Reconciling Software Extensibility with Modular Program Reasoning

Todd David Millstein

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

2003

Program Authorized to Offer Degree: Computer Science & Engineering

University of Washington

Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Todd David Millstein

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of Supervisory Committee:

_____

Craig Chambers

Reading Committee:

_____

Gaetano Borriello

_____

Craig Chambers

_____

David Notkin

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Bell and Howell Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, or to the author.

Signature_____

Date_____

University of Washington

Abstract

Reconciling Software Extensibility with Modular Program Reasoning

by Todd David Millstein

Chair of Supervisory Committee:

Professor Craig Chambers
Computer Science & Engineering

Programming languages that support the creation of reusable software components help make programs easier to create, maintain, and understand. To accrue these benefits in practice, a component must be extensible, able to be easily augmented from the outside by clients. A component must also support modular reasoning, providing an interface to clients that specifies the component's behavior and constraints for its proper usage.

Unfortunately, the goals of component extensibility and modular reasoning are inherently in conflict: the more extensible a component is, the harder it is to ensure properties of the component in isolation from its clients. Mainstream object-oriented (OO) and functional languages support modular reasoning in the form of modular static typechecking, which provides basic well-formedness guarantees about components. However, these languages lack some common forms of extensibility. For example, OO languages lack the functional ability to easily add new operations to existing abstractions, and functional languages lack OO-style subclassing. The conflict between extensibility and modular reasoning has forced previous languages that support these missing extensibility idioms to forgo modular typechecking.

In this dissertation, I show how programming languages can support both the OO and functional extensibility idioms while maintaining modular typechecking of components. I develop a modular type system for a simple but flexible calculus. I then apply this theoretical work to the design of practical extensions to mainstream OO and functional languages.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

Thinking back over the years, I am overwhelmed by the amount of support I've received. These acknowledgments can only scratch the surface.

My first thanks go to my advisor Craig Chambers. He has taught me by example what it means to be a great researcher, teacher, and advisor. In addition to the many technical things I've learned from Craig, I have especially valued his eye for both elegance and relevance, as well as the enormous energy, focus, and precision he brings to everything he does. I strive to emulate Craig in these ways and many more.

Paris Kanellakis was my undergraduate advisor. He generously gave me my first opportunities in computer-science research and teaching. He introduced me to topics and techniques that fascinate me as much today as then. His example made me want to pursue a career in computer science. His life and work continue to inspire me.

I have been fortunate enough to have several other mentors along the way. Tom Ball, Alon Halevy, Gary Leavens, Rustan Leino, Sriram Rajamani, Pascal Van Hentenryck, and Dan Weld have been wonderful research supervisors, broadening my horizons tremendously and helping to indulge my love of dabbling. David Notkin has been a crucial sounding board throughout my graduate career, always available to chat about matters big and small. Richard Anderson, Paul Beame, Alan Borning, Gaetano Borriello, Steve Gribble, and Ed Lazowska never failed to provide timely support and encouragement.

The research described in this dissertation is joint work with Colin Bleckner, Craig Chambers, Curt Clifton, Gary Leavens, and Mark Reay. Needless to say, the work would not be nearly as strong were it not for my interactions with these people. I would especially like to thank Curt and Gary for their long collaboration with Craig

and me on the MultiJava project. Curt deserves special kudos for his hard work in making a robust and practical MultiJava implementation come to life.

I have learned tons and had lots of fun collaborating with other graduate students on various research projects over the years. Thanks to Mike Ernst, Marc Friedman, Sorin Lerner, and Rupak Majumdar for making great research partners. My research has been informed by discussions with many more students (some of whom I'm sure I'm omitting), including Jonathan Aldrich, Greg Badros, William Chan, Dave Grove, Craig Kaplan, Keunwoo Lee, E Lewis, Vass Litvinov, Tapan Parikh, Matthai Philipose, and Sumeet Sobti.

Thanks to Jeff Hightower, Eric Lemar, Keunwoo Lee, Bart Niswonger, and Jing Su for enthusiastically programming with MultiJava and providing insightful feedback.

Thanks to Shannon Gilmore, Frankye Jones, Melody Kadenko, and Lindsay Michimoto for effortlessly navigating the university bureaucracy, so I could remain blissfully ignorant of it.

Finally, thanks to mom and dad for your tireless and unconditional love and support, which manifest in innumerable ways every single day.

DEDICATION

in memory of Paris Kanellakis (1953-1995) and William Chan (1972-1999)

Chapter 1

INTRODUCTION

1.1 Overview

Modern programming languages support the creation of programs as collections of *software components*. I use the term "software component" in a generic sense to denote any syntactically independent program fragment.[1] Classes in mainstream object-oriented (OO) languages like C++ [99], Java [5, 45], and C# [16] are a form of software component, as are modules in mainstream functional languages like ML [76] and Haskell [56]. Software components provide a number of software engineering benefits. For example, libraries of abstractions can be provided as components by one vendor and then used by others in a variety of programs. Each developer on a project can separately implement and validate some components, with all components later combined to form the complete application. Programs can evolve through the introduction of new components that interact with existing ones. Programs can be understood piecewise by understanding the behaviors of individual components.

To accrue these benefits in practice, clients of a software component must be able to easily customize the component to fit their needs. Languages support this ability by making components *extensible*, able to be augmented from the outside. For example, OO languages support extensibility through subclassing. Without the proper means of extensibility, a component must be modified in place in order to adapt it for use in some contexts, reducing its reusability. This modification requires source-code access to the component and requires the component to be recompiled. In-place modification also closely couples the component to the needs of particular clients, polluting the component for other clients and increasing

[1] Others (e.g., Szyperski et al. [100]) have used the term to denote more specific concepts.

maintenance costs.

It must also be possible to understand important properties of a component in isolation from the contexts in which it is used. This allows responsibility for ensuring that a component behaves properly to rest with the component's implementer, rather than with each client. Languages support this ability by providing an *interface* for each component, which provides behavioral guarantees about the component and specifies constraints on how the component may be used by clients. The component can be checked once by the implementer to ensure that it provides the guarantees specified in its interface, under the assumption that clients will obey the specified constraints. Separately, each client application can be checked to ensure that it respects the interface's constraints, without depending on or requiring access to the component's underlying implementation. Clients that respect the constraints can safely assume that the component will exhibit the behavioral properties guaranteed in its interface.

Unfortunately, the above requirements of software components are in direct conflict. The more extensible a component is, the harder it is to guarantee properties of the component separate from the contexts in which it is used. Each language therefore must trade off component extensibility and component reasoning to some extent. Mainstream OO and functional languages provide component reasoning in the form of modular static typechecking. Modular typechecking guarantees that if a component conforms to its interface, then the component is well formed in a particular sense: when used in any context satisfying its interface constraints, the component will never cause common kinds of run-time failures known as "type errors." For example, the component will not use an integer as if it were a string and will not attempt to invoke a function that does not exist. In order to provide these guarantees about a component's behavior, mainstream languages have been limited in the forms of component extensibility they can support.

For example, OO languages support component extensibility through subclassing. While subclassing is a useful idiom, there are several other useful extensibility idioms that are not supported. One well-known limitation of OO languages is the requirement that all of a class's methods be declared with the class. This requirement makes it difficult for clients to add application-specific behavior to an existing class or to evolve a class that

is frequently updated with new operations. There exist OO languages containing a form of *open classes*, which allows new methods to be declared externally. Unfortunately, this gain in component extensibility has been accompanied by a corresponding loss of modular typechecking. Therefore, component robustness is severely compromised. For example, a component's implementer loses early feedback about type errors, and clients can never trust that the component properly conforms to its interface. Instead, each client must take responsibility for ensuring that the component's implementation works properly in the client's context.

Another limitation of OO languages is the inability to easily write methods that *dynamically dispatch* on arguments other than the receiver. Dispatching on arguments, called *multiple dispatch*, is a useful form of extension whenever multiple class hierarchies cooperate to implement an operation. For example, the appropriate way to handle an event in an event-based system depends both on what kind of event occurs and on what kind of handler receives the event. There exist OO languages containing *multimethods*, which allow any subset of a method's arguments to be dispatched upon during method lookup. As with languages containing open classes, these languages have been forced to forgo modular typechecking of components.

Functional languages like ML and Haskell exhibit extensibility strengths and weaknesses that are complementary to those of OO languages. Abstractions are defined as datatypes consisting of a set of data variants. An operation over a datatype is written as an ordinary function, each of whose cases defines the behavior for a particular variant by *pattern matching* on the function's argument. Therefore, new functions are easily added to existing datatypes. Further, because functions have no distinguished receiver argument, functions may dispatch on all arguments via pattern matching. However, functional languages lack the traditional OO form of extensibility through subclassing. Adding a new variant to a datatype requires that the existing datatype be modified in place. Further, all functions on the datatype must be modified to include a case handling the new variant. There exist functional languages in which datatypes and functions are extensible, allowing new data variants and function cases to be written without modifying existing datatypes and functions. However, these languages have not achieved modular typechecking.

```
class AreaRectangle extends Rectangle {
  int area() { return length() * width(); }
}
```

Figure 1.1: Adding new methods via subclassing.

This dichotomy between the OO and functional styles of extensibility was pointed out by Reynolds as early as 1975 [93] and has subsequently received significant attention [30, 84, 61, 37, 44, 106]. In this dissertation, I show that it is possible to create languages that support the traditional extensibility idioms of both the OO and functional styles while retaining modular typechecking. These new languages greatly enhance the flexibility of software components: clients can easily add new subclasses to existing abstractions, new operations on existing abstractions, and operations that dynamically dispatch at multiple arguments, all without modifying the original abstractions or their clients. At the same time, components remain robust in the face of this heightened extensibility, as they are still guaranteed to be type-correct in any context satisfying their interface constraints.

1.2 Limitations of Traditional Languages

This section motivates the need for explicit language support to rectify the limitations of components in mainstream languages. I illustrate several approaches that have been proposed to achieve the desired extensibility idioms in traditional OO languages and describe their drawbacks.

1.2.1 Adding Methods To Existing Classes

The simplest way to add new methods to an existing class in traditional OO languages is via subclassing. Figure 1.1 illustrates this technique: an existing `Rectangle` class (not shown) is externally updated with a method for computing its area by creating a new subclass `AreaRectangle`, which inherits all the behavior of `Rectangle` and additionally contains the new method. `AreaRectangle` can then be used in place of `Rectangle` to achieve the desired functionality.

There are several drawbacks of this approach. First, the new method does not handle any existing instances of `Rectangle`, for example from a persistent store. Second, although the `area` method is added without modifying `Rectangle`, in general existing clients of `Rectangle` must still be modified: any code that constructs `Rectangle` instances must be updated to construct `AreaRectangle` instances. Third, if `AreaRectangle` instances are passed to and later returned from existing code, their static type upon return will be `Rectangle` (unless the existing code is modified). Therefore, run-time casts will be required to regain access to the `area` operation for these instances. Finally, this technique is tedious and awkward when an entire class hierarchy must be updated with a new operation. For example, suppose that `Rectangle` is part of a larger hierarchy of shape classes, with base class `Shape`. To provide area functionality for the entire hierarchy, one must also declare an `AreaShape` subclass of `Shape`. `AreaRectangle` must then subclass from `AreaShape` in addition to `Rectangle`, in order for `AreaRectangle`'s `area` method to override that of `AreaShape`, and this requires multiple inheritance of classes (which is not available in either Java or C#). Similarly, `AreaRectangle`'s `area` method is not inherited by any subclasses of `Rectangle`, so each of them must be provided with a new subclass that also inherits from `AreaRectangle`.

A different workaround, known as the *visitor design pattern* [43], is more suitable when a hierarchy of classes must be often updated with new operations. This approach is illustrated in figure 1.2. To allow clients to add application-specific operations to the shape hierarchy, the original implementer of `Shape` includes a "hook" via the `accept` method and provides the associated `ShapeVisitor` class. `ShapeVisitor` has one `visit*` operation per `Shape` subclass, and the `accept` method of each `Shape` subclass invokes its corresponding operation of the given visitor. Example implementations for `Rectangle` and `Rhombus` are shown. Given this infrastructure, new operations on the shape hierarchy can be created by providing a new subclass of `ShapeVisitor` that overrides each `visit*` method to provide the proper behavior for each `Shape` subclass. Figure 1.2 shows how area functionality is added to the shape hierarchy using the visitor pattern. The area operation can then be used as shown in the bottom-right corner of the figure.

Because the original classes in the shape hierarchy are augmented with the area func-

```
abstract class Shape {
  ... // various fields and methods
  abstract void
    accept(ShapeVisitor v);
}

class Rectangle extends Shape {
  ... // various fields and methods
  void accept(ShapeVisitor v) {
    v.visitRectangle(this);
  }
}

class Rhombus extends Shape {
  ... // various fields and methods
  void accept(ShapeVisitor v) {
    v.visitRhombus(this);
  }
}
```

```
abstract class ShapeVisitor {
  abstract void
    visitRectangle(Rectangle r);
  abstract void visitRhombus(Rhombus r);
  ... // one method per subclass of Shape
}

class AreaShapeVisitor
  extends ShapeVisitor {
  int area;
  void visitRectangle(Rectangle r) {
    area = r.length() * r.width();
  }
  void visitRhombus(Rhombus r) { ... }
}

// sample client code
Shape s = ...;
AreaShapeVisitor v =
  new AreaShapeVisitor();
// compute the shape's area
s.accept(v);
// use the shape's area
... v.area ...
```

Figure 1.2: Adding new methods with the visitor pattern.

```
class Rectangle extends Shape {
  void draw(Display d) { // default drawing for rectangles }
  void draw(ColorDisplay cd) { // draw rectangles to color displays }
  void draw(BitMapDisplay bmd) { // draw rectangles as bitmaps }
}
```

Figure 1.3: Simulating multiple dispatch with static overloading.

tionality, the problems of the previous approach are avoided. However, the visitor pattern has several new drawbacks. First, the need to add external operations must be planned for in advance by the implementer of the shape hierarchy, who must provide the `ShapeVisitor` class and the `accept` methods. Second, all external operations must subclass from the given `ShapeVisitor` class. Therefore, these operations must have identical argument and return types. In practice, the operations will take no arguments (other than the shape) and return no results. As illustrated by `AreaShapeVisitor`, operations that require arguments or results must simulate them through fields of the visitor. Also, all external operations must define one method per `Shape` subclass, via the `visit*` operations defined in `ShapeVisitor`. This requirement makes it difficult for a `Shape` subclass to inherit visiting functionality from its superclass. The requirement also makes it difficult to add an external operation that applies only to a subset of the classes in the `Shape` hierarchy. Finally, the visitor pattern loses the traditional OO idiom of modular subclass extensibility. For example, adding a new subclass `Triangle` of `Shape` requires that the `ShapeVisitor` class and *all* of its subclasses be updated with a new `visitTriangle` method, in order to provide the behavior of `Triangle`s for each external operation. Several variations and extensions of the basic visitor pattern have addressed these limitations [70, 81, 103, 87, 61, 106, 47]. However, most of these proposals complicate the already-complex visitor protocol. In addition — and further illustrating the tension between extensibility and modular reasoning — the proposals that retain support for modular subclass extensibility require a loss of static type safety in the form of run-time casts or reflection.

```
class Rectangle extends Shape {
  void draw(Display d) {
    if(d instanceof ColorDisplay) {
      ColorDisplay cd = (ColorDisplay) d;
      // draw rectangles to color displays
    } else if (d instanceof BitMapDisplay) {
      BitMapDisplay bmd = (BitMapDisplay) d;
      // draw rectangles as bitmaps
    } else {
      // default drawing for rectangles
    }
  }
}
```

Figure 1.4: Simulating multiple dispatch with run-time class tests and casts.

1.2.2 Multiple Dispatch

A simple attempt at simulating multiple dispatch is to use *static overloading*, a feature found in traditional OO languages including C++, Java, and C#. As an example, suppose the `Rectangle` class has functionality for drawing rectangles, whose behavior depends on the kind of display (subclasses of the base class `Display`) to which the rectangle should be drawn. The candidate solution is shown in figure 1.3. Unfortunately, static overloading does not provide the desired semantics. The three `draw` methods in the figure are completely unrelated, as if they had different names. At a call site, the *static*, rather than dynamic, type of the argument determines which of the three methods will be invoked. Therefore an invocation `r.draw(d)`, where `r` has static type `Rectangle` and `d` has static type `Display`, will *always* invoke the first `draw` method in figure 1.3, even if dynamically `d` refers to a `ColorDisplay` or `BitMapDisplay` instance.

Dispatch based on the dynamic, rather than static, types of the arguments to a call can be performed instead via run-time class tests and casts, as shown in figure 1.4. This approach achieves the desired semantics, but it is tedious and error-prone. First, the programmer must manually ensure that all appropriate `Display` subclasses are handled properly. For example, the programmer must order the `if` cases such that each class is tested before any of its superclasses. Second, static type safety is lost, forcing the entire burden of ensuring

that this manual dispatching logic is correct on the programmer. As a simple example, if the `instanceof` and associated cast in a given `if` case do not agree (e.g., because of a cut-and-paste error), the problem will not be detected until a run-time error occurs.[2] Third, the monolithic `if` statement limits the potential for inheritance. For example, it would be difficult for a subclass of `Rectangle` to inherit the behavior for drawing to a `ColorDisplay` but override the behavior for drawing to a `BitMapDisplay`. Finally, the extension of this technique to multiple arguments is even more complex and difficult to reason about.

A final technique for simulating multiple dispatch in traditional OO languages is *double dispatching* [54]. This technique achieves multiple dispatch via a cascaded sequence of single dispatches. The visitor pattern implementation of figure 1.2 uses double dispatching to dispatch on both the shape and the visitor. Invoking `accept` on a shape results in "learning" which shape is the receiver. The `accept` method sends a second message, to dispatch on the visitor argument. The invoked `visit*` method now "knows" both which shape and which visitor were passed and can therefore appropriately implement the desired functionality for that pair. Drawing can be analogously implemented, with `accept` renamed `draw`, the visitor hierarchy replaced by the display hierarchy, and each `visitC` method renamed `drawC`. Double dispatching also generalizes to any number of arguments. This dual usage of double dispatching, to add new methods to existing classes and to support multiple dispatch, illustrates the strong relationship between these extensibility idioms. Both idioms require the ability to dispatch on existing classes externally. In the former case the existing classes are the receivers of new external methods, while in the latter case the existing classes are arguments of another class's methods.

By using ordinary message sends, double dispatching retains static type safety, unlike the previous approach. However, double dispatching shares several drawbacks with the visitor pattern. First, existing `Display` subclasses may need to be modified whenever a new `Shape` subclass is added, in order to add methods for drawing the new kind of shape. Double dispatching also makes it hard for shapes to inherit functionality from their superclasses, because each `Display` class must implement a separate method for each `Shape` subclass.

---

[2]This problem can be avoided in languages that include a `typecase` construct, like Modula-3 [80].

```
int Shape.area() { ... }
int Rectangle.area() { ... }
int Rhombus.area() { ... }
```

Figure 1.5: Open classes in MultiJava

Finally, the protocol is tedious and error-prone, and the programmer must examine both the shape and display hierarchies in order to understand an operation's behavior.

1.3 Open Classes and Multimethods

The drawbacks of the workarounds presented above can be bypassed by augmenting traditional OO languages with new constructs that directly support the desired extensibility idioms. This section briefly previews such constructs — open classes and multimethods — in the context of MultiJava, an extension to Java developed by myself and colleagues. MultiJava is discussed in more detail in chapter 3.

Figure 1.5 shows how MultiJava's open classes support adding new methods to existing classes. The figure contains three *external methods*, each implementing the new `area` operation for a different member of the shape hierarchy. These declarations can be written in a new file, separate from the files containing the original shape classes. Further, clients invoke `area` the same way that any method in a shape class is invoked. Unlike the workarounds of section 1.2, the original shape hierarchy does not need to provide special "hooks," neither the shape classes nor their clients require modification, and no run-time type casts are necessary.

Figure 1.6 illustrates MultiJava's support for multimethods. Together the three `draw` methods have the same semantics as the implementation shown in figure 1.4. However, the MultiJava version does not suffer from the drawbacks of that approach. First, the methods can appear in any order, and the language does the work of dispatching an invocation to the appropriate one, based on the dynamic type of the argument. Second, the MultiJava version does not require the use of statically unsafe constructs like type casts. Third, multimethods can be easily inherited by subclasses. Finally, the multimethod syntax naturally extends to

```
class Rectangle extends Shape {
  void draw(Display@ColorDisplay cd) {
    // draw rectangles to color displays
  }
  void draw(Display@BitMapDisplay bmd) {
    // draw rectangles as bitmaps
  }
  void draw(Display d) {
    // default drawing for rectangles
  }
}
```

Figure 1.6: Multimethods in MultiJava

support dispatching on multiple arguments.

As mentioned earlier, there have been previous languages that support constructs similar to MultiJava's external methods and multimethods. However, all of these languages have been forced to forgo modular static typechecking. Supporting the desired extensibility idioms of the new language features while retaining safe modular typechecking is the subject of this dissertation.

1.4 Statement of the Thesis

My thesis is that *practical languages can be developed that support (1) the easy addition of new subclasses to existing abstractions; (2) the easy addition of new operations to existing abstractions; (3) multiple dispatch; and (4) modular typechecking of components.* I support my thesis in this dissertation in several ways:

- I define a small OO calculus, called Dubious, for formal study. Dubious naturally supports both open classes and multimethods via *generic functions*. I describe a type system for the calculus that provides purely modular typechecking while retaining both the traditional OO and functional extensibility idioms.

- I define MultiJava, a small backward-compatible extension to Java containing open classes and multimethods. MultiJava illustrates how the theoretical work of Dubious can be incorporated as a natural and practical extension to mainstream OO languages.

- I define Extensible ML (EML), an ML-like language containing extensible datatypes and functions. These constructs allow OO programming to be seamlessly incorporated into the functional style. As with MultiJava, EML relies on Dubious for its theoretical foundations. The foundations are extended and adapted to handle functional constructs including pattern matching and parametric polymorphism, while retaining modular typechecking. I present a formalization and type soundness proof for a core subset of EML called MINI-EML.

- I present several examples illustrating the benefits that MultiJava has provided others in a variety of real-world applications, including ubiquitous computing, graphical user interfaces, and compilers.

Chapters 2 discusses the Dubious language and its modular type system. Chapters 3 and 4 respectively describe MultiJava and EML. Chapter 4 also formalizes MINI-EML. Chapter 5 illustrates the ways in which MultiJava has been used and compares with the traditional Java solutions. Chapter 6 concludes the thesis. Appendix A provides the complete type soundness proof for MINI-EML.

Chapter 2

THEORETICAL FOUNDATIONS

This chapter describes Dubious [73, 74], a core OO language. Dubious is explicitly designed to contain only those language features necessary to express the programming idioms described in the previous chapter. This design makes Dubious simple enough that it can be studied formally but still rich enough that the results of such study can be adapted for use in full-scale programming languages. Dubious defines:

- a simple classless object model with explicitly declared inheritance, but not dynamically created objects or mutable state;
- first-class (generic) functions, but not lexically nested closures;
- explicitly declared function types, but not types independent of objects or polymorphic types; and
- modules to support separate typechecking and namespace management, but not encapsulation mechanisms, nested modules, or parameterized modules.

The syntax of Dubious is shown in figure 2.1. Metavariables *Mn*, *On*, and *I* denote module names, object names, and formal parameters, respectively. Brackets denote optional pieces of syntax. After introducing Dubious and illustrating how it supports the desired programming idioms discussed in chapter 1, I focus on the problem of modular typechecking. I illustrate several challenges for modular typechecking in Dubious and describe a modular type system that overcomes these challenges in a practical way.

P	::=	$M_1 \ldots M_n$; E	*programs*
M	::=	`module` Mn { $D_1 \ldots D_n$ }	*modules*
D	::=	[Q] `object` On [`isa` $O_1, \ldots, O_n$]	*declarations*
	|	Ob `has method` ($F_1, \ldots, F_n$) { E }	
Q	::=	`abstract` | `interface`	*object qualifiers*
O	::=	Ob | ($O_1, \ldots, O_n$)→ O	*objects*
F	::=	I [`@` Ob]	*formal arguments*
E	::=	I | Ob | E($E_1, \ldots, E_n$)	*expressions*
Ob	::=	$Mn.On$	*qualified object names*

Figure 2.1: Syntax of Dubious.

```
module SetMod {
  abstract object set
  object addElem isa (set, StdLibMod.int)→set
  object elems isa (set)→StdLibMod.list
  object union isa (set, set)→set
  union has method(s1, s2) {
    ... // union functionality using addElem and elems
  }
}

module ListSetMod {
  object listSet isa set
  addElem has method(s@listSet, i) { ... }
  elems has method(s@listSet) { ... }
  union has method(s1@listSet, s2@listSet) { ... }
}

module SListSetMod {
  object sListSet isa listSet
  addElem has method(s@sListSet, i) { ... }
  union has method(s1@slistSet, s2@slistSet) { ... }
  object getMin isa (sListSet)→int
  getMin has method(s@sListSet) { ... }
}
```

Figure 2.2: A hierarchy of integer sets in Dubious.

2.1 Dubious By Example

2.1.1 Informal Semantics

A Dubious program consists of a sequence of modules, followed by an expression to be evaluated in the context of those modules. Three Dubious modules that implement a simple hierarchy of integer sets are shown in figure 2.2. (Technically, all references to objects should be qualified by a module name, as is done for the references to `int` and `list` from the `StdLibMod` module, which is not shown in the figure. I elide the module names from object references when clear from context.) The body of a module is a sequence of declarations, and Dubious has only two kinds of declarations. The `object` declaration creates a fresh object with a unique identity and binds it to the given name. The declaration also names the (possibly multiple) objects from which the new object inherits. The *subclass* relation among objects is the reflexive, transitive closure of the declared `isa` relation. In the modules of figure 2.2, the `set` object is the top of the integer set hierarchy, `listSet` is a subclass for sets implemented with lists,[1] and `sListSet` is a subclass of `listSet` in which the underlying list of elements is kept in sorted order. As in other classless languages [62, 67, 101, 23], objects play the roles of both classes and instances, and `isa` accordingly plays the roles of both inheritance and instantiation.

An object in Dubious can also act as a *generic function* [77, 8, 86], a collection of methods of the same name and type signature. Because generic functions are objects, they are naturally first-class: they can be passed to and returned from other functions. An object acting as a generic function inherits from an *arrow object* that defines the type signature of the function. For example, the `addElem` object in `SetMod` can be invoked with a pair of one `set` subclass and one `int` subclass, and it returns a `set` subclass. For simplicity, Dubious requires that each generic function explicitly inherits from exactly one arrow object. The ordinary contravariant subtyping rule for function types [17] is used as the subclass relation among arrow objects.

Generic functions are implemented by *adding* methods to them, with the `has method`

[1] In a full-scale language, `listSet` would have a field for its underlying list, and possibly other fields. Read-only fields can be modeled in Dubious with methods.

declaration. In `ListSetMod`, the method added to the `addElem` generic function has two formal parameters, `s` and `i`. The first formal is *specialized* by providing the `@listSet` suffix, which specifies the object on which the argument is dynamically dispatched; that object is the argument's *specializer*. This method can only be invoked dynamically if the first actual argument is `listSet` or a subclass. Because the second argument is *unspecialized*, any integer may be passed to it. In traditional class-based object-oriented languages, this method would be modeled as the `addElem` method inside the `listSet` class. Similarly, the `addElem` method in `SListSetMod` would be modeled as an overriding `addElem` method inside the `sListSet` subclass. `SListSetMod` also introduces a new generic function for accessing the minimal element of an `sListSet` object, with an associated method.

