\ ok$ $\quad \Delta,\Delta'\vdash\overline{V}<:[\overline{V}/\overline{Y}]\overline{P}$ $\quad \Delta;\Gamma\vdash\overline{e}\in\overline{S}$

$\Delta,\Delta'\vdash\overline{S}<:[\overline{V}/\overline{Y}]\overline{U}$ $\quad [\overline{V}/\overline{Y}]U_0\Downarrow_{\Delta'}T$

By induction hypothesis,

$mtype((\Delta,\Delta',\overline{X}<:\overline{N}),\ m,\ C<\overline{T}>)=<\overline{Y}\lhd\overline{P}>\overline{U}\to U_0$

$\Delta,\overline{X}<:\overline{N};\Gamma\vdash e_0\in T_0$ $\quad \Delta,\overline{X}<:\overline{N}\vdash\overline{V}\ ok$ $\quad \Delta,\overline{X}<:\overline{N},\Delta'\vdash\overline{V}<:[\overline{V}/\overline{Y}]\overline{P}$ $\quad \Delta,\overline{X}<:\overline{N};\Gamma\vdash\overline{e}\in\overline{S}$

By Lemma 15, $\Delta,\overline{X}<:\overline{N}\vdash bound_{\Delta,\overline{X}<:\overline{N}}(T_0)\Uparrow^{\Delta'}C<\overline{T}>$

Conclusion follows from T-INVK.

Proof for $\Delta;\Gamma,x:U\vdash e_0.<\overline{V}>m(\overline{e})\in T$ follows the same structure.

Case T-NEW: $e=$new $C<\overline{T}>(\overline{e})\in C<\overline{T}>$

$\Delta\vdash C<\overline{T}>\ ok$ $\quad fields(\Delta,\ C<\overline{T}>)=\overline{U}\ \overline{f}$ $\quad \Delta;\Gamma\vdash\overline{e}\in\overline{S}$ $\quad \Delta\vdash\overline{S}<:\overline{U}$

By induction hypothesis,

$\Delta,\overline{X}<:\overline{N}\vdash C<\overline{T}>\ ok$ $\quad fields((\Delta,\overline{X}<:\overline{N}),\ C<\overline{T}>)=\overline{U}\ \overline{f}$ $\quad \Delta,\overline{X}<:\overline{N};\Gamma\vdash\overline{e}\in\overline{S}$ $\quad \Delta,\overline{X}<:\overline{N}\vdash\overline{S}<:\overline{U}$

It follows from T-NEW that $\Delta,\overline{X}<:\overline{N};\Gamma\vdash$new $C<\overline{T}>(\overline{e})\in C<\overline{T}>$.

Proof for $\Delta;\Gamma,x:U\vdash$new $C<\overline{T}>(\overline{e})\in C<\overline{T}>$ follows the same structure.

Case T-CAST, T-SCAST:

By straight-forward induction hypothesis.

4) By case analysis on MT-CLASS and MT-SUPER, and straight-forward induction hypothesis.

5) BY case analysis on F-OBJECT, F-CLASS, and F-CLASS-S, and straight forward induction hypothesis.

□

Lemma 6 (Narrowing)**.** *1) If* Δ,$\mathtt{X}$<:$\mathtt{S}\vdash\mathtt{T}_1$<:$\mathtt{T}_2$ *and* $\Delta\vdash\mathtt{U}$<:$\mathtt{S}$*, then* Δ,$\mathtt{X}$<:$\mathtt{U}\vdash\mathtt{T}_1$<:$\mathtt{T}_2$*.*

2) If Δ,$\mathtt{S}$<:$\mathtt{X}\vdash\mathtt{T}_1$<:$\mathtt{T}_2$ *and* $\Delta\vdash\mathtt{S}$<:$\mathtt{U}$*, then* Δ,$\mathtt{U}$<:$\mathtt{X}\vdash\mathtt{T}_1$<:$\mathtt{T}_2$*.*

3) If Δ,$\mathtt{X}$:(*,$\mathtt{S}$)$\vdash\mathtt{T}_1$<:$\mathtt{T}_2$*, then* Δ,($\mathtt{X}$:($\mathtt{v}$,$\mathtt{U}$))$\vdash\mathtt{T}_1$<:$\mathtt{T}_2$*.*

Proof. 1) By induction on the derivation of Δ,$\mathtt{X}$<:$\mathtt{S}\vdash\mathtt{T}_1$<:$\mathtt{T}_2$.

Case S-REFL, S-UBOUND, S-LBOUND: Trivial.

Case S-TRANS, S-VAR: By induction hypothesis.

Case S-CLASS: $\Delta\vdash\mathtt{C}\langle\overline{\mathtt{T}}\rangle<:\mathtt{T}$

$CT(\mathtt{C})$=`class C`$\langle\overline{\mathtt{Y}}\lhd\overline{\mathtt{N}}\rangle$ `<`$\overline{\mathtt{Y}}\lhd\overline{\mathtt{R}}$`>?D<`$\overline{\mathtt{S}}$`> { ... }`

Δ,$\mathtt{X}$<:$\mathtt{S}\vdash\mathtt{C}\langle\overline{\mathtt{T}}\rangle\Uparrow^{\Delta'}\mathtt{C}\langle\overline{\mathtt{U}}\rangle$ Δ,$\mathtt{X}$<:$\mathtt{S}$,$\Delta'\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{Y}}]\overline{\mathtt{R}}$ $([\overline{\mathtt{U}}/\overline{\mathtt{Y}}]\mathtt{D}\langle\overline{\mathtt{S}}\rangle)\Downarrow_{\Delta'}\mathtt{T}$

It is clear from the definition of $\Uparrow$ that Δ,$\mathtt{X}$<:$\mathtt{U}\vdash\mathtt{C}\langle\overline{\mathtt{T}}\rangle\Uparrow^{\Delta'}\mathtt{C}\langle\overline{\mathtt{U}}\rangle$.

By induction hypothesis, Δ,$\mathtt{X}$<:$\mathtt{U}$,$\Delta'\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{R}}$

Conclusion follows from S-CLASS that Δ,$\mathtt{X}$<:$\mathtt{U}\vdash\mathtt{C}\langle\overline{\mathtt{T}}\rangle<:\mathtt{T}$

2) can be similarly proven as above.

3) can be proven by treating Δ,$\mathtt{X}$:(*,$\mathtt{S}$) as Δ,$\mathtt{X}$<:`Object`.

Then Δ,$\mathtt{X}$:(+,$\mathtt{U}$) can be treated as Δ,$\mathtt{X}$<:$\mathtt{U}$, where $\Delta\vdash\mathtt{U}$<:`Object`.

Similarly, Δ,$\mathtt{X}$:(-,$\mathtt{U}$) can be treated as Δ,$\mathtt{U}$<:$\mathtt{X}$.

Then the proof follows from 1) and 2).

□

Lemma 7 (Type Substitution Preserves Subtyping)**.** *If* Δ_1,$\overline{\mathtt{X}}$<:$\overline{\mathtt{N}}$,$\Delta_2\vdash\mathtt{S}$<:$\mathtt{T}$ *and* $\Delta_1\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$*, with* $\Delta_1\vdash\overline{\mathtt{U}}$ *ok and none of* $\overline{\mathtt{X}}$ *appearing in* Δ_1*, then* Δ_1,$[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$.

Proof. Prove by induction on the derivation of Δ_1,$\overline{\mathtt{X}}$<:$\overline{\mathtt{N}}$,$\Delta_2\vdash\mathtt{S}$<:$\mathtt{T}$

Case S-REFL: Trivial.

Case S-TRANS:

Let $\mathtt{U}$ be a type such that: Δ_1,$\overline{\mathtt{X}}$<:$\overline{\mathtt{N}}$,Δ_2 $\mathtt{S}$<:$\mathtt{U}$ Δ_1,$\overline{\mathtt{X}}$<:$\overline{\mathtt{N}}$,Δ_2 $\mathtt{U}$<:$\mathtt{T}$

Then, Δ_1,$[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{U}$, and Δ_1,$[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{U}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$.

By S-TRANS, $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S} <: [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$.

Case S-UBOUND: $\mathtt{S}=\mathtt{X}$, $(\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2)(\mathtt{X})=(+,\mathtt{T})$.

If $\mathtt{S} \in dom(\Delta_1)$, then the result is obvious, since none of $\overline{\mathtt{X}}$ appears in Δ_1.

If $\mathtt{S} \in dom(\Delta_2)$, result is also obvious since substitution is applied to both Δ_2, S, and T.

If $\mathtt{S}=\mathtt{X}_i$, then $\Delta_1 \vdash \mathtt{U}_i <: [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{N}_i$, and $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}=\mathtt{U}_i$. Then by Lemma 5, $\Delta_1[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathtt{S} <: [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$.

Case S-LBOUND: $\mathtt{T}=\mathtt{X}$, $(\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2)(\mathtt{X})=(\mathtt{()\text{-}},\ \mathtt{S})$.

Proof follows the same structure as the proof for case S-UBOUND.

Case S-CLASS: $\mathtt{S}=\mathtt{C}\langle\overline{\mathtt{S}}\rangle$

Let $\Delta=\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2$.

$CT(\mathtt{C})=\mathtt{class\ C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle\ \langle\overline{\mathtt{X}}\lhd\overline{\mathtt{R}}\rangle\lhd\mathtt{D}\langle\overline{\mathtt{S}}\rangle\ \{...\}$

$\Delta \vdash \mathtt{C}\langle\overline{\mathtt{T}}\rangle \Uparrow^{\Delta'} \mathtt{C}\langle\overline{\mathtt{V}}\rangle \quad \Delta \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{X}}]\overline{\mathtt{R}} \quad ([\overline{\mathtt{V}}/\overline{\mathtt{X}}]\mathtt{D}\langle\overline{\mathtt{S}}\rangle) \Downarrow_{\Delta'} \mathtt{T}$

By Lemma 8(1) and (2), and the induction hypothesis, the result is obvious.

Case S-VAR: Easily follows from induction hypothesis.

□

Lemma 8. *1) If* $\Delta_1 \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ *and* $\Delta_1 \vdash \overline{\mathtt{S}}$ *ok,* $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2 \vdash \mathtt{C}\langle\overline{\mathtt{T}}\rangle \Uparrow^{\Delta'} \mathtt{C}\langle\overline{\mathtt{U}}\rangle$ *with none of* $\overline{\mathtt{X}}$ *appearing in* Δ_1 *and none of the type variables in* $dom(\Delta')$ *appearing in* $\overline{\mathtt{S}}$*, then*
$\Delta_1,[\overline{\mathtt{S}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{C}\langle\overline{\mathtt{T}}\rangle \Uparrow^{[\overline{\mathtt{S}}/\overline{\mathtt{X}}]\Delta'} [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{C}\langle\overline{\mathtt{U}}\rangle$.
2) If $\mathtt{S} \Downarrow_\Delta \mathtt{T}$ *where* $dom(\Delta)$ *and* $\overline{\mathtt{X}}$ *are distinct, then* $[\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{S} \Downarrow_{[\overline{\mathtt{S}}/\overline{\mathtt{X}}]\Delta} [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{T}$.
3) $[\overline{\mathtt{S}}/\overline{\mathtt{X}}]fields(\Delta,\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = fields(\Delta,\ [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{C}\langle\overline{\mathtt{T}}\rangle)$
4) $[\overline{\mathtt{S}}/\overline{\mathtt{X}}]mtype(\Delta,\ \mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = mtype(\Delta,\ \mathtt{m},\ [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{C}\langle\overline{\mathtt{T}}\rangle)$

Proof. 1) Prove by derivation of $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2 \vdash \mathtt{C}\langle\overline{\mathtt{T}}\rangle \Uparrow^{\Delta'} \mathtt{C}\langle\overline{\mathtt{U}}\rangle$.

Case O-REFL: Trivial

Case O-TRANS: Let there be T, where

$\Delta_2,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2 \vdash \mathtt{C}\langle\overline{\mathtt{T}}\rangle \Uparrow^{\Delta_3} \mathtt{T} \quad \Delta_2,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2,\Delta_3 \vdash \mathtt{T} \Uparrow^{\Delta_4} \mathtt{C}\langle\overline{\mathtt{U}}\rangle$

By induction hypothsis,

$\Delta_1,[\overline{S}/\overline{X}]\Delta_2 \vdash [\overline{S}/\overline{X}]C<\overline{T}> \Uparrow^{[\overline{S}/\overline{X}]\Delta_3} [\overline{S}/\overline{X}]T \quad \Delta_1,[\overline{S}/\overline{X}]\Delta_2,[\overline{S}/\overline{X}]\Delta_3 \vdash [\overline{S}/\overline{X}]T \Uparrow^{[\overline{S}/\overline{X}]\Delta_4} [\overline{S}/\overline{X}]C<\overline{U}>$

By S-TRANS, $\Delta_1,[\overline{S}/\overline{X}]\Delta_2 \vdash [\overline{S}/\overline{X}]C<\overline{T}> \Uparrow^{[\overline{S}/\overline{X}](\Delta_3,\Delta_4)} [\overline{S}/\overline{X}]C<\overline{U}>$

2) By induction on the derivation of $S \Downarrow_\Delta T$.

Case C-PROM: $\Delta(X)=(+,T)$, $X \Downarrow_\Delta T$

Since $\overline{X}$ do not appear in Δ, conclusion is trivial.

Case C-TVAR: Trivial.

Case C-CLASS: By induction hypothesis.

3) By derivation of $fields(\Delta,\ C<\overline{T}>)$

Case F-OBJECT: Trivial.

Case F-CLASS-S:

$CT(C)=\texttt{class}\ C<\overline{X}\lhd\overline{N}>\ <\overline{X}\lhd\overline{R}>?D<\overline{U}>\{\overline{G}\ \overline{f};\ \overline{M}\}$

$fields(\Delta,\ [\overline{T}/\overline{X}]D<\overline{U}>)=\overline{D}\ \overline{g} \quad \Delta \vdash \overline{T}<:[\overline{T}/\overline{X}]\overline{R}$

$[\overline{S}/\overline{X}]fields(\Delta,\ C<\overline{T}>)=[\overline{S}/\overline{X}]\overline{D}\ \overline{g},\ [\overline{S}/\overline{X}][\overline{T}/\overline{X}]\overline{G}\ \overline{f}$

By Lemma 7, $\Delta \vdash [\overline{S}/\overline{X}]\overline{T}<:[\overline{S}/\overline{X}][\overline{T}/\overline{X}]\overline{R}$

By induction hypothesis, $fields(\Delta,\ [\overline{S}/\overline{X}][\overline{T}/\overline{X}]D<\overline{U}>)=[\overline{S}/\overline{X}]\overline{D}\ \overline{g}$.

Conclusion follows from F-CLASS-S.

Case F-CLASS: By Lemma 7.

4) By derevation of $[\overline{S}/\overline{X}]mtype(\Delta,\ m,\ C<\overline{T}>)$. Both MT-CLASS and MT-SUPER can be shown through Lemma 7.

□

Lemma 9 (Type Substitution Preserves Type Well-formedness)**.** *If* $\Delta_1,\overline{X}<:\overline{N},\Delta_2 \vdash T$ *ok and* $\Delta_1 \vdash \overline{U}<:[\overline{U}/\overline{X}]\overline{N}$ *with* $\Delta_1 \vdash \overline{U}$ *ok, and none of* $\overline{X}$ *appearing in* Δ_1*, then* $\Delta_1,[\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]T$ *ok.*

Proof. We prove by induction on the derivation of $\mathtt{typeenv}_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2\vdash\mathtt{T}\ ok$.

Case WF-OBJECT: Trivial.

Case WF-VAR: $\mathtt{T}=\mathtt{X}$, $\mathtt{X}\in dom(\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2)$.

If $\mathtt{X}\notin\overline{\mathtt{X}}$, result is obvious.

If $\mathtt{X}=\mathtt{X}_i$, $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{X}=\mathtt{U}_i$. By assumptiong, $\Delta_1\vdash\mathtt{U}_i$ ok. By Lemma 5(2), $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash\mathtt{U}_i$ ok.

Case WF-CLASS: $\mathtt{T}=\mathtt{C}<\overline{\mathtt{T}}>$

$CT(\mathtt{C})=\mathtt{class\ C}<\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}>\ <\overline{\mathtt{Y}}\lhd\overline{\mathtt{R}}>\mathtt{?D}<\overline{\mathtt{S}}>\ \mathtt{\{...\}}$

$\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2\vdash\overline{\mathtt{T}}\ ok\quad\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2\vdash\overline{\mathtt{T}}<:[\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$

By induction hypothesis, $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}$ ok.

By Lemma 7, $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$. The result follows from WF-CLASS.

□

Lemma 10 (Type Substitution Preserves Typing)**.** *If both* Δ_1 *and* Δ_2 *have non-variable bounds and* $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2;\Gamma\vdash\mathtt{e}\in\mathtt{T}$ *and* $\Delta_1\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ *where* $\Delta_1\vdash\overline{\mathtt{U}}$ *ok and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1*, then* $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2;[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}\in\mathtt{S}$ *for some* S *such that* $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash\mathtt{S}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$*.*

Proof. Prove by induction on the derivation of $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2;\Gamma\vdash\mathtt{e}\in\mathtt{T}$.

Let $\Delta=\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2$.

Case T-VAR: Trivial.

Case T-FIELD: $\mathtt{e}=\mathtt{e}_0\mathtt{.f}_i$

$\Delta;\Gamma\vdash\mathtt{e}_0\in\mathtt{T}_0\quad\Delta\vdash bound_\Delta(\mathtt{T}_0)\Uparrow^{\Delta'}\mathtt{C}<\overline{\mathtt{T}}>\quad fields(\Delta,\ \mathtt{C}<\overline{\mathtt{T}}>)=\overline{\mathtt{S}}\ \overline{\mathtt{f}}\quad\mathtt{S}_i\Downarrow_{\Delta'}\mathtt{T}$

By induction hypothesis, there is a $\mathtt{S}_0$ such that

$\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2;[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma\vdash\mathtt{e}_0\in\mathtt{S}_0\quad\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash\mathtt{S}_0<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0$

By Lemma 11, $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash bound_{\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}(\mathtt{T}_0)<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}](bound_{\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2}(\mathtt{T}_0))$

By S-TRANS, $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash bound_{\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}(\mathtt{S}_0)<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}](bound_{\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2}(\mathtt{T}_0))$

By Lemma 2,

$\Delta_1,[\overline{U}/\overline{X}]\Delta_2 \vdash bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(S_0)\Uparrow^{\Delta''} D<\overline{S}'>$ $\quad fields((\Delta_1,[\overline{U}/\overline{X}]\Delta_2),\ D<\overline{S}'>)=\overline{S}''\ \overline{f}$

$S''_i \Downarrow_{\Delta''} T'$ $\quad \Delta_1,[\overline{U}/\overline{X}]\Delta_2 \vdash T'<:[\overline{U}/\overline{X}]T$

By T-FIELD, $\Delta_1,[\overline{U}/\overline{X}]\Delta_2;[\overline{U}/\overline{X}]\Gamma \vdash [\overline{U}/\overline{X}]e_0.f_i \in T'$.