However, methods in Dubious are more general than traditional OO methods. The various `union` methods in figure 2.2 illustrate this expressiveness. The `union` method in `SetMod` has no specializers and therefore acts as a regular procedure: it is applicable to any pair of sets. The `union` method in `ListSetMod` is a multimethod, because it specializes on more than one argument. This method provides an optimized union algorithm for the case when *both* arguments are `listSet` or a subclass. The `union` method in `SListSetMod` similarly provides a specialized algorithm for the case when both arguments are `sListSet` or a subclass. The `union` methods are examples of *binary methods* [15], which accept two arguments of the same type. These methods arise quite commonly and are naturally expressed by multimethods.

The `has method` declaration is an imperative operation. For example, `ListSetMod` adds a second method to `SetMod`'s `union` generic function, rather than creating a new generic function containing the extra method. This semantics is necessary to model the traditional OO style. For example, the imperative semantics allows clients of the set hierarchy to be aware only of the `set` object, with message sends to its `union` generic function dynamically dispatched to the appropriate methods of `set` subclasses. Similarly, the imperative semantics supports *downcalls*: the `addElem` and `elems` message sends from within `SetMod`'s `union` method are dispatched dynamically to methods of `set` subclasses.

To evaluate a generic function application (message send) $E\texttt{(}E_1, \ldots, E_n\texttt{)}$, Dubious employs the natural multimethod dispatching semantics, in which all arguments are treated

symmetrically. This semantics is used in the languages Kea [78], Cecil [23, 25], Dylan [36, 96], the λ&-calculus [21, 20], and ML$_{\leq}$ [10]. First E is evaluated to some generic function f and each E_i is evaluated to some object o_i. Then the methods added to the generic function f are extracted, and finally the *most-specific applicable method* for $(o_1, \ldots, o_n)$ is selected and invoked.

For uniformity in describing the semantics of Dubious throughout this chapter, I sometimes assume that all of a method's formal arguments are specialized; an unspecialized formal is equivalent to a formal that is specialized on the object in the corresponding position of the associated generic function's arrow object. For example, the `union` method in `SetMod` of figure 2.2 can be considered to specialize on `Set` at both arguments. Given this convention, a method in f is *applicable* to $(o_1, \ldots, o_n)$ if the method has n arguments and if $(o_1, \ldots, o_n)$ pointwise subclasses from the tuple of the method's specializers $(o'_1, \ldots, o'_n)$: for each i such that $1 \leq i \leq n$, o_i subclasses from o'_i. One method m_1 is *at least as specific* as another method m_2 if m_1's tuple of specializers pointwise subclasses from m_2's tuple of specializers. A method is the *most-specific* applicable method if it is the unique applicable method that is at least as specific as every other applicable method.

For example, consider the application `union(listSet,sListSet)`, evaluated in the context of the modules in figure 2.2. First, the `union`, `listSet`, and `sListSet` expressions are evaluated, yielding the corresponding objects. Then the methods that were added to the `union` generic function are extracted. Of the three methods, the first two methods in the figure are applicable: the third `union` method is not applicable because the first actual argument of the application is not a subclass of the method's first specializer, `sListSet`. Of the two applicable methods, the `(@listSet, @listSet)` method is the most-specific one, because `(@listSet, @listSet)` pointwise subclasses from `(@set, @set)`. Therefore, the `(@listSet, @listSet)` method is selected and invoked. If there are no applicable methods for a message send, a *message-not-understood* error occurs, while if there are applicable methods but no most-specific one, a *message-ambiguous* error occurs.

If an object is declared `abstract`, it is used solely as a template for other objects and may not be referred to in expressions. For example, the abstract `set` object is a template for integer sets that is implemented by the `listSet` and `sListSet` objects. Because abstract

```
module IsEmptyMod {
  object isEmpty isa (set)→bool
  isEmpty has method(s) { null(elems(s)) }
  isEmpty has method(s@listSet) { ... }
}
```

Figure 2.3: Open classes in Dubious.

objects may not be referred to in expressions, generic functions need not have method implementations handling these objects. For example, the `addElem` generic function in `SetMod` does not have a method for adding an element to the `set` object. In a traditional OO language, this *abstract-class* idiom would be implemented by declaring an abstract `addElem` method inside the `set` class.

Generic functions may optionally include methods that specialize on abstract objects, just as an abstract class in Java may contain method implementations; these methods are then inherited for use by subclasses. Dubious also supports the notion of an *interface object*, through the `interface` object qualifier. An interface object is like an abstract object, but it additionally may not be a specializer in any methods. This semantics is similar to interfaces in Java [5, 45], which act like abstract classes but additionally must have only abstract methods. For example, because `set` in figure 2.2 is not used as an explicit specializer (despite having an applicable `union` method), it could equivalently be declared an interface instead of an abstract object. Distinguishing between abstract and interface objects will become useful when considering modular typechecking later in this chapter. Arrow objects are implicitly treated as interface objects. An object that is neither abstract nor an interface is called *concrete*. (A *nonconcrete* object is either abstract or an interface, and a *noninterface* object is either concrete or abstract.)

2.1.2 Open Classes

Dubious's model of generic functions naturally supports the open-class idiom with no extra mechanism. An example is shown in figure 2.3, where the `set` hierarchy is extended with a function for checking emptiness. A method handling any set is provided, as well as an

```
module HashSetMod {
  object hashSet isa set
  addElem has method(s@hashSet, i) { ... }
  elems has method(s@hashSet) { ... }
}
```

Figure 2.4: A new `set` subclass.

```
module CListSetMod {
  object cListSet isa listSet
  addElem has method(s@cListSet, i) { ... }
  isEmpty has method(s@cListSet) { ... }
}
```

Figure 2.5: Another `set` subclass.

overriding method for `listSet`. The `sListSet` object implicitly inherits the second method.

Finally, the easy addition of new operations does not prevent the easy addition of new subclasses, as illustrated in figure 2.4. `HashSetMod` introduces a new `set` subclass that implements sets using hash tables, providing overriding methods of `addElem` and `elems` while inheriting the existing `union` functionality. `IsEmptyMod` and `HashSetMod` are completely independent: either, both, or neither module could be linked into the final application, giving four versions of the set hierarchy. When both modules are present, `hashSet` and subclasses can be passed to the `isEmpty` generic function, even though the implementer of the `HashSetMod` extension to the set hierarchy may be unaware of the `IsEmptyMod` extension, and vice versa. Figure 2.5 shows another `set` subclass, representing sets implemented as lists that additionally keep a count of the current elements. `CListSetMod` provides an overriding method of `addElem`. The implementer is aware of the `IsEmptyMod` extension and therefore also provides an overriding method for checking emptiness of `cListSet`. Unlike the visitor pattern, existing code does not have to be modified in order to implement this overriding method.

2.2 Static Typechecking for Dubious

2.2.1 Typechecking Generic Functions

Dubious's static type system is novel in its checking of generic functions. The checks are partitioned into *client-side* and *implementation-side* typechecking [26], also known as *inter-module* and *intra-module* typechecking [18]. Client-side typechecking is completely standard and ensures that clients of a generic function respect the function's declared type. For each message send expression $E\texttt{(}E_1, \ldots, E_n\texttt{)}$ in the program, Dubious checks that E's type is an arrow object $(o_1, \ldots, o_n) \rightarrow o$ and that the type of each E_i subclasses from o_i. The message send expression is then known to yield a value of type o.

While client-side checks ensure that actual arguments to a generic-function application always have appropriate types, they do not rule out message dispatch errors. That is the role of implementation-side typechecking (ITC), which ensures that each generic function properly implements its declared type. First, for each method declaration

$$Ob\ \texttt{has method (}I_1\ \texttt{@}\ Ob_1, \ldots, I_n\ \texttt{@}\ Ob_n\texttt{)}\ \{\ E\ \}$$

Dubious checks that Ob is declared to inherit from an arrow object $(o_1, \ldots, o_n) \rightarrow o$, each Ob_i subclasses from o_i, and E has type o on the assumption that each I_i has type Ob_i. Second, each generic function is checked to be *exhaustive* and *unambiguous*. A generic function is exhaustive if every possible type-correct invocation of the function finds at least one applicable method. An exhaustive generic function will never cause message-not-understood errors at run time. A generic function is unambiguous if every possible type-correct invocation of the function that finds at least one applicable method also finds a most-specific applicable method. An unambiguous generic function will never cause message-ambiguous errors at run time.

By slightly abusing the usual notation, throughout this dissertation I use the abbreviation ITC to refer only to the checks that generic functions are exhaustive and unambiguous; the first implementation-side check described above is straightforward and is not discussed further. Traditional OO languages can be viewed as performing a limited kind of ITC tailored to singly dispatched methods. For example, Java's typechecker produces a

compile-time error if a concrete class does not implement an abstract method of its superclass, because this scenario could cause a `NoSuchMethodException`, the Java equivalent of Dubious's message-not-understood error, to be thrown at run time. Java's lack of multiple inheritance among classes ensures the absence of message-ambiguous errors.

Suppose a Dubious generic function f has arrow object $(o_1, \ldots, o_n) \rightarrow o$. Define $(o'_1, \ldots, o'_n)$ to be a *legal argument tuple* to f if each o'_i is concrete and subclasses from o_i. ITC for f can then be straightforwardly performed by checking that each legal argument tuple has a most-specific applicable method added to f.[2] For example, consider ITC on the `union` generic function in the context of figures 2.2 through 2.5. There are 16 legal argument tuples: all possible pairs of the objects `listSet`, `sListSet`, `hashSet`, and `cListSet`. The `union` method in `SListSetMod` is most-specific for two `sListSet`s, the `union` method in `ListSetMod` is most-specific for all other pairs where each argument is a subclass of `listSet`, and the `union` method in `SetMod` is most-specific for all other pairs.

More efficient ITC algorithms exist than the naive strategy described above. For example, it is possible to define a subset of "interesting" legal argument tuples, such that uninteresting tuples need not be explicitly considered [26]. For the scenario when there is only single *implementation* inheritance (but possibly multiple *interface* inheritance), one can avoid tuple enumeration altogether, instead directly examining and comparing a generic function's methods [20, 63]. That style of ITC algorithm is used in the formalization of EML and is described in chapter 4.

2.2.2 Modular Typechecking

I define a language's typechecking scheme to be *modular* if it has two properties. First, each module m must be able to be typechecked given only the *signatures*[3] of other modules. This property ensures that m never relies on implementation details of another module m' for its well-formedness: the implementation of m' can change without affecting the type

[2] Because Dubious lacks a facility for dynamically creating objects, all objects are statically known. In a traditional class-based language, legal argument tuples would range over a set consisting of one arbitrary instance for each class.

[3] I employ the ML terminology *signature*, rather than the more generic *interface*, to avoid confusion with Dubious's interface objects.

correctness of m, as long as the signature of m' remains the same. Second, m must be able to be typechecked given only those signatures that m *statically depends upon*. Module m statically depends upon signature s if either of the following conditions holds:

- Module m refers to a name that is bound in s.
- Module m statically depends upon signature s', and s' refers to a name that is bound in s.[4]

This property ensures that m depends for its type correctness only on the signatures that are explicitly referenced in m (or in other depended-upon signatures); the rest of the program can be safely ignored. Therefore, m can be typechecked once and then safely reused in any program that includes modules implementing m's depended-upon signatures.

Traditional OO languages support modular typechecking as defined above. Each class can be typechecked given only the statically depended-upon class signatures. Informally, the signature of a class consists of its list of superclasses, the types of its visible fields, and the headers, but not bodies, of its visible methods. Signatures for modular typechecking in the context of Java have been formally defined by others [32, 2]. Traditional functional languages similarly support modular typechecking. For example, each structure in ML can be typechecked given only its statically depended-upon signatures. More details of ML's module system are described in the context of EML in chapter 4.

A modular typechecking scheme for Dubious must typecheck each module given only the signatures it statically depends upon. Dubious does not have an explicit notion of signature. The signature of a Dubious module M is implicitly defined to contain all of the module's `object` declarations and the headers, but not bodies, of its `has method` declarations. Consider modular typechecking for `HashSetMod` in figure 2.4. `HashSetMod` refers to the `set`, `addElem`, and `elems` objects, so it statically depends upon the signature of `SetMod` from figure 2.2. However, `HashSetMod` does not statically depend upon any other modules that are part of the set hierarchy. Therefore, it should be able to be safely typechecked without

[4]The module implementing s' may refer to names not mentioned in s', but these are implementation details that m should not depend upon for its type correctness.

access to any other subclasses of `set` nor to the `isEmpty` object. Indeed, those objects may not even exist when `HashSetMod` is written, and `HashSetMod` may be linked into programs that do not contain those objects. Similarly, `IsEmptyMod` statically depends upon `SetMod` and `ListSetMod`, but not on any other modules in the set hierarchy.

One undesirable feature of a Dubious signature is the fact that it reveals information about a generic function's method implementations. Ideally, clients of a module could be safely typechecked given only the types of objects (including generic functions) declared in the module, without requiring any knowledge of individual `has method` declarations. Such a revised Dubious signature would adhere to Cardelli's notion of modular typechecking as the ability to typecheck a module given only the types of its free variables [18]. However, Dubious's expressiveness makes this stronger notion of modularity infeasible. As discussed below, a client of a module may need information about the module's `has method` declarations in order to ensure that existing generic functions remain exhaustive and unambiguous in the face of the client's declared objects and methods.

Dubious is not alone in suffering from this problem. Indeed, Dubious's notion of modularity naturally generalizes the notion of modularity in traditional OO languages. As mentioned above, the signature for a class in such languages includes the headers of the class's methods, violating Cardelli's notion of modularity. This information is necessary so that subclasses can ensure the exhaustiveness and unambiguity of all operations they support. For example, an abstract class's signature must include information about which of the class's operations are implemented with a concrete method and which are not, so that concrete subclasses can be checked for exhaustiveness. Similarly, a class that inherits from multiple other classes requires knowledge of the method implementations in those classes in order to detect potential ambiguities.

2.2.3 Implementation-side Typechecking and Modularity

Most of the typechecks in Dubious are naturally modular, working on a single declaration or expression at a time. However, ITC as described earlier is a *global* check: for each generic function f, the check requires access to all of f's methods and to all legal argument tuples

```
module ShapeMod {
  abstract object shape
  object overlap isa (shape, shape)→bool
}
```

```
module RectMod {
  object rect isa shape
  overlap has method(s1@rect, s2)
    { ... }
}
```

```
module RhombusMod {
  object rhombus isa shape
  overlap has method(s1, s2@rhombus) { ... }
}
```

Figure 2.6: Challenges for modular implementation-side typechecking of multimethods.

to f. In the context of a modular typechecking scheme, ITC must instead be performed incrementally as part of the typechecks on each module M, given access only to M's *available* objects and methods. Those are the objects and methods that are either *local* to M (i.e., declared in M) or specified in a signature upon which M statically depends.

Therefore, Dubious performs ITC on each module M for every available generic function f whose exhaustiveness or ambiguity is potentially affected by the declarations in M. This is the case if M declares f, declares a method for f, or declares an object that is part of a legal argument tuple to f. Because ITC is modular, the check has access only to the available methods and legal argument tuples for f. For example, consider ITC for the modules in figure 2.2. When `SetMod` is typechecked, ITC for `addElem`, `elems`, and `union` is performed. Since `set` is abstract and is the only available `set` subclass, there are no legal argument tuples to these functions, so the checks succeed vacuously. When `ListSetMod` is typechecked, ITC is again performed on those functions, and all tuples involving `listSet` are checked. Finally, when `SListSetMod` is typechecked, `addElem`, `elems`, and `union` are checked with respect to tuples involving either `listSet` or `sListSet`. In addition, ITC is performed on `getMin` for the single legal argument tuple `(sListSet)`.

Unfortunately, the modular ITC strategy described above is unsound for Dubious. It is possible for modular ITC to succeed on a Dubious program but for that program to nonetheless incur message-not-understood or message-ambiguous errors at run time. The shape hierarchy in figure 2.6 illustrates the kinds of problems that can arise because of multimethods. Modular typechecking succeeds for the three modules in the figure. ITC

on `ShapeMod` succeeds vacuously, since `shape` is abstract. `RectMod` statically depends upon `ShapeMod`, so ITC on `RectMod` checks `overlap`. The only available legal argument tuple is a pair of `rect`s, so the check passes. ITC on `RhombusMod` succeeds analogously. However, `overlap` is neither exhaustive nor unambiguous. If `overlap(rhombus, rect)` is ever invoked at run time, a message-not-understood error will occur. If `overlap(rect, rhombus)` is ever invoked at run time, a message-ambiguous error will occur: both `overlap` methods are applicable to the call but neither is at least as specific as the other.

The exhaustiveness problem above illustrates a tension between multimethods and abstract classes. If `shape` were not declared abstract, then ITC on `ShapeMod` would ensure the existence of an `overlap` method handling two `shape`s, which would thereby also handle one `rhombus` and one `rect`. Therefore, one way to resolve the problem is to disallow objects from being declared `abstract`. However, I reject this solution because it prevents the abstract-class idiom of traditional OO languages from being expressed.

One way to resolve the multimethod ambiguity problem illustrated above is to break the symmetry of the dispatching semantics. For example, Dubious could compare specializers left-to-right instead of pointwise: a method with tuple of specializers $(o_1, \ldots, o_n)$ is at least as specific as a method with tuple of specializers $(o'_1, \ldots, o'_n)$ if the tuples are identical or there exists some i such that o_i subclasses from o'_i, $o_i \neq o'_i$, and the tuples agree on their first $i - 1$ elements. Under that semantics, which is used in Common Lisp [98, 86], the method in `RectMod` is at least as specific as the method in `RhombusMod` and is the most-specific applicable method for the invocation `overlap(rect, rhombus)`. However, a major design goal is for Dubious to retain the symmetric multimethod dispatching semantics, which I believe is more natural and less error-prone. The symmetric semantics reports all potential ambiguities, allowing (and requiring) a programmer to resolve the problem in an appropriate way for the particular application, rather than silently resolving ambiguities via a relatively arbitrary rule.

Figure 2.7 illustrates the challenges for modular ITC in the presence of open classes. `DrawMod` depends upon the signatures of all modules in figure 2.6, but not on the signature of `OtherShapesMod`. ITC on `DrawMod` checks `draw`, which is exhaustive and unambiguous for the two available legal arguments, `rect` and `rhombus`. `OtherShapesMod` also depends upon

```
module DrawMod {
  object draw isa (shape)→void
  draw has method (s@rect) { ... }
  draw has method (s@rhombus) { ... }
}
```

```
module OtherShapesMod {
  object circle isa shape
  object square isa rect, rhombus
}
```

Figure 2.7: Challenges for modular implementation-side typechecking of open classes.

the signatures of all modules in figure 2.6, but not on the signature of `DrawMod`. Therefore, ITC from `OtherShapesMod` does not check `draw`.[5] However, `draw` is neither exhaustive nor unambiguous. If `draw(circle)` is ever invoked at run time, a message-not-understood error will occur. If `draw(square)` is ever invoked at run time, a message-ambiguous error will occur.[6]

As above, the exhaustiveness error results from an interaction with abstract classes. If `shape` were not declared `abstract`, then `draw` would be forced to have a method handling `shape`, thereby also handling `circle`. Also, the ambiguity problem can again be resolved by introducing an asymmetric dispatching semantics. For example, the inheritance hierarchy could be linearized, as is done in Common Lisp and Dylan [36, 96], or the textual ordering of methods could be employed, as in parasitic multimethods [11]. However, I reject these solutions for the same reasons as described earlier.

It is because of the kinds of problems illustrated in figures 2.6 and 2.7 that all previous languages supporting symmetric multiple dispatch or open classes have been forced to give up modular typechecking. In these languages, ITC can only safely be performed when the entire program is available. Therefore, the increased component extensibility compared with traditional OO languages comes at a cost of decreased component understanding and reliability.

---

[5] `OtherShapesMod` would require some methods to pass ITC checks on `overlap`, but I elide these methods, as they are irrelevant to the example.

[6] A similar ambiguity problem can result from the interaction of multiple inheritance with multimethods.

2.3 Modular ITC for Dubious

This section describes a fully modular type system for Dubious, called *System M* (for *modular*), that resolves the problems illustrated in the previous section. However, there is no free lunch: in exchange for modular typechecking, System M must give up some of Dubious's expressiveness. It does this by imposing four *modularity requirements*, which are informally described below. These requirements are inspired by an understanding of the (often implicit) requirements in traditional OO languages like Java and C++ that facilitate modular ITC. System M's requirements generalize those of traditional OO languages to support the common uses of multimethods and open classes, while still retaining fully modular ITC.

A key notion in System M's modularity requirements is that of the *owner argument position* of a generic function. A function's owner position has some properties in common with the receiver position in standard OO languages. The owner position serves to coordinate otherwise independent extensions of a function, ensuring that they cannot interact to cause message-not-understood or message-ambiguous errors. The owner position does not impact Dubious's dispatching semantics, which remains completely symmetric. A generic function's owner position is signaled by a hash (`#`) mark at the corresponding position in the generic function's arrow object. The object at that position is called the generic function's *owner*. Similarly, the owner of a method is the object used as a specializer at the owner position of the associated generic function.

The revised grammar for Dubious objects is as follows:

$$\begin{array}{lcll} O & ::= & Ob \mid (A_1, \ldots, A_n) \rightarrow O & \textit{objects} \\ A & ::= & [\texttt{\#}]\ O & \textit{argument types} \end{array}$$

System M requires that exactly one argument type of a generic function is declared the function's owner. In figure 2.6, assume that the first argument position of `overlap` is the owner. In figure 2.7, the single argument position of `draw` is the owner.

2.3.1 Ensuring Unambiguity

In traditional OO languages, each method must dynamically dispatch at the receiver position on the enclosing class. This implicit requirement ensures that no ambiguities elude

modular detection. Consider method m_1 in class C_1 and m_2 in class C_2, where m_1 and m_2 are part of the same (implicit) generic function. In a language with only single inheritance, like Java, the requirement ensures that, if C_1 is not available to C_2 and vice versa, then m_1 and m_2 cannot be ambiguous. In particular, it must be the case that neither C_1 nor C_2 subclasses from the other (or else one would be available to the other), which implies that the methods are *disjoint*, applicable to completely distinct legal argument tuples. In a language with multiple inheritance, like C++, the methods may not be disjoint if a class C_3 later subclasses from both C_1 and C_2, but any ambiguity will be detected during ITC on C_3.

The Dubious analogue of the implicit requirement in traditional OO languages is that each method must have a *local owner*. This requirement resolves the ambiguity problem in figure 2.6. The `overlap` method in `RhombusMod` does not have a local owner — the method's owner position (the first argument) is unspecialized, so the owner is implicitly the nonlocal `shape` object. Therefore, `RhombusMod` fails modular ITC, and that method may not be written, resolving the ambiguity for `overlap(rect, rhombus)`. (An isomorphic error would occur in `RectMod` if the second argument position of `overlap` were instead designated the owner position.)

The local-owner requirement restricts the kinds of multimethods that may be written, but it still allows expression of the common form of OO extensibility while preserving safe modular ambiguity checking. In traditional OO languages, existing classes may be arbitrarily extended with new subclasses that provide overriding methods (for which that subclass is the receiver). Dubious's System M analogously allows existing objects to be arbitrarily extended with new subclasses that provide overriding methods for which that subclass is the owner. The local-owner requirement also allows idioms not supported by traditional OO languages. Owner-local methods are unrestricted in how multiple dispatch may be employed at nonowner positions, able to dispatch on arbitrary (local or nonlocal) objects. Further, Dubious modules are more flexible than classes, for example allowing multiple objects with their associated owner-local methods to be declared in a single module.

While the local-owner requirement resolves the multimethod ambiguity problem, it also has the effect of disallowing the open-class idiom. For example, none of the methods in

`DrawMod` of figure 2.7 has a local owner, so they would be forbidden under the requirement. To circumvent this problem, the requirement is generalized in a natural way: each method must either have a local owner *or a local generic function*. This generalization is intuitively safe because, if a method m belongs to a local generic function f, then the method will be available to any module M that adds a method m' to f. Therefore, checks on M can modularly detect any ambiguities between m' and m (except for ambiguities caused by multiple inheritance, which are discussed next). The generalized requirement provides the same solution as discussed above for handling the ambiguity in figure 2.6 while still safely allowing the open-class idiom of figure 2.7. As another example, all of the methods in the set hierarchy of figures 2.2 through 2.5 satisfy the generalized requirement.

Finally, the ambiguity problem illustrated in figure 2.7 must be addressed. To do so, System M restricts the forms of multiple inheritance to those that can be modularly reasoned about in the presence of open classes. Intuitively, arbitrary multiple implementation inheritance is allowed within a module, but only single implementation inheritance is allowed across module boundaries. More precisely, if a noninterface object o has a nonlocal noninterface superclass, then System M requires that o have a most-specific (according to the subclass relation) nonlocal noninterface superclass. The `square` object in `OtherShapesMod` does not satisfy the requirement: it has two nonlocal superclasses, `rect` and `rhombus`, neither of which is a subclass of the other. Therefore, `OtherShapesMod` fails modular ITC, and `square` cannot be programmed.

However, the requirement is not as limiting as it might appear. First, an object may still safely inherit from multiple nonlocal interfaces. For example, if `rect` were an interface then the declaration of `square` would pass the requirement, because `rhombus` would be its most-specific nonlocal noninterface superclass. In that case, the second `draw` method in `DrawMod` could not exist, since interfaces may not be specializers, thereby removing the ambiguity for `square`. In this way, System M safely retains the same form of multiple inheritance as is provided in Java and C#. Further, cross-module multiple implementation inheritance is still possible, as long as it is anticipated. For example, the implementer of `RectMod` could plan ahead for later multiple inheritance by introducing an object `rr` that inherits from both `rect` and `rhombus`. The `square` object can then singly inherit from `rr`,

thereby passing System M's requirement. At the same time, `rr` is available in `DrawMod`, so the ambiguity is modularly detected during ITC on `DrawMod`.

In summary, the ambiguity requirements are as follows:

M1. Each method must have either a local owner or a local generic function.

M2. If a noninterface object has a nonlocal noninterface superclass, then it has a most-specific nonlocal noninterface superclass.

2.3.2 Ensuring Exhaustiveness

For the purposes of the exhaustiveness requirements, it is useful to distinguish two kinds of generic functions. A generic function is called *internal* if it is declared in the same module as its owner; otherwise the generic function is *external*. An internal generic function can be thought of as an "initial" operation of its owner object and subclasses, while an external function is a later extension to their sets of operations. External functions have no analogue in traditional OO languages, in which a class's methods must all be declared with the class.

In traditional OO languages, a method whose receiver is an abstract class may be declared abstract. In turn, each concrete subclass of that receiver must declare or inherit an implementation of the method. System M adopts an analogous solution to ensure exhaustiveness for *internal* generic functions in Dubious. An abstract object o that is the owner of an internal generic function f need not have an applicable method of f. In turn, the module introducing a concrete subclass o' of o must declare or inherit a *local default* method of f for o'. A local default for o' is a method whose owner is o' but is unspecialized at all other argument positions. Local default methods are the Dubious analogue of the methods that are required to ensure exhaustiveness in traditional OO languages, which dispatch only on the receiver. Therefore, the local-default requirement is no more burdensome than the requirements in those languages.