Let $S=T'$

Case T-INVK: $e=e_0.<\overline{V}>m(\overline{e})\in T$.

$\Delta;\Gamma\vdash e_0\in T_0$ $\quad \Delta\vdash bound_\Delta(T_0)\Uparrow^{\Delta'} C<\overline{T}>$ $\quad mtype(\Delta,\ m,\ C<\overline{T}>)=<\overline{Y}\lhd\overline{P}>\overline{W}\to W_0$

$\overline{Y}\cap dom(\Delta')=\emptyset$ $\quad \Delta\vdash\overline{V}$ ok $\quad \Delta,\Delta'\vdash\overline{V}<:[\overline{V}/\overline{Y}]\overline{P}$

$\Delta;\Gamma\vdash\overline{e}\in\overline{S}$ $\quad \Delta,\Delta'\vdash\overline{S}<:[\overline{V}/\overline{Y}]\overline{U}$ $\quad [\overline{V}/\overline{Y}]W_0\Downarrow_{\Delta'}T$

By induction hypothesis, there is a S_0 such that

$\Delta_1,[\overline{U}/\overline{X}]\Delta_2;[\overline{U}/\overline{X}]\Gamma\vdash e_0\in S_0$ $\quad \Delta_1,[\overline{U}/\overline{X}]\Delta_2\vdash S_0<:[\overline{U}/\overline{X}]T_0$

Also by the induction hypothesis, there exists $\overline{S}'$such that

$\Delta_1,[\overline{U}/\overline{X}]\Delta_2;[\overline{U}/\overline{X}]\Gamma\vdash\overline{e}\in\overline{S}'$ $\quad \Delta_1,[\overline{U}/\overline{X}]\Delta_2\vdash\overline{S}'<:[\overline{U}/\overline{X}]\overline{S}$

By Lemma 11 and S-TRANS,

$\Delta_1,[\overline{U}/\overline{X}]\Delta_2\vdash bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(S_0)<:[\overline{U}/\overline{X}](bound_{\Delta_1,\overline{X}<:\overline{N},\Delta_2}(T_0))$

By Lemma 9, $\Delta_1,[\overline{U}/\overline{X}]\Delta_2\vdash[\overline{U}/\overline{X}]\overline{V}$ ok

By Lemma 7,

$\Delta_1,[\overline{U}/\overline{X}](\Delta_2,\Delta')\vdash[\overline{U}/\overline{X}]\overline{V}<:[\overline{U}/\overline{X}][\overline{V}/\overline{Y}]\overline{P}$ $\quad \Delta_1,[\overline{U}/\overline{X}](\Delta_2,\Delta')\vdash[\overline{U}/\overline{X}]\overline{S}<:[\overline{U}/\overline{X}][\overline{V}/\overline{Y}]\overline{W}$

By Lemma 8(2) and (4),

$mtype(\Delta,\ m,\ [\overline{U}/\overline{X}]C<\overline{T}>)=[\overline{U}/\overline{X}](<\overline{Y}\lhd\overline{P}>\overline{W}\to W_0)$ $\quad [\overline{U}/\overline{X}][\overline{V}/\overline{Y}]W_0\Downarrow_{[\overline{U}/\overline{X}]\Delta'}[\overline{U}/\overline{X}]T$

Because $\overline{X}$ does not appear in Δ_1, $[\overline{U}/\overline{X}][\overline{V}/\overline{Y}]T = [[\overline{U}/\overline{X}]\overline{V}/\overline{Y}]([\overline{U}/\overline{X}]T)$

By Lemma 3,

$\Delta_1,[\overline{U}/\overline{X}]\Delta_2\vdash bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(S_0)\Uparrow^{\Delta''} D<\overline{T}'>$

$mtype((\Delta_1,[\overline{U}/\overline{X}]\Delta_2),\ m,\ D<\overline{T}'>)=<\overline{Y}\lhd\overline{P}>\overline{W}'\to W'_0$

$\Delta_1,[\overline{U}/\overline{X}]\Delta_2,\Delta''\vdash[\overline{U}/\overline{X}]\overline{V}<:[[\overline{U}/\overline{X}]\overline{V}/\overline{Y}]\overline{W}'$ $\quad \Delta_1,[\overline{U}/\overline{X}]\Delta_2,\Delta''\vdash[\overline{U}/\overline{X}]\overline{S}<:[[\overline{U}/\overline{X}]\overline{V}/\overline{Y}]\overline{W}'$

$[[\overline{U}/\overline{X}]\overline{V}/\overline{Y}]W'_0\Downarrow_{\Delta''}T'$ $\quad \Delta_1,[\overline{U}/\overline{X}]\Delta_2\vdash T'<:[\overline{U}/\overline{X}]T$

By Lemma 5 and S-TRANS, $\Delta_1,[\overline{U}/\overline{X}]\Delta_2,\Delta''\vdash\overline{S}'<:[[\overline{U}/\overline{X}]\overline{V}/\overline{Y}]\overline{W}'$

By T-METHOD, $\Delta_1,[\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]e_0.\langle\overline{V}\rangle m(\overline{e}) \in T'$

Case T-NEW: $e = \texttt{new } C\langle\overline{T}\rangle(\overline{e})$, $T = C\langle\overline{T}\rangle$.

$\Delta \vdash C\langle\overline{T}\rangle\ ok \quad fields(\Delta,\ C\langle\overline{T}\rangle) = \overline{U}'\ \overline{f} \quad \Delta;\Gamma \vdash \overline{e} \in \overline{S} \quad \Delta \vdash \overline{S} <: \overline{U}'$

By Lemma 9, $\Delta_1,[\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]C\langle\overline{T}\rangle\ ok$.

By Lemma 8(3), $fields(\Delta,\ [\overline{U}/\overline{X}]C\langle\overline{T}\rangle) = [\overline{U}/\overline{X}]\overline{U}'\ \overline{f}$

By induction hypothesis, there is $\overline{S}'$,

$\Delta_1,[\overline{U}/\overline{X}]\Delta_2;[\overline{U}/\overline{X}]\Gamma \vdash [\overline{U}/\overline{X}]\overline{e} \in \overline{S}' \quad \Delta_1,[\overline{U}/\overline{X}]\Delta_2 \vdash \overline{S}' <: \overline{S}$

By Lemma 7 and S-TRANS, $\Delta_1,[\overline{U}/\overline{X}]\Delta_2 \vdash \overline{S}' <: [\overline{U}/\overline{X}]\overline{U}'$

By T-NEW, $\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash C\langle[\overline{U}/\overline{X}]\overline{T}\rangle([\overline{U}/\overline{X}]\overline{e}) \in [\overline{U}/\overline{X}]T$.

Case T-CAST, T-SCAST: Easy by induction hypothesis.

□

Lemma 11. *Suppose Δ has non-variable bounds and is of the form $\Delta_1,\overline{X}<:\overline{N},\Delta_2$. If $\Delta \vdash T\ ok$ and $\Delta_1 \vdash \overline{U} <: [\overline{U}/\overline{X}]\overline{N}$ with $\Delta_1 \vdash \overline{U}\ ok$ and none of the $\overline{X}$ appear in Δ_1, then,*
$\Delta_1,[\overline{U}/\overline{X}]\Delta_2 \vdash bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}([\overline{U}/\overline{X}]T) <: [\overline{U}/\overline{X}](bound_\Delta(T))$.

Proof. We prove for each case in the definition of $bound_\Delta$.

If T is a non-variable type, $bound_\Delta(T) = T$. The proof is trivial.

If T is a variable, $T = X$, $\Delta(X) = (+,S)$, then

$bound_\Delta(T) = bound_\Delta(S) \quad \Delta \vdash T <: S$

The proof is trivial if $X \notin \overline{X}$.

If $X = X_i$,

$bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}([\overline{U}/\overline{X}]X_i) = bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(U_i)$

$\Delta_1,[\overline{U}/\overline{X}]\Delta_2 \vdash bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(N_i) <: [\overline{U}/\overline{X}](bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(S))$

By assumption and Lemma 5, $\Delta_1,[\overline{U}/\overline{X}]\Delta_2 \vdash U_i <: N_i$

And S-TRANS finishes the case. □

Lemma 12 (Close Yields A Supertype Without Local Type Variables)**.** *If $\Delta,\Delta' \vdash S\ ok$ and $S \Downarrow_{\Delta'} T$, then $\Delta,\Delta' \vdash S <: T$ and $\Delta \vdash T\ ok$.*

Proof. We prove by induction on the $\Downarrow_{\Delta'}$ rules.

Case C-PROM: $\mathtt{S}{=}\mathtt{X}$, $\Delta'(\mathtt{X}){=}(+,\mathtt{T})$

By S-VAR and Lemma 5, $\Delta,\Delta'{\vdash}\mathtt{X}{<:}\mathtt{T}$.

Inspecting the cases when $\mathtt{X}{:}(\mathtt{v},\ \mathtt{T})$ are put into a Δ', it is only when opening a well-typed $\mathtt{C}\langle\mathtt{vT}\rangle$, where $\Delta{\vdash}\mathtt{T}$ *ok*(WF-CLASS).

Case C-TVAR: By S-REFL.

Case C-CLASS: $\mathtt{S}{=}\mathtt{C}\langle\overline{\mathtt{vT}}\rangle$, $\mathtt{T}{=}\mathtt{C}\langle\overline{\mathtt{wT}}'\rangle$.

There are three subcases, and we analyze without loss of generality only one $\mathtt{T}_i$.

Subcase 1: $\mathtt{T}_i{\Downarrow_{\Delta'}}\mathtt{T}_i$, $(\mathtt{w}_i,\mathtt{T}'_i){=}(\mathtt{v}_i,\mathtt{T}_i)$. Trivially true.

Subcase 2: $\mathtt{T}_i{\Downarrow_{\Delta'}}\mathtt{U}_i$, $\mathtt{T}_i{\neq}\mathtt{U}_i$, $(\mathtt{w}_i,\mathtt{T}'_i){=}(\mathtt{v}_i{\vee}{+},\mathtt{U}_i)$.

By induction hypothesis, $\Delta,\Delta'{\vdash}\mathtt{T}_i{<:}\mathtt{U}_i$.

$\mathtt{w}_i$ is either + or *. If $\mathtt{w}_i{=}{*}$, then by Lemma 5 and S-VAR, $\Delta,\Delta'{\vdash}\mathtt{C}\langle\overline{\mathtt{vT}}\rangle{<:}\mathtt{C}\langle\overline{\mathtt{wU}}\rangle$.

If $\mathtt{w}_i{=}{+}$, by S-VAR, and $\Delta,\Delta'{\vdash}\mathtt{T}_i{<:}\mathtt{U}_i$, we still have $\Delta,\Delta'{\vdash}\mathtt{S}{<:}\mathtt{T}$.

Subcase 3: $\mathtt{T}_i{=}\mathtt{X}$, $\Delta(\mathtt{X}){=}(\mathtt{v}'_i,\mathtt{U}_i)$.

If $\mathtt{v}'_i{=}\mathtt{o}$, or $\mathtt{v}_i$ and $\mathtt{v}'_i$ have opposite signs (i.e., one is + and the other -), then $\mathtt{w}_i{=}\mathtt{o}$, $\mathtt{T}_i{=}\mathtt{U}_i$. Conclusion easily falls out of S-VAR.

If $\mathtt{v}'_i{=}\mathtt{v}_i{=}{+}$ or $\mathtt{v}'_i{=}\mathtt{v}_i{=}{-}$, these cases easily falls out from the S-VAR rule, and S-UBOUND (for $\mathtt{v}'_i{=}\mathtt{v}_i{=}{+}$), and S-LBOUND (for $\mathtt{v}'_i{=}\mathtt{v}_i{=}{-}$).

If either $\mathtt{v}_i$ or $\mathtt{v}'_i$ is *, $\mathtt{w}_i{=}{*}$. Conclusion by S-VAR again.

□

Lemma 13. *Suppose* $\Delta,\Delta_1{\vdash}\mathtt{C}\langle\overline{\mathtt{S}}\rangle$ *ok and* $\mathtt{C}\langle\overline{\mathtt{S}}\rangle{\Downarrow_{\Delta_1}}\mathtt{T}$ *and* $\Delta{\vdash}\mathtt{T}{\Uparrow^{\Delta_2}}\mathtt{C}\langle\overline{\mathtt{U}}\rangle$ *and* $\Delta_3{\vdash}\mathtt{S}_0$ *ok where* $dom(\Delta_i)$ *(*$i=1,2,3$*) are distinct from each other.*

1) If $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}_0{\Downarrow_{\Delta_2}}\mathtt{S}'_0$ *and* $\overline{\mathtt{X}}$ *do not appear in* Δ_1 *or* Δ_2*, then* $[\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{S}_0{\Downarrow_{\Delta_1}}\mathtt{S}'_0$.

2) If $\Delta,\Delta_2{\vdash}\mathtt{U}_0{<:}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}_0$ *and* $\Delta{\vdash}\mathtt{U}_0$ *ok with* $\overline{\mathtt{X}}$ *not appearing in* Δ *or* Δ_1 *or* Δ_2*, then* $\Delta,\Delta_1{\vdash}\mathtt{U}_0{<:}[\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{S}_0$.

Proof. By inspecting the given conditions, one of these three properties must hold:

1. $\mathtt{S}_i \Downarrow_{\Delta_1} \mathtt{S}_i$, $\mathtt{U}_i = \mathtt{S}_i$.
2. $\mathtt{S}_i \Downarrow_{\Delta_1} \mathtt{S}'_i$, $\mathtt{S}_i \neq \mathtt{S}'_i$. By Lemma 12, $\Delta, \Delta_1 \vdash \mathtt{S}_i <: \mathtt{S}'_i$, and $\mathtt{U}_i$ is a type variable $\mathtt{Y}$ such that $\Delta_2(\mathtt{Y}) = (+, \mathtt{S}'_i)$.
3. $\mathtt{S}_i$ is a type variable $\mathtt{Z}$ and $\Delta_1(\mathtt{Z}) = (+, \mathtt{V})$, and $\mathtt{U}_i$ is a type variable $\mathtt{Y}$ and $\Delta_2(\mathtt{Y}) = (+, \mathtt{V})$.

1) We prove by the first part of the lemma by induction on $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}_0 \Downarrow_{\Delta_2} \mathtt{S}'_0$.

Case $\mathtt{S}_0 = \mathtt{X}$, $\mathtt{X} \notin dom(\Delta_2)$.

There are two cases:

Subcase $\mathtt{X} \in \overline{\mathtt{X}}$: $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}_0 = \mathtt{U}_i$, $[\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{S}_0 = \mathtt{S}_i$.

From the analysis of the three properties above,

1. $\mathtt{S}_i \Downarrow_{\Delta} \mathtt{S}_i$, $\mathtt{U}_i = \mathtt{S}_i$. Clearly, $\mathtt{U}_i \Downarrow_{\Delta_2} \mathtt{U}_i$.
2. $\mathtt{S}_i \Downarrow_{\Delta_1} \mathtt{S}'_i$, $\mathtt{U}_i \Downarrow_{\Delta_2} \mathtt{S}'_i$.
3. $\mathtt{S}_i \Downarrow_{\Delta_1} \mathtt{V}$, $\mathtt{U}_i \Downarrow_{\Delta_2} \mathtt{V}$.

Subcase $\mathtt{X} \notin \overline{\mathtt{X}}$: $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}_0 = [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{S}_0 = \mathtt{X}$. Proof is simple through induction hypothesis.

Case $\mathtt{S}_0 = \mathtt{C}\langle\overline{\mathtt{vT}}\rangle$, $\mathtt{C}\langle\overline{\mathtt{vT}}\rangle \Downarrow_{\Delta_2} \mathtt{C}\langle\overline{\mathtt{wT}'}\rangle$. Proofs for all cases of $\mathtt{v}_i$ and $\mathtt{w}_i$ falls out through induction hypothesis.

2) The second part is proven by inspecting the three properties above. There exists Δ', Δ'_1, $\overline{\mathtt{Y}}$, $\overline{\mathtt{T}}$, $\overline{\mathtt{S}}''$, such that:

$\Delta_1 = \Delta', \Delta'_1 \quad \Delta_2 = \Delta', \overline{\mathtt{Y}} <: \overline{\mathtt{T}} \quad \overline{\mathtt{S}} = [\overline{\mathtt{S}}''/\overline{\mathtt{Y}}]\overline{\mathtt{U}} \quad \Delta, \Delta_1 \vdash \overline{\mathtt{S}}'' <: \overline{\mathtt{T}}$

By Lemma 7, $\Delta, \Delta', \Delta'_1 \vdash [\overline{\mathtt{S}}''/\overline{\mathtt{Y}}]\mathtt{U}_0 <: [\overline{\mathtt{S}}''/\overline{\mathtt{Y}}][\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}_0$.

By definition $\overline{\mathtt{Y}}$ is fresh in $\mathtt{U}_0$ and $\mathtt{S}_0$, then,

$[\overline{\mathtt{S}}''/\overline{\mathtt{Y}}]\mathtt{U}_0 = \mathtt{U}_0 \quad [\overline{\mathtt{S}}''/\overline{\mathtt{Y}}][\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}_0 = [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{S}_0$

□

Lemma 14. *1. If* $\mathtt{S}\Downarrow_{\Delta',\mathtt{X}:(\mathtt{v},\mathtt{T})}\mathtt{S}''$*, and* $\Delta\vdash\mathtt{T}$ *ok where* $dom(\Delta',\mathtt{X}:(\mathtt{v},\mathtt{T}))$ *are fresh w.r.t.* Δ*, then* $[\mathtt{T}/\mathtt{X}]\mathtt{S}\Downarrow_{\Delta'}\mathtt{S}'$ *and* $\Delta\vdash\mathtt{S}'<:\mathtt{S}''$*.*

2. If $\Delta\vdash\mathtt{T}<:\mathtt{U}$ *and* $\mathtt{S}\Downarrow_{\Delta',\mathtt{X}:(+,\mathtt{T})}\mathtt{S}''$ *where* $dom(\Delta')$ *and* $\mathtt{X}$ *are fresh for* Δ*, then* $\mathtt{S}\Downarrow_{\Delta',\mathtt{X}:(+,\mathtt{T})}\mathtt{S}'$ *and* $\Delta\vdash\mathtt{S}'<:\mathtt{S}''$*.*

3. If $\Delta\vdash\mathtt{U}<:\mathtt{T}$ *and* $\mathtt{S}\Downarrow_{\Delta',\mathtt{X}:(-,\mathtt{T})}\mathtt{S}''$ *where* $dom(\Delta')$ *and* $\mathtt{X}$ *are fresh for* Δ*, then* $\mathtt{S}\Downarrow_{\Delta',\mathtt{X}:(-,\mathtt{T})}\mathtt{S}'$ *and* $\Delta\vdash\mathtt{S}'<:\mathtt{S}''$*.*

4. If $\mathtt{S}\Downarrow_{\Delta',\mathtt{X}:(*,\mathtt{U})}\mathtt{S}''$ *where* $dom(\Delta')\cup\{\mathtt{X}\}$ *are fresh for* Δ*, then* $\mathtt{S}\Downarrow_{\Delta',\mathtt{X}:(\mathtt{v},\mathtt{T})}\mathtt{S}'$ *and* $\Delta\vdash\mathtt{S}'<:\mathtt{S}''$*.*

Proof. We prove by structural induction on S.

1)

Case $\mathtt{S}=\mathtt{Y}$, where Y is a type variable. $\mathtt{Y}\Downarrow_{\Delta',\mathtt{X}:(\mathtt{v},\mathtt{T})}\mathtt{S}''$.

If $\mathtt{Y}\neq\mathtt{X}$, $\mathtt{Y}\Downarrow_{\Delta',\mathtt{X}:(\mathtt{v},\mathtt{T})}\mathtt{Y}$, and $\mathtt{Y}\Downarrow_{\Delta'}\mathtt{Y}$.

If $\mathtt{Y}=\mathtt{X}$, $\mathtt{v}=+$, $\mathtt{Y}\Downarrow_{\Delta',\mathtt{X}:(\mathtt{v},\mathtt{T})}\mathtt{T}$. $[\mathtt{T}/\mathtt{X}]\mathtt{Y}=\mathtt{T}$, $\mathtt{T}\Downarrow_{\Delta'}\mathtt{T}$.

If $\mathtt{Y}\notin dom(\Delta',\mathtt{X}:(\mathtt{v},\mathtt{T}))$, then $\mathtt{Y}\Downarrow_{\Delta',\mathtt{X}:(\mathtt{v},\mathtt{T})}\mathtt{Y}$, $\mathtt{Y}\Downarrow_{\Delta'}\mathtt{Y}$.

In all cases, conclusion follows through S-REFL.

Case $\mathtt{S}=\mathtt{C}<\overline{\mathtt{v}\mathtt{T}}>$, $\mathtt{S}''=\mathtt{C}<\overline{\mathtt{w}\mathtt{T}'}>$.

By inspecting the three cases in the definition of $(\mathtt{w}_i,\mathtt{T}'_i)$ and the induction hypothesis, it's easy to show that $[\mathtt{T}/\mathtt{X}]\mathtt{C}<\overline{\mathtt{v}\mathtt{T}}>\Downarrow_{\Delta'}\mathtt{C}<\overline{\mathtt{w}'\mathtt{T}''}>$ such that $\Delta\vdash\mathtt{C}<\overline{\mathtt{w}'\mathtt{T}''}><:\mathtt{C}<\overline{\mathtt{w}\mathtt{T}'}>$.

2), 3), and 4) can be proven similarly through induction hypothesis.

□

Lemma 15. *If* $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\overline{\mathtt{N}}$ *ok,* $\Delta\vdash bound_{\Delta}(\mathtt{T}_0)\Uparrow^{\Delta'}\mathtt{C}<\overline{\mathtt{U}}>$*, then* $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash bound_{\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}}(\mathtt{T}_0)\Uparrow^{\Delta'}\mathtt{C}<\overline{\mathtt{U}}>$*.*

Proof. By straight-forward induction on the derivation of open rules. □

Appendix B

Featherweight MorphJ (FMJ): Proof of Soundness

Theorem 4 (Subject Reduction)**.** *If* $\Delta;\Lambda;\Gamma \vdash \mathtt{e}{\in}\mathtt{T}$ *and* $\mathtt{e} \rightarrow \mathtt{e}'$*, then* $\Delta;\Lambda;\Gamma \vdash \mathtt{e}'{\in}\mathtt{S}$ *and* $\Delta \vdash \mathtt{S}{<:}\mathtt{T}$ *for some* S*.*

Proof. Prove by structural induction on the reduction rules.

Case R-FIELD: $\mathtt{e}{=}\mathtt{new\ C{<}\overline{T}{>}(\overline{e}).f}_i$, $\mathtt{e}'{=}\mathtt{e}_i$

By T-NEW,

$\Delta;\Lambda;\Gamma{\vdash}\mathtt{new\ C{<}\overline{T}{>}(\overline{e})}{\in}\mathtt{C{<}\overline{T}{>}}$ $\quad \Delta{\vdash}\mathit{fields}(\mathtt{C{<}\overline{T}{>}}){=}\overline{\mathtt{U}}\ \overline{\mathtt{f}}$

$\Delta;\Lambda;\Gamma{\vdash}\overline{\mathtt{e}}{\in}\overline{\mathtt{S}}$ $\quad \Delta{\vdash}\overline{\mathtt{S}}{<:}\overline{\mathtt{U}}$

By T-FIELD and definition of bound,

$\mathit{bound}_\Delta(\mathtt{C{<}\overline{T}{>}}){=}\mathtt{C{<}\overline{T}{>}}$ $\quad \Delta;\Lambda;\Gamma{\vdash}\mathtt{new\ C{<}\overline{T}{>}(\overline{e}).f}_i{\in}\mathtt{U}_i$

Let S be $\mathtt{S}_i$, T be $\mathtt{U}_i$.

Case R-INVK: $\mathtt{e}{=}\mathtt{new\ C{<}\overline{T}{>}(\overline{e}).m(\overline{d})}$, $\mathtt{e}'{=}[\overline{\mathtt{d}}/\overline{\mathtt{x}},\mathtt{new\ C{<}\overline{T}{>}/this}]\mathtt{e}_0$

By R-INVK, T-NEW, T-INVK

$\mathit{mbody}(\mathtt{m},\ \mathtt{C{<}\overline{T}{>}}){=}(\overline{\mathtt{x}},\mathtt{e}_0)$ $\quad \Delta{\vdash}\mathtt{new\ C{<}\overline{T}{>}(\overline{e})}{\in}\mathtt{C{<}\overline{T}{>}}$ $\quad \Delta{\vdash}\mathtt{C{<}\overline{T}{>}}\ \mathit{ok}$

$\Delta;\Lambda{\vdash}\mathit{mtype}(\mathtt{m},\ \mathtt{C{<}\overline{T}{>}}){=}\overline{\mathtt{T}}'{\rightarrow}\mathtt{T}$ $\quad \Delta;\Gamma;\Lambda{\vdash}\overline{\mathtt{d}}{\in}\overline{\mathtt{S}}$ $\quad \Delta{\vdash}\overline{\mathtt{S}}{<:}\overline{\mathtt{T}}'$

By Lemma 16, for some S, $\Delta{\vdash}\mathtt{S}{<:}\mathtt{T}$, $\Delta{\vdash}\mathtt{S}\ \mathit{ok}$, $\Delta;\Lambda;\overline{\mathtt{x}}{\mapsto}\overline{\mathtt{T}}',\mathtt{this}{\mapsto}\mathtt{C{<}\overline{T}{>}}{\vdash}\mathtt{e}_0{\in}\mathtt{S}$.

By Lemma 19, for some $\mathtt{S}'$ where $\Delta;\Lambda;\Gamma{\vdash}\mathtt{S}'{<:}\mathtt{S}$,

$\Delta;\Lambda;\Gamma{\vdash}[\overline{\mathtt{d}}/\overline{\mathtt{x}},\mathtt{new\ C{<}\overline{T}{>}/this}]\mathtt{e}_0{\in}\mathtt{S}'$,

Case RC-FIELD: $\mathtt{e}{=}\mathtt{e}_0\mathtt{.f}$, $\mathtt{e}'{=}\mathtt{e}'_0\mathtt{.f}$

By T-FIELD,

$\Delta;\Lambda;\Gamma{\vdash}\mathtt{e}_0{\in}\mathtt{T}_0$ $\quad \Delta{\vdash}\mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0)){=}\overline{\mathtt{T}}\ \overline{\mathtt{f}}$ $\quad \Delta;\Lambda;\Gamma{\vdash}\mathtt{e}_0\mathtt{.f}{\in}\mathtt{T}_i$

By induction hypothesis,

$\Delta;\Lambda;\Gamma\vdash \mathtt{e}'_0\in \mathtt{S}_0 \quad \Delta\vdash \mathtt{S}_0<:\mathtt{T}_0$

By Lemma 26, $\Delta\vdash \mathit{fields}(\mathit{bound}_\Delta(\mathtt{S}_0))=\overline{\mathtt{S}}\ \overline{\mathtt{g}}$, $\Delta;\Lambda;\Gamma\vdash \mathtt{e}'_0.\mathtt{f}\in\mathtt{S}_i$, $\mathtt{S}_i=\mathtt{T}_i$.

By S-REFL, $\Delta\vdash \mathtt{S}_i<:\mathtt{T}_i$.

Let T be $\mathtt{T}_i$, S be $\mathtt{S}_i$.

Case RC-INV-RECV: $\mathtt{e}=\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})$, $\mathtt{e}'=\mathtt{e}'_0.\mathtt{m}(\overline{\mathtt{e}})$

By T-INVK,

$\Delta;\Lambda;\Gamma\vdash \mathtt{e}_0\in\mathtt{T}_0 \qquad \Delta;\Lambda;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{T}}'$

$\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{T}_0)=\overline{\mathtt{T}}\to\mathtt{T} \quad \Delta\vdash\overline{\mathtt{T}}'<:\overline{\mathtt{T}}$

By induction hypothesis, Lemma 34, and T-INVK,

$\Delta;\Lambda;\Gamma\vdash \mathtt{e}'\in\mathtt{S}_0 \qquad \Delta\vdash \mathtt{S}_0<:\mathtt{T}_0$

$\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{S}_0)=\overline{\mathtt{T}}\to\mathtt{T} \quad \Delta;\Lambda;\Gamma\vdash \mathtt{e}'.\mathtt{m}(\overline{\mathtt{e}})\in\mathtt{T}$

Let S be T.

Case RC-INV-ARG: Easy by induction hypothesis and T-INVK.

Case RC-NEW-ARG: Easy by induction hypothesis and T-NEW.

□

Theorem 5 (Progress)**.** *Let* e *be a well-typed expression. 1. If* e *has* new C<$\overline{\mathtt{T}}$>($\overline{\mathtt{e}}$).f *as a subexpression, then* $\emptyset\vdash \mathit{fields}(\mathtt{C}<\overline{\mathtt{T}}>)=\overline{\mathtt{U}}\ \overline{\mathtt{f}}$*, and* f = $\mathtt{f}_i$*. 2. If* e *has* new C<$\overline{\mathtt{T}}$>($\overline{\mathtt{e}}$).m($\overline{\mathtt{d}}$) *as a subexpression, then* $\mathit{mbody}(\mathtt{m},\ \mathtt{C}<\overline{\mathtt{T}}>)=(\overline{\mathtt{x}},\mathtt{e}_0)$ *and* $|\overline{\mathtt{x}}| = |\overline{\mathtt{d}}|$.

Proof. 1. It follows easily from T-FIELD and the well-typedness of subexpressions.

2. Also using well-typedness of subexpression and T-INVK, we have $\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{C}<\overline{\mathtt{T}}>)=\overline{\mathtt{U}}\to\mathtt{U}_0$. It is then easy using the MB-* rules to show that $\Delta;\Lambda\vdash \mathit{mbody}(\mathtt{m},\ \mathtt{C}<\overline{\mathtt{T}}>)=(\overline{\mathtt{x}},\mathtt{e}_0)$, since MB-* and MT-* rules have a one-to-one correspondence for non-variable types C<$\overline{\mathtt{T}}$>. □

Theorem 6 (Type Soundness)**.** *If* $\emptyset;\emptyset;\emptyset\vdash \mathtt{e}\in\mathtt{T}$ *and* $\mathtt{e}\longrightarrow^*\mathtt{e}'$*, then* e′ *is a value* v *such that* $\emptyset;\emptyset;\emptyset\vdash \mathtt{v}\in\mathtt{S}$ *and* $\emptyset\vdash \mathtt{S}<:\mathtt{T}$ *for some type* S.

Proof. Conclusion follows from Theorem 4 and Theorem 5 □

Lemma 16. *If* $\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{C<\overline{T}>})=\overline{\mathtt{S}}\to\mathtt{S}$, $mbody(\mathtt{m},\ \mathtt{C<\overline{T}>})=(\overline{\mathtt{x}},\mathtt{e}_0)$, *where* $\Delta\vdash\mathtt{C<\overline{T}>}\ ok$, *then there exists a type* $\mathtt{S}'$ *such that* $\Delta\vdash\mathtt{S}'<:\mathtt{S}$, $\Delta\vdash\mathtt{S}'\ ok$, $\Delta;\Lambda;\overline{\mathtt{x}}\mapsto\overline{\mathtt{S}},\mathtt{this}\mapsto\mathtt{C<\overline{T}>}\vdash\mathtt{e}_0\in\mathtt{S}'$.

Proof. By induction on the derivation of $mbody(\mathtt{m},\ \mathtt{C<\overline{T}>})=(\overline{\mathtt{x}},\mathtt{e}_0)$:

Case MB-CLASS-S:

$CT(\mathtt{C})=\mathtt{class\ C<\overline{X}\lhd\overline{N}>\lhd N\ \{...\overline{M}\}}\quad \mathtt{U}_0\ \mathtt{m}\ \mathtt{(\overline{U}\ \overline{x})}\ \mathtt{\{\uparrow e;\}}\in\overline{\mathtt{M}}\quad \mathtt{e}_0=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e}$

By MT-CLASS-S,

$\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{C<\overline{T}>})=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{U}}\to\mathtt{U}_0)$

By WF-CLASS, $\Delta\vdash\overline{\mathtt{T}}<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$

By T-METH-S,

$\overline{\mathtt{X}}<:\overline{\mathtt{N}};\emptyset;\overline{\mathtt{x}}\mapsto\overline{\mathtt{S}},\mathtt{this}\mapsto\mathtt{C<\overline{X}>}\vdash\mathtt{e}\in\mathtt{U}_0'\quad \overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\mathtt{U}_0'<:\mathtt{U}_0$

By Lemma 23 and 20, $\Delta;\Lambda;[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{x}}\mapsto\overline{\mathtt{S}},\mathtt{this}\mapsto\mathtt{C<\overline{X}>})\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e}\in[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0'$

By Lemma 21 and 20, $\Delta\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0'<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0$.

Let $\overline{\mathtt{S}}=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}}$, $\mathtt{S}=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0$, $\mathtt{S}'=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0'$.

Case MB-CLASS-R:

$CT(\mathtt{C})=\mathtt{class\ C<\overline{X}\lhd\overline{N}>\lhd T\ \{\ ...\ \ \overline{\mathfrak{M}}\}}$

$\mathtt{<\overline{Y}\lhd\overline{P}>for(}\mathbb{M}_p\mathtt{;}o\mathbb{M}_f\mathtt{)\ S''\ }\eta\ \mathtt{(\overline{S''}\ \overline{x})\ \{\uparrow e_0';\}}\in\overline{\mathfrak{M}}$

$R_p''=range(\mathbb{M}_p,\mathtt{<\overline{Y}\lhd\overline{P}>})\quad R_n''=range(\mathbb{M}_f,\bullet)\qquad \Lambda''=\langle R_p'',R_n''\rangle$

$\Delta''=\overline{\mathtt{X}}<:\overline{\mathtt{N}};\overline{\mathtt{Y}}<:\overline{\mathtt{P}}\qquad \Lambda_d=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\langle R_p'',R_n''\rangle)\qquad \Delta'';\emptyset\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda_r$

$\Delta'';[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda\Lambda_d\qquad \mathtt{e}_0=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta]\mathtt{e}_0'$

By MT-CLASS-R and Lemma 17 (thus there is only one method $\mathtt{m}$),

$\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{C<\overline{T}>})=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{S}''}\to\mathtt{S}'')$

By T-METH-R,

$\Gamma''=\overline{\mathtt{x}}\mapsto\overline{\mathtt{S}''},\mathtt{this}\mapsto\mathtt{C<\overline{X}>}\quad \Delta'';\Lambda'';\Gamma''\vdash\mathtt{e}_0'\in\mathtt{T}''\quad \Delta''\vdash\mathtt{T}''<:\mathtt{S}''\quad \Delta''\vdash\overline{\mathtt{N}}\ \mathtt{OK}$

By WF-CLASS and Lemma 20, $\Delta,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta''\vdash\overline{\mathtt{T}}<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$

By Lemma 21 and Lemma 20, $\Delta,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta''\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}''<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S}''$

By Lemma 23, $\Delta,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta'';[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Lambda'';[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Gamma''\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e}'_0\in[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_0$

By Lemma 35, and the obvious facts that $[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Delta''=\emptyset$, $[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Gamma''=\Gamma$,

$\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta]\mathtt{e}'_0\in[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}_0$

By Lemma 22, $\Delta\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}''<:\mathtt{S}''$

Let $\overline{\mathtt{S}}=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{S}}''$, $\mathtt{S}=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{S}''$, $\mathtt{S}'=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}''$.

Case MB-SUPER-S, MB-SUPER-R: Follows from induction hypothesis.