The existence of local defaults is ensured in System M by treating even abstract and interface objects as concrete at argument positions other than the owner position, for the purposes of ITC. The abstract and interface objects act as surrogates for their potentially

unavailable concrete subclasses. Under this revised ITC algorithm, `RhombusMod` in figure 2.6 fails modular ITC because `overlap` is not exhaustive for the (now legal) argument tuple `(rhombus, shape)`. (An isomorphic error would occur in `RectMod` if the second argument position of `overlap` were instead designated the owner position.) If `RhombusMod` contained a local default method of the form

```
overlap has method(s1@rhombus, s2) { ... }
```

then modular ITC would succeed and the exhaustiveness for `overlap(rhombus, rect)` would be resolved. As another example, in figure 2.2 `set` safely remains unimplemented for `addElem` and `elems`, and each concrete subclass implements the appropriate local defaults.

Unfortunately, the above requirement cannot work in general for external generic functions. The problem is that an external generic function f may not be available in every module containing a concrete subclass of f's owner. For example, in figure 2.7 `draw` is not available in `OtherShapesMod`, so ITC is not performed on `draw` during `OtherShapesMod`'s checks. Therefore, no local default method is required for `circle`. To circumvent this problem, System M instead places responsibility for ensuring exhaustiveness on the module introducing an external generic function f. That module must declare a *global default* method for f, which is a completely unspecialized method. Again this is done by treating nonconcrete objects as concrete for the purposes of ITC, but this time it is done at all argument positions, including the owner position. Given the new requirement, `DrawMod` fails modular ITC because it is not exhaustive for `shape`. Including a global default method allows ITC to succeed and resolves the exhaustiveness problem for `circle`. As another example, the external `isEmpty` generic function in figure 2.3 satisfies the requirement because its first method is a global default, thereby handling unavailable `set` subclasses like `hashSet` in figure 2.4.

In order to safely support the expressiveness of external generic functions, System M's global-default requirement gives up the abstract-class idiom on these functions. However, this is no restriction on traditional OO languages, as those languages do not even allow external generic functions to be written. Abstract classes may still safely employ the abstract-class idiom on all internal generic functions.

In summary, the exhaustiveness requirements have the following effect:[7]

M3. If f is an internal generic function and o is a concrete object that subclasses f's owner, then the module declaring o must declare or inherit a local default of f for o.

M4. If f is an external generic function, then the module declaring f must declare a global default of f.

Although System M's exhaustiveness requirements are no more restrictive than those of traditional OO languages, they are not always easy to respect. The ability to write an appropriate global default method depends heavily on an object's initial set of operations. In particular, the global-default requirement only works well for external functions whose behavior can be expressed solely in terms of that initial set. For example, the requirement is easy to obey for `isEmpty`, as shown in figure 2.3. However, had `SetMod` in figure 2.2 not included the `elems` function, it is unlikely that a reasonable default case for `isEmpty` could be written. Instead, the implementer may have no choice but to make the global default method simply throw an exception (assuming Dubious had exceptions), which is not much different from the run-time message-not-understood error being prevented by the global-default requirement. I have explored two approaches to overcoming this problem, in the context of MultiJava and Eml, respectively; they are described in the next two chapters.

2.3.3 Discussion

System M supports all of the extensibility idioms of traditional OO languages like Java and C#. It additionally supports owner-local multimethods and arbitrary open classes. Therefore, System M allows both new subclasses and new operations to be added to existing objects without modifying existing code, resolving a longstanding tension between these two forms of extensibility. At the same time, System M retains completely modular

[7] Treating some nonconcrete objects as concrete during ITC is necessary to catch ambiguities due to multiple inheritance. For example, if the `rr` object described at the end of section 2.3.1 is abstract, then treating it as concrete is necessary in order to detect the ambiguity for `draw` during ITC on `DrawMod` in figure 2.7. In the absence of multiple inheritance, the exhaustiveness requirements can be expressed by directly requiring the appropriate local and global defaults. Chapter 4 uses this style to formalize Eml's type system.

typechecking. The cost of the extra expressiveness provided by owner-local multimethods and open classes is a loss of unanticipated multiple implementation inheritance; multiple interface inheritance is allowed arbitrarily.

Although the owner position of a generic function f does not affect f's dispatching semantics, it has a large impact on the ways in which clients can interact with and extend the function. Through requirement M1, the owner position affects what methods may be added to f by clients. Through requirement M3, the owner position determines the client objects for which f may safely remain unimplemented. Therefore, effective use of f by clients requires some advance planning by the original implementer of f, who must choose an appropriate owner position based on the kinds of extensibility that are anticipated. Choosing incorrectly can disallow clients from extending f as desired, making it less reusable. This advance planning is analogous to the need to choose which argument of a generic function should be the receiver in traditional OO languages. That choice has the same effects as the choice of the owner position in System M and additionally determines the sole argument that may be dynamically dispatched upon.

Intuitively, an object o in f's arrow object is a good choice as the function's owner if the following conditions hold:

- It is expected that f's behavior will depend on the particular subclass of o that is passed as an argument.

- It is expected that clients will declare new subclasses of o.

The first condition implies that methods will likely need to dispatch at o's position. The second condition then implies that clients will likely need to declare overriding methods of f for new subclasses of o. Making o the owner allows these overriding methods to satisfy requirement M1. The first condition also implies that it may be difficult to implement f for o itself, if o is nonconcrete. Making o the owner will obviate the need for such a default implementation when f is internal, instead requiring concrete subclasses of o to provide their own local defaults by requirement M3.

The System M ambiguity and exhaustiveness requirements together enforce an important

monotonicity property on Dubious programs: the local view of a program from one module is always consistent with the global view in a particular sense. Specifically, if the view from one module suggests that some available legal argument tuple t to an available generic function f will invoke the available method m, then that will be the case at run time, no matter what other modules are part of the complete program. This property ensures that ITC can safely be performed piecewise on a generic function, from each module's partial view of the program. It also validates a programmer's understanding given only a partial view of the program, ensuring that an argument tuple cannot be "hijacked" by an unseen method.

I have formalized Dubious and proven its type system *sound* [74]. This is the formal guarantee that a well-typed Dubious program cannot incur type errors, including message-not-understood and message-ambiguous errors, at run time. Therefore, Dubious's soundness proof validates the correctness of System M's modular ITC requirements. A simplified formalization of System M in the context of EML is presented in chapter 4, and its soundness proof is provided in appendix A.

2.4 Related Work

This section describes other multimethod calculi and work on modular typechecking for symmetric multimethods. Later chapters describe source-level languages containing multimethods and open classes.

Chambers and Leavens made the first effort toward modular typechecking of symmetric multimethods [26], in the Cecil language [24, 25]. They defined the notions of client- and implementation-side typechecking. They provided a global implementation-side typechecking algorithm and sketched informal ideas for modular typechecking. BeCecil [27] is a core language for multimethods, intended as a formalization of this earlier work. It includes the same core features as Dubious, as well as types and subtyping separate from objects and inheritance and a block structure that allows arbitrarily nested declarations. BeCecil's dispatching semantics is more complicated than Dubious's, with `inherits` (BeCecil's version of `isa`) and `has method` declarations associated with particular scopes and only visible to

certain call sites. As a result, modular typechecking was not achieved. Dubious treats `isa` and `has method` declarations as having global extent dynamically, simplifying the semantics enough for us to develop modular typechecking rules and prove their soundness.

The λ&-calculus and variants [21, 20] are a family of calculi based on symmetric multimethods. Dispatching is performed on types which, along with the subtyping relation, are predefined rather than programmer-constructed. Some of the calculi have second-order type systems, which Dubious lacks. The language λ_object augments the calculi with the ability to define new types and subtyping relations. The λ&-calculi and λ_object trivially support modular typechecking, because the declaration of a generic function includes the declaration of all of its methods, as in traditional functional languages.

Chapter 3

MULTIJAVA

This chapter describes MultiJava [29, 28, 79], a backward-compatible extension to Java [5, 45] supporting multimethods and open classes. Dubious's System M allows the new idioms to be naturally incorporated into Java while preserving Java's existing modular typechecking strategy.

3.1 MultiJava By Example

3.1.1 Multimethods

Figure 3.1 shows a translation of the first two Dubious modules in figure 2.2 to MultiJava.[1] The only part of the code that is not written in standard Java is the `union` method in `ListSet`, which illustrates how multimethod dispatch is declared in MultiJava. A formal parameter is specialized via the syntax `<StaticType>@<DynamicType>`. The static type part of a specialized formal parameter is used to identify to which generic function the new method belongs, in the absence of explicit generic functions. A generic function can be viewed as implicitly being formed in Java by a *top method*, which does not override any other methods, and all the methods that override that top method. In Java, a method m_1 overrides another method m_2 if m_1's receiver is a subtype of m_2's receiver and the methods have the same static argument types and return type. `Set`'s `union` method is a top method, because it does not override another method; it implicitly introduces a new generic function. The argument to `ListSet`'s `union` method has a static type of `Set`, which signifies that the method overrides the `union` method in `Set`; they are part of the same generic function.

The dynamic type, or specializer, of a specialized argument plays the same role as a

[1] The third module and the `elems` generic function are omitted because they do not add anything interesting to the example.

```
// file Set.java
package sets;
public abstract class Set {
  public abstract Set addElem(int i);
  public Set union(Set s) { ... }
}

// file ListSet.java
package sets;
public class ListSet extends Set {
  public Set addElem(int i) { ... }
  public Set union(Set@ListSet ls) { ... }
}
```

Figure 3.1: An implementation of integer sets in MultiJava.

specializer in Dubious. Because `ListSet` is the dynamic type of the argument to `ListSet`'s `union` method, the method is only applicable to an invocation at run time if both the receiver and argument values are instances of `ListSet` or a subclass. As in Dubious's System M, a specializer must be a class rather than an interface.

Had `ListSet`'s `union` method simply used an argument type of `ListSet` instead of `Set@ListSet`, Java would not consider the method to override the `union` method in `Set`. Instead the methods would be statically overloaded, as described in section 1.2.2. In Dubious's terminology, the `union` method in `ListSet` would implicitly declare its own generic function. MultiJava's syntax for a specialized argument cleanly separates static overloading, which uses the given static type, from dynamic dispatch, which uses the given dynamic type.

As in Dubious, MultiJava uses the symmetric multimethod dispatching semantics. Also like Dubious, a method may specialize on any subset of its arguments, and different methods of the same generic function may specialize on different arguments. This flexibility allows regular Java methods to be seamlessly overridden by multimethods. For example, one can override `java.lang.Object`'s `equals` method with a method in `Set` that tests equality of two sets as follows:

```
public boolean equals(Object@Set s) { ... }
```

The syntax for message sends is unchanged from Java, so clients are unaffected by whether

```
// file isEmpty.java
package extension1;
import sets.*;
public boolean Set.isEmpty() { ... }
public boolean ListSet.isEmpty() { ... }
```

Figure 3.2: External methods in MultiJava.

or not a given generic function is implemented using multimethods.

3.1.2 Open Classes

Open classes in MultiJava are illustrated in figure 3.2, a translation of the Dubious code in figure 2.3. MultiJava extends Java with the ability to write *external methods*, which are methods declared outside of their respective receiver classes. As shown in the figure, external methods can be declared in separate files and separate packages from their receivers. The first external method in figure 3.2 implicitly declares a new generic function, and the second method is part of that same generic function. External methods may also be multimethods. In MultiJava a generic function is said to be *external* if its top method is external; otherwise the generic function is *internal*.[2] All generic functions are invoked using the ordinary Java message-send syntax, so call sites are insensitive to whether an external or internal generic function is being invoked.

Unlike ordinary methods of a class, external methods have no special access privileges to their receivers. Instead, an external method has the same access as any other client of its receiver. For example, an external method may not access private fields and methods of its receiver class. Similarly, an external method may not access members of its receiver class that have package-level visibility unless the external method is defined in the same package as its receiver. While these rules restrict the utility of open classes in some cases, they are critical for ensuring modular reasoning, both by tools (like typecheckers and compilers) and by programmers. For example, if an external method had access to the private data of

[2] As a special case, an external method is treated as a regular Java method if it is declared in the same file as its receiver. In that case, the MultiJava compiler simply inserts the method inside the class.

```
// file CListSet.java
package extension2;
import sets.*;
import extension1.isEmpty;
public class CListSet extends ListSet {
  public Set addElem(int i) { ... }
  public boolean isEmpty() { ... }
}
```

Figure 3.3: Internal methods overriding external methods in MultiJava.

its receiver, then `private` would cease to mean private: any client could get access to the private members of a class simply by writing an external method on the class. Any invariants established and maintained on private data could then be potentially compromised.

In a similar vein, the semantics of a privacy modifier on an external method is defined relative to the file in which the method resides, rather than the file containing the method's receiver. A private external method is only accessible in the file in which it is declared, and an external method with no privacy modifier is only accessible in the package in which it is declared. For simplicity MultiJava requires that all methods belonging to the same external generic function have identical privacy modifiers. This requirement ensures that a method will never be invoked from a scope in which the method is inaccessible. It would be safe to relax this requirement, allowing a method to optionally be more accessible than the methods it overrides. Java uses this relaxed rule when checking the privacy modifiers of ordinary methods.

Like classes in Java and objects in Dubious, external generic functions are *scoped*. Clients that invoke or add methods to an external generic function must first *import* it. For example, consider figure 3.3, which translates the Dubious code in figure 2.5 to MultiJava. The `CListSet` class contains a regular Java method that overrides the external methods in figure 3.2. To do so, the associated external generic function is explicitly imported. Had it not been imported, the generic function would not have been visible to `CListSet`. Therefore, the `isEmpty` method in `CListSet` would instead implicitly declare a new generic function. There can also exist `isEmpty` external generic functions in other packages with the same type signature as the one in figure 3.2. Regular static scoping disambiguates these generic

```
public class Fib {
  int fib(int@@0 i) { return 0; }
  int fib(int@@1 i) { return 1; }
  int fib(int i) { return fib(i-1) + fib(i-2); }
}
```

Figure 3.4: Value dispatching in MultiJava

functions: at most one of them may be imported by a given client, or else a compile-time error is signaled.

The receiver of an external generic function's top method may be an interface. However, the receivers of all other methods in the generic function must be classes. This requirement is the analogue of the rule disallowing interface specializers in multimethods: the receiver type of the generic function's top method is the static type of the generic function's receiver position and so is not really being dispatched upon.

3.1.3 Value Dispatching

Multimethods in MultiJava as presented so far are useful whenever a generic function's behavior depends upon the particular class of an actual argument. However, a generic function's behavior sometimes depends upon an actual argument's *value* rather than its class. MultiJava's multimethods generalize naturally to support a common case of dispatch on values, in which the depended-upon values are compile-time constants.[3]

Figure 3.4 illustrates how value dispatching is used to compute the Fibonacci numbers. Value dispatch is denoted using `@@` instead of `@` on a specialized position. The first method in the figure is only applicable dynamically if the argument `i` has the value 0, and similarly for the second method. The existing multimethod dispatch semantics generalizes seamlessly to handle value dispatching by viewing each dispatched-upon value as a singleton concrete subclass of its associated type. For example, the first `fib` method overrides the third one because 0 is a "subclass" of `int`. Any compile-time constant expression as defined by Java

[3]Dubious already supports this form of value dispatching with no extra mechanism, because its methods dispatch on objects, which act both as classes and as values.

can be used as a value specializer, in addition to simple literals. Examples that make good use of this ability are described in chapter 5.

Value dispatching as shown in the example is similar to Java's existing `switch` statement, but value dispatching has a number of advantages. First, any subset of a method's arguments can employ value dispatching, and a method can employ value dispatching on some arguments and ordinary class dispatching on others. Second, value dispatching is supported for all of Java's primitive types, instead of just the integral ones, as well as for `java.lang.String`. Finally, value dispatching interacts with subclassing to provide a form of extensible `switch`. For example, a subclass of `Fib` can inherit some of the `fib` methods, override others, and add new multimethods handling other interesting integer values.

The only Java literal that is currently not supported as a value specializer is `null`. Several possible semantics for this extension are possible, but to date we have not settled on one. In MultiJava currently, only unspecialized argument positions are deemed to be applicable to `null`. This semantics violates the notion of `null` as a singleton "subclass" of every type, but it preserves the semantic correspondence between multimethods and Java's `instanceof` tests (as shown, for example, in figure 1.4), since the value of `(null instanceof C)` is `false` for every class `C`.

3.1.4 Resends

Inspired by a similar construct in Cecil [25], MultiJava augments Java with a `resend` expression, which is similar to Java's `super` construct but walks up the multimethod specificity ordering. An invocation of `resend` from a method invokes the unique most-specific method that is overridden by the resending method. (It is a compile-time error if such a method does not exist.) For example, an invocation of `this.resend(ls)` from `ListSet`'s `union` method in figure 3.1 invokes `Set`'s `union` method. In this example, the `resend` has the same semantics as invoking `super.union(ls)`, but that is not always the case. A `super` send will always invoke a method whose receiver is a strict superclass of the current receiver, while this need not be true for `resend`. For example, suppose `ListSet` also had a singly dispatched `union` method, accepting any `Set` instance as an argument. In that case, the

earlier `resend` would invoke the new method, because it would be the most-specific method overridden by the resending method, while the `super` send would still invoke `Set`'s `union` method.

To ensure that the unique most-specific method that is overridden by the resending one will be applicable to the `resend`'s arguments, MultiJava requires that the receiver of a `resend` be `this` and that the ith argument to the `resend` be the ith formal parameter of the enclosing method. Further, these formals must be declared `final` in the enclosing method, to guarantee that they are not modified before the call to `resend`.

An alternative design for resends is to modify `super`'s semantics to act like `resend`, instead of introducing a new keyword. An earlier version of MultiJava did exactly that. That design ensures that the "right" semantics for invoking overridden methods is always used. Further, it is still backward-compatible with Java, differing in the semantics of `super` only in the presence of MultiJava's new constructs. However, that design has several drawbacks compared with the use of `resend`. First, some users were confused by the generalized `super`'s semantics, still expecting it to always invoke a method of the current receiver's superclass. Second, the new `resend` keyword allows us to better control the idiom's usage. For example, `super` sends may invoke generic functions other than the current one. On the other hand, the appropriate semantics of `resend`s to arbitrary generic functions is unclear, so the `resend` syntax simply disallows it. Finally, retaining `super`'s traditional semantics aids a program's migration from Java to MultiJava. For example, in the original version of MultiJava, the target method of a `super` send inside some Java method could be altered if the method were later converted into a set of multimethods.

3.2 Modular Implementation-side Typechecking

Java programs can be safely typechecked modularly, as defined in section 2.2.2: each file can be typechecked in isolation, given only the signatures of the classes it statically depends upon. MultiJava augments Java's modular typechecks to include implementation-side typechecking for multimethods and external methods. For each available generic function in a file, MultiJava checks that every available legal argument tuple has an available most-

specific applicable method. As with Dubious, an unrestricted MultiJava program may pass modular ITC while still signaling message-not-understood or message-ambiguous errors at run time. Therefore, MultiJava adapts Dubious's System M from section 2.3 in order to safely perform ITC in the context of Java's modular typechecking scheme. Rather than having each generic function specify its owner, the receiver position is implicitly used as the owner position. A key property of the System M requirements is that they still allow all of the idioms of traditional OO programming to be expressed. Therefore, although some multimethod and open-class idioms are disallowed in MultiJava, the language is still strictly more expressive than Java.

First consider the System M ambiguity requirements, described in section 2.3.1. In the context of MultiJava, requirement M1 ensures that each external method is declared in the same file as its top method. For example, `ListSet`'s `isEmpty` method in figure 3.2 is declared in the same file as its top method. Requirement M1 also allows that method to be declared as a regular method inside of the `ListSet` class in figure 3.1. If the method is declared in any other place, a compile-time error is reported when the enclosing file is typechecked. Requirement M2 is unnecessary in MultiJava, because Java classes support only single inheritance. As in Dubious, interfaces support unrestricted multiple inheritance, and that remains safe because interfaces may not be dispatched upon (either in a multimethod argument or as the receiver of an overriding external method).

Now consider the System M exhaustiveness requirements. Requirement M3 ensures the existence of a local default method for each internal generic function declared or inherited by a concrete class. This requirement is the same one that Java already uses to ensure that generic functions are exhaustively implemented. For example, suppose `Set`'s `union` method in figure 3.1 were declared `abstract`. In that case, M3 requires `ListSet` to contain a regular singly dispatched implementation of `union`, just as a Java typechecker would require. Requirement M4 ensures the presence of a global default method for each external generic function. For example, the first `isEmpty` method in figure 3.2 serves as the global default. To ensure that a global default method is applicable to all legal argument tuples, the method

must not have any abstract overriding methods.[4] Therefore, MultiJava disallows a method belonging to an external generic function from being declared `abstract`. Similarly, to ensure the sufficiency of local default methods, MultiJava disallows a multimethod from being declared `abstract`. All of Java's abstract-class idioms are still expressible in MultiJava.

ITC is naturally generalized to support modular exhaustiveness and ambiguity checking in the presence of value dispatching. As mentioned earlier, each value used as a specializer is considered to be a singleton concrete subclass of its associated type. Further, this type is considered to be an abstract class for the purposes of ITC. For example, during ITC on `fib` in figure 3.4, the values 0 and 1 are checked to have a most-specific applicable method. This checking ensures, among other things, that an ambiguity will be signaled statically if multiple `fib` methods dispatch on 0. Requirement M3 ensures that `int` is treated as concrete during ITC, thereby requiring the third `fib` method in figure 3.4, which serves as the local default. This method is necessary to handle arguments other than 0 and 1.

MultiJava currently uses the above strategy to handle all value dispatching, but it is possible to safely omit some default methods when dispatching on booleans. For example, a generic function that accepts a single argument of type `boolean` is exhaustive if it contains two methods, one dispatching on `true` and the other on `false`; a default method would be superfluous. To safely omit default methods, ITC in the presence of boolean dispatching should always consider both `true` and `false` values explicitly as concrete subclasses of `boolean`, even if only one of them is dispatched upon. Since all boolean values are thereby represented during ITC, the `boolean` type need not be considered concrete as dictated by requirements M3 and M4, thereby obviating the need for some default methods.

3.3 Implementation

A MultiJava compiler, *mjc*, has been written as an extension to the Kopi compiler for Java [60], which is itself written in Java. The MultiJava compiler is freely available for download [79]. In addition to modular typechecking via Dubious's System M, mjc also

[4] This requirement is necessary because in Java, a concrete method that is applicable to a message send may not be invoked if there is an overriding abstract method that is also applicable.

supports modular compilation of MultiJava programs. As in Java, each file can be compiled separately, to standard Java bytecode class files. Because no extension to the bytecode language is required, MultiJava code can be run on a stock Java virtual machine, and MultiJava code interacts seamlessly with third-party Java libraries. These properties were critical for allowing users to easily migrate from Java to MultiJava. User experience with MultiJava is described in chapter 5.

MultiJava's implementation and modular compilation strategy are largely the work of Curt Clifton and are discussed in detail in his master's thesis [28]. The modular compilation strategy relies on System M's requirements for its correctness. Intuitively, the monotonicity property provided by System M, mentioned at the end of chapter 2, ensures that a generic function's dynamic dispatching logic can safely be generated piecewise, given only partial knowledge of the program.

3.4 Related Work

There are several other source-level languages supporting multimethods or open classes. Cecil [24, 25] is a statically typed, prototype-based object-oriented language supporting multimethods written external to their associated objects. Cecil requires the whole program to safely perform ITC [68], and it is this problem that Dubious was originally created to resolve. Dubious is essentially a distillation of Cecil down to a core set of constructs that illustrate the problem for modular typechecking of multimethods and open classes. Vortex [31], the compiler for Cecil (and other languages), employs a global compilation strategy that makes heavy use of whole-program optimization.

Common Lisp [98, 86] and Dylan [96, 36] are both multimethod-based languages with generic functions. All methods are written external to their classes. To avoid run-time ambiguities, Common Lisp totally orders the arguments of a generic function; Dylan uses the symmetric semantics, as in MultiJava. Both Common Lisp and Dylan totally order the inheritance hierarchy, eliminating the potential for multiple-inheritance ambiguities. The languages are dynamically typed, so they do not consider the issue of static typechecking, modular or otherwise.

Polyglot [1] is a database programming language akin to Common Lisp with a first-order static type system. There are no abstract methods, so there is no possibility of message-not-understood errors. Further, the dispatching semantics uses Common Lisp-style total ordering of multimethod arguments and inheritance, avoiding all ambiguities. Therefore, only the monotonicity of the result types [21, 94] of multimethods needs to be checked to ensure modular type safety.

Kea [78] and Tuple [63] are statically typed, class-based languages with symmetric multimethods. Kea has a notion of separate compilation, but this requires run-time ITC of generic functions. Tuple requires the whole program to be available in order to perform ITC statically. In Tuple, all multimethods are written as external methods that dispatch on an explicit tuple of arguments as the receiver, thereby cleanly separating specialized from unspecialized argument positions. An early design for MultiJava adapted this style in conjunction with Dubious's System M, but colleagues and I rejected that in favor of the current design for a few reasons. Tuple requires that all methods of the same generic function have identical specialized and unspecialized argument positions. This means, for example, that a multimethod cannot override an existing Java method. Further, because the distinction between specialized and unspecialized arguments is visible to clients in Tuple, converting an unspecialized argument of a generic function to be specialized, or vice versa, requires modifying all callers.

Encapsulated multimethods [22, 15] are a design for adding multimethods to an existing singly dispatched object-oriented language. An encapsulated multimethod is written inside of its receiver's class; external (multi)methods are not supported. Encapsulated multimethods involve two levels of dispatch. The first level is just like regular single dispatch to the class of the receiver object. The second level of dispatch is performed within this class to find the best multimethod applicable to the dynamic classes of the remaining arguments. The encapsulated style can lead to duplication of code, since multimethods in a class cannot be inherited for use by subclasses.

Parasitic methods [11] and Half & Half [7] are both extensions to Java that support encapsulated multimethods. Both augment the encapsulated style with the ability to inherit multimethods from superclasses. The resulting expressiveness is comparable to that of Mul-

tiJava's internal multimethods, but MultiJava additionally retains the natural symmetric multimethod dispatch semantics. MultiJava also supports open classes and value dispatching. Both parasitic methods and Half & Half support the use of interfaces as specializers in multimethods. Because it is difficult to modularly check multimethod ambiguity in the presence of interface specializers, parasitic methods modify the multimethod dispatching semantics so that ambiguities cannot exist, employing the textual order of methods to break ties. Half & Half resolves the problem by performing ITC on entire packages at a time, rather than on individual classes. For such package-level checking to be safe, Half & Half must also limit the visibility of some interfaces to their associated packages, thereby disallowing outside clients from employing them as specializers. In addition to multimethods, Half & Half supports *retroactive abstraction*, the ability to add new superclasses and superinterfaces to existing classes.