□

Lemma 17. *Suppose* $\Delta\vdash disjoint(\Lambda_1, \Lambda_2)$*, where*

$\Lambda_1=\langle R_{p_1}, oR_{n_1}\rangle \quad R_{p_1}=(\mathtt{T}_1, <\overline{\mathtt{X}}\lhd\overline{\mathtt{Q}}>\overline{\mathtt{U}}\rightarrow\mathtt{U}_0) \quad R_{n_1}=(\mathtt{T}'_1, \overline{\mathtt{U}}'\rightarrow\mathtt{U}'_0)$

$\Lambda_2=\langle R_{p_2}, o'R_{n_2}\rangle \quad R_{p_2}=(\mathtt{T}_2, <\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}>\overline{\mathtt{V}}'\rightarrow\mathtt{V}_0) \quad R_{n_2}=(\mathtt{T}'_2, \overline{\mathtt{V}}'\rightarrow\mathtt{V}'_0)$

Then for any Λ_3*, if* $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash\Lambda_3\sqsubseteq_\Lambda\Lambda_1$*, then there does not exist* $\overline{\mathtt{W}}'$ *such that* $\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash\Lambda_3\sqsubseteq_\Lambda\Lambda_2$

Proof. We prove by contradiction. Let there by such $\overline{\mathtt{W}}'$ such that $\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash\Lambda_3\sqsubseteq_\Lambda\Lambda_2$

Let $\Lambda_3=\langle R_{p_3}, o''R_{n_3}\rangle$, where

$R_{p_3}=(\mathtt{T}_3, <\overline{\mathtt{Z}}\lhd\overline{\mathtt{N}}>\overline{\mathtt{S}}\rightarrow\mathtt{S}_0) \quad R_{n_3}=(\mathtt{T}'_3, <\overline{\mathtt{Z}}\lhd\overline{\mathtt{N}}>\overline{\mathtt{S}}\rightarrow\mathtt{S}'_0)$

By DS-Λ, one of the following mutually exclusive range conditions must hold:

$\Delta\vdash{+}R_{p_1}\otimes{+}R_{p_2} \quad \Delta\vdash{+}R_{p_1}\otimes o'R_{n_2} \quad \Delta\vdash{+}R_{p_2}\otimes oR_{n_2} \quad \Delta\vdash oR_{n_1}\otimes o'R_{n_2}$

By SB-Λ and SB-R, $\Delta\vdash\overline{\mathtt{P}},\overline{\mathtt{Q}},\overline{\mathtt{N}}$ OK

$\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash R_{p_3}\sqsubseteq_R R_{p_1} \quad \Delta,\overline{\mathtt{X}}<:\overline{\mathtt{Q}},\overline{\mathtt{Z}}<:\overline{\mathtt{N}};[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash unify(\mathtt{S}_0:\overline{\mathtt{S}}, \mathtt{U}_0:\overline{\mathtt{U}})$

$\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash R_{p_3}\sqsubseteq_R R_{p_2} \quad \Delta,\overline{\mathtt{Y}}<:\overline{\mathtt{P}},\overline{\mathtt{Z}}<:\overline{\mathtt{N}};[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash unify(\mathtt{S}_0:\overline{\mathtt{S}}, \mathtt{V}_0:\overline{\mathtt{V}})$

By UNI,

$[\overline{\mathtt{W}}/\overline{\mathtt{X}}](\mathtt{S}_0:\overline{\mathtt{S}})=[\overline{\mathtt{W}}/\overline{\mathtt{X}}](\mathtt{U}_0:\overline{\mathtt{U}})$ for all $\mathtt{X}_i\in\overline{\mathtt{X}}$, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{Q}},\overline{\mathtt{Z}}<:\overline{\mathtt{N}}\vdash\mathtt{W}_i\prec:_{\overline{\mathtt{X}}}\mathtt{X}_i$

$[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{S}_0:\overline{\mathtt{S}}=[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{V}_0:\overline{\mathtt{V}}$ for all $\mathtt{Y}_i\in\overline{\mathtt{Y}}$, $\Delta,\overline{\mathtt{Y}}<:\overline{\mathtt{P}},\overline{\mathtt{Z}}<:\overline{\mathtt{N}}\vdash\mathtt{W}'_i\prec:_{\overline{\mathtt{Y}}}\mathtt{Y}_i$

Since neither $\overline{\mathtt{X}}$ or $\overline{\mathtt{Y}}$ appear in $\mathtt{S}_0:\overline{\mathtt{S}}$,

$[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\mathtt{S}_0:\overline{\mathtt{S}}=[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{S}_0:\overline{\mathtt{S}}=\mathtt{S}_0:\overline{\mathtt{S}} \quad [\overline{\mathtt{W}}/\overline{\mathtt{X}}]\mathtt{U}_0:\overline{\mathtt{U}}=[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{V}_0:\overline{\mathtt{V}}$

By Lemma 20,

for all $\mathtt{X}_i\in\overline{\mathtt{X}}$, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{P}},\overline{\mathtt{Y}}<:\overline{\mathtt{Q}}\vdash\mathtt{W}_i\prec:_{\overline{\mathtt{X}}:\overline{\mathtt{Y}}}\mathtt{X}_i$

for all $\mathtt{Y}_i\in\overline{\mathtt{Y}}$, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{P}},\overline{\mathtt{Y}}<:\overline{\mathtt{Q}}\vdash\mathtt{W}'_i\prec:_{\overline{\mathtt{X}}:\overline{\mathtt{Y}}}\mathtt{Y}_i$,

It follows that,

$\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{P}},\overline{\mathtt{Y}}<:\overline{\mathtt{Q}};[(\overline{\mathtt{W}}:\overline{\mathtt{W}}')/(\overline{\mathtt{X}}:\overline{\mathtt{Y}})]\vdash \mathit{unify}(\mathtt{U}_0:\overline{\mathtt{U}},\ \mathtt{V}_0:\overline{\mathtt{V}})$

It directly contradicts $\Delta\vdash{+}R_{p_1}\otimes{+}R_{p_2}$ Thus, one of the other mutually exclusive range conditions must hold.

By SB-Λ,

$\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash\Lambda_3\sqsubseteq_\Lambda\Lambda_1$ implies $o{=}o''{=}+$ or $o{=}o''{=}-$

$\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash\Lambda_3\sqsubseteq_\Lambda\Lambda_2$ implies $o'{=}o''{=}+$ or $o'{=}o''{=}-$

Thus, there can only be two options for $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash\Lambda_3\sqsubseteq_\Lambda\Lambda_1$ and $\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash\Lambda_3\sqsubseteq_\Lambda\Lambda_2$:

$o{=}o'{=}o''{=}+$ or $o{=}o'{=}o''{=}-$

We now analyze these two cases:

Case $o{=}o'{=}o''{=}+$

By SB-Λ and SB-R,

$\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash R_{n_3}\sqsubseteq_R R_{n_1}$ $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{Q}},\overline{\mathtt{Z}}<:\overline{\mathtt{N}};[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash \mathit{unify}(\mathtt{S}'_0:\overline{\mathtt{S}}',\ \mathtt{U}'_0:\overline{\mathtt{U}}')$

$\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash R_{p_3}\sqsubseteq_R R_{p_2}$ $\Delta,\overline{\mathtt{Y}}<:\overline{\mathtt{P}},\overline{\mathtt{Z}}<:\overline{\mathtt{N}};[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash \mathit{unify}(\mathtt{S}'_0:\overline{\mathtt{S}}',\ \mathtt{V}'_0:\overline{\mathtt{V}}')$

By UNI,

$[\overline{\mathtt{W}}/\overline{\mathtt{X}}](\mathtt{S}'_0:\overline{\mathtt{S}}')=[\overline{\mathtt{W}}/\overline{\mathtt{X}}](\mathtt{U}'_0:\overline{\mathtt{U}}')$ for all $\mathtt{X}_i\in\overline{\mathtt{X}}$, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{Q}},\overline{\mathtt{Z}}<:\overline{\mathtt{N}}\vdash\mathtt{W}_i\prec:_{\overline{\mathtt{X}}}\mathtt{X}_i$

$[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{S}'_0:\overline{\mathtt{S}}'=[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{V}'_0:\overline{\mathtt{V}}'$ for all $\mathtt{Y}_i\in\overline{\mathtt{Y}}$, $\Delta,\overline{\mathtt{Y}}<:\overline{\mathtt{P}},\overline{\mathtt{Z}}<:\overline{\mathtt{N}}\vdash\mathtt{W}'_i\prec:_{\overline{\mathtt{Y}}}\mathtt{Y}_i$

Since neither $\overline{\mathtt{X}}$ or $\overline{\mathtt{Y}}$ appear in $\mathtt{S}'_0:\overline{\mathtt{S}}'$,

$[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\mathtt{S}'_0:\overline{\mathtt{S}}'=[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{S}'_0:\overline{\mathtt{S}}'=\mathtt{S}'_0:\overline{\mathtt{S}}'$ $[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\mathtt{U}'_0:\overline{\mathtt{U}}'=[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{V}'_0:\overline{\mathtt{V}}'$

By Lemma 20,

for all $\mathtt{X}_i\in\overline{\mathtt{X}}$, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{P}},\overline{\mathtt{Y}}<:\overline{\mathtt{Q}}\vdash\mathtt{W}_i\prec:_{\overline{\mathtt{X}}:\overline{\mathtt{Y}}}\mathtt{X}_i$

for all $\mathtt{Y}_i\in\overline{\mathtt{Y}}$, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{P}},\overline{\mathtt{Y}}<:\overline{\mathtt{Q}}\vdash\mathtt{W}'_i\prec:_{\overline{\mathtt{X}}:\overline{\mathtt{Y}}}\mathtt{Y}_i$,

It follows that

$\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{P}},\overline{\mathtt{Y}}<:\overline{\mathtt{Q}};[(\overline{\mathtt{W}}:\overline{\mathtt{W}}')/(\overline{\mathtt{X}}:\overline{\mathtt{Y}})]\vdash \mathit{unify}(\mathtt{U}'_0:\overline{\mathtt{U}}',\ \mathtt{V}'_0:\overline{\mathtt{V}}')$

This directly contradicts $\Delta\vdash{+}R_{n_1}\otimes{+}R_{n_2}$

Thus, it must be true that $\Delta\vdash{+}R_{p_1}\otimes{+}R_{n_2}$, or $\Delta\vdash{+}R_{p_2}\otimes{+}R_{n_1}$.

Assume that $\Delta\vdash{+}R_{p_1}\otimes{+}R_{n_2}$. By $\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash R_{n_3}\sqsubseteq_R R_{n_2}$, and Lemma 18, $\Delta\vdash{+}R_{p_1}\otimes{+}R_{n_3}$, which makes $\Delta\vdash \mathit{disjoint}(\Lambda_1,\ \Lambda_3)$. This contradicts the assumption.

$\Delta\vdash{+}R_{p_2}\otimes{+}R_{n_1}$ results in a similar contradiction.

Case $o{=}o'{=}o''{=}$-

By Lemma 18 and $\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash R_{n_2}\sqsubseteq_R R_{n_3}$, $\Delta\vdash{+}R_{p_1}\otimes$-$R_{n_3}$, which makes $\Delta\vdash \mathit{disjoint}(\Lambda_1,\ \Lambda_3)$. This contradicts the assumption.

Similar contradiction results from $\Delta\vdash{+}R_{p_2}\otimes$-$R_{n_1}$.

□

Lemma 18. *1) Suppose* $\Delta\vdash{+}R_1\otimes{+}R_2$, $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_3\sqsubseteq_R R_2$, *where R3 has no pattern type variables, then* $\Delta\vdash{+}R_1\otimes{+}R_3$.

2) Suppose $\Delta\vdash{+}R_1\otimes$-$R_2$, $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_2\sqsubseteq_R R_3$, *where R3 has no pattern type variables, then* $\Delta\vdash{+}R_1\otimes$-$R_3$.

Proof. Let $R_1{=}(\mathtt{T}_1,\ {<}\overline{\mathtt{X}}{\lhd}\overline{\mathtt{Q}}{>}\overline{\mathtt{V}}{\rightarrow}\mathtt{V})$, $R_2{=}(\mathtt{T}_2,\ {<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>}\overline{\mathtt{U}}{\rightarrow}\mathtt{U})$, $R_3{=}(\mathtt{T}_3,\ \overline{\mathtt{S}}{\rightarrow}\mathtt{S})$

1) We prove by contradiction. Suppose $\Delta\not\vdash{+}R_1\otimes{+}R_3$

That means for some $\overline{\mathtt{W}}'$, $\Delta,\overline{\mathtt{X}}{<:}\overline{\mathtt{Q}};[\overline{\mathtt{W}}'/\overline{\mathtt{Z}}]\vdash \mathit{unify}(\mathtt{V}{:}\overline{\mathtt{V}},\ \mathtt{S}{:}\overline{\mathtt{S}})$

By definition of SB-R, $\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\vdash R_3\sqsubseteq_R R_1$

By UNI, and the fact that neither $\overline{\mathtt{X}}$ or $\overline{\mathtt{Y}}$ appear in $\mathtt{S}{:}\overline{\mathtt{S}}$,

$[\overline{\mathtt{W}}'/\overline{\mathtt{X}}](\ \mathtt{S}{:}\overline{\mathtt{S}}){=}[\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\ \mathtt{S}{:}\overline{\mathtt{S}})\quad [\overline{\mathtt{W}}'/\overline{\mathtt{X}}](\mathtt{V}{:}\overline{\mathtt{V}}){=}[\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\mathtt{U}{:}\overline{\mathtt{U}})$

Since $\overline{\mathtt{X}}$ do not appear in $\mathtt{U}{:}\overline{\mathtt{U}}$, and $\overline{\mathtt{Y}}$ do not appear in $\mathtt{V}{:}\overline{\mathtt{V}}$,

$[\overline{\mathtt{W}}'/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\mathtt{V}{:}\overline{\mathtt{V}}){=}[\overline{\mathtt{W}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\mathtt{U}{:}\overline{\mathtt{U}})$

Thus, $\overline{\mathtt{W}}{:}\overline{\mathtt{W}}'$ contradicts the condition in ME-1 for $\Delta\vdash{+}R_1\otimes{+}R_2$.

2) Proof follows from Lemma 38 and ME-2.

□

Lemma 19 (Term Substitution Preserves Typing)**.** *If* $\Delta;\Lambda;\Gamma,\overline{\mathtt{x}}{\mapsto}\overline{\mathtt{T}}\vdash\mathtt{e}{\in}\mathtt{T}$, η *does not appear in* $\mathtt{e}$, $\Delta;\Lambda;\Gamma\vdash\overline{\mathtt{d}}{\in}\overline{\mathtt{S}}$, $\Delta\vdash\overline{\mathtt{S}}{<:}\overline{\mathtt{T}}$, *then* $\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}{\in}\mathtt{T}'$, *for some* $\mathtt{T}'$ *where* $\Delta;\Lambda;\Gamma\vdash\mathtt{T}'{<:}\mathtt{T}$.

Proof. By induction on the derivation of $\Delta;\Lambda;\Gamma,\overline{\mathtt{x}}{\mapsto}\overline{\mathtt{T}}\vdash\mathtt{e}{\in}\mathtt{T}$.

Case T-VAR: $\mathtt{e}{=}\mathtt{x}_i$

$[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{x}{=}\mathtt{d}_i\quad \Delta;\Lambda;\Gamma\vdash\mathtt{d}_i{\in}\mathtt{S}_i$

Let $\mathtt{T}'{=}\mathtt{S}_i$

Case T-FIELD: $\mathtt{e}=\mathtt{e}_0\,.\,\mathtt{f}_i$

By induction hypothesis and T-FIELD,

$\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0\in\mathtt{T}_0 \quad \Delta;\Lambda;\Gamma\vdash\mathtt{T}_0<:\mathtt{T}$

Conclusion follows from Lemma 26.

Case T-INVK: $\mathtt{e}=\mathtt{e}_0\,.\,\mathtt{n}(\overline{\mathtt{e}})$

By premesis of the lemma, $\mathtt{n}=\mathtt{m}$

By T-INVK:

$\Delta;\Lambda;\Gamma,\overline{\mathtt{x}}\mapsto\overline{\mathtt{T}}\vdash\mathtt{e}_0\in\mathtt{T}_0 \qquad \Delta;\Lambda;\Gamma,\overline{\mathtt{x}}\mapsto\overline{\mathtt{T}}\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}}$

$\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{T}_0)=\overline{\mathtt{T}}\to\mathtt{T} \quad \Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{T}}$

By induction hypothesis,

$\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0\in\mathtt{T'}_0 \quad \Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}}\in\overline{\mathtt{T}'}$

$\Delta\vdash\mathtt{T'}_0<:\mathtt{T}_0 \qquad \Delta\vdash\overline{\mathtt{T}'}<:\overline{\mathtt{T}}$

By Lemma 34, $\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{T'}_0)=\overline{\mathtt{T}}\to\mathtt{T}$

By T-INVK, $\Delta;\Lambda\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}](\mathtt{e}_0\,.\,\mathtt{m}(\overline{\mathtt{e}}))\in\mathtt{T}$

Case T-NEW: $\mathtt{e}=\mathtt{new\ C}\!<\!\overline{\mathtt{T}}\!>\!(\overline{\mathtt{e}})$

$\Delta\vdash\mathtt{C}\!<\!\overline{\mathtt{T}}\!>\ ok \quad \Delta\vdash \mathit{fields}(\mathtt{C}\!<\!\overline{\mathtt{T}}\!>)=\overline{\mathtt{U}}\ \overline{\mathtt{f}}$

$\Delta;\Lambda;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}} \quad \Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{U}}$

By induction hypothesis, $\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}}\in\overline{\mathtt{S}'}$, for some $\overline{\mathtt{S}'}$where $\Delta\vdash\overline{\mathtt{S}'}<:\overline{\mathtt{S}}$.

By S-TRANS, $\Delta\vdash\overline{\mathtt{S}'}<:\overline{\mathtt{U}}$.

By Lemma 26 and T-NEW, $\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}](\mathtt{new\ C}\!<\!\overline{\mathtt{T}}\!>\!(\overline{\mathtt{e}}))\in\mathtt{C}\!<\!\overline{\mathtt{T}}\!>$

□

Lemma 20 (Weakening)**.** *Suppose* $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\overline{\mathtt{N}}$ *ok, none of* $\overline{\mathtt{X}}$ *appears in* Δ*, and* $\Delta\vdash\mathtt{U}$ *ok.*

1. *If* $\Delta\vdash\mathtt{S}<:\mathtt{T}$*, then* $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\mathtt{S}<:\mathtt{T}$*.*
2. *If* $\Delta\vdash\mathtt{S}$ *ok, then* $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\mathtt{S}$ *ok.*
3. *If* $\Delta;\Lambda;\Gamma\vdash\mathtt{e}\in\mathtt{T}$*, then* $\Delta;\Lambda;\Gamma,\mathtt{x}\in\mathtt{U}\vdash\mathtt{e}\in\mathtt{T}$ *and* $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}};\Lambda;\Gamma\vdash\mathtt{e}\in\mathtt{T}$*.*

4. *If* $\Delta;\emptyset;\Gamma\vdash \mathtt{e}\in\mathtt{T}$, *then* $\Delta;\Lambda;\Gamma\vdash \mathtt{e}\in\mathtt{T}$.