Jiazzi [71] is an extension to Java based on *units* [40, 37], a powerful form of parameterized module. Jiazzi supports extensibility idioms not provided by MultiJava, such as the ability to implement a *mixin* [12, 37, 41], which is a class parameterized by its superclass. The authors also show how to encode an *open class pattern* in Jiazzi, whereby a module imports a class and exports a version of that class modified to contain a new method or field. Open classes in MultiJava allow two clients of a class to augment the class in independent ways, without having to be aware of one another. In contrast, in Jiazzi there must be a single module that integrates all augmentations, thereby creating the final version of the class. Module linking in Jiazzi is performed statically, so it is not possible to dynamically add new methods to existing classes. Dynamic augmentation is possible in MultiJava, since open classes are integrated with Java's regular dynamic loading process.

Recently several languages have emerged that provide direct support for separation of concerns. For example, AspectJ [58] is an aspect-oriented [59] extension to Java, whose *aspects* can extend existing classes in powerful ways. Hyper/J [85] is a subject-oriented [48] extension to Java that provides *hyperslices*, which are fine-grained modular units that are composed to form classes. Both languages support open classes; for example, this ability corresponds to AspectJ's *introduction* methods. The languages additionally support many more flexible extensibility mechanisms than MultiJava. For example, AspectJ's *before* and

after methods provide ways of modifying existing methods externally. To cope with this level of expressiveness, these languages employ non-modular typechecking and compilation strategies. For example, AspectJ's compiler "weaves" the aspects into their associated classes; only when all aspects that can possibly affect a class are available for weaving are typechecking and compilation performed.

Binary Component Adaptation (BCA) [57] allows programmers to define *adaptation specifications* for their classes, which can include the addition of new methods, thereby supporting open classes. Adaptation specifications can also include modifications not supported by MultiJava, like retroactive abstraction. The typechecking and compilation strategy is similar to the aspect weaving approach described above, requiring access to all adaptation specifications that can affect a given class in order to typecheck and compile the class. The authors describe an on-line implementation of BCA, whereby the weaving is performed dynamically using a specialized class loader.

3.5 Current and Future Work

Further work on MultiJava is focused on increasing its expressiveness while retaining support for modular reasoning. Colleagues and I are currently developing an extension to MultiJava called Relaxed MultiJava (RMJ) [75]. We observe that violations of the System M requirements indicate only the *potential* for a generic function to be nonexhaustively or ambiguously implemented, rather than an actual error, because System M must be conservative given only a modular view of the program. RMJ performs the same modular static typechecking as MultiJava, but RMJ treats a violation of the System M requirements as a *warning* rather than an error. The programmer can choose to resolve the violation, as he would be forced to do in MultiJava, thereby obtaining a modular guarantee of type safety. Alternatively, the programmer can choose to retain the desired expressiveness and take responsibility for ensuring that the potential error does not arise. A custom class loader [65] for RMJ incrementally checks that potentially erroneous generic functions remain exhaustive and unambiguous as classes are loaded, thereby ensuring that all errors are still detected no later than class load time.

```
// file HashSet.java
package extension3;
import sets.*;
public class HashSet extends Set {
  public Set addElem(int i) { ... }
}
```

Figure 3.5: Another `Set` subclass

```
// file isEmpty.java
package client;
import extension1.isEmpty;
import extension3.HashSet;
public HashSet.isEmpty() { ... }
```

Figure 3.6: Glue methods in RMJ

RMJ provides several useful programming idioms that are disallowed in MultiJava. For example, consider the `HashSet` class in figure 3.5, which is the MultiJava analogue of the Dubious module in figure 2.4. The implementer of this extension is unaware of the `isEmpty` extension to `Set` provided in figure 3.2, and vice versa. The System M requirements ensure that `isEmpty` has a most-specific applicable method for `HashSet`, namely the global default `isEmpty` method handling any instance of `Set` or a subclass. As described in the previous chapter, if the operations defined on `Set` are not rich enough, there may be no appropriate default behavior except to throw an exception. In that case (and even if there is an appropriate default behavior), a client of the `isEmpty` and `HashSet` extensions to `Set` may desire special-purpose behavior for `isEmpty` on `HashSet`, as shown in figure 3.6. Such a method is called a *glue method*, because it serves to integrate two previously independent libraries. System M's requirement M1 disallows the creation of glue methods, forcing each external method to be declared in the same file that introduces its generic function. Indeed, it is difficult to modularly check glue methods for ambiguity. For example, if a second client also implements `isEmpty` for `HashSet`, this duplication will not be detected modularly. Rather than prohibiting the idiom, RMJ allows glue methods to be written, performs as much modular ambiguity checking as possible at compile time, and completes ambiguity checking

incrementally at load time.

RMJ relaxes the other System M requirements analogously. For example, the global default `isEmpty` method in figure 3.2 may instead be declared `abstract`. This relieves the implementer of the burden of providing reasonable default functionality and documents the fact that all concrete subclasses of `Set` should provide an appropriate `isEmpty` implementation. The RMJ class loader will ensure this is the case for each subclass of `Set` that is actually loaded in the program. As another example, RMJ allows interfaces to be dispatched upon and inserts load-time ambiguity checking to ensure safety. In future work, I plan to gain experience with the strengths and limitations of RMJ by providing it for use by others. Several projects have already been using MultiJava regularly, as described in chapter 5, and this user experience was one motivation for designing RMJ. The current prototype implementation of RMJ is freely available as part of the mjc compiler [79].

There are several practical kinds of extensibility that are missing from both MultiJava and RMJ. It would be useful to support retroactive abstraction, like some of the related systems described above. Retroactive abstraction is particularly natural in the presence of open classes, whereby the addition of new methods to a class can allow that class to implement an interface that it otherwise would not. This combination of features can potentially allow independently developed components to be integrated from the outside by adding the appropriate external operations that make them meet a common interface, resolving a well-known problem [51]. The abilities to add static external methods and static external fields to a class are simple but useful extensions. External constructors can also be supported, as long as their bodies begin with an invocation of another constructor of the same class. Adding instance fields to a class from the outside would be useful but is more challenging to implement efficiently. In the longer term, it will be interesting to investigate how to incorporate some of the additional extensibility of systems like AspectJ and Hyper/J, particularly the ability to extend individual methods with additional "before" and "after" behavior from the outside, while retaining support for modular program reasoning.

Chapter 4

EXTENSIBLE ML

As mentioned in chapter 1, the datatypes and functions in traditional functional languages naturally support a form of open classes and multiple dispatch, but they lack the traditional OO extensibility idioms like subclassing. This chapter describes an ML-like [76] language called Extensible ML (Eml) [72], which generalizes ML-style datatypes and functions to support the OO extensibility idioms, adapting Dubious's System M to preserve modular typechecking. As a proof of concept, Colin Bleckner has developed a prototype interpreter for Eml.

4.1 EML By Example

Figure 4.1 shows an Eml version of (relevant portions of) the Dubious code for integer sets in figures 2.2 and 2.3.[1] Much of Eml's syntax is essentially a translation of Dubious's syntax into the ML style. Dubious modules are replaced by ML-style structures. They can contain declarations of classes, functions, and function cases, the analogues of Dubious's objects, generic functions (which are just special kinds of Dubious objects), and methods. For now I assume that structures do not contain any of the ordinary ML declarations, but only the new Eml declarations. This assumption is lifted in section 4.4, which describes the interaction of Eml's features with an ML-style module system.

4.1.1 Classes

Each class declares a record type of its instance variables, using the `of` clause. Superclass instance variables are inherited: the *representation type* of a class C is the representation type (recursively) of its direct superclass (if any) concatenated with the type in the `of`

[1] As in Dubious, all uses of an identifier should be qualified by a module name, but I elide this detail for readability.

```
structure SetMod = struct
  abstract class Set() of {}
  fun addElem:(#Set * int) → Set
  fun union:(#Set * Set) → Set
  extend fun union(s1, s2) = ...
end

structure ListSetMod = struct
  class ListSet(elems:int list) extends Set()
    of {es:int list = elems}
  extend fun addElem(ls as ListSet {es=es}, i) =
    if (member i es) then ls else ListSet(i::es)
  extend fun union(ListSet _, ListSet _) = ...
end

structure SListSetMod = struct
  class SListSet(elems:int list)
    extends ListSet(elems) of {}
  extend fun addElem(SListSet {es=es}, i) = ...
  extend fun union(SListSet _, SListSet _) = ...
  fun getMin:#SListSet → int
  extend fun getMin(s) = ...
end

structure IsEmptyMod = struct
  fun isEmpty:#Set → bool
  extend fun isEmpty s = ...
  extend fun isEmpty(ListSet {es=nil}) = true
  extend fun isEmpty(ListSet _) = false
end
```

Figure 4.1: A hierarchy of integer sets in EML.

clause in C's declaration. For example, the representation type of `SListSet` in figure 4.1 is {`es:int list`}. The `of`-clause syntax suggests a correspondence between EML's classes and ordinary ML-style datatype variants, which is described below.

Each class declaration implicitly declares a constructor, similar to constructor declarations in OCaml [90] and XMOC [39], a core language for Moby [38]. For example, the `ListSet` constructor expects an argument `elems` of type `int list`, initializes inherited instance variables (although there are none in this case) via the call `Set()` to the superclass constructor, and initializes the new `es` instance variable to `elems`. In general, the arguments to the superclass constructor call and the instance-variable initializers may be arbitrary expressions. It would be straightforward to allow a class to have multiple constructors by introducing a separate `constructor` declaration, similar to "makers" in Moby.

Classes are as expressive as ordinary ML-style datatypes. An ML datatype of the form

$$\texttt{datatype DT = } \mathtt{C}_1 \texttt{ of } \{\mathtt{L}_{11}\texttt{:}\mathtt{T}_{11}\texttt{,...,}\mathtt{L}_{1m}\texttt{:}\mathtt{T}_{1m}\} \texttt{ | } \cdots \texttt{ | } \mathtt{C}_r \texttt{ of } \{\mathtt{L}_{r1}\texttt{:}\mathtt{T}_{r1}\texttt{,...,}\mathtt{L}_{rn}\texttt{:}\mathtt{T}_{rn}\}$$

is encoded in EML by the following class declarations:

$$\texttt{abstract class DT of } \{\}$$
$$\texttt{class } \mathtt{C}_1\texttt{(}\mathtt{I}_{11}\texttt{:}\mathtt{T}_{11}\texttt{,...,}\mathtt{I}_{1m}\texttt{:}\mathtt{T}_{1m}\texttt{) extends DT() of } \{\mathtt{L}_{11}\texttt{:}\mathtt{T}_{11}\texttt{=}\mathtt{I}_{11}\texttt{,...,}\mathtt{L}_{1m}\texttt{:}\mathtt{T}_{1m}\texttt{=}\mathtt{I}_{1m}\}$$
$$\ldots$$
$$\texttt{class } \mathtt{C}_r\texttt{(}\mathtt{I}_{r1}\texttt{:}\mathtt{T}_{r1}\texttt{,...,}\mathtt{I}_{rn}\texttt{:}\mathtt{T}_{rn}\texttt{) extends DT() of } \{\mathtt{L}_{r1}\texttt{:}\mathtt{T}_{r1}\texttt{=}\mathtt{I}_{r1}\texttt{,...,}\mathtt{L}_{rn}\texttt{:}\mathtt{T}_{rn}\texttt{=}\mathtt{I}_{rn}\}$$

For example, the declarations of the `Set` and `ListSet` classes in figure 4.1 could have been written equivalently as follows:

```
datatype Set = ListSet of {es:int list}
```

Classes additionally generalize ML-style datatypes to be *extensible*, whereby new variants can be written in modules other than the one declaring the datatype, and *hierarchical*, whereby variants can have their own "subvariants." The `SListSet` subclass of `ListSet` in figure 4.1 illustrates both of these generalizations. In addition to being extensible and hierarchical, classes are also full-fledged types while ML variants are not. For example, `SListSet` is the argument type of the `getMin` function in figure 4.1.

A concrete class is instantiated by invoking its constructor. For example, the result of evaluating `ListSet([5,3])` is an instance of `ListSet` representing the set {5,3}. An instance's record of instance variables is called its *representation*. Like values of ML datatypes, class instances have no special object identity or mutable state; ML `ref`s can be used in a class's representation for this purpose.

Classes support only single inheritance. Single inheritance of classes is compatible with the ML style, in which each data variant conceptually singly inherits from the corresponding datatype, as shown in the earlier encoding of datatypes into classes. However, like Dubious and MultiJava, EML supports *interfaces*. For example, the abstract `Set` class in figure 4.1 could instead be declared to be an interface as follows:

```
interface Set
```

Interfaces do not have instance variables and consequently additionally do not declare a constructor. As in Java, an interface can inherit from multiple interfaces via an `extends` clause, and a class can inherit from multiple interfaces via an `implements` clause. Given the above declaration of the `Set` interface, the declaration of `ListSet` would be modified as follows:

```
class ListSet(elems:int list) implements Set of {es:int list = elems}
```

4.1.2 Functions and Function Cases

The `fun` declaration introduces a new generic function and specifies its type. The `#` in a function's type plays the same role as in Dubious's System M, identifying the function's owner position. Any class type within the function's type may be the function's owner, no matter how deeply nested it is. For example, the owner of a function accepting two pairs of class instances would be specified to be one of the component class types of one of the pairs. The `extend fun` declaration adds a new case to a generic function; this declaration plays the same role as Dubious's `has method` declaration.

The `fun` and `extend fun` declarations together generalize ML functions to be extensible, allowing new cases to be declared in later modules. An ordinary (but explicitly-typed) ML

function consisting of n function cases can be desugared in EML as a `fun` declaration followed by n `extend fun` declarations. EML functions can be passed to and returned from other functions, like ML functions. However, a function's extensibility is second-class: new cases may only be added to statically known functions.

The `extend fun` declaration specifies the name of the generic function, a *pattern* guard, and the new case's body. The pattern specifies the dynamic dispatching behavior of the new case. The `union` function case in `ListSetMod` performs Dubious-style multiple dispatching: the case is only applicable to two arguments that are instances of `ListSet` or some subclass. However, EML patterns also support the expressiveness of patterns in ML. For example, EML patterns can dispatch recursively on the substructure of class instances. Each *class pattern* includes the name of the class being dispatched upon, as well as a pattern for the class's instance variables. Analogous to the restrictions against interface specializers in Dubious and MultiJava, the class pattern may refer only to classes, not interfaces. The *wildcard pattern* (`_`) is used in `ListSet`'s `union` case to signify that no properties are required of the two arguments' instance variables. On the other hand, the `addElem` case uses a *representation pattern* to provide a name to the given `ListSet` instance's underlying list of elements, which can then be referred to in the case's body. EML patterns also subsume MultiJava's value dispatching, as described in section 3.1.3. For example, the second `isEmpty` case in `IsEmptyMod` dispatches on the constant `nil`, which denotes the empty list.

Consider a call to a generic function f with argument value v. Analogous to Dubious, the most-specific applicable function case in f for v is invoked. EML's dispatching semantics naturally generalizes the dispatching semantics of Dubious to handle patterns. A function case is applicable to v if v matches the case's pattern. A function case c_1 is at least as specific as case c_2 if the set of values matching c_1's pattern is a subset of the set of values matching c_2's pattern. For example, consider the invocation `isEmpty(ListSet([5,3]))`. The first and third `isEmpty` cases in `IsEmptyMod` are applicable, but the second case is not applicable because it requires that the argument's `es` instance variable be the empty list. Of the other two cases, the later one is most specific, because it accepts only instances of `ListSet` and subclasses, while the first case accepts any argument of type `Set`. This "best-match" semantics contrasts with the traditional "first-match" semantics of function

```
abstract class 'a Set() of {}
class 'a ListSet(elems:'a list) extends 'a Set()
  of {es:'a list = elems}
fun 'a addElem:(#'a Set * 'a) → 'a Set
extend fun 'a addElem(ls as ListSet {es=es}, i) =
  if (member i es) then ls else 'a ListSet(i::es)
```

Figure 4.2: Parametric polymorphism in EML.

cases in ML. The first-match semantics does not generalize naturally to handle extensible datatypes and functions, where typically the more-specific function cases are written *after* the less-specific ones, as new data variants are defined.

4.1.3 Parametric Polymorphism

EML supports a polymorphic type system. Class, function, and function case declarations optionally bind *type variables*. References to a polymorphic class or function specify a particular *type instantiation*. As an example, figure 4.2 shows a polymorphic version of some declarations from figure 4.1. The `Set` and `ListSet` classes are now parameterized by the element type. The declaration of the `addElem` function binds a type variable for use in its declared type. As a convenience, a function case locally rebinds its function's type variables, for use in the case's body. In the figure, the `addElem` case uses the same name (`'a`) for the type variable as does the `addElem` function declaration. Classes referred to in patterns do not contain type instantiations; the unique type-correct instantiation is determined by the declared argument type of the associated function.

EML's polymorphic type system is deliberately simple. Subclasses must have the same type variables as their superclasses. This requirement is consistent with polymorphism in ML, where data variants have the same type variables as their associated datatype. Also, type parameters are *invariant*; for example, $\texttt{T}_1$ `ListSet` is a subtype of $\texttt{T}_2$ `Set` if and only if $\texttt{T}_1$=$\texttt{T}_2$. Finally, there is no support for *bounded polymorphism* [19]. I have chosen to make the polymorphic type system simple because polymorphism is orthogonal to the problems of modular ITC that I address in this work. Those problems arise from the fact that some related classes, functions, and function cases are not modularly available to one another;

the problems are neither reduced nor exacerbated by polymorphic types. Therefore, EML's polymorphic type system could be generalized in standard ways without affecting modular typechecking. For example, EML could adopt ML$_{\leq}$'s subtype-constrained polymorphic type system for extensible datatypes and functions [10].

EML is also explicitly typed. This contrasts with ML's polymorphic type system, which supports type inference. Unfortunately, supporting both subtyping and polymorphic type inference is known to be difficult [42, 50, 82]. It would be useful to explore forms of *local* type inference [89] to ease the type-annotation burden somewhat. Recent work of Bonniot [9] has presented a simplified account of ML$_{\leq}$'s type system and shown how to incorporate a form of local type inference, so this could be a promising foundation for augmenting EML.

4.2 Modular Implementation-side Typechecking

ML supports modular typechecking of structures. EML augments ML's modular typechecks to support ITC for extensible datatypes and functions. The requirements from Dubious's System M in section 2.3 are imposed in order to make this checking safe.

The ambiguity requirement M1 ensures that each function case either has a local owner or a local generic function. Analogous with Dubious, a case's owner is the class that is dispatched upon in the owner position of the case's pattern. For example, the owner of the `union` case in `ListSetMod` of figure 4.1 is `ListSet`. Because `ListSet` is local to `ListSetMod`, requirement M1 is satisfied. A case whose pattern does not dispatch on a class in the owner position does not have a well-defined owner. For example, the `union` case in `SetMod` lacks an owner. However, the case still satisfies requirement M1 because the `union` function is local to `SetMod`. Requirement M1 does not restrict the ordinary ML style, because all ML function cases have a local generic function. Requirement M1 safely generalizes that implicit requirement to also allow OO-style subclasses with overriding methods. As in MultiJava, requirement M2 need not be imposed because EML supports only single inheritance of classes.

Requirement M1 guarantees that if two cases are declared in structures that do not statically depend upon one another, then the two cases are disjoint and hence unambiguous.

To complete modular ambiguity checking, each case c declared in S is checked for ambiguity with each available case c' other than itself. Because EML classes lack multiple inheritance, explicit enumeration of legal argument tuples, as is done in Dubious (section 2.2), is not necessary. Let the patterns of c and c' be *pat* and *pat*$'$, respectively. Cases c and c' are checked for ambiguity as follows:

- If *pat* and *pat*$'$ are *congruent*, meaning that they are identical when all identifier bindings are removed, then the cases are ambiguous.

- Otherwise if the patterns are *disjoint*, meaning that no value can match both *pat* and *pat*$'$, then the cases are unambiguous.

- Otherwise there must be some pattern *pat*$''$ that represents the *intersection* of *pat* and *pat*$'$: all values matching both *pat* and *pat*$'$ also match *pat*$''$. The cases are unambiguous only if there exists an available case c'' whose pattern is congruent to *pat*$''$. Case c'' safely resolves the ambiguity between c and c'.

A degenerate form of the third scenario above occurs when one of *pat* and *pat*$'$ is strictly more specific than the other, so the resolving case is one of the original two. For example, the last two `isEmpty` cases in figure 4.1 are neither congruent nor disjoint. Their intersection pattern is congruent to the first case's pattern, and that case itself is the resolving case.

Modular exhaustiveness checking simply amounts to enforcement of requirements M3 and M4, which ensure the existence of appropriate local and global default cases. As an example, because `isEmpty` in figure 4.1 is an external generic function, M4 ensures that it has a global default case. The first case in `IsEmptyMod` serves this purpose, handling any unavailable concrete subclasses of `Set`. Unlike the ambiguity requirements, M3 and M4 do pose an additional burden on the ordinary ML style, because ML does not always require default cases. For example, consider the ML datatype that is the analogue of the `Set` and `ListSet` classes, as illustrated in section 4.1.1 above. In the context of that datatype, the analogue of the `isEmpty` function in ML would be deemed exhaustive given only the last two function cases in `IsEmptyMod`. This would be safe because datatypes in ML are

nonextensible, so `ListSet` would be guaranteed to be the sole variant of the `Set` datatype. An analogous situation exists for local defaults. Therefore, the cost of making ML datatypes extensible is the need for default cases.

As discussed in previous chapters, it is not always possible to write a reasonable default case. Relaxed MultiJava, described in section 3.5, handles this problem by allowing defaults to be omitted and inserting load-time checks to ensure exhaustiveness. ML illustrates an alternative solution: default cases can be safely omitted without load-time checks as long as datatypes are not extensible. In section 4.4, I describe a mechanism for *sealing* classes in EML, which can be used to obtain the semantics of ordinary nonextensible datatypes in EML. An external generic function on a sealed class hierarchy can safely omit the global default case, and an analogous relaxation of the local-default requirement is also safe.

One practical issue that arises with checking for defaults in EML is the need to avoid being overly conservative. A simple way to enforce requirement M4 is to check that every external generic function has a case whose pattern is the wildcard pattern (`_`). While using this algorithm in EML would be safe, it would also cause many exhaustive generic functions to nonetheless be rejected. For example, the `isEmpty` function in figure 4.1 has a global default case, but it fails the above check. To remedy this imprecision, EML instead requires that each external generic function have a case whose pattern is a *global default pattern*: any argument of the function's type matches the pattern. In the `isEmpty` example, there are several possible global default patterns, including `_`, `s`, `(Set _)`, and `(Set {})`.

EML performs the revised check by generating a precise global default pattern and then requiring that this pattern be at least as specific as some case's pattern. The algorithm retains precision via the notion of a pattern's *depth*, which is effectively the depth of the pattern's abstract syntax tree representation. A valid global default case will not be overlooked by the algorithm as long as the generated pattern has a depth no smaller than that of the case's pattern. Therefore, EML generates the pattern to have a depth equal to the maximum depth of any available case on the generic function.[2] In the `isEmpty` example,

[2]Because class patterns allow pattern matching on a class's representation, which may recursively involve class patterns, it is possible for patterns to have arbitrary depth. Therefore, there is in general no *a priori* maximal depth for the patterns of a given function.

$$
\begin{array}{rcll}
\tau & ::= & \alpha \mid Ct \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 * \cdots * \tau_k & \\
Mt & ::= & \#Ct \mid \tau_1 * \cdots * \tau_{i-1} * Mt * \tau_{i+1} * \cdots * \tau_k & \\
E & ::= & I \mid Fv \mid E_1\ E_2 \mid Ct(\overline{E}) \mid (\overline{E}) \mid Ct\{\overline{V} = \overline{E}\} & \\
Pat & ::= & \_ \mid I\ \texttt{as}\ Pat \mid C\{\overline{V} = \overline{Pat}\} \mid (\overline{Pat}) & \\
Ct & ::= & \overline{\tau}\ C & C ::= Sn.Cn \\
Fv & ::= & \overline{\tau}\ F & F ::= Sn.Fn \\
V & ::= & Sn.Vn &
\end{array}
$$

Figure 4.3: MINI-EML types, expressions, and patterns.

EML generates the global default pattern `(Set {})`, matching the depth of the second case's pattern. The check for a global default case succeeds because the generated pattern is at least as specific as the first case's pattern. A similar algorithm is used for checking for local default cases. These algorithms are implemented in the EML prototype interpreter and are formalized in section 4.3.

4.3 Mini-EML

This section describes MINI-EML, a core language that formalizes EML and its modular type system. It is similar in scope to Dubious but additionally supports ML-style patterns and parametric polymorphism.

4.3.1 Syntax

Figure 4.3 defines the syntax of types, expressions, and patterns in MINI-EML. The syntax is essentially that of EML as informally presented so far. As in Dubious, standard constructs that don't impact modular ITC are omitted, including base types, conditionals, anonymous and lexically nested functions, local variables, references, and exceptions. Metavariable α ranges over type variable names, I over identifier names, Sn over structure names, Cn over class names, Vn over instance variable names, and Fn over function names. $\overline{X}$ denotes a comma-separated sequence of elements of the domain X (and is independent of any variable named X); the empty sequence is denoted $\bullet$. By a slight abuse of notation, sometimes $\overline{X}$ is part of a larger abbreviation, and that larger abbreviation supersedes the default

interpretation of $\overline{X}$. For example, the notation $\overline{V} = \overline{E}$ abbreviates $V_1 = E_1, \ldots, V_k = E_k$ where $\overline{V}$ is $V_1, \ldots, V_k$ and $\overline{E}$ is $E_1, \ldots, E_k$ for some $k \geq 0$, and similarly for $\overline{V} = \overline{Pat}$. All such abbreviations are defined as needed below.

MINI-EML types include type variables, class types, function types, and tuple types. The domain *Mt* represents *marked types*, which contain a `#` mark on a single component class type. Expressions include identifiers, function values, function application, constructor calls, tuples, and *instance expressions*. The instance expression $Ct\{\overline{V} = \overline{E}\}$ is used only to formalize the semantics of constructor calls and is not available at the source level.

The construct $\{\overline{V} = \overline{E}\}$ differs from an ordinary record in two ways. First, the labels are *scoped*: the name of the structure in which an instance variable was introduced becomes part of the instance variable's name. Scoping allows a class to introduce an instance variable with the same name as one in a superclass declared in another structure. While this ability provides only a minor convenience in MINI-EML, scoping provides a foundation for allowing a class to safely hide instance variables, as described in section 4.4. Instance variables in EML use this mechanism implicitly; regular static scoping rules determine which instance variable is referred to. Second, for simplicity the components of $\{\overline{V} = \overline{E}\}$ are ordered, unlike traditional records.

Patterns include the wildcard pattern, identifier binding, class patterns, and tuple patterns. For simplicity, the representation pattern $\{\overline{V} = \overline{Pat}\}$ within a class pattern must mention all of the associated class's instance variables. A pattern of the form I, used in figure 4.1, is syntactic sugar for `(`I `as _)`.

The notation and semantic style of MINI-EML were influenced by Featherweight Java [53], a core language for Java. As in that language, classes are formally represented by their names. A class is uniquely represented as $Sn.Cn$, where Cn is the name of the class and Sn is the name of the structure that declares Cn. Extensible functions are represented similarly.