5. *If* $\Delta\vdash \mathtt{T}\prec:_{\overline{\mathtt{Z}}}\mathtt{S}$, *then* $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash \mathtt{T}\prec:_{\overline{\mathtt{Z}}}\ \mathtt{S}$.

Proof. 1. Follows by induction on the derivation of subtyping.

2. Follows by induction on well-formed types rules.

3. Follows by induction on expression typing rules.

4. Follows by induction on expression typing rules.

5. Follows by pattern matching rules.

□

Lemma 21 (Type Substitution Preserves Subtyping)**.** *If* $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2\vdash \mathtt{S}<:\mathtt{T}$, *and* $\Delta_1\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ *with* $\Delta_1\vdash\overline{\mathtt{U}}$ *ok and none of* $\overline{\mathtt{X}}$ *appearing in* Δ_1, *then* $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$.

Proof. By induction on the derivation of $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2\vdash \mathtt{S}<:\mathtt{T}$

Case S-REFL: Trivial

Case S-VAR:

If $\mathtt{X}\notin\overline{\mathtt{X}}$, then it is trivial.

If $\mathtt{X}\in\overline{\mathtt{X}}$, $bound_\Delta(\mathtt{X}_i)=\mathtt{N}_i$.

$[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{X}_i=\mathtt{U}_i \quad [\overline{\mathtt{U}}/\overline{\mathtt{X}}]bound_\Delta(\mathtt{X}_i)=[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{N}_i$.

By assumption and Lemma 20, $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash\mathtt{U}_i<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{N}_i$.

Case S-TRANS: By induction hypothesis.

Case S-CLASS: Trivial.

□

Lemma 22 (Pattern-matching Type Substitution Preserves Subtyping)**.** *If* $\Delta\vdash\mathtt{S}<:\mathtt{T}$, $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda\sqsubseteq_\Lambda\Lambda'$, $\Delta\vdash\overline{\mathtt{Y}}<:\overline{\mathtt{P}}$, $\Delta\vdash\overline{\mathtt{W}}$ *ok, and* $\overline{\mathtt{Y}}$ *do not appear in* Λ, *then* $\Delta\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{S}<:[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}$.

Proof. By induction on the derivation of subtyping relation:

Case S-REFL: Trivial.

Case S-VAR:

If $\mathtt{X} \notin \overline{\mathtt{Y}}$, conclusion is thus trivial.

If $\mathtt{X} \in \overline{\mathtt{Y}}$, let $\mathtt{X} = \mathtt{Y}_i$. $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{X} = \mathtt{W}_i$. By Lemma 37, $\Delta \vdash \mathtt{W}_i <: [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{P}_i$.

Case S-TRANS: By induction hypothesis and S-TRANS.

Case S-CLASS: Easy by S-CLASS.

□

Lemma 23 (Type Substitution Preserves Typing)**.** *If* $\Delta_1, \overline{\mathtt{X}} <: \overline{\mathtt{N}}, \Delta_2; \Lambda; \Gamma \vdash \mathtt{e} \in \mathtt{T}$ *and* $\Delta_1 \vdash \overline{\mathtt{U}} <: [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ *where* $\mathtt{e}$ *is fully grounded,* $\Delta_1 \vdash \overline{\mathtt{U}}$ *ok, and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1 *and* Λ*,*
then $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2; \Lambda; [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e} \in [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$*.*

Proof. By induction on the derivation of $\Delta_1, \overline{\mathtt{X}} <: \overline{\mathtt{N}}, \Delta_2; \Lambda; \Gamma \vdash \mathtt{e} \in \mathtt{T}$

Let $\Delta_o = \Delta_1, \overline{\mathtt{X}} <: \overline{\mathtt{N}}, \Delta_2$, $\Delta_n = \Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2$

Case T-VAR: Trivial

Case T-FIELD: $\Delta_o; \Lambda; \Gamma \vdash \mathtt{e}_0.\mathtt{f}_i \in \mathtt{T}_i$

$\Delta_o; \Lambda; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \quad \Delta \vdash \mathit{fields}(\mathit{bound}_{\Delta_o}(\mathtt{T}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$

By induction hypothesis, $\Delta_n; \Lambda; [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0 \in [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0$

By Lemma 25, $\Delta_n \vdash \mathit{bound}_{\Delta_n}([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0) <: [\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathit{bound}_{\Delta_o}(\mathtt{T}_0))$

By Lemma 26,

$\Delta_n \vdash \mathit{fields}(\mathit{bound}_{\Delta_n}([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0)) = \overline{\mathtt{S}}\ \overline{\mathtt{g}}$,

$\mathtt{f}_j = \mathtt{g}_j$, $\mathtt{S}_j = [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_j$ for $j \leq \#(\overline{\mathtt{f}})$.

By T-FIELD, $\Delta_n; \Lambda; [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i \in [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_i$.

Case T-INVK: $\Delta_o \vdash \mathtt{e}_0.\mathtt{n}(\overline{\mathtt{e}}) \in \mathtt{T}$

$\Delta_o; \Lambda; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta_o; \Lambda; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}}$

$\Delta_o; \Lambda \vdash \mathit{mtype}(\mathtt{n}, \mathtt{T}_0) = \overline{\mathtt{T}} \to \mathtt{T} \quad \Delta_o \vdash \overline{\mathtt{S}} <: \overline{\mathtt{T}}$

By induction hypothesis,

$\Delta_n;\Lambda;[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0\in[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0 \quad \Delta_n;\Lambda;[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{e}}\in[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}$

By Lemma 21, $\Delta_n\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}$

By Lemma 28, and that $\mathtt{e}$ is fully grounded, and thus $\mathtt{n}$ must be $\mathtt{m}$ and not η,

$\Delta_n\vdash mtype(\mathtt{n},\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0)=[\overline{\mathtt{U}}/\overline{\mathtt{X}}](\overline{\mathtt{T}}\to\mathtt{T})$

By T-INVK, $\Delta_n;\Lambda;[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathtt{e}_0.\mathtt{n}(\overline{\mathtt{e}}))\in[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$

Case T-NEW: $\Delta_o\vdash\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}\in\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$

$\Delta_o\vdash\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ ok \quad \Delta_o\vdash fields(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=\overline{\mathtt{S}}\ \overline{\mathtt{f}}$

$\Delta_o;\Lambda;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}}' \quad \Delta_o\vdash\overline{\mathtt{S}}'<:\overline{\mathtt{S}}$

By Lemma 24, $\Delta_n\vdash\mathtt{C<}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}\mathtt{>}\ ok$

By definition of *fields*:

$CT(\mathtt{C})=\mathtt{class\ C<}\overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{N}}\mathtt{><:S\ \{}\overline{\mathtt{D}}\ \overline{\mathtt{g}}\ \mathtt{...\}} \quad \Delta_n\vdash fields(bound_{\Delta_o}([\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\mathtt{S}))=\overline{\mathtt{D}}'\ \overline{\mathtt{g}}'$

$\overline{\mathtt{S}}\ \overline{\mathtt{f}}=[\overline{\mathtt{T}}/\overline{\mathtt{Y}}](\overline{\mathtt{D}}\ \overline{\mathtt{g}},\ \overline{\mathtt{D}}'\ \overline{\mathtt{g}}')$

Since $\overline{\mathtt{X}}$ cannot appear in $\mathtt{S}$, by the definition of *bound*,

$bound_{\Delta_o}([\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\mathtt{S})=bound_{\Delta_n}([\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\mathtt{S})=[\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\mathtt{S}$

By Lemma 27, $\Delta_n\vdash fields([\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\mathtt{S})=\overline{\mathtt{D}}'\ \overline{\mathtt{g}}'$

It follows that, $\Delta_n\vdash fields([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\ \overline{\mathtt{f}}$.

By Lemma 23 and 21,

$\Delta_n;\Lambda;[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{e}}\in[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}' \quad \Delta_n\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}'<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}$

The conclusion follows from T-NEW.

□

Lemma 24. *If* $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2\vdash\mathtt{T}\ ok$*, and* $\Delta_1\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ *with* $\Delta_1\vdash\overline{\mathtt{U}}\ ok$*, none of* $\overline{\mathtt{X}}$ *appearing in* Δ_1*, then* $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}\ ok$.

Proof. By induction on the derivation of $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2\vdash\mathtt{T}\ ok$

Case WF-OBJECT: Trivial.

Case WF-VAR: If $\mathtt{X}$ not in $\overline{\mathtt{X}}$, then it's trivial. Otherwise, $\mathtt{X}{=}\mathtt{X}_i$, $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{X}_i{=}\mathtt{U}_i$, by assumption, $\Delta_1{\vdash}\mathtt{U}_i$ *ok*. And by Lemma 20, conclusion follows.

Case WF-CLASS: By induction hypothesis and Lemma 21.

□

Lemma 25. *Suppose* $\Delta_1,\overline{\mathtt{X}}{<:}\overline{\mathtt{N}},\Delta_2{\vdash}\mathtt{T}\,ok$*, and* $\Delta_1{\vdash}\overline{\mathtt{U}}{<:}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ *with* $\Delta_1{\vdash}\overline{\mathtt{U}}$ *ok, and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1*. Then,* $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2{\vdash}bound_{\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}){<:}[\overline{\mathtt{U}}/\overline{\mathtt{X}}](bound_{\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2}(\mathtt{T}))$

Proof. If $\mathtt{T}$ is a non-variable type, then the conclusion is trivial.

If $\mathtt{T}$ is a variable type, but $\mathtt{T}{\notin}\overline{\mathtt{X}}$, then the conclusion is also trivial.

If $\mathtt{T}{\in}\overline{\mathtt{X}}$,

$bound_{\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}){=}\mathtt{U}_i \quad [\overline{\mathtt{U}}/\overline{\mathtt{X}}](bound_{\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2}(\mathtt{T})){=}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{N}_i$

By assumption and Lemma 20, conclusion follows.

□

Lemma 26. *If* $\Delta{\vdash}\mathtt{S}{<:}\mathtt{T}$*, and* $\Delta{\vdash}fields(bound_\Delta(\mathtt{T})){=}\overline{\mathtt{T}}\ \overline{\mathtt{f}}$*, then* $\Delta{\vdash}fields(bound_\Delta(\mathtt{S})){=}\overline{\mathtt{S}}\ \overline{\mathtt{g}}$*, and* $\mathtt{S}_i{=}\mathtt{T}_i$*, and* $\mathtt{g}_i{=}\mathtt{f}_i$ *for all* $i \leq \#(\overline{\mathtt{f}})$*.*

Proof. By induction on the derivation of $\Delta{\vdash}\mathtt{S}{<:}\mathtt{T}$.

Case S-REFL: Trivial.

Case S-VAR: $bound_\Delta(\mathtt{S}){=}bound_\Delta(\mathtt{T})$. Conclusion follows.

Case S-TRANS: By induction hypothesis.

Case S-CLASS: $\Delta{\vdash}\mathtt{C}{<}\overline{\mathtt{T}}{>}{<:}[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}$

By FD-CLASS,

$\mathtt{class\ C}{<}\overline{\mathtt{X}}{\lhd}\overline{\mathtt{N}}{>}{\lhd}\mathtt{T}\ \{\overline{\mathtt{U}}\ \overline{\mathtt{f}}\ \mathtt{...}\} \qquad \Delta{\vdash}fields(bound_\Delta([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T})){=}\overline{\mathtt{T}}\ \overline{\mathtt{f}}$

$\Delta{\vdash}fields(\mathtt{C}{<}\overline{\mathtt{T}}{>}){=}\overline{\mathtt{T}}\ \overline{\mathtt{f}},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}}\ \overline{\mathtt{f}}$

Conclusion is obvious.

□

Lemma 27. *If* $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2\vdash \mathit{fields}(\mathtt{T})=\overline{\mathtt{S}}\ \overline{\mathtt{f}}$ $\Delta_1\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$*, where* $\Delta_1\vdash\overline{\mathtt{U}}$ *ok, and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1*, then* $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash \mathit{fields}([\overline{\mathtt{U}}/\overline{\mathtt{T}}]\mathtt{T})=[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\ \overline{\mathtt{f}},\overline{\mathtt{S}}'\ \overline{\mathtt{f}}'$*, for some* $\overline{\mathtt{S}}'$ *and* $\overline{\mathtt{f}}'$*.*

Proof. We prove by case analysis on the definition of *fields*.

Case FD-OBJ: $\mathtt{T}=\mathtt{Object}$. Trivial.

Case FD-CLASS: $\mathtt{T}=\mathtt{C}<\overline{\mathtt{T}}>$

$CT(\mathtt{C})=\mathtt{class\ C}<\overline{\mathtt{Y}}\lhd\overline{\mathtt{N}}>\lhd\mathtt{S\ \{\ \overline{S}\ \overline{f};\ \}}\quad \Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{U}},\Delta_2\vdash \mathit{fields}(\mathit{bound}_{\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{U}},\Delta_2}([\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\mathtt{S}))=\overline{\mathtt{D}}\ \overline{\mathtt{g}}$

By induction hypothesis, for some $\overline{\mathtt{D}}'$, $\overline{\mathtt{g}}'$,

$\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash \mathit{fields}([\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathit{bound}_{\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2}([\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\mathtt{S})))=[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{D}}\ \overline{\mathtt{g}},\ \overline{\mathtt{D}}'\ \overline{\mathtt{g}}'$.

By Lemma 25,

$\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash \mathit{bound}_{\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}([\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\mathtt{S})<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathit{bound}_{\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2}([\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\mathtt{S}))$

Conclusion then follows from Lemma 26.

□

Lemma 28. *If* $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{T})=\overline{\mathtt{V}}\to\mathtt{V}_0$*,* $\Delta_1\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$*, where* $\Delta_1\vdash\overline{\mathtt{U}}$ *ok, and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1 *or* Λ*, then* $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T})=[\overline{\mathtt{U}}/\overline{\mathtt{X}}](\overline{\mathtt{V}}\to\mathtt{V}_0)$.

Proof. By induction on the derivation of $\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{T})=\overline{\mathtt{V}}\to\mathtt{V}_0$.

Let $\Delta_o=\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2$, $\Delta_n=\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2$

Case MT-VAR-S: $\Delta_o;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{X})=\overline{\mathtt{V}}\to\mathtt{V}_0$

If $\mathtt{X}$ not in $\overline{\mathtt{X}}$, easy.

If $\mathtt{X}$ is in $\overline{\mathtt{X}}$, let $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{X}=\mathtt{U}_i$

$\Delta_o;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathit{bound}_{\Delta_o}(\mathtt{X}))=\overline{\mathtt{V}}\to\mathtt{V}_0$

By induction hypothesis, $\Delta_n;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathit{bound}_{\Delta_o}(\mathtt{X}))=[\overline{\mathtt{U}}/\overline{\mathtt{X}}](\overline{\mathtt{V}}\to\mathtt{V}_0)$

By Lemma 25, $\Delta_n\vdash \mathit{bound}_{\Delta_n}(\mathtt{U}_i)<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathit{bound}_{\Delta_o}(\mathtt{X}))$.

By Lemma 34 $\Delta_n;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathit{bound}_{\Delta_n}(\mathtt{U}_i))=[\overline{\mathtt{U}}/\overline{\mathtt{X}}](\overline{\mathtt{V}}\to\mathtt{V}_0)$

By MT-VAR-S again, the conclusion follows.

Case MT-CLASS-S, MT-SUPER-S: Trivial through type substitutions.

Case MT-CLASS-R:

$\Delta_o;\Lambda\vdash \mathit{specialize}(\mathtt{m},\ \Lambda_d)=\Lambda_r \quad \Delta_o;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda\Lambda_d$

By Lemma 29,

$\Delta_n;\Lambda\vdash \mathit{specialize}(\mathtt{m},\ \Lambda_d)=\Lambda'_r \quad \Delta_n;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash\Lambda'_r\sqsubseteq_\Lambda\Lambda_d$

for some $\overline{\mathtt{W}}'$. Conclusion follows by MT-CLASS-R.

Case MT-SUPER-R:

$CT(\mathtt{C})=\mathtt{class\ C<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{>\lhd T\ \{...}\ \ \overline{\mathfrak{M}}\mathtt{\}}$

for all $\mathfrak{M}_i\in\overline{\mathfrak{M}}$, $\Lambda_d=\mathit{reflectiveEnv}(\mathfrak{M}_i)$

$\Delta_o;\Lambda\vdash \mathit{specialize}(\mathtt{m},\ \Lambda_d)=\Lambda_r \quad \Delta_o\vdash \mathit{disjoint}(\Lambda_r,\ \Lambda_d)$

By Lemma 30,

$\Delta_n;\Lambda\vdash \mathit{specialize}(\mathtt{m},\ \Lambda_d)=\Lambda'_r \quad \Delta_n\vdash \mathit{disjoint}(\Lambda'_r,\ \Lambda_d)$

Conclusion follows from induction hypothesis.

□

Lemma 29. *If* $\Delta_1,\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}},\Delta_2;\Lambda\vdash \mathit{specialize}(\mathtt{m},\ \Lambda_d)=\Lambda_r$, $\Delta_1,\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}},\Delta_2;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda\Lambda_d$, $\Delta_1\vdash\overline{\mathtt{T}}\mathtt{<:}[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$, *where* $\Delta_1\vdash\overline{\mathtt{T}}$ *ok, and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1 *or* Λ, *then* $\Delta_1,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta_2;\Lambda\vdash \mathit{specialize}(\mathtt{m},\ \Lambda_d)=\Lambda'_r$, *and* $\Delta_1,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta_2;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash\Lambda'_r\sqsubseteq_\Lambda\Lambda_d$, *where* $\overline{\mathtt{W}}'=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{W}}$.