The subset of expressions that are MINI-EML values is described by the following grammar, which includes class instances, function values, and tuple values:

$$v ::= Ct\{\overline{V} = \overline{v}\} \mid Fv \mid (\overline{v})$$

The syntax of structures and declarations is shown in figure 4.4a. A structure consists

$$
\begin{array}{rcl}
S & ::= & \texttt{structure}\ Sn = \texttt{struct}\ \overline{D}\ \texttt{end} \\
D & ::= & \texttt{<abstract>}\ \texttt{class}\ \overline{\alpha}\ Cn(\overline{I} : \overline{\tau}) \\
 & & \quad \texttt{<<extends}\ Ct(\overline{E})\ \texttt{>>} \\
 & & \quad \texttt{of}\ \{\overline{Vn} : \overline{\tau_0} = \overline{E_0}\} \\
 & | & \texttt{fun}\ \overline{\alpha}\ Fn : Mt \to \tau \\
 & | & \texttt{extend fun}_{Mn}\ \overline{\alpha}\ F\ Pat = E
\end{array}
$$

(a)

$$
\begin{array}{rcl}
Sig & ::= & \texttt{signature}\ Sn = \texttt{sig}\ \overline{Sp}\ \texttt{end} \\
Sp & ::= & \texttt{<abstract>}\ \texttt{class}\ \overline{\alpha}\ Cn(\overline{I} : \overline{\tau}) \\
 & & \quad \texttt{<<extends}\ Ct\ \texttt{>>} \\
 & & \quad \texttt{of}\ \{\overline{Vn} : \overline{\tau_0}\} \\
 & | & \texttt{fun}\ \overline{\alpha}\ Fn : Mt \to \tau \\
 & | & \texttt{extend fun}_{Mn}\ \overline{\alpha}\ F\ Pat
\end{array}
$$

(b)

Figure 4.4: (a) MINI-EML structures and declarations. (b) MINI-EML signatures and specifications.

of a sequence of class, function, and function case declarations. The syntax of the three declarations is faithful to that of EML, except that cases now contain a *case name*, ranged over by metavariable *Mn*. This name is used in the formal semantics to uniquely identify each function case declaration. Angle brackets (<>) and double angle brackets (<<>>) denote independent optional pieces of syntax. The notation $\overline{Vn} : \overline{\tau} = \overline{E}$ abbreviates $Vn_1 : \tau_1 = E_1, \ldots, Vn_k : \tau_k = E_k$, and similarly for $\overline{I} : \overline{\tau}$.

The declared inheritance graph is assumed to be acyclic. The class, function, and case names introduced in a given structure are assumed to be distinct. The type variables parameterizing a given declaration are assumed to be distinct. All the instance variable names introduced in a given structure are assumed to be distinct. The identifiers introduced in a given function case's pattern are assumed to be distinct.

MINI-EML also includes an explicit notion of signature, as defined in figure 4.4b. In ML, the name of a signature is completely independent of the names of structures that implement the signature. However, there is a one-to-one mapping between structures and signatures in a MINI-EML program (see below), so for simplicity the name of a MINI-EML structure is also used as the name of its associated signature. A signature contains a sequence of class, function, and function case *specifications*. The syntax for each specification is identical to its corresponding declaration, but with all expressions removed. The notation $\overline{Vn} : \overline{\tau}$ abbreviates $Vn_1 : \tau_1, \ldots, Vn_k : \tau_k$.

Inspired by Featherweight Java, a MINI-EML program is a pair of a *structure table* and an expression to be evaluated. A structure table is a finite function from structure names to

```
signature SListSetMod = sig
  class SListSet(elems:int list) extends ListSet of {}
  extend fun addElem(SListSet {es=es}, i)
  extend fun union(SListSet _, SListSet _)
  fun getMin:#SListSet → int
  extend fun getMin(s)
end
```

Figure 4.5: The principal signature of `SListSetMod`.

the associated structure declarations. The formal semantics assumes a fixed structure table ST. The domain of ST is denoted $\mathrm{dom}(ST)$. The structure table is assumed to satisfy some sanity conditions: (1) $ST(Sn) = (\texttt{structure}\ Sn = \texttt{struct} \cdots \texttt{end})$ for every $Sn \in \mathrm{dom}(ST)$; (2) for every structure name Sn appearing anywhere in the program, $Sn \in \mathrm{dom}(ST)$.

The static semantics of Mini-Eml also relies on a fixed *signature table* $SigT$, which maps each structure name Sn in $\mathrm{dom}(ST)$ to the *principal signature* of $ST(Sn)$. The principal signature of a structure S is a signature of the same name as S whose body is identical to S's body, but with all expressions removed. For example, the principal signature of `SListSetMod` from figure 4.1 is shown in figure 4.5. Each structure in the range of ST is typechecked in the context of $SigT$, without access to ST, thereby ensuring that the first criterion for modular typechecking is met, as defined in section 2.2.2: a structure is typechecked only against the signatures, rather than the implementations, of other structures. The domain of $SigT$ is denoted $\mathrm{dom}(SigT)$.

4.3.2 Dynamic Semantics

Mini-Eml's dynamic semantics is defined as a mostly standard small-step operational semantics. The structure table is accessed whenever information about a declaration is required in order to execute an expression. The metavariable ρ ranges over *environments*, which are finite functions from identifiers to values. $|\overline{X}|$ denotes the length of the sequence $\overline{X}$. The notation $[I_1 \mapsto E_1, \ldots, I_k \mapsto E_k]X$ denotes the expression resulting from the simultaneous substitution of E_i for each occurrence of I_i in X, for all $1 \leq i \leq k$, and similarly for $[\alpha_1 \mapsto \tau_1, \ldots, \alpha_k \mapsto \tau_k]X$. $[\overline{I} \mapsto \overline{E}]X$ is used as a shorthand when $\overline{I}$ and $\overline{E}$ have the same

length, and similarly for $[\overline{\alpha} \mapsto \overline{\tau}]X$. In a given inference rule, fragments enclosed in <> must either be all present or all absent, and similarly for <<>>. Sequences are sometimes treated as if they were sets. For example, $D \in \overline{D}$ means that D is one of the declarations in $\overline{D}$. Finally, $D \in ST(Sn)$ is shorthand for the two facts $ST(Sn) = (\texttt{structure}\ Sn = \texttt{struct}\ \overline{D}\ \texttt{end})$ and $D \in \overline{D}$.

Figure 4.6 contains the rules for evaluating expressions. The notation $(\overline{I}, \overline{v})$ abbreviates $(I_1, v_1), \ldots, (I_k, v_k)$, and $Sn.\overline{Vn} = \overline{E}$ abbreviates $Sn.Vn_1 = E_1, \ldots, Sn.Vn_k = E_k$. For simplicity in the semantics, a constructor call is treated as syntactic sugar for the instance expression obtained by expanding the constructor's definition. Rule E-New specifies this semantics, and rule E-Rep evaluates instance expressions. It would be straightforward to instead use a call-by-value semantics for constructor calls, at the cost of some additional mechanism. Rule E-New uses the first two auxiliary rules at the bottom of the figure. Rule Concrete checks that the class to be instantiated was declared without the `abstract` keyword. Rule Rep initializes the fields of the new instance as directed by the class's implicit constructor, substituting the actual arguments to the constructor call for the formals. The rule also substitutes the new instance's type parameters for the class's type variables. Types have no dynamic effect in Mini-Eml, but maintaining types in the dynamic semantics eases the type system's proof of soundness (see appendix A).

The last rule in figure 4.6 formalizes function-case lookup, used in E-AppRed. The first premise of Lookup specifies the case to invoke. The second premise ensures that this case is applicable: the argument value matches the case's pattern. That premise also produces an environment mapping each identifier in the case's pattern to the appropriate "pieces" of the argument value. This environment is used by E-AppRed to evaluate the case's body. The remaining premise ensures that the chosen case is most-specific: the case is strictly more specific than any other applicable case. The condition $Sn.Mn \neq Sn'.Mn'$ uses the case names to ensure that the chosen case is not compared for specificity with itself.

The rules for pattern matching and specificity are shown in figure 4.7. The notation $\text{match}(\overline{v}, \overline{Pat}) = \overline{\rho}$ abbreviates $\text{match}(v_1, Pat_1) = \rho_1 \cdots \text{match}(v_k, Pat_k) = \rho_k$, and similarly for $\overline{Pat_1} \leq \overline{Pat_2}$. The matching rules are straightforward except for E-MatchClass. The judgment $C \leq C'$ is defined at the bottom of figure 4.7 as the reflexive, transitive closure

$\boxed{E \longrightarrow E'}$

$$\frac{Ct = (\overline{\tau}\ C) \qquad \text{concrete}(C) \qquad \text{rep}(Ct(\overline{E_0})) = \{\overline{V} = \overline{E_1}\}}{Ct(\overline{E_0}) \longrightarrow Ct\{\overline{V} = \overline{E_1}\}}\ \text{E-New}$$

$$\frac{E \longrightarrow E'}{Ct\{\overline{V_0} = \overline{v_0}, V = E, \overline{V_1} = \overline{E_1}\} \longrightarrow Ct\{\overline{V_0} = \overline{v_0}, V = E', \overline{V_1} = \overline{E_1}\}}\ \text{E-Rep}$$

$$\frac{E \longrightarrow E'}{(\overline{v_0}, E, \overline{E_1}) \longrightarrow (\overline{v_0}, E', \overline{E_1})}\ \text{E-Tup}$$

$$\frac{E_1 \longrightarrow E_1'}{E_1\ E_2 \longrightarrow E_1'\ E_2}\ \text{E-App1} \qquad \frac{E_2 \longrightarrow E_2'}{v_1\ E_2 \longrightarrow v_1\ E_2'}\ \text{E-App2}$$

$$\frac{\text{most-specific-case-for}(Fv, v) = (\{(\overline{I}, \overline{v})\}, E)}{Fv\ v \longrightarrow [\overline{I} \mapsto \overline{v}]E}\ \text{E-AppRed}$$

$\boxed{\text{concrete}(C)}$

$$\frac{(\texttt{class}\ \overline{\alpha}\ Cn\ \ldots) \in ST(Sn)}{\text{concrete}(Sn.Cn)}\ \textsc{Concrete}$$

$\boxed{\text{rep}(Ct(\overline{E_0})) = \{\overline{V} = \overline{E}\}}$

$$\frac{\begin{array}{c}(\texttt{<<abstract>> class}\ \overline{\alpha}\ Cn(\overline{I} : \overline{\tau_1})\ \texttt{<extends}\ Ct(\overline{E_0})\ \texttt{> of}\ \{\overline{Vn} : \overline{\tau_2} = \overline{E_2}\}) \in ST(Sn) \\ < \text{rep}(Ct(\overline{E_0})) = \{\overline{V} = \overline{E_1}\} >\end{array}}{\text{rep}((\overline{\tau}\ Sn.Cn)(\overline{E})) = [\overline{I} \mapsto \overline{E}][\overline{\alpha} \mapsto \overline{\tau}]\{ < \overline{V} = \overline{E_1}, > Sn.\overline{Vn} = \overline{E_2}\}}\ \textsc{Rep}$$

$\boxed{\text{most-specific-case-for } (Fv, v) = (\rho, E)}$

$$\frac{\begin{array}{c}(\texttt{extend fun}_{Mn}\ \overline{\alpha}\ F\ Pat = E) \in ST(Sn) \qquad \text{match}(v, Pat) = \rho \\ \forall Sn' \in \text{dom}(ST).\forall(\texttt{extend fun}_{Mn'}\ \overline{\alpha'}\ F\ Pat' \ldots) \in ST(Sn').\forall \rho'. \\ ((\text{match}(v, Pat') = \rho' \wedge Sn.Mn \neq Sn'.Mn') \Rightarrow Pat < Pat')\end{array}}{\text{most-specific-case-for } ((\overline{\tau}\ F), v) = (\rho, [\overline{\alpha} \mapsto \overline{\tau}]E)}\ \textsc{Lookup}$$

Figure 4.6: Evaluation rules for Mini-Eml expressions.

$\boxed{\text{match}(v, Pat) = \rho}$

$$\frac{}{\text{match}(v, \_) = \{\}}\ \textsc{E-MatchWild}$$

$$\frac{\text{match}(v, Pat) = \rho}{\text{match}(v, I\ \texttt{as}\ Pat) = \rho \cup \{(I, v)\}}\ \textsc{E-MatchBind}$$

$$\frac{C \leq C' \qquad \text{match}(\overline{v}, \overline{Pat}) = \overline{\rho}}{\text{match}(\overline{\tau}\ C\{\overline{V} = \overline{v}, \overline{V_1} = \overline{v_1}\}, C'\{\overline{V} = \overline{Pat}\}) = \bigcup \overline{\rho}}\ \textsc{E-MatchClass}$$

$$\frac{\text{match}(\overline{v}, \overline{Pat}) = \overline{\rho}}{\text{match}((\overline{v}), (\overline{Pat})) = \bigcup \overline{\rho}}\ \textsc{E-MatchTup}$$

$\boxed{Pat \leq Pat'}$

$$\frac{}{Pat \leq \_}\ \textsc{SpecWild}$$

$$\frac{Pat_1 \leq Pat_2}{I\ \texttt{as}\ Pat_1 \leq Pat_2}\ \textsc{SpecBind1} \qquad \frac{Pat_1 \leq Pat_2}{Pat_1 \leq I\ \texttt{as}\ Pat_2}\ \textsc{SpecBind2}$$

$$\frac{C \leq C' \qquad \overline{Pat_1} \leq \overline{Pat_2}}{C\{\overline{V} = \overline{Pat_1}, \overline{V_3} = \overline{Pat_3}\} \leq C'\{\overline{V} = \overline{Pat_2}\}}\ \textsc{SpecClass}$$

$$\frac{\overline{Pat_1} \leq \overline{Pat_2}}{(\overline{Pat_1}) \leq (\overline{Pat_2})}\ \textsc{SpecTup}$$

$\boxed{C \leq C'}$

$$\frac{}{C \leq C}\ \textsc{SubRef}$$

$$\frac{C_1 \leq C_2 \qquad C_2 \leq C_3}{C_1 \leq C_3}\ \textsc{SubTrans}$$

$$\frac{(\texttt{<abstract>}\ \texttt{class}\ \overline{\alpha}\ Cn(\overline{I_1 : \tau_1})\ \texttt{extends}\ \overline{\tau}\ C\ \ldots) \in ST(Sn)}{Sn.Cn \leq C}\ \textsc{SubExt}$$

Figure 4.7: EML pattern matching, pattern specificity, and subclassing.

of the declared class `extends` relation. Therefore, an instance of class C matches a class pattern of class C' if C subclasses C' and the instance's representation recursively matches the given representation pattern. The instance may have more instance variables than are mentioned in the given representation pattern, so that subclass instances can match superclass patterns.

The judgment $Pat \leq Pat'$ means that Pat is at least as specific as Pat'. Rule LOOKUP uses $Pat < Pat'$ as shorthand for $Pat \leq Pat' \wedge Pat' \not\leq Pat$. The pattern specificity semantics generalizes OO-style best-match semantics to support ML-style patterns. Any pattern is at least as specific as the wildcard, and identifier binding has no effect on specificity. Class pattern specificity (SPECCLASS) follows the ordering induced by subclassing. Analogous with E-MATCHCLASS, the more-specific pattern may contain extra instance variables. The natural rule SPECTUP for specificity of tuple patterns is analogous to the symmetric multi-method dispatching semantics of Dubious. When a tuple is used to send multiple arguments to a function, tuple patterns allow all arguments to be dynamically dispatched upon, and no argument position is more important than the rest.

4.3.3 Static Semantics

Figure 4.8 contains the rules for typechecking structures and declarations. Γ is a *type environment*, mapping identifiers to types. The notation $\hat{Mt}$ denotes the type τ identical to Mt, but with the `#` mark removed. The notation $\overline{Sn} \vdash \overline{D}$ OK in Sn abbreviates $\overline{Sn} \vdash D_1$ OK in $Sn \cdots \overline{Sn} \vdash D_k$ OK in Sn; the notation $\overline{\alpha} \vdash \overline{\tau}$ OK abbreviates $\overline{\alpha} \vdash \tau_1$ OK $\cdots$ $\overline{\alpha} \vdash \tau_k$ OK; the notation $(\overline{I}, \overline{\tau})$ abbreviates $(I_1, \tau_1), \ldots, (I_k, \tau_k)$; the notation $\Gamma; \overline{\alpha} \vdash \overline{E} : \overline{\tau}$ abbreviates $\Gamma; \overline{\alpha} \vdash E_1 : \tau_1 \cdots \Gamma; \overline{\alpha} \vdash E_k : \tau_k$; the notation $\overline{\tau_1} \leq \overline{\tau_0}$ abbreviates $\tau_{11} \leq \tau_{01} \cdots \tau_{1k} \leq \tau_{0k}$.

Structures are typechecked (STRUCTOK) by checking each declaration in turn. It is assumed that S OK holds for each structure S in the range of ST. $\overline{Sn}$ in the premise of STRUCTOK denotes those signatures in $SigT$ that may be accessed during ITC on the current structure. As defined in section 2.2.2, ITC is only modular if the current structure statically depends upon each signature in $\overline{Sn}$. Later judgments ensure the well-formedness

$\boxed{S \text{ OK}}$

$$\frac{\overline{Sn} \subseteq \mathrm{dom}(SigT) \qquad \overline{Sn} \vdash \overline{D} \text{ OK in } Sn}{\texttt{structure } Sn = \texttt{struct } \overline{D} \texttt{ end OK}} \textsc{StructOK}$$

$\boxed{\overline{Sn} \vdash D \text{ OK in } Sn}$

$$\frac{\begin{array}{c} < Ct = \overline{\alpha}\ C > \qquad < \Gamma; \overline{\alpha} \vdash Ct(\overline{E}) \text{ OK} > \\ \overline{\alpha} \vdash \overline{\tau} \text{ OK} \qquad \overline{\alpha} \vdash \overline{\tau_0} \text{ OK} \qquad \Gamma = \{(\overline{I}, \overline{\tau})\} \qquad \Gamma; \overline{\alpha} \vdash \overline{E_0} : \overline{\tau_1} \qquad \overline{\tau_1} \leq \overline{\tau_0} \\ \overline{Sn} \vdash Sn.Cn \text{ ITCTransUses} \qquad \text{concrete}(Sn.Cn) \Rightarrow \overline{Sn} \vdash \text{funs-have-ldefault-for } Sn.Cn \end{array}}{\overline{Sn} \vdash \texttt{<<abstract>> class } \overline{\alpha}\ Cn(\overline{I} : \overline{\tau}) \texttt{ <extends } Ct(\overline{E}) \texttt{ > of } \{\overline{Vn} : \overline{\tau_0} = \overline{E_0}\} \text{ OK in } Sn} \textsc{ClassOK}$$

$$\frac{\begin{array}{c} \overline{\alpha} \vdash \hat{Mt} \text{ OK} \\ \overline{\alpha} \vdash \tau \text{ OK} \qquad \text{owner}(Sn.Fn) = Sn'.Cn \qquad Sn \neq Sn' \Rightarrow \overline{Sn} \vdash Sn.Fn \text{ has-gdefault} \end{array}}{\overline{Sn} \vdash \texttt{fun } \overline{\alpha}\ Fn : Mt \to \tau \text{ OK in } Sn} \textsc{FunOK}$$

$$\frac{\begin{array}{c} (\texttt{fun } \overline{\alpha'}\ Fn : Mt \to \tau) \in SigT(Sn') \\ \text{matchType}([\overline{\alpha'} \mapsto \overline{\alpha}]\hat{Mt}, Pat) = (\Gamma, \tau_0) \qquad \Gamma; \overline{\alpha} \vdash E : \tau' \qquad \tau' \leq [\overline{\alpha'} \mapsto \overline{\alpha}]\tau \\ \overline{Sn} \vdash Sn'.Fn \text{ ITCUses} \qquad Sn; \overline{Sn} \vdash \texttt{extend fun}_{Mn}\ \overline{\alpha}\ Sn'.Fn\ Pat \text{ unambiguous} \end{array}}{\overline{Sn} \vdash \texttt{extend fun}_{Mn}\ \overline{\alpha}\ Sn'.Fn\ Pat = E \text{ OK in } Sn} \textsc{CaseOK}$$

Figure 4.8: Static semantics of Mini-Eml structures and declarations.

of the chosen $\overline{Sn}$, as described below.[3] The formalism does not explicitly enforce modularity of the rest of static typechecking, as those checks are standard and are naturally modular.

The rules for typechecking the three declaration forms are largely straightforward. Rule ClassOK checks that a class's superclass constructor call is well-typed, that all types mentioned in the class declaration are well-formed, and that the instance-variable initializer expressions have the appropriate types. The first premise in the rule ensures that the new class has the same type variables as its superclass, as mentioned in section 4.1.3. Rule FunOK checks that a function's declared type is well-formed. Rule CaseOK ensures that the case's pattern and body are compatible with the associated function's declared type. The "ITC-TransUses" and "ITCUses" judgments in ClassOK and CaseOK ensure well-formedness of the signatures $\overline{Sn}$ to be accessed during ITC of the enclosing structure; these judgments

[3]This "guess and check" style is used for simplicity. An alternative would be to take an initial pass over each structure's declarations in the static semantics, in order to compute the appropriate $\overline{Sn}$ before typechecking the structure.

$\boxed{\overline{\alpha} \vdash \tau \text{ OK}}$

$$\frac{\alpha \in \overline{\alpha}}{\overline{\alpha} \vdash \alpha \text{ OK}} \text{ TVarOK}$$

$$\frac{\overline{\alpha} \vdash \overline{\tau} \text{ OK} \qquad (\texttt{<abstract> class } \overline{\alpha_0}\ Cn \ldots) \in SigT(Sn) \qquad |\overline{\alpha_0}| = |\overline{\tau}|}{\overline{\alpha} \vdash \overline{\tau}\ Sn.Cn \text{ OK}} \text{ ClassTypeOK}$$

$$\frac{\overline{\alpha} \vdash \tau_1 \text{ OK} \qquad \overline{\alpha} \vdash \tau_2 \text{ OK}}{\overline{\alpha} \vdash \tau_1 \to \tau_2 \text{ OK}} \text{ FunTypeOK}$$

$$\frac{\overline{\alpha} \vdash \tau_1 \text{ OK} \ \cdots\ \overline{\alpha} \vdash \tau_k \text{ OK}}{\overline{\alpha} \vdash \tau_1 * \cdots * \tau_k \text{ OK}} \text{ TupTypeOK}$$

$\boxed{\tau \leq \tau'}$

$$\frac{}{\tau \leq \tau} \text{ SubTRef} \qquad \frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3} \text{ SubTTrans}$$

$$\frac{(\texttt{<abstract> class } \overline{\alpha}\ Cn(\overline{I_1} : \overline{\tau_1})\ \texttt{extends}\ Ct \ldots) \in SigT(Sn)}{\overline{\tau}\ Sn.Cn \leq [\overline{\alpha} \mapsto \overline{\tau}]Ct} \text{ SubTExt}$$

$$\frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'} \text{ SubTFun}$$

$$\frac{\tau_1 \leq \tau_1' \ \cdots\ \tau_k \leq \tau_k'}{\tau_1 * \cdots * \tau_k \leq \tau_1' * \cdots * \tau_k'} \text{ SubTTup}$$

Figure 4.9: Static semantics of Mini-Eml types.

are described below. Finally, each rule enforces one of the three requirements for modular ITC: ClassOK enforces the local-default requirement M3 ("funs-have-ldefault-for") if the class is concrete; FunOK enforces the global-default requirement M4 ("has-gdefault") if the function is external; CaseOK performs ambiguity checking ("unambiguous") for the given case, which includes enforcement of requirement M1 as described in section 4.2. The judgment for each requirement has $\overline{Sn}$ in the context, to ensure that only those signatures are accessed during enforcement of the requirement.

Figure 4.9 contains the static semantics of types. The judgment $\overline{\alpha} \vdash \tau$ OK ensures that τ refers only to type variables in $\overline{\alpha}$ and that each class in τ has the correct number of type

parameters. The subtyping relation $\tau \leq \tau'$ is completely standard [17].

Figure 4.10 contains the rules for typechecking expressions. The notation $Sn.\overline{Vn} : \overline{\tau}$ abbreviates $Sn.Vn_1 : \tau_1, \ldots, Sn.Vn_k : \tau_k$. The judgment $\Gamma; \overline{\alpha} \vdash E : \tau$ ensures that an expression is well-typed in the context of the type environment and sequence of type variables currently in scope. Most of the rules are standard. Rule T-Fun looks up a generic function's declared type in the signature table and substitutes the given use's type parameters for the function's type variables. Rule T-New uses T-Constr to check that a constructor invocation includes a well-formed class type and that the actual arguments have appropriate types, as dictated by the class's specification. Similarly, rule T-Rep ensures that an instance expression includes a well-formed class type and that the instance-variable expressions have appropriate types, as dictated by the class's specification. Rule T-Rep uses RepType, which computes a class's representation type.

Figure 4.11 contains the rules for typechecking patterns. The notation matchType$(\overline{\tau_0}, \overline{Pat}) = (\overline{\Gamma}, \overline{\tau_1})$ abbreviates matchType$(\tau_1, Pat_1) = (\Gamma_1, \tau_1') \cdots$ matchType$(\tau_k, Pat_k) = (\Gamma_k, \tau_k')$. The judgment matchType$(\tau, Pat) = (\Gamma, \tau')$ checks that Pat can be matched by values of type τ. The judgment produces a type environment mapping any identifiers in Pat to their types. This type environment is used in CaseOK (figure 4.8) to typecheck the associated case's body. The type τ' represents the particular subtype of τ to which Pat conforms; it is used to give precise types to any identifiers bound to Pat, as shown in rule T-MatchBind.