Proof. Let $\Delta_o=\Delta_1,\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}},\Delta_2$, $\Delta_n=\Delta_1,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta_2$

By the definition of *specialize*,

$\Lambda_d=\langle R_p,oR_n\rangle \qquad R_p=(\mathtt{T}_i,\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to U}) \quad R_n=(\mathtt{T}_j,\ \overline{\mathtt{V}}\mathtt{\to V})$

$\Delta_o;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{T}_i)=\overline{\mathtt{U}}'\mathtt{\to U}' \quad R'_p=(\mathtt{T}_i,\ \overline{\mathtt{U}}'\mathtt{\to U}')$

By Lemma 28, $\Delta_n;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_i)=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{U}}'\mathtt{\to U}')$

Let $R''_p=([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_i,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}}'\mathtt{\to U}')$

By Lemma 31, $\Delta_n;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash R''_p\sqsubseteq_R R_p$, where $\overline{\mathtt{W}}'=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{W}}$.

Case o=+, $\Delta_o;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{T}_j)=\overline{\mathtt{V}}'\mathtt{\to V}'_0$,

$R'_n=(\mathtt{T}_j,\ \overline{\mathtt{V}}'\mathtt{\to V}_0)$

By Lemma 28,

$\Delta_n;\Lambda\vdash mtype(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_j)=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{V}'}\to\mathtt{V}')$

Let $R''_n=([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_j,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{V}'}\to\mathtt{V}'))$

By Lemma 31, $\Delta_n;[\overline{\mathtt{W}'}/\overline{\mathtt{Y}}]\vdash R''_n\sqsubseteq_R R_n$.

Case o=-,

$R'_n=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R_n$.

Let $R''_n=[\overline{\mathtt{W}'}/\overline{\mathtt{Y}}]R_n$, clearly, $\Delta_n;[\overline{\mathtt{W}'}/\overline{\mathtt{Y}}]\vdash R_n\sqsubseteq_R R''_n$.

Since none of $\overline{\mathtt{X}}$ appears in Λ, $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_i=\mathtt{T}_i$, $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_j=\mathtt{T}_j$.

Then let $\Delta_n;\Lambda\vdash specialize(\mathtt{m},\ \Lambda_d)=\langle R''_p,R''_n\rangle$. By SB-$\Lambda$, $\Delta_n;[\overline{\mathtt{W}'}/\overline{\mathtt{Y}}]\vdash\langle R''_p,R''_n\rangle\sqsubseteq_\Lambda\Lambda_d$.

□

Lemma 30. *If* $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2;\Lambda\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda_r$, $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2\vdash disjoint(\Lambda_r,\ \Lambda_d)$, $\Delta_1\vdash\overline{\mathtt{T}}<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$, *where* $\Delta_1\vdash\overline{\mathtt{T}}$ *ok, and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1 *or* Λ, *then* $\Delta_1,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta_2;\Lambda\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda'_r$, *and* $\Delta_1,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta_2\vdash disjoint(\Lambda'_p,\ \Lambda_d)$.

Proof. Let $\Delta_o=\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2$, $\Delta_n=\Delta_1,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta_2$

By the definition of *specialize*,

$\Lambda_d=\langle R_p,oR_n\rangle$ $\quad R_p=(\mathtt{T}_i,\ <\overline{\mathtt{Y}}<\overline{\mathtt{P}}>\overline{\mathtt{U}}\to\mathtt{U})$ $\quad R_n=(\mathtt{T}_j,\ \overline{\mathtt{V}}\to\mathtt{V})$

$\Delta_o;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{T}_i)=\overline{\mathtt{U}'}\to\mathtt{U}'$ $\quad R'_p=(\mathtt{T}_i,\ \overline{\mathtt{U}'}\to\mathtt{U}')$

By Lemma 28,

$\Delta_n;\Lambda\vdash mtype(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_i)=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{U}'}\to\mathtt{U}')$

Let $R''_p=([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_i,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}'}\to\mathtt{U}')$

Case o=+, $\Delta_o;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{T}_j)=\overline{\mathtt{V}'}\to\mathtt{V}'_0$,

$R'_n=(\mathtt{T}_j,\ \overline{\mathtt{V}'}\to\mathtt{V}_0)$

By Lemma 28,

$\Delta_n;\Lambda\vdash mtype(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_j)=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{V}'}\to\mathtt{V}')$

Let $R''_n=([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_j,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{V}'}\to\mathtt{V}'))$

Case o=-,

R'_n=$[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R_n$.

Let R''_n=$[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]R_n$, clearly, $\Delta_n;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash R_n \sqsubseteq_R R''_n$.

Since none of $\overline{\mathtt{X}}$ appears in Λ, $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_i$=$\mathtt{T}_i$, $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_j$=$\mathtt{T}_j$. The let $\Delta_n;\Lambda\vdash \mathit{specialize}(\mathtt{m},\ \Lambda_d)=\langle R''_p, R''_n\rangle$.

The by Lemma 33 and DS-Λ, if there was originally a pair of mutually exclusive range conditions in Λ_r, then there is a pair of mutually exclusive range conditions in Λ'_r.

□

Lemma 31 (Substitution Preserves Single Range Containment)**.** *If* $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_1 \sqsubseteq_R R_2$, $\Delta_1\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$, *where* $\Delta_1\vdash\overline{\mathtt{U}}$ *ok, and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1, *none of* $\overline{\mathtt{X}}$ *appears on* $\overline{\mathtt{Y}}$, *then* $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash R_1 \sqsubseteq_R R_2$, *where* $\overline{\mathtt{W}}'$=$[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{W}}$.

Proof. Let Δ_o=$\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2$, Δ_n=$\Delta_1,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta_2$

By SB-R,

R_1=$(\mathtt{T}_1,\ <\overline{\mathtt{X}}<\overline{\mathtt{Q}}>\overline{\mathtt{U}}\rightarrow\mathtt{U}_0)$ $\quad R_2$=$(\mathtt{T}_2,\ \overline{\mathtt{V}}\rightarrow\mathtt{V}_0)$

$\Delta_o\vdash\mathtt{T}_2<:\mathtt{T}_1$ $\quad \Delta'_o$=$\Delta_o,\overline{\mathtt{X}}<:\overline{\mathtt{Q}},\overline{\mathtt{Y}}<:\overline{\mathtt{P}}$ $\quad \Delta'_o;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash \mathit{unify}(\mathtt{U}_0{:}\overline{\mathtt{U}},\ \mathtt{V}_0{:}\overline{\mathtt{V}})$

By Lemma 21, $\Delta_n\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_2<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_1$

By UNI, $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{U}_0{:}\overline{\mathtt{U}}$=$[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{V}_0{:}\overline{\mathtt{V}}$, for all $\mathtt{Y}_i\in\overline{\mathtt{Y}},\Delta_o\vdash\mathtt{W}_i\prec:_{\overline{\mathtt{Y}}}\mathtt{Y}_i$

Clearly, $[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{U}_0{:}\overline{\mathtt{U}}$=$[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{V}_0{:}\overline{\mathtt{V}}$

By Lemma 32, $\Delta_n\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{W}_i\prec:_{\overline{\mathtt{Y}}}\mathtt{Y}_i$ for all $\mathtt{W}_i$.

It follows that $\Delta_n;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash \mathit{unify}([\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{U}_0{:}\overline{\mathtt{U}},\ [\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{V}_0{:}\overline{\mathtt{V}})$

Conclusion follows from SB-R. □

Lemma 32 (Substitution Preserves Pattern Type Pattern Match)**.** $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2\vdash\mathtt{W}\prec:_{\overline{\mathtt{Y}}}\mathtt{Y}$, *where* $\Delta_1\vdash\overline{\mathtt{U}}$ *ok, and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1, *none of* $\overline{\mathtt{X}}$ *appears on* $\overline{\mathtt{Y}}$, *then* $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{W}\prec:_{\overline{\mathtt{Y}}}\mathtt{Y}$.

Proof. We prove by induction on pattern matching rules PM-*.

Case PM-REFL: Trivial

Case PM-CL: By induction hypothesis.

Case PM-CL-S: By induction hypothesis.

Case PM-VAR: By inspection of definition of *bound*, $bound_\Delta([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{W})=[\overline{\mathtt{U}}/\overline{\mathtt{X}}]bound_\Delta(\mathtt{W})$. Conclusion follows by induction hypothesis.

Case PM-PVARS: By induction hypothesis and inspection of definition of *bound*, similar to above case.

□

Lemma 33 (Substitution Preserves Range Mutual Exclusion)**.** *If* $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2\vdash oR_1\otimes o'R_2$, $\Delta_1\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$, *where* $\Delta_1\vdash\overline{\mathtt{U}}$ *ok, and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1,R_1, *or* R_2, *then* $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash oR_1\otimes o'R_2$.

Proof. Let $\Delta_o=\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2$, $\Delta_n=\Delta_1,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta_2$

We prove by case analysis of ME-1 and ME-2.

Case ME-1:

$R_1=(\mathtt{T}, <\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}>\overline{\mathtt{U}}\to\mathtt{U}_0)$ $R_2=(\mathtt{S}, <\overline{\mathtt{Z}}\lhd\overline{\mathtt{Q}}>\overline{\mathtt{V}}\to\mathtt{V}_0)$

$\Delta_o\vdash\mathtt{T}<:\mathtt{S}$ or $\mathtt{S}<:\mathtt{T}$ $\Delta_o'=\Delta_o,\overline{\mathtt{Y}}<:\overline{\mathtt{P}},\overline{\mathtt{Z}}<:\overline{\mathtt{Q}}$

For all $\overline{\mathtt{W}}$, $\Delta_o';[\overline{\mathtt{W}}/(\overline{\mathtt{Y}}:\overline{\mathtt{Z}})]\vdash unify(\overline{\mathtt{U}},\overline{\mathtt{V}})$ implies $[\overline{\mathtt{W}}/(\overline{\mathtt{Y}}:\overline{\mathtt{Z}})]\mathtt{U}_0\neq[\overline{\mathtt{W}}/(\overline{\mathtt{Y}}:\overline{\mathtt{Z}})]\mathtt{V}_0$

Since $\overline{\mathtt{X}}$ do not appear in R_1 or R_2, by Lemma 21, $\Delta_n\vdash\mathtt{T}<:\mathtt{S}$ or $\mathtt{S}<:\mathtt{T}$.

By definition of UNI and the PM-* rules,

for all $\overline{\mathtt{W}}$, $\Delta_n';[\overline{\mathtt{W}}/(\overline{\mathtt{Y}}:\overline{\mathtt{Z}})]\vdash unify(\overline{\mathtt{U}},\overline{\mathtt{V}})$ implies $[\overline{\mathtt{W}}/(\overline{\mathtt{Y}}:\overline{\mathtt{Z}})]\mathtt{U}_0\neq[\overline{\mathtt{W}}/(\overline{\mathtt{Y}}:\overline{\mathtt{Z}})]\mathtt{V}_0$

Case ME-2:

$R_1=(\mathtt{T}, <\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}>\overline{\mathtt{U}}\to\mathtt{U}_0)$ $R_2=(\mathtt{S}, <\overline{\mathtt{Z}}\lhd\overline{\mathtt{Q}}>\overline{\mathtt{V}}\to\mathtt{V}_0)$

$\Delta_o'=\Delta_o,\overline{\mathtt{Y}}<:\overline{\mathtt{P}},\overline{\mathtt{Z}}<:\overline{\mathtt{Q}}$ $\Delta_o';[\overline{\mathtt{W}}/(\overline{\mathtt{Y}}:\overline{\mathtt{Z}})]\vdash R_1\sqsubseteq_R R_2$

By Lemma 31 and the fact that $\overline{\mathtt{X}}$ do not appear in R_1 or R_2, $\Delta_n,\overline{\mathtt{Y}}<:\overline{\mathtt{P}},\overline{\mathtt{Z}}<:\overline{\mathtt{Q}}\vdash R_1\sqsubseteq_R R_2$.

□

Lemma 34. *If* $\Delta;\Lambda\vdash mtype(\mathtt{m},\mathtt{T})=\overline{\mathtt{U}}\to\mathtt{U}_0$, $\Delta\vdash\mathtt{S}<:\mathtt{T}$, *then* $\Delta;\Lambda\vdash mtype(\mathtt{m},\mathtt{S})=\overline{\mathtt{U}}\to\mathtt{U}_0$.

Proof. By induction on the derivation of $\Delta\vdash\mathtt{S}<:\mathtt{T}$

Case S-REFL: Trivial.

Case S-VAR: $\Delta \vdash \mathtt{X} <: \Delta(\mathtt{X})$

MT-VAR-R1 and MT-VAR-R2 do not apply, since $\mathtt{m} \neq \eta$.

By definition, $bound_\Delta(\mathtt{X}) = \Delta(\mathtt{X})$. Conclusion follows from MT-VAR-S.

Case S-TRANS: Easy by induction hypothesis.

Case S-CLASS: $\mathtt{T} = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}$, $\mathtt{S} = \mathtt{C}\langle\overline{\mathtt{T}}\rangle$

$\Delta;\Lambda \vdash mtype(\mathtt{m}, \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = \overline{\mathtt{V}} \to \mathtt{V}_0$ can oly be retrieved via MT-CLASS-* and MT-SUPER-*.

If retrieved using MT-CLASS-R:

$CT(\mathtt{C}) = \mathtt{class\ C}\langle\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\rangle \lhd \mathtt{T}\ \{\ \ldots\ \ \overline{\mathfrak{M}}\ \}$

$\mathfrak{M}_i = \langle\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\rangle \mathtt{for}(\mathbb{M}_p\,; o\mathbb{M}_f)\ \mathtt{S}_0\ \eta\ (\overline{\mathtt{S}}\ \overline{\mathtt{x}})\ \{\uparrow \mathtt{e};\}\ \mathfrak{M}_i \in \overline{\mathfrak{M}}$

$\mathbb{M}_p = \mathtt{U}_0\ \eta\ (\overline{\mathtt{U}})\ :\ \ \mathtt{X}_i\mathtt{.methods}$ $\quad \mathbb{M}_f = \mathtt{U'}_0\ \eta\ (\overline{\mathtt{U}'})\ :\ \ \mathtt{X}_j\mathtt{.methods}$

$R_p = (\mathtt{X}_i,\ \overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}(\overline{\mathtt{U}} \to \mathtt{U}_0))$ $\quad R_n = (\mathtt{X}_j,\ \overline{\mathtt{U}'} \to \mathtt{U'}_0)$

$\Lambda' = \langle R_p, o R_n \rangle$ $\quad \Lambda_d = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Lambda'$

$\Delta;\Lambda \vdash specialize(\mathtt{m}, \Lambda_d) = \Lambda_r$ $\quad \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}] \vdash \Lambda_r \sqsubseteq_\Lambda \Lambda_d$

$\overline{\mathtt{V}} = [\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{S}}$ $\quad \mathtt{V}_0 = [\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{S}_0$

By T-METH-R and the definition of *override*,

$\Delta' = \overline{\mathtt{X}} <: \overline{\mathtt{N}}, \overline{\mathtt{Y}} <: \overline{\mathtt{P}}$ $\quad \Delta' \vdash \overline{\mathtt{N}}, \overline{\mathtt{P}}$ ok $\quad \Delta';\Lambda' \vdash override(\eta, \mathtt{T}, \overline{\mathtt{S}} \to \mathtt{S}_0)$

$\Delta';\Lambda' \vdash mtype(\eta, \mathtt{T}) = \overline{\mathtt{S}} \to \mathtt{S}_0$ $\quad \Delta' \vdash validRange(\Lambda', \mathtt{T})$

What we need to show now is that in all cases of MT-*,

$\Delta';\Lambda' \vdash mtype(\eta, \mathtt{T}) = \overline{\mathtt{S}} \to \mathtt{S}_0$ implies $\Delta;\Lambda \vdash mtype(\mathtt{m}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{S}} \to \mathtt{S}_0)$

- MT-VAR-R1: $\Delta';\Lambda' \vdash mtype(\eta, \mathtt{X}_i) = \overline{\mathtt{S}} \to \mathtt{S}_0$

  By definition of *specialize*,

  $\Lambda_d = \langle [\overline{\mathtt{T}}/\overline{\mathtt{X}}]R_p, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]R_n \rangle$ $\quad [\overline{\mathtt{T}}/\overline{\mathtt{X}}]R_p = (\mathtt{T}_i,\ \overline{\mathtt{Y}} \lhd [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{P}}([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}} \to [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0))$

  $\Delta;\Lambda \vdash mtype(\mathtt{m}, \mathtt{T}_i) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{V}} \to \mathtt{V}_0)$

  Since $\overline{\mathtt{Y}}$ do not appear in $\mathtt{V}_0$ or $\overline{\mathtt{V}}$, $\Delta;\Lambda \vdash mtype(\mathtt{m}, \mathtt{T}_i) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{V}} \to \mathtt{V}_0)$.

  By $\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}] \vdash \Lambda_r \sqsubseteq_\Lambda [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Lambda_d$, $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\mathtt{S}_0 : \overline{\mathtt{S}}) = [\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{V}_0 : \overline{\mathtt{V}})$

- MT-VAR-R2: Using similar technique to MT-VAR-R1. The information we use from the definition of *specialize* is when $o = +$, and $\Delta;\Lambda \vdash mtype(\mathtt{m}, \mathtt{T}_j)$ is similarly defined.

- MT-VAR-S: By the fact that if $\mathtt{X}$ is not the reflective type of R_p, then $\mathtt{T}_i$ is not the reflective type of $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]R_p$, either. And conclusion follows from induction hypothesis.
- MT-CLASS-R: $\Delta';\Lambda' \vdash mtype(\eta, \mathtt{D}\langle\overline{\mathtt{Q}}\rangle) = \overline{\mathtt{S}} \rightarrow \mathtt{S}_0$.