Figure 4.12 contains the well-formedness rules for the signatures $\overline{Sn}$ (which are "guessed" in StructOK) to be accessed during ITC of a structure Sn. Rule ClassITCTransUses is used by ClassOK in figure 4.8 to ensure that $\overline{Sn}$ contains all the signatures that specify a (reflexive, transitive) superclass of a class declared in Sn. Rule FunITCUses is used by CaseOK to ensure that $\overline{Sn}$ contains the signature specifying the associated function for a case declared in Sn. In either case, if $\overline{Sn}$ is required to include some signature Sn', then Sn does indeed statically depend upon Sn' according to the definition of static dependency given in section 2.2.2. The rules do not ensure that all statically depended upon signatures are in $\overline{Sn}$, but only those required for safe modular ITC. The rules also do not forbid $\overline{Sn}$ from including signatures that are not statically depended upon. However, the type soundness

$\boxed{\Gamma; \overline{\alpha} \vdash E : \tau}$

$$\frac{(I, \tau) \in \Gamma}{\Gamma; \overline{\alpha} \vdash I : \tau}\ \text{T-Id}$$

$$\frac{(\texttt{fun}\ \overline{\alpha_0}\ Fn : Mt \to \tau) \in SigT(Sn) \qquad \overline{\alpha} \vdash \overline{\tau_0}\ \text{OK}}{\Gamma; \overline{\alpha} \vdash \overline{\tau_0}\ Sn.Fn : [\overline{\alpha_0} \mapsto \overline{\tau_0}](\hat{Mt} \to \tau)}\ \text{T-Fun}$$

$$\frac{\Gamma; \overline{\alpha} \vdash E_1 : \tau_2 \to \tau \qquad \Gamma; \overline{\alpha} \vdash E_2 : \tau_2' \qquad \tau_2' \leq \tau_2}{\Gamma; \overline{\alpha} \vdash E_1\ E_2 : \tau}\ \text{T-App}$$

$$\frac{\Gamma; \overline{\alpha} \vdash Ct(\overline{E})\ \text{OK} \qquad Ct = (\overline{\tau}\ C) \qquad \text{concrete}(C)}{\Gamma; \overline{\alpha} \vdash Ct(\overline{E}) : Ct}\ \text{T-New}$$

$$\frac{\Gamma; \overline{\alpha} \vdash E_1 : \tau_1\ \cdots\ \Gamma; \overline{\alpha} \vdash E_k : \tau_k}{\Gamma; \overline{\alpha} \vdash (E_1, \ldots, E_k) : \tau_1 * \cdots * \tau_k}\ \text{T-Tup}$$

$$\frac{\begin{array}{c}\overline{\alpha} \vdash Ct\ \text{OK} \\ Ct = (\overline{\tau_0}\ C) \qquad \text{concrete}(C) \qquad \text{repType}(Ct) = \{\overline{V} : \overline{\tau}\} \qquad \Gamma; \overline{\alpha} \vdash \overline{E} : \overline{\tau_1} \qquad \overline{\tau_1} \leq \overline{\tau}\end{array}}{\Gamma; \overline{\alpha} \vdash Ct\{\overline{V} = \overline{E}\} : Ct}\ \text{T-Rep}$$

$\boxed{\Gamma; \overline{\alpha} \vdash Ct(\overline{E})\ \text{OK}}$

$$\frac{\begin{array}{c}\overline{\alpha} \vdash Ct\ \text{OK} \qquad Ct = (\overline{\tau_0}\ Sn.Cn) \qquad (<\texttt{abstract}>\ \texttt{class}\ \overline{\alpha_0}\ Cn(\overline{I} : \overline{\tau})\ \ldots) \in SigT(Sn) \\ \Gamma; \overline{\alpha} \vdash \overline{E} : \overline{\tau_1} \qquad \overline{\tau_1} \leq [\overline{\alpha_0} \mapsto \overline{\tau_0}]\overline{\tau}\end{array}}{\Gamma; \overline{\alpha} \vdash Ct(\overline{E})\ \text{OK}}\ \text{T-Constr}$$

$\boxed{\text{repType}(Ct) = \{\overline{V} : \overline{\tau}\}}$

$$\frac{\begin{array}{c}(<<\texttt{abstract}>>\ \texttt{class}\ \overline{\alpha}\ Cn(\overline{I} : \overline{\tau_1})\ <\texttt{extends}\ Ct>\ \texttt{of}\ \{\overline{Vn} : \overline{\tau_2}\}) \in SigT(Sn) \\ < \text{repType}(Ct) = \{\overline{V} : \overline{\tau_3}\} >\end{array}}{\text{repType}(\overline{\tau}\ Sn.Cn) = [\overline{\alpha} \mapsto \overline{\tau}]\{ < \overline{V} : \overline{\tau_3}, > Sn.\overline{Vn} : \overline{\tau_2}\}}\ \text{RepType}$$

Figure 4.10: Static semantics of Mini-Eml expressions.

$\boxed{\text{matchType}(\tau, Pat) = (\Gamma, \tau')}$

$$\frac{}{\text{matchType}(\tau, \_) = (\{\}, \tau)}\ \text{T-MatchWild}$$

$$\frac{\text{matchType}(\tau, Pat) = (\Gamma, \tau')}{\text{matchType}(\tau, I\ \texttt{as}\ Pat) = (\Gamma \cup \{(I, \tau')\}, \tau')}\ \text{T-MatchBind}$$

$$\frac{C \leq C' \qquad \text{repType}(\overline{\tau}\ C) = \{\overline{V} : \overline{\tau_0}\} \qquad \text{matchType}(\overline{\tau_0}, \overline{Pat}) = (\overline{\Gamma}, \overline{\tau_1})}{\text{matchType}((\overline{\tau}\ C'), C\{\overline{V} = \overline{Pat}\}) = (\bigcup \overline{\Gamma}, (\overline{\tau}\ C))}\ \text{T-MatchClass}$$

$$\frac{\text{matchType}(\tau_1, Pat_1) = (\Gamma_1, \tau_1') \ \cdots\ \text{matchType}(\tau_k, Pat_k) = (\Gamma_k, \tau_k')}{\text{matchType}(\tau_1 * \cdots * \tau_k, (Pat_1, \ldots, Pat_k)) = (\Gamma_1 \cup \ldots \cup \Gamma_k, \tau_1' * \cdots * \tau_k')}\ \text{T-MatchTup}$$

Figure 4.11: Static semantics of Mini-Eml patterns.

$\boxed{\overline{Sn} \vdash Sn.Cn\ \text{ITCTransUses}}$

$$\frac{(<<\texttt{abstract}>>\ \texttt{class}\ \overline{\alpha}\ Cn(\overline{I} : \overline{\tau})\ <\texttt{extends}\ (\overline{\tau_0}\ C) > \ldots) \in SigT(Sn) \qquad Sn \in \overline{Sn} \qquad < \overline{Sn} \vdash C\ \text{ITCTransUses} >}{\overline{Sn} \vdash Sn.Cn\ \text{ITCTransUses}}\ \text{ClassITCTransUses}$$

$\boxed{\overline{Sn} \vdash F\ \text{ITCUses}}$

$$\frac{Sn \in \overline{Sn}}{\overline{Sn} \vdash Sn.Fn\ \text{ITCUses}}\ \text{FunITCUses}$$

Figure 4.12: Well-formedness of the signatures to be accessed during ITC of a structure.

$\boxed{Sn; \overline{Sn} \vdash \texttt{extend fun}_{Mn}\ \overline{\alpha}\ F\ Pat \text{ unambiguous}}$

$$\frac{\begin{array}{c}(\texttt{fun}\ \overline{\alpha_1}\ Fn : Mt \to \tau) \in SigT(Sn_1) \qquad Sn = Sn_1 \vee \text{owner}(Mt, Pat) = Sn.Cn \\ \forall Sn' \in \overline{Sn}.\forall(\texttt{extend fun}_{Mn'}\ \overline{\alpha_1}\ F\ Pat') \in SigT(Sn'). \\ (Sn.Mn \neq Sn'.Mn' \Rightarrow \overline{Sn} \vdash (Pat, Pat') \text{ unambiguous})\end{array}}{Sn; \overline{Sn} \vdash \texttt{extend fun}_{Mn}\ \overline{\alpha}\ Sn_1.Fn\ Pat \text{ unambiguous}}\ \textsc{Amb}$$

$\boxed{\overline{Sn} \vdash (Pat, Pat') \text{ unambiguous}}$

$$\frac{\begin{array}{c}Pat \ncong Pat' \\ \forall Pat_0.((Pat \cap Pat' = Pat_0) \Rightarrow \\ \exists Sn'' \in \overline{Sn}.\exists(\texttt{extend fun}_{Mn''}\ \overline{\alpha_2}\ F\ Pat'') \in SigT(Sn'').(Pat_0 \cong Pat''))\end{array}}{\overline{Sn} \vdash (Pat, Pat') \text{ unambiguous}}\ \textsc{PairAmb}$$

$\boxed{Pat_1 \cap Pat_2 = Pat}$

$$\frac{}{\_ \cap Pat = Pat}\ \textsc{PatIntWild} \qquad \frac{Pat_1 \cap Pat_2 = Pat}{I\ \texttt{as}\ Pat_1 \cap Pat_2 = Pat}\ \textsc{PatIntBind}$$

$$\frac{C \leq C' \qquad \overline{Pat_1} \cap \overline{Pat_2} = \overline{Pat}}{C\{\overline{V} = \overline{Pat_1}, \overline{V_3} = \overline{Pat_3}\} \cap C'\{\overline{V} = \overline{Pat_2}\} = C\{\overline{V} = \overline{Pat}, \overline{V_3} = \overline{Pat_3}\}}\ \textsc{PatIntClass}$$

$$\frac{\overline{Pat_1} \cap \overline{Pat_2} = \overline{Pat}}{(\overline{Pat_1}) \cap (\overline{Pat_2}) = (\overline{Pat})}\ \textsc{PatIntTup} \qquad \frac{Pat_2 \cap Pat_1 = Pat}{Pat_1 \cap Pat_2 = Pat}\ \textsc{PatIntRev}$$

Figure 4.13: Modular ambiguity checking for Mini-Eml.

proof for Mini-Eml (see section 4.3.4) can only rely on the two properties of $\overline{Sn}$ enforced by rules ClassITCTransUses and FunITCUses, thereby ensuring that modular ITC is sufficient to achieve type safety.

Figure 4.13 formalizes the portion of modular ITC that ensures functions are unambiguous. The notation $\overline{Pat'} \cap \overline{Pat''} = \overline{Pat}$ abbreviates $Pat'_1 \cap Pat''_1 = Pat_1 \cdots Pat'_k \cap Pat''_k = Pat_k$. The top-level rule is Amb. That rule enforces requirement M1, ensuring that the given function case is declared in the same module as either its associated function or its owner. The final premise in Amb uses PairAmb to check that the given case is unambiguous with each available function case other than itself. PairAmb uses the pairwise ambiguity algorithm described in section 4.2. $Pat \cong Pat'$ denotes that Pat is congruent to Pat'; it is an

abbreviation for the two facts $Pat \leq Pat'$ and $Pat' \leq Pat$. Pattern intersection is formalized by the judgment $Pat \cap Pat' = Pat''$. If Pat and Pat' are disjoint then there is no Pat'' such that $Pat \cap Pat' = Pat''$. The rules for pattern intersection are straightforward. Rule PatIntClass is simplified by the fact that classes support only single inheritance.

Figure 4.14 formalizes the portion of modular ITC that ensures functions are exhaustive. The notation $\text{defaultPat}(\overline{\tau}, C, d-1) = \overline{Pat}$ abbreviates $\text{defaultPat}(\tau_1, C, d-1) = Pat_1 \cdots \text{defaultPat}(\tau_k, C,\ d-1) = Pat_k$. Metavariable T ranges over both types and marked types, and metavariable d ranges over nonnegative integers. Rule GDefault checks that a given function has a global default case, and LDefault checks that all available functions whose owners are superclasses of a given class C have a local default case for C. Since a global default case of F is equivalent to a local default case of F for C, where C is the owner of F, the two requirements are able to share the helper rule Default that performs the checks.

The global- and local-default requirements are enforced by the algorithm described in section 4.2. Rule Default generates a (global or local) default pattern and checks that this pattern is at least as specific as the pattern of some available function case. The judgment $\text{defaultPat}(T, C, d) = Pat$ generates a default pattern of (possibly marked) type T. The default pattern dispatches on C in the marked position (if any) of T and accepts any type-correct argument in the other positions. The integer d represents the depth which the generated pattern should have. Rule Default chooses the depth non-deterministically, so the Mini-Eml type soundness proof implies that any depth can safely be used. However, as discussed in section 4.2, an implementation of the algorithm should choose a large-enough depth to ensure precision.

Figure 4.15 contains the helper judgments for accessing the class at the owner position of a function, marked type, and pattern. Finally, the rules defining the judgments $\text{concrete}(C)$, $Pat \leq Pat'$, and $C \leq C'$ in figures 4.6 and 4.7 are borrowed from the dynamic semantics.[4]

[4]Technically, the version of rules Concrete and SubExt in the static semantics should access the signature table rather than the structure table. It is clear by inspection that the rules do not access anything from a structure that is not also available in its principal signature.

$\boxed{\overline{Sn} \vdash F \text{ has-gdefault}}$

$$\frac{\text{owner}(F) = C \qquad \overline{Sn} \vdash F \text{ has-default-for } C}{\overline{Sn} \vdash F \text{ has-gdefault}} \text{ GDefault}$$

$\boxed{\overline{Sn} \vdash \text{funs-have-ldefault-for } C}$

$$\frac{\forall F, C'.[(\overline{Sn} \vdash F \text{ ITCUses} \wedge \text{owner}(F) = C' \wedge C \leq C') \Rightarrow \overline{Sn} \vdash F \text{ has-default-for } C]}{\overline{Sn} \vdash \text{funs-have-ldefault-for } C} \text{ LDefault}$$

$\boxed{\overline{Sn} \vdash F \text{ has-default-for } C}$

$$\frac{\begin{array}{c}(\texttt{fun}\ \overline{\alpha}\ Fn : Mt \to \tau) \in SigT(Sn) \qquad \text{defaultPat}(Mt, C, d) = Pat \\ (\texttt{extend fun}_{Mn}\ \overline{\alpha_0}\ Sn.Fn\ Pat') \in SigT(Sn') \qquad Pat \leq Pat' \qquad Sn' \in \overline{Sn}\end{array}}{\overline{Sn} \vdash Sn.Fn \text{ has-default-for } C} \text{ Default}$$

$\boxed{\text{defaultPat}(T, C, d) = Pat}$

$$\frac{}{\text{defaultPat}(T, C, 0) = \_} \text{ DefZero}$$

$$\frac{d > 0}{\text{defaultPat}(\alpha, C, d) = \_} \text{ DefTypeVar}$$

$$\frac{\text{repType}(\overline{\tau}\ C') = \{\overline{V} : \overline{\tau_0}\} \qquad \text{defaultPat}(\overline{\tau_0}, C, d-1) = \overline{Pat} \qquad d > 0}{\text{defaultPat}((\overline{\tau}\ C'), C, d) = (C'\{\overline{V} = \overline{Pat}\})} \text{ DefClassType}$$

$$\frac{\text{repType}(\overline{\tau}\ C) = \{\overline{V} : \overline{\tau_0}\} \qquad \text{defaultPat}(\overline{\tau_0}, C, d-1) = \overline{Pat} \qquad d > 0}{\text{defaultPat}(\#(\overline{\tau}\ C'), C, d) = (C\{\overline{V} = \overline{Pat}\})} \text{ DefOwnerClassType}$$

$$\frac{\text{defaultPat}(T_1, C, d-1) = Pat_1\ \cdots\ \text{defaultPat}(T_k, C, d-1) = Pat_k \qquad d > 0}{\text{defaultPat}(T_1 * \ldots * T_k, C, d) = (Pat_1, \ldots, Pat_k)} \text{ DefTupType}$$

$$\frac{d > 0}{\text{defaultPat}(\tau_1 \to \tau_2, C, d) = \_} \text{ DefFunType}$$

Figure 4.14: Modular exhaustiveness checking for Mini-Eml.

$\boxed{\text{owner}(F) = C}$

$$\frac{(\texttt{fun}\ \overline{\alpha}\ Fn : Mt \to \tau) \in SigT(Sn) \qquad \text{owner}(Mt) = C}{\text{owner}(Sn.Fn) = C}\ \textsc{OwnerFun}$$

$\boxed{\text{owner}(Mt) = C}$

$$\frac{}{\text{owner}(\texttt{\#}\overline{\tau}\ C) = C}\ \textsc{OwnerClass}$$

$$\frac{\text{owner}(Mt) = C}{\text{owner}(\tau_1 * \cdots * \tau_{i-1} * Mt * \tau_{i+1} * \cdots * \tau_k) = C}\ \textsc{OwnerTup}$$

$\boxed{\text{owner}(Mt, Pat) = C}$

$$\frac{\text{owner}(Mt, Pat) = C}{\text{owner}(Mt, I\ \texttt{as}\ Pat) = C}\ \textsc{OwnerBindPat}$$

$$\frac{\text{owner}(Mt, Pat_i) = C}{\text{owner}(\tau_1 * \cdots * \tau_{i-1} * Mt * \tau_{i+1} * \cdots * \tau_k, (Pat_1, \ldots, Pat_k)) = C}\ \textsc{OwnerTupPat}$$

$$\frac{}{\text{owner}(\texttt{\#}Ct, C\{\overline{V} = \overline{Pat}\}) = C}\ \textsc{OwnerClassPat}$$

Figure 4.15: Accessing the owner.

4.3.4 Type Soundness

MINI-EML's type system is *sound*: a well-typed MINI-EML program cannot incur type errors at run time. MINI-EML's type errors are defined implicitly by the *stuck* expressions, which are those expressions that are not values but cannot be further evaluated (because there is no applicable rule in the MINI-EML dynamic semantics). The message-not-understood and message-ambiguous type errors are represented by the fact that function invocations lacking a most-specific applicable function case are stuck in the MINI-EML dynamic semantics. MINI-EML's soundness therefore validates the correctness of Dubious's System M (in the absence of multiple inheritance): modular ITC with System M's requirements is sufficient to ensure that function-case lookup always succeeds at run time.

Let $\vdash E : \tau$ denote the typechecking of E in the context of the empty type environment and empty sequence of type variables. Let $\rightarrow^*$ denote the reflexive, transitive closure of the $\rightarrow$ relation in the MINI-EML dynamic semantics. Finally, let $E \uparrow$ denote the fact that evaluation of E *diverges*: for all E', if $E \rightarrow^* E'$ then there exists some E'' such that $E' \rightarrow E''$. MINI-EML's type system is sound if the evaluation of a well-typed expression cannot result in a stuck expression: evaluation either diverges or ends in a value (of the appropriate type).

Theorem (Type Soundness) If $\vdash E : \tau$, then either $E \uparrow$ or there exist v and τ' such that $E \rightarrow^* v$ and $\vdash v : \tau'$, where $\tau' \leq \tau$.

As is standard [105], I prove type soundness via a *progress* theorem and a *type preservation* theorem, which together imply the above theorem. The progress theorem says that a well-typed expression can always take a step in the dynamic semantics:

Theorem (Progress) If $\vdash E : \tau$ and E is not a value, then there exists E' such that $E \longrightarrow E'$.

The type preservation theorem says that evaluation preserves well-typedness:

Theorem (Type Preservation) If $\vdash E : \tau$ and $E \longrightarrow E'$, then there exists τ' such that $\vdash E' : \tau'$ and $\tau' \leq \tau$.

The proofs of these two theorems are provided in appendix A. Proving progress requires reasoning about modular ITC, in order to show that function applications can always make

```
structure BadMod = struct
  class C() of {}
  fun f:#C → unit
  val bad = f(C())
  extend fun f(C {}) = ()
end
```

Figure 4.16: The effect of value declarations on modular ITC.

progress. The key lemma says that a most-specific applicable function case exists in the dynamic semantics for each type-correct function application:

Lemma If $\vdash Fv : \tau_1 \rightarrow \tau_2$ and $\vdash v : \tau_1'$ and $\tau_1' \leq \tau_1$, then there exist ρ and E such that most-specific-case-for(Fv,v) = (ρ, E).

Proving type preservation is relatively straightforward, as it is completely independent of modular ITC.

4.4 ML-style Modules

This section discusses the interaction of Eml with an ML-style module system [69, 76], including structures, signatures, and functors. I describe the problems that can arise for modular ITC in this context and sketch some possible solutions.

4.4.1 Structures

Thus far I have assumed that Eml structures contain only a sequence of class, function, and function case declarations. Eml structures should also accommodate the ordinary ML declarations. These include the ability to bind a name to a value, provide a synonym for a type, declare an exception, and declare an inner structure. The latter three kinds of declarations can be straightforwardly incorporated, but special care is needed to handle value declarations. Figure 4.16 shows an example of the problems that can occur. As presented so far, ITC on `BadMod` succeeds, because function `f` has an appropriate case for `C`. However, at run time a message-not-understood error occurs when the `val` declaration is executed, because `f`'s function case has not yet been declared.

There are several approaches to handling this problem. One solution would be to adopt a two-pass style of structure evaluation. The first pass would evaluate all of the declarations except the value declarations, and the second pass would evaluate the value declarations. In figure 4.16, this semantics would ensure that `f`'s function case is declared before `f` is invoked. An alternative approach would be to make the unit of modular ITC more fine-grained than an entire structure, with `val` declarations forming the boundaries of these units. For example, `BadMod` would consist of two units, one of which contains the first two declarations and the other containing the last declaration. When ITC is performed on the first unit, the exhaustiveness problem for `f` on `C` would result in a static error. The prototype EML interpreter uses a variant of this approach.

4.4.2 Signature Ascription

By default, clients of an ML structure *Sn* access its components through the view provided by *Sn*'s principal signature. Intuitively, a structure's principal signature provides complete interface information to clients. Information hiding in ML is achieved by explicitly *ascribing* a signature to a structure. Let metavariable *SigN* range over *signature names*. Clients of a structure

$$\texttt{structure}\ Sn\ \texttt{:>}\ SigN = \texttt{struct} \cdots \texttt{end}$$

may only access *Sn*'s components through the view provided by the signature *SigN*.[5] The ascribed signature may include less information than the structure's principal signature. For example, in ML an ascribed signature may omit specifications for some of the structure's declarations, making them inaccessible to clients.

Signature specifications in EML are analogous to those of MINI-EML as shown in figure 4.4b. Each specification has identical syntax to its corresponding EML declaration, but with all expressions removed. Signature ascription for EML provides forms of OO-style encapsulation. For example, classes, functions, and function cases can be hidden from clients, making them private to their enclosing structure. However, these declarations cannot be

[5]Standard ML includes two kinds of ascription: *transparent* and *opaque* [76]. Transparent ascription can be desugared into opaque ascription, so I focus exclusively on the latter form.

```
signature ShapeSig = sig
  abstract class Shape() of {}
  fun bad:#Shape → unit
  extend fun bad s
end

structure ShapeMod :> ShapeSig = struct
  abstract class Shape() of {}
  fun draw:#Shape → unit
  fun bad:#Shape → unit
  extend fun bad s = draw s
end

structure CircleMod
  class Circle() extends Shape() of {}
end
```

Figure 4.17: The effect of signature ascription on modular ITC.

hidden arbitrarily, or else modular ITC would become unsound. Figure 4.17 shows a simple example of the problems that can occur. `ShapeMod` creates the abstract `Shape` class and two associated functions, `draw` and `bad`. ITC in `ShapeMod` succeeds for both functions: `draw` is found to be exhaustive and unambiguous because `Shape` is abstract. Because `ShapeSig` is ascribed to `ShapeMod`, `draw` is hidden from `ShapeMod`'s clients. Therefore, ITC succeeds for `CircleMod`: `bad` has a most-specific applicable case for `Circle`. If `bad` is ever invoked with a `Circle` instance, however, `draw` will be invoked, causing a message-not-understood error.

The example is purposely similar to the exhaustiveness problem with open classes illustrated for Dubious in figure 2.7. In that case, the global-default requirement ensures that the problem is modularly detected. Intuitively, the modular ITC problems of signature ascription can be solved in the same way: a set of declarations can be safely hidden if that set could have been declared in a separate structure that passes modular ITC [74]. The `draw` function in figure 4.17 does not satisfy this condition. If `draw` were declared in its own structure, requirement M4 would force the existence of a global default case for `draw`, since `draw` would now be an external function. If `draw` had such a case, then the function (and that case) could be hidden via signature ascription, and the problem for `Circle` would be resolved.

Aside from hiding entire declarations, it is useful to hide properties of a declaration. Several properties of classes may be hidden. First, any subset of a class's instance variables may be hidden. This hiding does not preclude clients from creating instances of the class, because instances are created only by invoking constructors, and neither the class's constructor nor any constructor arguments have been hidden. As mentioned in section 4.3, instance variables are scoped — the name of the structure declaring an instance variable is implicitly part of the name of the instance variable. Therefore, there is no conflict if a subclass in a new module creates an instance variable of the same name as a hidden one in the superclass. Second, a concrete class can be viewed as an abstract one, thereby disallowing clients from instantiating the class. System M's requirements M3 and M4 ensure that clients will provide the appropriate default methods in the face of these supposedly abstract classes. Treating a concrete class as abstract could be useful, for example, to enforce the *singleton pattern* [43], in which a class has a single instance.

A signature can also declare a class C `sealed` [96], which hides C's extensibility: classes declared outside of C's structure may not directly subclass from C. This construct can be used to faithfully model ML-style (nonextensible) datatypes. For example, suppose the first two structures in figure 4.1 were combined into one. If that structure were ascribed to a signature that specified both `Set` and `ListSet` as `sealed`, then other structures would be disallowed from declaring new (direct or indirect) subclasses of `Set`. In that case, sets form a *sealed hierarchy*. As discussed in section 4.2, functions in ML can safely omit (local and global) default cases, because datatypes are nonextensible. EML's modular requirements can be analogously relaxed for functions on sealed hierarchies. For example, if the set hierarchy were sealed, then `IsEmptyMod` could safely omit the first `isEmpty` case, which is the global default. Because there would never exist concrete subclasses of `Set` other than `ListSet`, the other `isEmpty` cases would be sufficient to ensure exhaustiveness. The local-default requirement can be relaxed similarly.

Hiding other properties of classes is more problematic. It would be useful for a signature to expose only a transitive, rather than the direct, superclass of a class. This ability would reduce the dependence of clients on a class's particular implementation. It would similarly be useful to ascribe an ML-style type specification to a class, possibly augmenting the

```
structure PointMod = struct
  abstract class Point() of {}
  fun draw:#Point → unit
end

signature APointSig = sig
  class APoint(x:int,y:int)
    extends Point of {x:int,y:int}
  extend fun draw(APoint {x=x,y=y})
end

functor Colorize(M:APointSig) = struct
  class ColorPoint(x:int,y:int,color:int)
    extends M.APoint(x,y) of {color:int=color}
  extend fun draw
    (ColorPoint {x=x,y=y,color=color}) = ...
  fun getColor:#ColorPoint → int
  extend fun getColor
    (ColorPoint {x=x,y=y,color=color}) = color
end
```

Figure 4.18: Encoding mixins with EML functors.

specification with *partial revelations* [80] to reveal some of the class's underlying structure. Unfortunately, these abilities make modular ITC unsound. For example, a client of two classes C and C' can write ambiguous function cases that appear to be unambiguous if the fact that C subclasses from C' is hidden.

Finally, a function may be sealed by ascribing an ordinary ML-style value specification to the function and its cases. For example, `isEmpty` and its three cases in figure 4.1 could be represented in a signature by the specification `val isEmpty : Set → bool`. Clients can still invoke the sealed `isEmpty` function, but its extensibility is hidden (and the # mark is no longer necessary): clients may not add new cases to `isEmpty` and do not perform ITC on it. In this way, function sealing allows EML to model ML-style (nonextensible) functions. Function sealing is allowed under the same circumstances that the function and its cases may be hidden, thereby ensuring that the sealed function is exhaustive and unambiguous.

4.4.3 Functors

Standard ML supports *functors*, which are structures parameterized by other structures. In the presence of EML's features, functors can provide a great deal of flexibility. Figure 4.18 illustrates some of the idioms that would be useful to express. The `PointMod` structure contains a `Point` base class with an associated `draw` function. The `Colorize` functor implements a mixin [12, 37, 41], since the `ColorPoint` class is parameterized by its superclass. The functor can be *instantiated* by applying it to any structure that matches the `APointSig` signature, thereby creating a colored version of that structure's `Point` subclass. An overriding case for the existing `draw` function is given, in order to draw colored points specially. The functor also introduces a new function for accessing the color of a colored point, with an associated case.

In ML, each functor body can be safely typechecked once, given only the signature of the argument structure. EML should similarly perform modular ITC once on a functor body, guaranteeing exhaustiveness and unambiguity of all relevant functions no matter how the functor is instantiated. The major challenge for modular ITC of functors like `Colorize` is the fact that the identities of some classes, for example `M.APoint`, are unknown. Instead only partial information is known about the relationship between `M.APoint` and other classes. While others have investigated the integration of functors (or other forms of *external linking*) with traditional OO-style classes (e.g. [38, 3, 71]), the interaction of functors with EML's extensible datatypes and functions has not been considered previously.