  $CT(\mathtt{D}) = \mathtt{class\ D}\langle\overline{\mathtt{Z}} \lhd \overline{\mathtt{R}}\rangle \lhd \mathtt{R\ \{\ ...\ \ } \overline{\mathfrak{M}}\mathtt{\}}$

  $\mathfrak{M}_i \in \overline{\mathfrak{M}} \quad \mathfrak{M}_i = \langle\overline{\mathtt{Y}}' \lhd \overline{\mathtt{P}}'\rangle\mathtt{for\ S}'_0\ \eta\ \mathtt{(}\overline{\mathtt{S}}'\ \overline{\mathtt{x}}\mathtt{)\ \{\ ...\}}$

  $\Lambda_d{}' = [\overline{\mathtt{Q}}/\overline{\mathtt{Z}}](reflectiveEnv(\mathfrak{M}_i)) \quad \Delta';[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}'] \vdash \Lambda' \sqsubseteq_\Lambda \Lambda_d{}'$

  $\overline{\mathtt{S}} = [\overline{\mathtt{Q}}/\overline{\mathtt{Z}}][\overline{\mathtt{W}}'/\overline{\mathtt{Y}}']\overline{\mathtt{S}}' \qquad \mathtt{S}_0 = [\overline{\mathtt{Q}}/\overline{\mathtt{Z}}][\overline{\mathtt{W}}'/\overline{\mathtt{Y}}']\mathtt{S}_0{}'$

  By Lemma 31 and Lemma 20, $\Delta,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta';[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}'/\overline{\mathtt{Y}}'] \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Lambda' \sqsubseteq_\Lambda [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Lambda'_d$

  By Lemma 38, $\Delta,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta';[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}'][\overline{\mathtt{W}}/\overline{\mathtt{Y}}] \vdash \Lambda_r \sqsubseteq_\Lambda [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Lambda_d{}'$

  Since $\overline{\mathtt{Y}}$ do not appear anywhere in Λ_r, or $\Lambda_d{}'$, $\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}'][\overline{\mathtt{W}}/\overline{\mathtt{Y}}] \vdash \Lambda_r \sqsubseteq_\Lambda [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Lambda_d{}'$

  It follows from MT-CLASS-R that $\Delta;\Lambda \vdash mtype(\mathtt{m}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D}\langle\overline{\mathtt{Q}}\rangle) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{S}} \rightarrow \mathtt{S}_0)$
- MT-SUPER-R: By induction hypothesis.
- MT-CLASS-S and MT-SUPER-S: Do not apply.

If retrieved using MT-SUPER-S or MT-SUPER-R, then the conclusion is obvious from these definitions.

□

Lemma 35. *If* $\Delta;\Lambda \vdash specialize(\mathtt{m}, \Lambda_d) = \Lambda_r$, $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}] \vdash \Lambda_r \sqsubseteq_\Lambda \Lambda_d$, *and* $\Delta;\Lambda;\Gamma \vdash \mathtt{e} \in \mathtt{T}$.
Then $\Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Gamma \vdash [\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta]\mathtt{e} \in [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}$.

Proof. By induction on the derivation of $\Delta;\Lambda;\Gamma \vdash \mathtt{e} \in \mathtt{T}$

Case T-VAR: Trivial.

Case T-FIELD: $\Delta;\Lambda;\Gamma \vdash \mathtt{e}_0\mathtt{.f}_i \in \mathtt{T}_i$.

$\Delta;\Lambda;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \quad \Delta \vdash fields(bound_\Delta(\mathtt{T}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$

By induction hypothesis, $\Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Gamma \vdash [\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta]\mathtt{e}_0 <: [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}_0$

By Lemma 26 and Lemma 20, $\Delta \vdash fields([\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}_0) = [\overline{\mathtt{W}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}\ \overline{\mathtt{f}}, \overline{\mathtt{S}}\ \overline{\mathtt{g}}$.

By T-FIELD, $\Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Gamma \vdash [\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta]\mathtt{e}_0\mathtt{.f}_i \in [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}_i$

Case T-INVK: $\Delta;\Lambda;\Gamma \vdash \mathtt{e}_0\mathtt{.n(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{T}$

$\Delta;\Lambda;\Gamma\vdash \mathtt{e}_0\in\mathtt{T}_0 \quad \Delta;\Lambda\vdash mtype(\mathtt{n},\ \mathtt{T}_0)=\overline{\mathtt{T}}\to\mathtt{T} \quad \Delta;\Lambda;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}} \quad \Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{T}}$

By induction hypothesis, $\Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Gamma\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta]\overline{\mathtt{e}}\in[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{S}}$

By Lemma 22, $\Delta\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{S}}<:[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{T}}$.

Also by induction hypothesis, $\Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Gamma\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta]\mathtt{e}_0\in[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}_0$

If $\mathtt{n}=\mathtt{m}$, or $\mathtt{n}=\eta$, $[\mathtt{m}/\eta]\mathtt{n}=\mathtt{m}$, by Lemma 36, $\Delta;\Lambda\vdash mtype(\mathtt{m},\ [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}_0)=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{T}}\to\mathtt{T})$.

If $\mathtt{n}=\mathtt{m}'$, where $\mathtt{m}'\neq\mathtt{m}$, $\overline{\mathtt{Y}}$ do not appear in $\overline{\mathtt{T}}$ or $\mathtt{T}$, thus,

$\Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Gamma\vdash mtype(\mathtt{m}',\ [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T})=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{T}}\to\mathtt{T})$

In either case, it follows from T-INVK that $\Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Gamma\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta](\mathtt{e}_0\mathtt{.n(}\overline{\mathtt{e}}\mathtt{)})\in[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}$.

Case T-NEW: $\Delta;\Lambda;\Gamma\vdash\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}\in\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$

$\Delta\vdash\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ ok $\quad \Delta\vdash fields(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=\overline{\mathtt{U}}\ \overline{\mathtt{f}} \quad \Delta;\Lambda;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}} \quad \Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{U}}$

By induction hypothesis and Lemma 22,

$\Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Gamma\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta]\overline{\mathtt{e}}\in[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{S}} \quad \Delta\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{S}}<:[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}$

By definition of *fields* and the fact that its definition does not involve any pattern matching variables, $\Delta\vdash fields([\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}\ \overline{\mathtt{f}}$

Conclusion follows from T-NEW.

□

Lemma 36. *Suppose* $\Delta;\Lambda\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda_r$, $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda\Lambda_d$. *If* $\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{T})=\overline{\mathtt{S}}\to\mathtt{S}$, *then* $\Delta;\Lambda\vdash mtype(\mathtt{m},\ [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T})=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{(}\overline{\mathtt{S}}\to\mathtt{S)}$.

Proof. By induction on the derivation of $\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{T})=\overline{\mathtt{S}}\to\mathtt{S}$:

Case MT-VAR-R1 and MT-VAR-R2 do not apply.

Case MT-VAR-S: $\mathtt{T}=\mathtt{X}$.

If $\mathtt{X}\notin\overline{\mathtt{Y}}$, then $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{X}=\mathtt{X}$. It is also impossible from method typing rules for $\overline{\mathtt{Y}}$ to appear in the method definitions of $bound_\Delta(\mathtt{X})$, or its method types. The conclusion follows naturally.

If $\mathtt{X}\in\overline{\mathtt{Y}}$, $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{X}=\mathtt{W}_i$, $\Delta;\Lambda\vdash mtype(\mathtt{m},\ bound_\Delta(\mathtt{Y}_i))=\overline{\mathtt{S}}\to\mathtt{S}$, $\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{Y}_i)=\overline{\mathtt{S}}\to\mathtt{S}$.

It follows from induction hypothesis and $bound_\Delta(\mathtt{Y}_i)=\mathtt{P}_i$ that,

$\Delta;\Lambda\vdash mtype(\mathtt{m},\ [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{P}_i)=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{S}}\to\mathtt{S})$.

By Lemma 37, $\Delta\vdash\mathtt{W}_i<:[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{P}_i$.

By Lemma 34, $\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{W}_i)=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{S}}\to\mathtt{S})$.

Case MT-CLASS-S, MT-SUPER-S: Eas following type substitution.

Case MT-CLASS-R: Conclusion follows easily from the definition of *specialize*.

Case MT-SUPER-R: By induction hypothesis.

□

Lemma 37 (Unification Mapping Preserves Type Variable Bounds)**.** *If* $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda\sqsubseteq_\Lambda\Lambda'$*, and* $\overline{\mathtt{Y}}$ *do not appear anywhere in* Λ*,* $\Lambda'=\langle R'_p,o'R'_n\rangle$*, where* $R'_p=(\mathtt{T},\ <\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}>\overline{\mathtt{U}}\to\mathtt{U})$*, then* $\Delta\vdash\mathtt{W}_i<:[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{P}_i$*, for all* $\mathtt{W}_i\in\overline{\mathtt{W}}$*.*

Proof. Let $\Lambda=\langle R_p,oR_n\rangle$, where $R_p=(\mathtt{S},\ \overline{\mathtt{V}}\to\mathtt{V})$, $\Delta'=\Delta,\overline{\mathtt{Y}}<:\overline{\mathtt{P}}$.

By SB-R, $\Delta';[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash unify(\mathtt{U}:\overline{\mathtt{U}},\ \mathtt{V}:\overline{\mathtt{V}})$.

By UNI, $\Delta'\vdash\mathtt{W}_i\prec:_{\overline{\mathtt{Y}}}\mathtt{Y}_i$ for all $\mathtt{W}_i\in\overline{\mathtt{W}}$. We inspect the rules of PM-*:

Since $\overline{\mathtt{Y}}$ cannot appear in $\overline{\mathtt{W}}$, the first rule that applies is PM-VAR:

$bound_{\Delta'}(\mathtt{W}_i)=\mathtt{C}<\overline{\mathtt{T}}>\quad\Delta'\vdash\mathtt{C}<\overline{\mathtt{T}}>\prec:_{\overline{\mathtt{Y}}}[\mathtt{C}<\overline{\mathtt{T}}>/\mathtt{Y}_i]bound_\Delta(\mathtt{Y}_i)$, where $bound_\Delta(\mathtt{Y}_i)=\mathtt{P}_i$

We next prove by derivation of $\Delta'\vdash\mathtt{C}<\overline{\mathtt{T}}>\prec:_{\overline{\mathtt{Y}}}[\mathtt{C}<\overline{\mathtt{T}}>/\mathtt{Y}_i]\mathtt{P}_i$

Case PM-REFL: $[\mathtt{C}<\overline{\mathtt{T}}>/\mathtt{Y}_i]\mathtt{P}_i=\mathtt{C}<\overline{\mathtt{T}}>=bound_{\Delta'}(\mathtt{W}_i)$

Since $\overline{\mathtt{Y}}$ do not appear in Δ and consequently, $\mathtt{C}<\overline{\mathtt{T}}>$, $\Delta\vdash\mathtt{W}_i<:\mathtt{C}<\overline{\mathtt{T}}>$.

Case PM-CL: Let $\mathtt{P}_i=\mathtt{C}<\overline{\mathtt{S}}>$. Since $\mathtt{Y}_i$ do not appear in $\overline{\mathtt{S}}$ after substitution, for $\overline{\mathtt{T}}\prec:_{\overline{\mathtt{Y}}}\overline{\mathtt{S}}$, without loss of generality, we assume all other $\overline{\mathtt{Y}}$ has been substituted, then $\overline{\mathtt{T}}=\overline{\mathtt{S}}$. Then $\mathtt{C}<\overline{\mathtt{T}}>=\mathtt{C}<\overline{\mathtt{S}}>$, and by S-REFL, conclusion follows.

Case PM-CL-S: Follows from induction hypothesis.

□

Lemma 38 (Single Range Containment is Transitive)**.** *If* $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_1 \sqsubseteq_R R_2$, $\Delta;[\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\vdash R_2 \sqsubseteq_R R_3$, *then* $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\vdash R_1 \sqsubseteq_R R_3$,

Proof. By SB-R,

$R_1 = (\mathtt{T}_1, <\overline{\mathtt{X}} \lhd \overline{\mathtt{N}} > \overline{\mathtt{U}} \to \mathtt{U}_0)$ $\quad R_2 = (\mathtt{T}_2, <\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}} > \overline{\mathtt{V}} \to \mathtt{V}_0)$

$\Delta \vdash \mathtt{T}_2 <: \mathtt{T}_1$ $\quad \Delta, \overline{\mathtt{X}} <: \overline{\mathtt{N}}, \overline{\mathtt{Y}} <: \overline{\mathtt{P}}; [\overline{\mathtt{W}}/\overline{\mathtt{Y}}] \vdash \mathit{unify}(\mathtt{U}_0 : \overline{\mathtt{U}},\ \mathtt{V}_0 : \overline{\mathtt{V}})$

$R_3 = (\mathtt{T}_3, <\overline{\mathtt{Z}} \lhd \overline{\mathtt{O}} > \overline{\mathtt{V}}' \to \mathtt{V}'_0)$

$\Delta \vdash \mathtt{T}_3 <: \mathtt{T}_2$ $\quad \Delta, \overline{\mathtt{Y}} <: \overline{\mathtt{P}}, \overline{\mathtt{Z}} <: \overline{\mathtt{O}}; [\overline{\mathtt{Q}}/\overline{\mathtt{Z}}] \vdash \mathit{unify}(\mathtt{V}_0 : \overline{\mathtt{V}},\ \mathtt{V}'_0 : \overline{\mathtt{V}}')$

By S-TRANS, $\Delta \vdash \mathtt{T}_3 <: \mathtt{T}_1$

By UNI,

$[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{U}_0 : \overline{\mathtt{U}} = [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{V}_0 : \overline{\mathtt{V}}$ for all $\mathtt{Y}_i \in \overline{\mathtt{Y}}$, $\Delta, \overline{\mathtt{X}} <: \overline{\mathtt{N}}, \overline{\mathtt{Y}} <: \overline{\mathtt{P}} \vdash \mathtt{W}_i \prec :_{\overline{\mathtt{Y}}} \mathtt{Y}_i$

$[\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\mathtt{V}_0 : \overline{\mathtt{V}} = [\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\mathtt{V}'_0 : \overline{\mathtt{V}}'$ for all $\mathtt{Z}_i \in \overline{\mathtt{Z}}$, $\Delta, \overline{\mathtt{Y}} <: \overline{\mathtt{P}}, \overline{\mathtt{Z}} <: \overline{\mathtt{O}} \vdash \mathtt{Q}_i \prec :_{\overline{\mathtt{Z}}} \mathtt{Z}_i$

By construction of R_1 and R_2, no $\overline{\mathtt{Y}}$ appear in $\mathtt{U}_0 : \overline{\mathtt{U}}$, no $\overline{\mathtt{Z}}$ appear in $\mathtt{V}_0 : \overline{\mathtt{V}}$.

Thus, $\mathtt{U}_0 : \overline{\mathtt{U}} = [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{V}_0 : \overline{\mathtt{V}}$, $\mathtt{V}_0 : \overline{\mathtt{V}} = [\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\mathtt{V}'_0 : \overline{\mathtt{V}}'$

Then $\mathtt{U}_0 : \overline{\mathtt{U}} = [\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\mathtt{V}'_0 : \overline{\mathtt{V}}'$

Since no $\overline{\mathtt{Y}}$ appear in $\mathtt{V}'_0 : \overline{\mathtt{V}}'$, $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\mathtt{V}'_0 : \overline{\mathtt{V}}' = [([\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{Q}})/\overline{\mathtt{Z}}]\mathtt{V}'_0 : \overline{\mathtt{V}}'$

By inspecting PM-*, and by Lemma 20, it is clear that for all $\mathtt{Z}_i \in \overline{\mathtt{Z}}$, $\Delta, \overline{\mathtt{X}} <: \overline{\mathtt{N}}, \overline{\mathtt{Y}} <: \overline{\mathtt{P}}, \overline{\mathtt{Z}} <: \overline{\mathtt{O}} \vdash [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{Q}_i \prec :_{\overline{\mathtt{Z}}} \mathtt{Z}_i$

The conclusion follows. □

Lemma 39 (Method Type Lookup Terminates)**.** *Method type lookup* $\mathit{mtype}(\mathtt{n},\ \mathtt{T})$ *for all* $\mathtt{T}$ *with a finite chain of reflective dependency either terminates with* $\Delta;\Lambda \vdash \mathit{mtype}(\mathtt{n},\ \mathtt{T}) = \overline{\mathtt{U}} \to \mathtt{U}_0$*, or ends with none of the MT-* rules applicable.*

Proof. The chain of reflective dependency is a sequence of types defined to be:

$$
\begin{aligned}
\mathit{refchain}(\mathtt{Object}) &= \mathtt{Object} \\
\mathit{refchain}(\mathtt{X}) &= \mathtt{X} : \mathit{refchain}(\mathit{bound}_\Delta(\mathtt{X})) \\
\mathit{refchain}(\mathtt{C}\!<\!\overline{\mathtt{T}}\!>) &= \mathtt{C}\!<\!\overline{\mathtt{T}}\!> : \mathit{refchain}(\mathtt{T}_i) : \mathit{refchain}([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}) \\
&\quad \text{where } \overline{\mathtt{T}} = \mathtt{T}_0, \ldots, \mathtt{T}_n
\end{aligned}
$$

The chain is constructed so that if a reoccurrence of the same type, in any form of instantiation happens, the chain construction is terminated, and the chain is deemed not finite. Since there is a finite number of classes, the chain construction either terminates with a finite chain, or a reoccurrence as described above must happen.

We define the measure function to be:

$measure(mtype(\mathtt{n},\ \mathtt{T}))=length(refchain(\mathtt{T}))$.

It is simple to see that with each recursive call, the measure must decrease:

Case MT-VAR-S:

$measure(mtype(\eta,\ \mathtt{X}))=length(refchain(\mathtt{X}))$

$measure(mtype(\eta,\ bound_{\Delta}(\mathtt{X})))=length(refchain(bound_{\Delta}(\mathtt{X})))$

Since $length(refchain(\mathtt{X}))=1+length(refchain())bound_{\Delta}(\mathtt{X})$, clearly measure decreases.

Case MT-SUPER-S,

$measure(mtype(\mathtt{m},\ \mathtt{C}\text{<}\overline{\mathtt{T}}\text{>}))=length(refchain(\mathtt{C}\text{<}\overline{\mathtt{T}}\text{>}))$

$measure(mtype(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}))=length(refchain([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}))$

$refchain([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N})$ is embedded in $refchain(\mathtt{C}\text{<}\overline{\mathtt{T}}\text{>})$ by construction. Thus, the measure decreases.

Case MT-CLASS-R:

mtype is recursively invoked through *specialize* on one of the type parameters of $\mathtt{C}\text{<}\overline{\mathtt{T}}\text{>}$. By construction, again, $refchain(\mathtt{T}_i)$ is embedded in $refchain(\mathtt{C}\text{<}\overline{\mathtt{T}}\text{>})$. Thus, again, measure decreases.

Case MT-SUPER-R: Similar to MT-CLASS-R and MT-SUPER-S.

Since the chains are finite, then the measure function cannot decrease infinitely. Thus, the recursion must terminate.

□

References

[1] *Apache Xerces Project, http://xerces.apache.org/.* Accessed June 2009.

[2] *Guice: A Lightweight Dependency Injection Framework, http://code.google.com/p/google-guice/.* Accessed June 2009.