A possible approach to conservatively performing ITC in the presence of partial information in EML is to generalize the subclass relation in the static semantics to be *three-valued*, saying "don't know" when the partial class hierarchy information is inconclusive. The pattern specificity relation is then also generalized to be three-valued, making use of the generalized subclass relation. Lastly, modular ITC is modified to be conservative with respect to three-valued subclassing and pattern specificity. In ambiguity checking, all tests for pattern congruence and disjointness should succeed only if the given patterns are definitely congruent and disjoint, respectively. Similarly, the intersection of two patterns should result in a pattern that is definitely the intersection pattern. If the two patterns are neither

```
structure APointMod = struct
  class APoint(x:int,y:int)
    extends Point() of {x:int = x, y:int = y}
  extend fun draw(APoint {x=x,y=y})
end

functor Negate(M:APointSig) = struct
  fun negate:#Point → Point
  extend fun negate(Point _) = ...
  extend fun negate(APointMod.APoint _) = ...
  extend fun negate(M.APoint _) = ...
end
```

Figure 4.19: Three-valued modular ITC of functor bodies.

definitely disjoint nor definitely intersecting, then the associated cases are conservatively considered ambiguous. In exhaustiveness checking, a local default should be required for any class that *may* subclass an available function's owner. Finally, the generated (local or global) default pattern should definitely be at least as specific as some function case's pattern.

As an example, consider three-valued ITC on `negate` in `Negate` of figure 4.19. The first case is definitely a global default, so `negate` is exhaustive. As usual, the first two cases are found to be unambiguous. The first and third cases are similarly found to be unambiguous, even though the identity of `M.APoint` is not known. From the definition of `APointSig` in figure 4.18 it is clear that `M.APoint` strictly subclasses `Point`, and this is enough information to compute the definite intersection of the two patterns. Finally, the second and third cases are found to be ambiguous: their patterns are neither definitely disjoint nor definitely intersecting. Indeed, if `Negate` is ever instantiated with `APointMod`, the resulting `negate` function will be ambiguous for `APoint`.

The restrictions on signature ascription described earlier can severely limit the reusability of functors. For example, the `Colorize` and `Negate` functors can only be instantiated with a class `APoint` that is a direct subclass of `Point`, rather than an indirect (transitive) one. Also, `APoint`'s module must contain a `draw` case with the pattern specified in `APointSig`. While relaxing these restrictions would increase functor reusability, it would also force ITC

on functor bodies to become much more conservative, in order to account for the new expressiveness of clients. The relaxation could therefore reduce the overall benefit of functors by requiring the type system to disallow too many of them.

A pragmatic way to avoid the restrictions on signature ascription could be to move some of the burden of ITC to clients of the functor. In the limit, EML would perform modular ITC once per instantiation of the functor, on the structure resulting from the instantiation. At the point of instantiation, all the identities of classes and functions in the functor's argument are known, so ordinary modular ITC as described in section 4.2 would suffice. It is possible that in practice most of ITC could still be performed on the functor body in isolation, with only a few additional checks performed per instantiation. Such a scheme would confer benefits akin to Relaxed MultiJava as described in section 3.5, providing early feedback about a functor's type correctness while still safely supporting the desired expressiveness.

4.5 Related Work

There have been several other research efforts to integrate the benefits of the functional and OO styles. OCaml [90] adds OO features including class and method definitions to ML. The OO constructs form their own sublanguage which is largely separate from the existing ML `datatype` and `fun` constructs. In a similar vein, Pizza [84] adds ML-style datatypes and functions (among other things) to Java. Adding a set of new constructs has the advantage that existing language constructs are minimally affected by the extension, retaining their traditional semantics and modular typechecking scheme, including type inference. However, such simplicity comes at a cost to programmers, who are forced to choose up front whether to represent an abstraction with datatypes or with classes. This decision impacts the kind of extensibility allowable for the abstraction. It may be difficult to determine *a priori* which kind of extensibility will be required, and it is difficult to change the decision after the fact. Further, it is not possible for the abstraction to enjoy both kinds of extensibility at once.

OML [91] introduces an `objtype` construct to ML, which is a form of hierarchical and extensible datatype similar to EML's `class` construct. However, OML still maintains a distinction between methods and functions, which have different benefits. New methods

may not be added to existing `objtype`s without modifying existing code, while ordinary ML functions may be. Methods dynamically dispatch on their associated `objtype`, while functions support ML-style pattern matching. Unlike Eml, OML retains ML-style type inference. OML has no direct support for field and method inheritance, but the authors show how to encode inheritance and other features of traditional classes via the ML module system [92].

Zenger and Odersky [106] describe an extensible datatype mechanism in the context of an OO language. Extending a datatype has the effect of creating a new datatype that subtypes from the original one. To ensure exhaustiveness in the presence of datatype extension, all functions on extensible datatypes must include the equivalent of a global default case, while Eml often requires only local defaults. Because Zenger's functions are not extensible, if new data variants require overriding function cases, a new function must be created that inherits the existing function cases, and clients must be modified to invoke the new function. Like OML, Zenger's language includes both OO-style methods and ML-style functions. Zenger's language also retains a distinction between datatype "cases" and regular OO classes. Garrigue shows how to use *polymorphic variants*, which are variants defined independent of any particular datatype, to obtain both modular data-variant and function extensibility in ML [44]. As in Zenger's language, when a function is extended any clients that require the new functionality must be modified. Polymorphic variants are not hierarchical and do not support subtyping, but they preserve ML-style type inference.

Mixin modules [33] allow datatype and function declarations to be split across multiple modules, thereby providing a form of extensible datatypes and functions. Mixin modules must be explicitly combined to form the complete datatypes and functions. Therefore, there must be a single place in the program where all extensions to a given datatype or function are known. This contrasts with the "nonlinear" extensibility of Eml: there need not be a single structure where all of a class's subclasses or all of a function's cases are available. Mixin modules are also not hierarchical and do not support subtyping. Later work [34] incorporates mixin modules into a traditional class-based OO language. A distinction is made between class-based and mixin-based objects. Mixin-based objects do not support subtyping, but they support a form of type specialization for methods that Eml lacks.

$ML_{\leq}$ [10] is the closest language to EML, generalizing ML datatypes to be hierarchical and extensible and simulating (multi)methods via function cases that use OO-style dispatching semantics. However, patterns in function cases are restricted to be top-level datatype constructor tests, which are the analogue of dynamic dispatch tests in OO languages. $ML_{\leq}$ also does not support extensible functions: all function cases are provided when a function is declared. The authors sketch a source-level language that supports extensible functions, but this generalization requires whole-program ITC for safety. In contrast, EML uses Dubious's System M to support extensible functions with fully modular typechecking. $ML_{\leq}$'s polymorphic type system includes a form of bounded quantification, while EML contains only ML-style parametric polymorphism.

Several recent languages support pattern matching for XML-like [13] data. For example, XDuce [52] augments ML-style pattern matching with support for regular expression patterns. XDuce does not include hierarchical or extensible datatypes, but regular expression types naturally incorporate a form of structural subtyping. ITC for XDuce is formalized by operations on tree automata. Like ML, a first-match semantics is used and functions are nonextensible, so modular ITC is straightforward. HydroJ [64] provides "handlers" that support pattern matching similar to that of XDuce, but in the context of Java. Unlike XDuce, HydroJ use OO-style dispatching semantics, and HydroJ's handlers are extensible via subclasses. HydroJ handlers are guaranteed to be exhaustive, because the argument type of the handler "generic function" is simply the union of the types of its methods' patterns. Unambiguity is checked using an adaptation of EML's strategy involving intersection patterns. Requirement M1 is unnecessary because a handler may not be declared outside of its receiver class.

Work on predicate dispatching [35] describes ITC for patterns that are arbitrary predicates, subsuming EML's pattern language. An OO-style dispatching semantics is used, with predicate implication as the specificity relation among patterns, and functions are extensible. However, the ITC algorithm is not modular, instead requiring access to the entire program to ensure safety.

4.6 Future Work

There are several directions for future work. First, section 4.1.3 discussed future work related to EML's polymorphic type system. Second, EML currently does not allow aliasing of classes or extensible functions. This restriction ensures that each class and function is statically known, so new subclasses or function cases can be statically reasoned about. A general approach to handling aliasing would allow classes and extensible functions to be less second-class and would better integrate them with ML-style types and functions. For example, a form of linear types [104] could allow some statically unknown classes and functions to be safely extended. Third, more work is also needed to integrate EML with ML-style modules, particularly functors. Section 4.4 sketched some of the challenges and solutions, but more study and experience are necessary to find practical requirements that balance expressiveness and modular typechecking.

Finally, the EML prototype interpreter is not suitable for practical experimentation with the language. Because it is written as a standalone application, it lacks support for many of ML's primitives and library functions, which are necessary for large-scale program development. I would like to create a full-fledged implementation of EML as an extension to an existing ML interpreter or compiler. This will allow me to more easily gain experience and gauge the utility of the language in practice, to play with alternative designs, and hopefully to attract existing ML users.

Chapter 5

EXPERIENCE

MultiJava has been used by others in the implementation of several applications since March 2002. This chapter illustrates the ways in which MultiJava's features have been employed and reports on user feedback about the benefits and limitations of the language. The applications span several domains. First, MultiJava has been used to implement reliable ubiquitous computing systems. The one.world system [46] is a platform for building ubiquitous systems. Labscape [6] is a ubiquitous computing environment for cell biology laboratories that is built on top of one.world. The Location Stack [49] is a framework for combining and representing measurements from a heterogeneous network of sensors that track the locations of objects in an environment. Second, MultiJava has been used to implement two compilers. Hydro is a domain-specific language for XML data processing, and HydroJ [64] is an extension to Java supporting Hydro's features. Finally, MultiJava was used to implement a graphical user interface (GUI) for manipulating a reconfigurable chip that performs machine learning tasks [14].

5.1 Multimethods

5.1.1 Binary Methods

One of the simplest uses of multimethods is in the implementation of binary methods. In Java, all classes have at least one binary operation, the `equals` generic function inherited from `java.lang.Object`. Figure 5.1a shows a common idiom for implementing `equals` methods in Java, and figure 5.1b shows the MultiJava equivalent. The Location Stack as well as the Hydro and HydroJ compilers implement `equals` methods using MultiJava's multimethods. Hydro and HydroJ also include an expressive sublanguage for pattern matching on XML data, as discussed in section 4.5. A variant of the MultiJava style shown in

```
class C {
  boolean equals(Object o) {
    if (o instanceof C) {
      C c = (C) o;
      ... // check equality of two Cs
    } else {
      return false;
    }
  }
}
```

(a)

```
class C {
  boolean equals(Object@C c) {
    ... // check equality of two Cs
  }
  boolean equals(Object o) {
    return false;
  }
}
```

(b)

Figure 5.1: (a) Binary methods in Java. (b) Binary methods in MultiJava.

figure 5.1b is used in the Hydro and HydroJ compilers to implement binary methods on patterns, including pattern specificity and pattern intersection.

Even on the small example in the figure, the MultiJava version enjoys several advantages. Each method in the MultiJava version declaratively expresses its dispatching behavior in its header. This allows programmers to more easily understand the functionality and allows MultiJava to check statically for exhaustiveness errors and ambiguities. In contrast, the dispatching in the Java version is performed manually through run-time type tests and casts, and dispatching errors are not caught until run time. The MultiJava version also allows functionality to be more easily inherited. For example, the second `equals` method in figure 5.1b is typically not written. Instead, MultiJava's dispatching semantics ensures that the `equals` method in `java.lang.Object` will be invoked whenever the receiver is a `C` instance (or a subclass) but the argument is not. Because `Object`'s `equals` method implements pointer equality, the simplified code has the same semantics as the original. In contrast, the Java version of `C`'s `equal` method must always include an `else` case to ensure exhaustiveness.

5.1.2 Event Dispatching

Many kinds of applications are naturally structured in an event-based style. In this style, components do not communicate by directly sending messages to one another. Instead, each

```
abstract class Component {
  abstract void handleEvent(Event e);
}
```

Figure 5.2: The base class of components in an event-based system.

component is able to *announce* a set of events. Separately, other components can register to receive notification whenever a certain event is announced by providing a *handler* procedure for the event. When an event is announced, the system invokes all the handlers that are associated with that event.

The canonical example of an event-based system is a GUI. Events are announced in response to user actions (e.g., clicking a button), and these events trigger the appropriate actions of components (e.g., updating the display). Java's Abstract Window Toolkit (AWT) [55] is a library for building GUIs that employs the event-based style, and the reconfigurable chip's GUI is built on top of AWT. The event-based style is also used by all of the ubiquitous systems described above. In these systems, extensibility is at a premium: it must be possible for new components to easily join and leave the system dynamically. The event-based style facilitates this extensibility by keeping components loosely coupled, since components communicate only indirectly through events.

Basic Event Dispatching

In the context of an object-oriented language like Java, an event-based system typically includes an abstract class or interface that defines the required functionality of all components, as shown in figure 5.2. Each component is a concrete subclass of `Component`, and each event is similarly a subclass of an abstract `Event` class. A component's `handleEvent` method is its event handler: when an event occurs, the components that have registered for that event are notified by having their `handleEvent` methods invoked, passing the announced event as an argument.

The `handleEvent` operation is a natural application for multiple dispatch in MultiJava. The functionality for handling an event depends both on which component is handling the

```
class EditorGUI extends Component {
  void handleEvent(Event e) {
    if (e instanceof Open) {
      Open o = (Open) e;
      ... // open a file
    } else if (e instanceof Save) {
      Save s = (Save) e;
      ... // save the currently opened file
    } else if (e instanceof Quit) {
      Quit q = (Quit) e;
      ... // quit the application
    } else {
      ... // handle unexpected events
    }
  }
}
```

(a)

```
class EditorGUI extends Component {
  void handleEvent(Event@Open o) {
    ... // open a file
  }
  void handleEvent(Event@Save s) {
    ... // save the currently opened file
  }
  void handleEvent(Event@Quit q) {
    ... // quit the application
  }
  void handleEvent(Event e) {
    ... // handle unexpected events
  }
}
```

(b)

Figure 5.3: (a) Event handling in Java. (b) Event handling in MultiJava.

event and on which event has been announced, but Java only allows one of these hierarchies to be dispatched upon. Therefore, programmers must manually dispatch on the other hierarchy, usually via run-time type tests and casts. An example of Java and MultiJava event handlers in a hypothetical GUI for a text editor is shown in figure 5.3.

The benefits of MultiJava illustrated for binary methods are accrued to an even greater extent in the context of event handling. MultiJava allows each conceptual handler to be encapsulated as its own method, rather than buried in an `if` case of one monolithic method. Users report that this style of implementing handlers exactly matches their high-level view of a component as containing a set of handlers, each handling a particular event. The header of each multimethod characterizes the conditions under which that handler will be invoked, and static checking ensures that there is a most-specific applicable handler for each possible event. As a simple example of static checking, the lack of a default handler accepting any `Event` would signal a static exhaustiveness error in MultiJava. In contrast, the Java version would compile without error but fail dynamically in unexpected ways if the final `else` case were missing. Users report such errors to be common, particularly in the ubiquitous computing context, where the system can easily become misconfigured as

components enter and exit.

The MultiJava style also had the unexpected effect of encouraging programmers to write better documentation. For example, in one.world, "Handles events" is the typical comment for a monolithic `handleEvent` method. A one.world developer reported that MultiJava made it natural for him to instead document the actual behavior of each handler in comments. Such documentation can then be displayed by *mjdoc* [79], a web-based documentation tool for MultiJava programs developed by David Cok, which is similar to Java's *javadoc* tool.

Event Dispatching in Practice

Event handlers are often significantly more complicated than the example shown in figure 5.3, and MultiJava's advantages over Java increase with this complexity. For example, the various `Event` subclasses may form a deep hierarchy, with subclasses of `Event` having their own subclasses. This scenario occurs in both one.world and the Location Stack. In the text editor example above, suppose that `Save` has a subclass `SaveAs` for saving to a new file. If special behavior for `SaveAs` is required for `EditorGUI` in figure 5.3, the Java version must be updated carefully with a new `if` case. The programmer must ensure that that the `SaveAs` case comes before the case for `Save`, or else the new case will never be invoked. In general, the programmer must always ensure that a class is tested before any of its superclasses. In the MultiJava version, a new `handleEvent` multimethod specializing on `SaveAs` can be added anywhere in `EditorGUI`, and MultiJava's dispatching semantics ensure that it will be invoked properly.

Deep hierarchies also naturally utilize `resend`. For example, suppose that `EditorGUI` on `SaveAs` should do everything that it does for `Save`, preceded by some extra statements $s_1, \ldots, s_n$ (e.g., initializing the new file) and followed by some other statements $s'_1, \ldots, s'_m$. In the MultiJava version, the `handleEvent` method for `SaveAs` simply performs $s_1, \ldots, s_n$, invokes `resend`, and then performs $s'_1, \ldots, s'_m$. The Location Stack employs this style to reuse code across handlers. In contrast, in the Java version one must either duplicate the code for `Save` in the case for `SaveAs`, or one must create a helper method that both cases invoke, which requires that the case for `Save` be modified.

Another common source of complexity in practice is when a handler depends upon more than just the kind of event announced in order to determine what action to take. For example, events in one.world have a `closure` field of type `Object`. The closure is used in request-response interactions to distinguish among several response events processed by the same event handler. Many handlers dispatch based on both which event is passed and what kind of closure that event contains. MultiJava generalizes naturally to handle this scenario, via dispatch on multiple arguments. Instead of defining `handleEvent` by a set of multi-methods, a single `handleEvent` method now invokes a helper method `handleWithClosure`, passing both the announced event and the event's closure field. The `handleWithClosure` method then performs the desired dispatching:

```
void handleWithClosure(Event@Event1 e1, Object@LocalClosure closure) { ... }
void handleWithClosure(Event@Event2 e2, Object@RemoteClosure closure) { ... }
...
```

Each `handleWithClosure` method cleanly documents the conditions under which it will be invoked, and static typechecking ensures a most-specific applicable handler for each (event, closure) pair. The only limitation is the need to create this helper method, in order to dispatch on the `closure` field. Aside from being more verbose, the helper method also breaks the bond between the event and the closure: there is nothing in the `handleWithClosure` operation documenting the fact that the closure argument should be the value of the `closure` field of the event argument. Using the helper method also makes it hard to inherit handlers from superclasses (as described below), unless those superclasses also use `handleWithClosure` helper methods. An extension to MultiJava to incorporate ML-style pattern matching as illustrated in chapter 4, or more generally a form of predicate dispatching [35], would allow fields to be directly dispatched upon.

To handle dispatch on `closure` fields, the monolithic Java version gets even more obfuscated than it already is, requiring additional type tests and casts. In one.world this is typically done by nested `if`s: the outer `if` dispatches on the event, and each case of that `if` dispatches on the various kinds of closures. This approach is tedious and also asymmetric, making it harder to understand the conditions under which each "handler" triggers. Fur-

ther, a one.world developer reports that this style made it too easy to omit enforcement of some dispatching requirements in the code. For example, dispatching on the closure can be omitted in outer `if` cases that don't use the closure (or don't depend on its run-time class), even if it is still the intent that the closure be of a particular class. Because of the tedium of the Java style, this kind of shortcut occurs often in one.world. Revisiting his own code that exhibits this shortcut, the developer was not sure whether his intent was that any closure should be allowed or not. In the MultiJava version, there is no advantage to omitting dispatch requirements, because each `handleWithClosure` method explicitly mentions both the event and its closure, and expressing a dispatching requirement is lightweight and declarative.

Component Hierarchies

MultiJava also allows new ways of structuring handlers that were not considered previously by the developers of the event-based systems. Because multimethods can be inherited, it is possible to have deep hierarchies of components. Each component inherits all of the handlers of its superclasses, optionally overrides any of these handlers, and adds new handlers.

Both the reconfigurable chip's GUI and the Location Stack employ this style. For example, in the GUI, the abstract `Component` class represents an arbitrary GUI element. It has a default method handling any `Event` that acts as the "error handler," freeing subclasses from having to handle unexpected events. An abstract `HighlightedComponent` subclass represents a component that is able to be highlighted. It inherits the error handler and adds a handler that responds to the act of highlighting by updating the GUI properly. A `TerminalComponent` is a subclass of `HighlightedComponent` representing a wire connection point on the chip. It inherits the error handler and the highlighting functionality, and it has additional handlers for events specific to terminals.

Simulating this idiom of fine-grained handlers inheriting functionality from superclasses is very awkward in the Java version, where each handler is a monolithic `if` block. Each subclass has to explicitly invoke `super` in the right places to manually dispatch to superclass handlers when inheritance is desired. That style is so unnatural that the developers of these

```
void handleEvent(ActionEvent e) {
  String cmd = e.getActionCommand();
  if (cmd.equals("open")) {
    ... // open a file
  } else if (cmd.equals("save")) {
    ... // save the currently opened file
  } else if (cmd.equals("quit")) {
    ... // quit the application
  } else {
    ... // handle unexpected events
  }
}
```

(a)

```
void handleEvent(ActionEvent e) {
  handleCmd(e, e.getActionCommand());
}
void handleCmd(ActionEvent e,
               String@@"open" cmd) {
  ... // open a file
}
void handleCmd(ActionEvent e,
               String@@"save" cmd) {
  ... // save the currently opened file
}
void handleCmd(ActionEvent e,
               String@@"quit" cmd) {
  ... // quit the application
}
void handleCmd(ActionEvent e,
               String cmd) {
  ... // handle unexpected events
}
```

(b)

Figure 5.4: (a) Dispatching on primitive events in Java. (b) Dispatching on primitive events in MultiJava.

systems did not even consider handler inheritance to be an option before they started using MultiJava.

Value Dispatching

The event-based style presented so far uses a hierarchy of `Event` subclasses to represent the various events in a system. An alternative approach employs a single `Event` class with no subclasses, with an integer or string field signifying which kind of event a particular instance represents. Although this latter style is less object-oriented and less expressive (e.g., it does not allow deep hierarchies of events), it is perceived to be a more lightweight solution and is fairly common. If an event-based system uses this style, MultiJava's value dispatching can be used to declaratively dispatch on the event "tags."

For example, the Java AWT sometimes uses strings to distinguish events. Its `ActionEvent` class has an "action command" string that is set in the constructor and

specifies the event being represented. Figure 5.4 shows how event handlers in this style are written in Java and MultiJava, for the hypothetical text editor. The MultiJava version has the same advantages as described for the code in figure 5.3b. Labscape employs MultiJava in this way to handle events related to its GUI. Value dispatching allows Labscape to use the existing AWT library while still enjoying the benefits of the MultiJava style.

There are opportunities for value dispatching even when events are written in a class hierarchy. An earlier example illustrated event handlers that depend upon an event's `closure` field in addition to the event's run-time class. Some one.world events, subclasses of `TypedEvent`, contain an integer `type` field which is used to distinguish among events of the same general kind. MultiJava handlers in this case look similar to the `handleWithClosure` methods presented above, but with the second argument employing value dispatching on the `type` field of the received event.

Limitations

Users did point out some limitations of MultiJava in the context of event dispatching. First, dispatching on properties of an event other than its run-time class requires the creation of helper methods, as mentioned earlier. Second, it is not easy to update a component when a new event type enters the system. The component must be augmented in place to contain a multimethod specializing on the new event type. This is still better than the Java version, in which the programmer must find the right place in the `if` chain to place a new case. However, it would be nice to write the new multimethod external to the component, thereby allowing new events to be incorporated without modifying existing code. This ability is a step toward allowing a running system to be updated on the fly with new events, which is important for the ubiquitous computing applications. Noninvasive handler extensibility is conceptually possible using glue methods in RMJ, as described in section 3.5, but the current RMJ implementation does not allow glue methods for internal generic functions because of challenges for modular compilation.

Second, the need for default methods limits MultiJava's ability to perform useful exhaustiveness checking. It is impossible for a component to document the fact that it handles

```
abstract class Visitor {
  abstract void apply(Shape s);
}

class AreaVisitor extends Visitor {
  int area;
  void apply(Shape s) {
    if (s instanceof Rectangle) {
      Rectangle r = (Rectangle) s;
      area = r.length() * r.width();
    } else if (s instanceof Circle) {
      Circle c = (Circle) s;
      ...
    }
  }
}
```

Figure 5.5: A noninvasive version of the visitor pattern.

exactly three kinds of events, and no others. Instead, it must always include a default method, to handle any unexpected events. Some programmers found it useful for the language to force them to think about exceptional situations, but others thought it more of a nuisance. Again, RMJ would allow defaults to be omitted, with additional load-time checking for safety.

5.1.3 Noninvasive Visitors

Figure 1.2 illustrated the standard form of the visitor design pattern [43], which allows new methods to be added to existing classes. A variant of that pattern uses type tests and casts instead of double dispatching, obviating the need for "hooks" inside the original classes. The translation of figure 1.2's visitors to this "noninvasive visitor pattern" is shown in figure 5.5. A `Shape` variable `s` has its area computed in the new style as follows:

```
new AreaVisitor().apply(s)
```

MultiJava allows the noninvasive visitor pattern to be implemented in a declarative and statically typed manner. `AreaVisitor` would instead have several `apply` multimethods, each dispatching on a different subclass of `Shape`. This use has obvious similarities to the

event dispatching scenario and has the same benefits. The HydroJ compiler is built on top of the Polyglot extensible compiler framework [83], which uses a form of noninvasive visitors over the hierarchy representing abstract syntax tree (AST) nodes. HydroJ provides new subclasses of Polyglot's visitors, using multimethods instead of type tests and casts to implement the node dispatch. A comparison of noninvasive visitors with open classes is presented in section 5.2.3 below.

One limitation of MultiJava that arises in the HydroJ compiler is the requirement that specializers be classes. Polyglot provides its AST nodes as a hierarchy of interfaces, with the intent that the associated implementation classes should remain hidden from clients. The HydroJ compiler must break this abstraction boundary and dispatch directly on the implementation classes. The Java style does not suffer from this problem, because `instanceof` tests are allowed on interfaces. The current implementation of RMJ allows interfaces to be specializers and includes load-time ambiguity checking for safety, providing one solution to MultiJava's limitation.

5.1.4 Finite-state Machines

A common way to implement a finite-state machine (FSM) in Java is to associate a constant with each state. The FSM's class has a field recording the current state, and a method in the class implements the FSM's transition function: based on the given input and the current state, the method performs some actions and transitions to a new state. MultiJava's value dispatching provides a declarative way to implement this transition function.

As a simple example, figure 5.6 implements an FSM that keeps track of the number of consecutive alternations of 0 and 1 that have been input. There are two states, which respectively track whether a 0 or 1 is expected as input. The transition function has three transitions, each nicely encapsulated in its own multimethod. For example, the first `transition` method "fires" when the input is 0 and the FSM is in the EXPECT_ZERO state. In that case, the FSM moves to the EXPECT_ONE state. The method uses the ability to put any compile-time constant expression can appear after a `@@`. The last `transition` method is the local default required by MultiJava to ensure exhaustiveness. It handles the

```
class ZeroOneFSM {
  static final int EXPECT_ZERO = 0;
  static final int EXPECT_ONE = 1;
  int currState = EXPECT_ZERO;
  int numOccurrences = 0;
  void readAndTransition(int input) {
    transition(input, currState);
  }
  void transition(int@@0 input, int@@EXPECT_ZERO state) {
    currState = EXPECT_ONE;
  }
  void transition(int@@1 input, int@@EXPECT_ONE state) {
    currState = EXPECT_ZERO;
    numOccurrences++;
  }
  void transition(int input, int state) {
    currState = EXPECT_ZERO;
    numOccurrences = 0;
  }
}
```

Figure 5.6: Implementing finite-state machines in MultiJava.

case when unexpected data is input (or an unexpected state is reached), in which case the FSM resets.