[3] *Hibernate: Relational Persistence for Java and .NET, http://www.hibernate.org.* Accessed June 2009.

[4] *Java Collections Framework Design FAQ, http://java.sun.com/javase/6/docs/technotes/guides /collections/designfaq.html.* Accessed June 2009.

[5] *Java Collections Framework Web site, http://java.sun.com/javase/6/docs/technotes/guides /collections/index.html.* Accessed June 2009.

[6] *Ruby on Rails Website, http://rubyonrails.org/.* Accessed June 2009.

[7] Alexandrescu, A., *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley Professional, February 2001.

[8] Allen, E., Bannet, J., and Cartwright, R., "A first-class approach to genericity," in *OOPSLA 2003: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, (New York, NY), pp. 96–114, ACM Press, 2003.

[9] Apache Software Foundation, *Byte-code engineering library. "http://jakarta.apache.org/bcel /manual.html".* Accessed June 2009.

[10] Attardi, G. and Cisternino, A., "Reflection support by means of template metaprogramming," in *Proceedings of Third International Conference on Generative and Component-Based Software Engineering, LNCS*, (London, UK), pp. 118–127, Springer-Verlag, 2001.

[11] Bachrach, J. and Playford, K., "The Java syntactic extender (JSE)," in *OOPSLA 2001: Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, (New York, NY), pp. 31–42, ACM Press, 2001.

[12] Baker, J. and Hsieh, W. C., "Maya: multiple-dispatch syntax extension in Java," in *PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, (New York, NY), pp. 270–281, ACM Press, 2002.

[13] Batory, D., Lofaso, B., and Smaragdakis, Y., "JTS: tools for implementing domain-specific languages," in *Proceedings of the Fifth International Conference on Software Reuse*, (New York, NY), pp. 143–153, IEEE, 1998.

[14] Batory, D., Sarvela, J. N., and Rauschmayer, A., "Scaling step-wise refinement," in *ICSE 2003: Proceedings of the 25th International Conference on Software Engineering*, (Washington, DC), pp. 187–197, IEEE Computer Society, 2003.

[15] Bracha, G. and Cook, W., "Mixin-based inheritance," in *OOPSLA/ECOOP 1990: Proceedings of the European Conference on Object-Oriented Programming and Object-Oriented Programming, Systems, Languages, and Applications*, (New York, NY), pp. 303–311, ACM Press, 1990.

[16] Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P., "Making the future safe for the past: Adding genericity to the Java programming language," in *OOPSLA 1998: ACM Symposium on Object-Oriented Programming, Systems, Languages, and Applications* (Chambers, C., ed.), (New York, NY), pp. 183–200, 1998.

[17] Bracha, G. and Ungar, D., "Mirrors: design principles for meta-level facilities of object-oriented programming languages," in *OOPSLA 2004: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, (New York, NY), pp. 331–344, ACM, 2004.

[18] Bravenboer, M., Dolstra, E., and Visser, E., "Preventing injection attacks with syntax embeddings," in *GPCE 2007: Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, (New York, NY), pp. 3–12, ACM, 2007.

[19] Bruenton, E. and others, *ASM Java Bytecode Engineering Library: http://asm.ow2.org/*. Accessed June 2009.

[20] Burke, B. and others, *JBoss AOP Web site, http://www.jboss.org/jbossaop/*. Accessed June 2009.

[21] Calcagno, C., Taha, W., Huang, L., and Leroy, X., "Implementing multi-stage languages using ASTs, gensym, and reflection," in *GPCE 2003: Proceedings the 2nd International Conference on Generative Programming and Component Engineering*, (New York, NY), pp. 57–76, Springer-Verlag New York, Inc., 2003.

[22] Cameron, N., Drossopoulou, S., and Ernst, E., "A Model for Java Wildcards," in *ECOOP 2008: Proceedings of the 22nd European Conference on Object-Oriented Programming*, (Heidelberg, Germany), pp. 2–26, Springer-Verlag, 2008.

[23] Cannon, H. I., "Flavors: A non-hierarchical approach to object-oriented programming," tech. rep., 1982.

[24] Cook, W. R., Hill, W., and Canning, P. S., "Inheritance is not subtyping," in *POPL 1990: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (New York, NY), pp. 125–135, ACM Press, 1990.

[25] Csallner, C. and Smaragdakis, Y., "JCrasher: An automatic robustness tester for Java," *Software—Practice and Experience*, vol. 34, pp. 1025–1050, Sept. 2004.

[26] Czarnecki, K. and Eisenecker, U. W., "Synthesizing objects," in *ECOOP 1999: Proceedings of the 13th European Conference on Object-Oriented Programming*, (London, UK), pp. 18–42, Springer-Verlag, 1999.

[27] Danforth, S. and Forman, I. R., "Reflections on metaclass programming in SOM," in *OOPSLA 1994: Proceedings of the 9th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, (New York, NY), pp. 440–452, ACM Press, 1994.

[28] Dijkstra, E. W., "On the role of scientific thought," in *Selected Writings on Computing: A Personal Perspective*, pp. 60–66, Springer-Verlag, 1982.

[29] Draheim, D., Lutteroth, C., and Weber, G., "A type system for reflective program generators," in *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, LNCS 3676, (Heidelberg, Germany), pp. 327–341, Springer-Verlag, 2005.

[30] Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A. P., "Traits: A mechanism for fine-grained reuse," *TOPLAS: ACM Transactions on Programming Languages and Systems*, vol. 28, no. 2, pp. 331–388, 2006.

[31] Ekman, T. and Hedin, G., "The JastAdd extensible Java compiler," in *OOPSLA 2007: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, (New York, NY), pp. 1–18, ACM, 2007.

[32] Emir, B., Kennedy, A., Russo, C., and Yu, D., "Variance and generalized constraints for C# generics," in *ECOOP 2006: Proceedings of the European Conference on Object-Oriented Programming*, (Nantes, France), Springer, 2006.

[33] Ernst, M. D., Badros, G. J., and Notkin, D., "An empirical analysis of c preprocessor use," *IEEE Transactions on Software Engineering*, vol. 28, no. 12, pp. 1146–1170, 2002.

[34] Fähndrich, M., Carbin, M., and Larus, J. R., "Reflective program generation with patterns," in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, (New York, NY), pp. 275–284, ACM Press, 2006.

[35] Gamma, E., Helm, R., and Johnson, R., *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, Addison-Wesley, 1995.

[36] Garrido, A. and Johnson, R., "Analyzing multiple configurations of a C program," in *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, (Washington, DC), pp. 379–388, IEEE Computer Society, 2005.

[37] Gosling, J. and others, *The Java Language Specification*. 5F, No.7, Lane 50, Sec.3 Nan Kang Road Taipei, Taiwan: GOTOP Information Inc.

[38] Halfond, W. G. J. and Orso, A., "Amnesia: analysis and monitoring for neutralizing sql-injection attacks," in *ASE 2005: Proceedings of the 20th IEEE/ACM International Conference on Automated software engineering*, (New York, NY), pp. 174–183, ACM, 2005.

[39] Hejlsberg, A., Wiltamuth, S., and Golde, P., *C# Language Specification*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[40] Herlihy, M., *SXM: C# Software Transactional Memory. "http://www.cs.brown.edu/ mph/SXM/"*. Accessed May 2007.

[41] Herlihy, M., Luchangco, V., and Moir, M., "A flexible framework for implementing software transactional memory," in *OOPSLA 2006: Proceedings of the 21st annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, (New York, NY), pp. 253–262, ACM, 2006.

[42] Hoffman, K. and Eugster, P., "Bridging Java and AspectJ through explicit join points," in *PPPJ 2007: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, (New York, NY), pp. 63–72, ACM, 2007.

[43] Huang, S. S. and Smaragdakis, Y., "Easy language extension with Meta-AspectJ," in *ICSE 2006: Proceedings of International Conference on Software Engineering*, (New York, NY), pp. 865–868, ACM, May 2006.

[44] Huang, S. S., Zook, D., and Smaragdakis, Y., "Statically safe program generation with SafeGen.," in *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, LNCS 3676, pp. 309–326, Springer-Verlag, 2005.

[45] Huang, S. S., Zook, D., and Smaragdakis, Y., "cJ: Enhancing Java with safe type conditions," in *Proc. of the 6th Intl. Conf. on Aspect-Oriented Software Development*, (Vancouver, British Columbia, Canada), pp. 185–198, ACM Press, 2007.

[46] Huang, S. S., Zook, D., and Smaragdakis, Y., "Morphing: Safely shaping a class in the image of others," in *ECOOP 2007: 21st European Conference on Object Oriented Programming* (Ernst, E., ed.), vol. 4609 of *Lecture Notes in Computer Science*, pp. 303–329, Springer, July 2007.

[47] Igarashi, A., Pierce, B., and Wadler, P., "Featherweight Java: A minimal core calculus for Java and GJ," *TOPLAS: ACM Transactions on Programming Languages and Systems*, vol. 23, no. 3, pp. 396–450, 2001.

[48] Igarashi, A. and Viroli, M., "Variant parametric types: A flexible subtyping scheme for generics," *TOPLAS: ACM Transactions on Programming Languages and Systems*, vol. 28, no. 5, pp. 795–847, 2006.

[49] ISO Standards Committee, "ISO/IEC standard 14882: Programming languages – C++," 1998.

[50] Jansson, P. and Jeuring, J., "PolyP - a polytypic programming language extension," in *POPL 1997: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (New York, NY), pp. 470–482, ACM Press, 1997.

[51] Järvi, J., Willcock, J., and Lumsdaine, A., "Associated types and constraint propagation for mainstream object-oriented generics.," in *OOPSLA 2005: Proceedings of the 2005 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, (New York, NY), pp. 1–19, ACM Press, 2005.

[52] Jones, S. P., *Haskell 98 Language and Libraries: the Revised Report*. Cambridge, England: Cambridge University Press, 1999.

[53] Jones, S. P., Jones, M., and Meijer, E., "Type classes: An exploration of the design space," in *In Haskell Workshop*, 1997.

[54] Kästner, C. and Apel, S., "Type-checking software product lines - a formal approach," in *ASE 2008: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 258–267, IEEE Computer Society, Sept. 2008.

[55] Kästner, C., Apel, S., and Batory, D., "A case study implementing features using AspectJ," in *SPLC 2007: Proceedings of the 11th International Software Product Line Conference* , (Los Alamitos, CA), pp. 223–232, IEEE Computer Society, Sept. 2007.

[56] Kästner, C., Apel, S., and Kuhlemann, M., "Granularity in software product lines," in *ICSE 2008: Proceedings of the 30th International Conference on Software Engineering*, (New York, NY), pp. 311–320, ACM, 2008.

[57] Kennedy, A. and Syme, D., "Design and implementation of generics for the .NET Common Language Runtime," in *PLDI 2001: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, (New York, NY), ACM, 2001.

[58] Kiczales, G., des Rivieres, J., and Bobrow, D. G., *The Art of the Metaobject Protocol*. MIT Press, 1991.

[59] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W., "Getting started with AspectJ," *Communications of the ACM*, vol. 44, no. 10, pp. 59–65, 2001.

[60] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G., "An overview of AspectJ," in *ECOOP 2001: Proceedings of the 15th European Conference on Object Oriented Programming*, (London, UK), pp. 327–353, Springer-Verlag, 2001.

[61] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J., "Aspect-Oriented Programming," in *ECOOP 1997: Proceedings of the 11th European Conference on Object Oriented Programming* (Akşit, M. and Matsuoka, S., eds.), vol. 1241, pp. 220–242, Heidelberg, Germany, and New York: Springer-Verlag, 1997.

[62] Lam, P., Kuncak, V., and Rinard, M., "Crosscutting techniques in program specification and analysis," in *AOSD 2005: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, (New York, NY), pp. 169–180, ACM Press, 2005.

[63] Lämmel, R. and Jones, S. P., "Scrap your boilerplate: a practical design pattern for generic programming," in *TLDI 2003: Proceedings of the 2003 ACM SIGPLAN International workshop on Types in Languages Design and Implementation*, (New York, NY), pp. 26–37, ACM, 2003.

[64] Leroy, X., "A modular module system," *Journal of Functional Programming*, vol. 10, no. 3, pp. 269–303, 2000.

[65] Lieberman, H., "Using prototypical objects to implement shared behavior in object-oriented systems," *SIGPLAN Not.*, vol. 21, no. 11, pp. 214–223, 1986.

[66] Liskov, B., *CLU Reference Manual*. Secaucus, NJ: Springer-Verlag New York, Inc., 1983.

[67] Liskov, B. and Zilles, S., "Programming with abstract data types," in *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, (New York, NY), pp. 50–59, ACM, 1974.

[68] Litvinov, V., "Constraint-based polymorphism in Cecil: Towards a practical and static type system," in *OOPSLA 1998: Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, vol. 33(10), (New York, NY), pp. 388–411, ACM Press, 1998.

[69] McNamara, B. and Smaragdakis, Y., "Static interfaces in C++," in *C++ Template Programming Workshop*, Oct. 2000.

[70] Mezini, M. and Ostermann, K., "Conquering aspects with caesar," in *AOSD 2003: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, (New York, NY), pp. 90–99, ACM, 2003.

[71] Milner, R., Tofte, M., and Macqueen, D., *The Definition of Standard ML*. Cambridge, MA: MIT Press, 1997.

[72] Mohnen, M., "Interfaces with default implementations in Java," in *Proceedings of the Inaugural Conference on the Principles and Practice of Programming*, (Maynooth, County Kildare, Ireland, Ireland), pp. 35–40, National University of Ireland, 2002.

[73] Murphy, G. C., Lai, A., Walker, R. J., and Robillard, M. P., "Separating features in source code: an exploratory study," in *ICSE 2001: Proceedings of the 23rd International Conference on Software Engineering*, (Washington, DC), pp. 275–284, IEEE Computer Society, 2001.

[74] Myers, A. C., Bank, J. A., and Liskov, B., "Parameterized types for Java," in *POPL 1997: the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (New York, NY), pp. 132–145, ACM, 1997.

[75] Parnas, D. L., "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[76] Pearce, D. J. and Noble, J., "Relationship aspects," in *AOSD 2006: Proceedings of the 5th International Conference on Aspect-oriented Software Development*, (New York, NY), pp. 75–86, ACM Press, 2006.

[77] Rémy, D., "Type checking records and variants in a natural extension of ML," in *POPL 1989: Symposium on Principles of Programming Languages*, (Austin, Texas, United States), pp. 77–88, ACM Press, 1989.

[78] Reppy, J. and Turon, A., "Metaprogramming with traits," in *ECOOP 2007: Proceedings of the European Conference on Object Oriented Programming*, (Berlin,Germany), pp. 373–398, Springer-Verlag, Aug. 2007.

[79] Scharli, N., Ducasse, S., Nierstrasz, O., and Black, A., "Traits: Composable units of behavior," in *ECOOP 2003: European Conference on Object Oriented Programming*, Springer LNCS 2743, 2003.

[80] Sheard, T. and Jones, S. P., "Template meta-programming for Haskell," in *Proceedings of the ACM SIGPLAN Workshop on Haskell*, (Pittsburgh, Pennsylvania), pp. 1–16, ACM Press, 2002.

[81] Siek, J. and Lumsdaine, A., "Concept checking: Binding parametric polymorphism in C++," in *C++ Template Programming Workshop*, Oct. 2000.

[82] Smaragdakis, Y. and Batory, D., "Implementing layered designs with mixin layers," in *ECOOP 1998: Proceedings of the 12th European Conference on Object Oriented Programming*, pp. 550–570, Springer-Verlag LNCS 1445, 1998.

[83] Smaragdakis, Y. and Batory, D., "Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs," *TOSEM: ACM Transactions on Software Engineering and Methodologies*, vol. 11, pp. 215–255, Apr. 2002.

[84] Stein, L. A., "Delegation is inheritance," in *OOPSLA 1987: Proceedings on ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, (New York, NY), pp. 138–146, ACM, 1987.

[85] Strniša, R., Sewell, P., and Parkinson, M., "The Java module system: core design and semantic definition," in *OOPSLA 2007: Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, (New York, NY), pp. 499–514, ACM, 2007.

[86] Su, Z. and Wassermann, G., "The essence of command injection attacks in web applications," in *POPL 2006: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (New York, NY), pp. 372–382, ACM, 2006.

[87] Taha, W. and Sheard, T., "Multi-stage programming with explicit annotations," in *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and semantics-based Program Manipulation*, (Amsterdam, The Netherlands), pp. 203–217, ACM Press, 1997.

[88] Tarr, P., Ossher, H., Harrison, W., and Stanley M. Sutton, J., "N degrees of separation: multi-dimensional separation of concerns," in *ICSE 1999: Proceedings of the 21st International Conference on Software Engineering*, (Los Alamitos, CA), pp. 107–119, IEEE Computer Society Press, 1999.

[89] Torgersen, M., Ernst, E., and Hansen, C. P., "Wild FJ," in *FOOL 2005: Foundations of Object-Oriented Languages*, (New York, NY), ACM Press, 2005.

[90] Torgersen, M., Hansen, C. P., Ernst, E., von der Ahe, P., Bracha, G., and Gafter, N., "Adding wildcards to the java programming language," in *SAC 2004: Proceedings of the 2004 ACM Symposium on Applied Computing*, (Nicosia, Cyprus), pp. 1289–1296, ACM Press, 2004.

[91] Visser, E., "Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9.," in *Domain-Specific Program Generation* (Lengauer, C., Batory, D., Consel, C., and Odersky, M., eds.), pp. 216–238, Springer-Verlag, 2004. LNCS 3016.

[92] Wehr, S., Lämmel, R., and Thiemann, P., "JavaGI: Generalized interfaces for Java," in *ECOOP 2007: 21st European Conference on Object Oriented Programming* (Ernst, E., ed.), vol. 4609 of *LNCS*, pp. 347–372, Springer-Verlag, 2007.

[93] Weirich, S. and Huang, L., "A design for type-directed Java," in *Workshop on Object-Oriented Developments (WOOD)* (Bono, V., ed.), ENTCS, 2004.

[94] Wirth, N., *Programming in Modula 2*. Springer, 1982.

[95] Wright, A. K. and Felleisen, M., "A syntactic approach to type soundness," *Information and Computation*, vol. 115, no. 1, pp. 38–94, 1994.

[96] Yu, D., Kennedy, A., and Syme, D., "Formalization of generics for the .NET common language runtime," in *POPL 2004: 31st Symposium on Principles of Programming Languages*, (New York, NY), ACM, 2004.