The Location Stack uses this style to implement the FSMs that parse readings from the various location sensors. The developer reports that it is much easier to understand the behavior of an FSM written in this way, versus the typical Java style. The MultiJava version also enjoys all the benefits described earlier for multimethods. For example, the FSM is easily extensible by subclasses, which can add new transitions and optionally override existing ones.

5.2 Open Classes

The Hydro and HydroJ compilers exploit open classes for several purposes, which are described in this section. I end the section by describing some potential uses of open classes that have yet to be explored by users.

```
interface Procedure {
  void apply(Object o);
}
void Iterator.doEach(Procedure p) {
  while (this.hasNext()) {
    p.apply(this.next());
  }
}

// sample client code
Iterator i = ...
i.doEach(new Procedure() {
           void apply(Object o) { ... } } );
```

Figure 5.7: Adding closure-based iteration to Java.

5.2.1 Unavailable Source

A common use of open classes has been to augment classes whose source is not available (or not easily modified). The Hydro and HydroJ compilers add several methods to classes in the Java standard library. For example, a method for removing whitespace from a string is defined as follows:

```
public String String.deleteWhitespace() { ... }
```

Clients can import the new generic function and then invoke it as if it were part of the original functionality of strings. In Java, this idiom is typically simulated by a static method in a dummy class, which is a bit more tedious and has a different invocation syntax from the original methods of `String`.

A more interesting example is illustrated in figure 5.7, which is a variant of code from the Hydro compiler. The `doEach` method augments `java.util.Iterator` with closure-based iteration. The closure is defined as a (typically anonymous) class that implements the `Procedure` interface. A side benefit of open classes illustrated in this example is the ability to add methods to interfaces (like `Iterator`).

The Hydro compiler uses the SableCC parser generator [95], which builds the AST node hierarchy automatically from a description of Hydro's grammar. Modifying the resulting AST classes is undesirable, because any changes will be lost the next time SableCC is run.

The nodes generated by SableCC provide a visitor-like framework so clients can implement external traversals over the AST hierarchy, and the Hydro developer used these visitors to implement the major passes in the compiler. However, he preferred using open classes for functionality that is not meant to traverse the entire AST hierarchy. For example, Hydro includes a rich language for pattern matching, and open classes make it easy to add new behavior to the pattern nodes. Using the visitor infrastructure would require that the external operations for patterns actually be able to handle an arbitrary node, which is more tedious and loses some static type safety.

A limitation of open classes is the lack of privileged access to the receiver class. To use open classes successfully, the public functionality of the receiver must be rich enough to allow clients to implement unanticipated behaviors. For example, `String` has methods that provide access to each character, and this is enough to allow whitespace to be removed by clients, as shown in the first example above. This limitation is also shared by the Java workarounds for open classes.

5.2.2 Client-specific Extensions

It can make sense to make an operation external even if the source code for its receiver class is available. One such scenario is when the new functionality is client-specific rather than general-purpose. With open classes, the new functionality can be implemented without polluting the view of the original receiver class as seen by other clients. In general, each client can have his own library of extensions to an existing class hierarchy. The HydroJ compiler implements client-specific operations in this way. For example, the compiler includes a method for deep copying a list of AST nodes. Although in this case the source code of the receiver (`java.util.List`) is actually not available, the developer reports that he would use external methods for this operation even if he had source-code access to `List`. Indeed, an AST-specific operation does not belong in the view of `List` as seen by all clients.

```
// file print.java
public String TreePatternNode.print() { ... }
public String ListPatternNode.print() { ... }
public String StarPatternNode.print() { ... }
...
```

Figure 5.8: Structuring code by algorithm with open classes.

5.2.3 Flexible Code Structuring

Open classes also allow code decompositions other than by receiver class. It is sometimes useful to encapsulate an entire generic function's methods as a unit, rather than spreading the methods across the various receiver classes. This can be especially helpful when the methods implement a single conceptual algorithm. As an example, the code for printing AST nodes in the HydroJ compiler is implemented as an external generic function, a portion of which is illustrated in figure 5.8. The developer felt that this decomposition was more natural than the by-class view. In addition, the HydroJ compiler contains a few different algorithms for printing AST nodes. Each is implemented as an external generic function, and clients import the one appropriate to their needs.

Operations like `print` are natural candidates for use of the visitor pattern in a language lacking open classes. It is interesting to compare open classes with the noninvasive visitor pattern described in section 5.1.3, which resolves some problems of the original visitor pattern. Each style has advantages over the other. Open classes allow clients to invoke the `print` operation using the same syntax that regular methods of the AST classes are invoked with. Further, open classes are extensible while the visitor pattern is not. With open classes, a new `Node` subclass can be given an overriding `print` method as a regular method inside the class. With the visitor pattern, the only option is to modify the visitor class in place to handle the new kind of node.

On the other hand, the noninvasive visitor pattern provides the advantages of classes. Helper fields can easily be included in the visitor class, while these must be simulated through extra parameters in external methods. More importantly, visitors can inherit functionality from superclasses, analogous to the inheritance of event handlers described earlier.

For example, the Polyglot compiler framework underpinning HydroJ provides an abstract `HaltingVisitor` class, which performs a boilerplate traversal over AST nodes that also supports bypassing certain nodes during the traversal. Concrete visitors that require the functionality to bypass nodes simply subclass `HaltingVisitor` and provide overriding methods to customize its behavior as necessary. In contrast, each external generic function must implement its own traversal behavior from scratch.

5.2.4 Potential Future Uses

There are a few ways in which external methods may be profitably used that have not yet been explored.

Application Versioning

The uses of open classes described above are mainly small extensions to an existing hierarchy. The open-class idiom also has the potential to allow for statically typed modular versioning of entire applications. A canonical benchmark in the literature is the idea of a modular interpreter (e.g., Liang et al. [66]). An original interpreter is later augmented from the outside to handle both extensions to the language being interpreted and new functionality (e.g., the addition of a pretty-print pass). As an example, suppose that a Java compiler were built in MultiJava, with an external generic function for each compiler pass. A HydroJ compiler could then be built as a noninvasive extension of this compiler. New AST nodes would be introduced for each of HydroJ's new language constructs, and each of these nodes would include an overriding method for each existing compiler pass, as a regular internal method.[1] This contrasts with the use of the visitor pattern, which would require that all existing visitors be modified in place to contain the new overriding methods. MultiJava also allows new passes like pretty printing to be implemented noninvasively over the entire AST hierarchy as external generic functions, as long as there are appropriate default methods to handle any unavailable AST nodes.

---

[1] In RMJ, glue methods could be used to instead put all the overriding methods pertaining to a particular pass together in a single file.

As another example of modular versioning, consider the event-based style described in section 5.1.2. In this style, components register to be notified about events, and notification occurs via their `handleEvent` methods. Suppose one wishes to noninvasively augment an event-based system to allow components to also specify priorities among events, which the system will use to determine the order in which events should be announced. Subclassing can be used to create a new version of the system that makes use of priorities. However, subclassing the base `Component` class to contain a new method for specifying priorities is not desirable, because then existing components would not be accepted by the new version of the system. In MultiJava, one can instead create an external method for the `Component` class, providing a default event priority function. Existing components will continue to function properly in the new version of the event-based system, while new clients that enter the system can make use of the ability to specify priorities.

Application Design with Open Classes

A broad area for future study is the integration of open classes into traditional program development. Most of the examples in this chapter use open classes for after-the-fact augmentation of an existing class. However, open classes can also potentially impact the ways in which a programmer thinks about structuring classes in the first place. For example, the original implementer of a class can explicitly create multiple versions of the class. Functionality that all clients will likely require would be placed inside the class as usual, while secondary functionality would be written as external operations. Further, these external operations would be structured in several different packages, based on what "concern" of the original class they affect.

This kind of structuring of classes could be quite beneficial, supporting a form of role-based programming [4, 102, 97]. Clients could mix and match among the various extensions to the original class, based on the needs of a particular application. Further, a client could easily "swap in" his own version of some external operation, if the existing one were not suitable. This scheme would also allow classes to be more easily adaptable to different environments, which could be useful in the context of ubiquitous computing. For example,

a client could use the "full" version of a class when running an application on a desktop machine but switch to the "thin" version containing only the bare-bones functionality when moving to a handheld computer.

Making this vision a reality will require a better integration of external methods into traditional OO programming language methodology. In a language like Java, classes effectively play the role of modules (among other roles), structuring and encapsulating sets of method, field, and other class declarations. External methods live outside of this dominant class structure and consequently can feel strange to Java programmers. Many MultiJava programmers who effectively use multiple dispatch in their programs report in contrast that they do not know the right way to think about using external methods.

More experience with open classes is necessary in order to better understand how to naturally incorporate them into the OO programming style. It is possible that further language extensions would help to better integrate external methods with Java. For example, if Java had a richer module system than its packages, like Jiazzi [71], then it might be natural for programmers to think of modules, rather than classes, as the unit of structure and encapsulation. Such a module system could make Java feel more like ML, in which functions are naturally external to their datatypes, and both constructs are encapsulated in modules. As another example, the incorporation of retroactive abstraction could provide the necessary complement to open classes that provides the impetus for users to experiment more broadly with external methods.

Chapter 6

CONCLUSIONS

In this dissertation, I have illustrated one approach to resolving some well-known conflicts between component extensibility and component reasoning in programming languages. I have naturally integrated the OO and functional styles of extensibility while preserving fully modular typechecking. The Dubious calculus serves as a simple but expressive platform for formal study. MultiJava and Eml illustrate that Dubious can serve as the theoretical foundation for practical extensions to mainstream OO and functional languages.

This dissertation is a step in my broader research agenda to incorporate expressive programming idioms into mainstream programming languages. The last few years have seen an explosion of proposals for augmenting the extensibility of mainstream languages, particularly OO languages. Many of these proposals fall in the category of aspect-oriented languages, aiming to provide better support for separation of concerns. While these proposals have the potential to greatly increase the expressiveness of software components, to date there has been little work in modular reasoning for the new forms of extensibility, providing a large research challenge. My experience with Dubious, MultiJava, and Eml has taught me many valuable lessons about programming language design and evaluation, and more generally about the nature of problem solving. I end this dissertation by describing these lessons. Hopefully they will prove useful in addressing modular typechecking for aspect-oriented languages, and in my future research in general.

- **Simplify.** My initial attempts to solve the modularity problem for open classes and multimethods were performed in the context of BeCecil [27], described in section 2.4. BeCecil includes some fairly sophisticated constructs, including mutable state, information hiding, block structure, and subtyping independent of subclassing. These features make BeCecil more "real" than Dubious, but they only serve to obscure and complicate the underlying modularity problem, and I found it difficult to make

progress.

Frustration with BeCecil as a research platform led me to design Dubious, with the explicit intent that it contain the minimal set of constructs needed to illustrate the modularity problem for open classes and multimethods. This design allowed me to focus on the problem's essence and understand it well enough to find a reasonable solution. Further, because System M isolates and solves the fundamental modularity problem, it later proved quite robust in the face of the many constructs missing from the Dubious formalism, as illustrated by MultiJava and Eml.

- **Follow Peres' Law.** A recent technical report [88] dubs the following quote from Shimon Peres as Peres' Law: "If a problem has no solution, it may not be a problem, but a fact, not to be solved, but to be coped with over time." System M does not really solve the longstanding modularity problem for multimethods and open classes: it is still the case that arbitrary use of these constructs conflicts with modular typechecking. The insight illustrated by System M is that, although the general problem may be intractable, many of the extensibility idioms commonly desired in practice can be reconciled with modular typechecking. I believe this approach, which follows Peres' Law, is a useful one for many aspects of programming language design, and indeed for computer science research in general, where tradeoffs are plentiful and fundamental. Despite (or maybe because of) the inability to completely resolve a tradeoff, important progress can be made by accepting the tradeoff as a given and letting practical considerations guide exploration of the design space.

- **Theory matters in practice.** Formalizing System M and proving its soundness were much more than mere theoretical exercises. The formalization served to define notions that are otherwise open to interpretation. For example, the notion of modular typechecking itself, which is fundamental to this research, is quite slippery and has different meanings to different people. Formalizing System M allowed me to be precise in my use of that term. The formalization was similarly valuable in making System M's requirements, which were originally based on intuition, precise.

The soundness proof validates the formalization's correctness, thereby also validating the original intuition. The proof also helped to debug the formalization and to identify parts of the type system that are unnecessary or overly restrictive. As an example, the formalization of ITC for Mini-Eml originally included explicit exhaustiveness checking: in each structure, each available function must have at least one available applicable function case for each available legal argument tuple. To my surprise, the soundness proof did not depend upon this check at all; exhaustiveness could be proven solely from the local- and global-default requirements. This realization shed new light on System M's correctness. It also allowed me to simplify both the formalization of ITC and its implementation in Eml's interpreter.

- **Users are critical.** Formalisms and soundness proofs can give confidence that a language does not allow "bad" things to happen, but user experience is necessary to provide evidence that the language allows the desired "good" things to happen. Feedback from MultiJava users has been invaluable in helping to understand the strengths and weaknesses of the underlying theory. Often aspects of the language that I considered relatively unimportant were found quite useful, and vice versa. For example, splitting up an event handler into multiple methods encouraged better documentation in the form of comments, as described in section 5.1.2, and this is an advantage that I had not previously considered. Weaknesses identified by users are obvious candidates for future research. For example, user feedback led to the design of value dispatching and was a motivation for the design of Relaxed MultiJava. More broadly, open classes have not been as widely used as expected, and user feedback is critical to understanding the deficiencies and how they can be corrected.

  The strategy of extending mainstream languages was crucial in attracting users to evaluate Dubious's System M. In particular, the fact that MultiJava is backward-compatible with Java greatly reduced both the risk and the burden on potential users. Existing Java programmers were able to incrementally migrate their applications from Java to MultiJava, as they became more comfortable with MultiJava's features. Had I designed a new source-level language from scratch, it is unlikely I would have been

nearly as successful in attracting users.

- **Where they diverge, utility trumps purity.** Although languages are interesting objects of study in their own right, their value ultimately lies in how useful they are to programmers. This fact sometimes requires that a bit of elegance be sacrificed for the sake of pragmatism. This is especially true when designing an extension to an existing language, because the extender must adhere to the stylistic constraints imposed by the original designer. For example, initially it seemed inexcusable to me for MultiJava's multimethods to live inside of their receivers, because this design uses an asymmetric (i.e., receiver-based) syntax but a symmetric dispatching semantics. A more uniform alternative considered by colleagues and myself is to use a style based on Tuple [63], in which all multimethods are external and have a symmetric syntax. However, as discussed in section 3.4, this style does not integrate nearly as well with existing Java programs. Aside from being less expressive than MultiJava, Tuple also deviates from the "feel" of Java, imposing an extra cognitive burden on programmers. While Tuple is closer to the "right" syntax for symmetric multimethods, MultiJava's current style is decidedly more practical in the context of Java.

BIBLIOGRAPHY

[1] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. Static Type Checking of Multi-Methods. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 113–128, November 1991.

[2] Davide Ancona, Giovanni Lagorio, and Elena Zucca. A formal framework for Java separate compilation. In *Proceedings of the 2002 European Conference on Object-Oriented Programming*, LNCS 2374, Malaga, Spain, June 2002. Springer-Verlag.

[3] Davide Ancona and Elena Zucca. True modules for Java-like languages. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001–Object-Oriented Programming*, LNCS 2072. Springer, 2001.

[4] Egil P. Andersen and Trygve Reenskaug. System design by composing structures of interacting objects. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 133–152. Springer-Verlag, June 1992.

[5] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language Third Edition*. Addison-Wesley, Reading, MA, third edition, 2000.

[6] Larry Arnstein, Chia-Yang Hung, Robert Franza, Qing Hong Zhou, Gaetano Borriello, Sunny Consolvo, and Jing Su. Labscape: A smart environment for the cell biology laboratory. *IEEE Pervasive Computing*, 1(3):13–21, July 2002.

[7] Gerald Baumgartner, Martin Jansche, and Konstantin Läufer. Half & half: Multiple dispatch and retroactive abstraction for java. Technical Report OSU-CISRC-5/01-TR08, Department of Computer and Information Science, The Ohio State University, March 2002.

[8] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Commonloops: merging lisp and object-oriented programming. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 17–29. ACM Press, 1986.

[9] Daniel Bonniot. Type-checking multi-methods in ML (a modular approach). In *The Ninth International Workshop on Foundations of Object-Oriented Languages, FOOL 9*, Portland, Oregon, USA, January 2002.

[10] François Bourdoncle and Stephan Merz. Type-checking higher-order polymorphic multi-methods. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 302–315, Paris, France, 15–17 January 1997.

[11] John Boyland and Giuseppe Castagna. Parasitic methods: Implementation of multi-methods for Java. In *Conference Proceedings of OOPSLA '97, Atlanta*, volume 32(10) of *ACM SIGPLAN Notices*, pages 66–76. ACM, October 1997.

[12] Gilad Bracha and William Cook. Mixin-based inheritance. In *ECOOP/OOPSLA '90*, pages 303–311, 1990.

[13] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. eXtensible Markup Language (XML) 1.0 (Second Edition). World Wide Web Consortium recommendation, `http://www.w3.org/TR/REC-xml`, 2000.

[14] Seth Bridges, Miguel Figueroa, David Hsu, and Chris Diorio. Field-programmable learning arrays. *Advances in Neural Information Processing Systems 15*, 2003.

[15] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):217–238, 1995.

[16] C# Language Specification, Second Edition. ECMA International, Standard ECMA-334, December 2002.

[17] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February 1988.

[18] Luca Cardelli. Program fragments, linking, and modularization. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277, Paris, France, 15–17 January 1997.

[19] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[20] Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkhauser, Boston, 1997.

[21] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, February 1995.

[22] Guiseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, March 1995.

[23] Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *ECOOP '92: European Conference on Object-Oriented Programming*, LNCS 615, pages 33–56. Springer-Verlag, 1992.

[24] Craig Chambers. Object-oriented multi-methods in cecil. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 33–56. Springer-Verlag, June 1992.

[25] Craig Chambers. The Cecil language specification and rationale: Version 2.1. `http://www.cs.washington.edu/research/projects/cecil/www/pubs/cecil-spec.html`, March 1997.

[26] Craig Chambers and Gary T. Leavens. Typechecking and modules for multimethods. *ACM Transactions on Programming Languages and Systems*, 17(6):805–843, November 1995.

[27] Craig Chambers and Gary T. Leavens. BeCecil, a core object-oriented language with block structure and multimethods: Semantics and typing. In *4th Workshop on Foundations of Object-Oriented Languages*, January 1997.

[28] Curtis Clifton. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical Report 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, November 2001.

[29] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10) of *ACM SIGPLAN Notices*, pages 130–145, October 2000.

[30] William R. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, LNCS 489, pages 151–178. Springer-Verlag, New York, NY, 1991.

[31] Jeffrey Dean, Greg DeFouw, Dave Grove, Vassily Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of the 1996 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 83–100, San Jose, CA, October 1996.

[32] Sophia Drossopoulou, Susan Eisenbach, and David Wragg. A fragment calculus —

towards a model of separate compilation, linking and binary compatibility. In *Logic in Computer Science*, pages 147–156, 1999.

[33] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 262–273, Philadelphia, Pennsylvania, May 1996.

[34] Dominic Duggan and Ching-Ching Techaubol. Modular mixin-based inheritance for application frameworks. In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, pages 223–240. ACM Press, 2001.

[35] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In Eric Jul, editor, *ECOOP '98–Object-Oriented Programming*, LNCS 1445, pages 186–211. Springer, 1998.

[36] Neal Feinberg, Sonya E. Keene, Robert O. Mathews, and P. Tucker Withington. *The Dylan Programming Book*. Addison-Wesley Longman, Reading, Mass., 1997.

[37] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 94–104. ACM, June 1998.

[38] Kathleen Fisher and John Reppy. The design of a class mechanism for MOBY. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 37–49, Atlanta, Georgia, May 1–4, 1999.

[39] Kathleen Fisher and John Reppy. Extending Moby with inheritance-based subtyping. In *14th European Conference on Object-Oriented Programming*, LNCS 1850, pages 83–107, June 2000.

[40] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 236–248, Montreal, Canada, 17–19 June 1998.

[41] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.

[42] You-Chin C. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(2):155–175, June 1990.

[43] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.

[44] J. Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, November 2000.

[45] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[46] Robert Grimm, Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson, Steven Gribble, Tom Anderson, Brian Bershad, Gaetano Borriello, and David Wetherall. Programming for pervasive computing environments. Technical Report UW-CSE-01-06-01, Department of Computer Science and Engineering, University of Washington, June 2001.

[47] Christian Grothoff. Walkabout revisited: The runabout. In *Proceedings of the 2003 European Conference on Object-Oriented Programming*, LNCS 2743, Darmstadt, Germany, July 2003. Springer-Verlag.

[48] William Harrison and Harold Ossher. Subject-oriented programming (A critique of pure objects). In Andreas Paepcke, editor, *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428. ACM Press, October 1993.

[49] Jeffrey Hightower, Barry Brumitt, and Gaetano Borriello. The location stack: A layered model for location in ubiquitous computing. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems & Applications (WMCSA 2002)*, pages 22–28, Callicoon, NY, June 2002. IEEE Computer Society Press.

[50] My Hoang and John C. Mitchell. Lower bounds on type inference with subtypes. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.*, pages 176–185, New York, NY, January 1995. ACM.

[51] Urs Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings of the 1993 European Conference on Object-Oriented Programming*, LNCS 707, pages 36–56, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[52] Haruo Hosoya and Benjamin Pierce. Regular expression pattern matching for xml. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 67–80. ACM Press, 2001.

[53] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[54] Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 347–349. ACM Press, 1986.

[55] Java 2 Platform, Standard Edition, version 1.4.2 API Specification. `http://java.sun.com/j2se/1.4.2/docs/api/index.html`.

[56] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[57] Ralph Keller and Urs Hölzle. Binary component adaptation. In Eric Jul, editor, *ECOOP '98—Object-Oriented Programming*, LNCS 1445, pages 307–329. Springer, 1998.

[58] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 2001 European Conference on Object-Oriented Programming*, LNCS 2072, Budapest, Hungary, June 2001. Springer-Verlag.

[59] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, LNCS 1241, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[60] Kopi project home page. `http://www.dms.at/kopi`.

[61] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In Eric Jul, editor, *ECOOP'98–Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, LNCS 1445, pages 91–113. Springer-Verlag, July 1998.

[62] Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh. An exemplar based smalltalk. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 322–330. ACM Press, 1986.

[63] Gary T. Leavens and Todd D. Millstein. Multiple dispatch as dispatch on tuples. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 374–387, October 1998.

[64] Keunwoo Lee, Anthony LaMarca, and Craig Chambers. Hydroj: Object-oriented pattern matching for evolvable distributed systems. In *Proceedings of the 2003 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Anaheim, CA, October 2003.

[65] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the 1998 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 36–44, Vancouver, Canada, October 1998.

[66] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: San Francisco, California, January 22–25, 1995*, pages 333–343, New York, NY, USA, 1995. ACM Press.

[67] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 214–223. ACM Press, 1986.

[68] Vassily Litvinov. Constraint-based polymorphism in cecil: Towards a practical and static type system. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, October 1998.

[69] David MacQueen. Modules for standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 198–207. ACM, ACM, August 1984.

[70] Robert Martin. Acyclic visitor. In *Pattern Languages of Program Design 3*, pages 93–104. Addison-Wesley, 1998.

[71] Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. Jiazzi: new-age components for old-fashioned java. In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, pages 211–222. ACM Press, 2001.

[72] Todd Millstein, Colin Bleckner, and Craig Chambers. Modular typechecking for hierarchically extensible datatypes and functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*, volume 37(9) of *ACM SIGPLAN Notices*, pages 110–122, New York, NY, September 2002. ACM.

[73] Todd Millstein and Craig Chambers. Modular statically typed multimethods. In Rachid Guerraoui, editor, *ECOOP '99 – Object-Oriented Programming 13th European Conference, Lisbon Portugal*, LNCS 1628, pages 279–303. Springer-Verlag, New York, NY, June 1999.

[74] Todd Millstein and Craig Chambers. Modular statically typed multimethods. *Information and Computation*, 175(1):76–118, May 2002.

[75] Todd Millstein, Mark Reay, and Craig Chambers. Relaxed MultiJava: Balancing extensibility and modular typechecking. In *Proceedings of the 2003 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Anaheim, CA, October 2003.

[76] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[77] David A. Moon. Object-oriented programming with flavors. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–8. ACM Press, 1986.

[78] Warwick B. Mugridge, John Hamer, and John G. Hosking. Multi-methods in a statically-typed programming language. In Pierre America, editor, *ECOOP '91: European Conference on Object-Oriented Programming*, LNCS 512, pages 307–324. Springer-Verlag, 1991.

[79] MultiJava home page. `http://multijava.sourceforge.net`.

[80] Greg Nelson. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[81] Martin E. Nordberg. Default and extrinsic visitor. In *Pattern Languages of Program Design 3*, pages 105–123. Addison-Wesley, 1998.

[82] Johan Nordlander. Pragmatic subtyping in polymorphic languages. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 216–227, 1999.

[83] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *Proceedings of CC 2003: 12'th International Conference on Compiler Construction*. Springer-Verlag, April 2003.

[84] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, 15–17 January 1997.

[85] Harold Ossher and Peri Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *Proceedings of the 22nd International Conference on Software Engineering, June 4-11, 2000, Limerick, Ireland*, pages 734–737, 2000.

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

[87] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC*, pages 9–15, 19–21 August 1998.

[88] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB//CSD-02-1175, Division of Computer Science, UC Berkeley, March 2002.

[89] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000.

[90] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension of ML. *Theory and Practice of Object Systems*, 4(1):27–52, 1998.

[91] John Reppy and Jon Riecke. Simple objects for Standard ML. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 171–180, Philadelphia, Pennsylvania, 21–24 May 1996.

[92] John H. Reppy and Jon G. Riecke. Classes in object ML via modules. In *Third Workshop on Foundations of Object-Oriented Languages (FOOL)*, July 1996.

[93] John. C. Reynolds. User-defined types and procedural data structures as complementary approaches to type abstraction. In S. A. Schuman, editor, *New Directions in Algorithmic Languages*, pages 157–168. IRIA, Rocquencourt, 1975.

[94] John C. Reynolds. Using category theory to design implicit conversions and generic operators. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark*, LNCS 94, pages 211–258. Springer-Verlag, January 1980.

[95] SableCC home page. `http://www.sablecc.org`.

[96] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass., 1997.

[97] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In Eric Jul, editor, *ECOOP '98–Object-Oriented Programming*, LNCS 1445, pages 550–570. Springer, 1998.

[98] Guy L. Steele Jr. *Common Lisp: The Language, Second Edition*. Digital Press, Bedford (MA), USA, 1990.

[99] Bjarne Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley, Reading, Mass., 1997.

[100] Clemens Szyperski, Domiinik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, second edition edition, 2002.

[101] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, 1987.

[102] Michael VanHilst and David Notkin. Using C++ templates to implement role-based designs. In *Proceedings of 2nd International Symposium on Object Technologies for Advanced Software*, March 1996.

[103] John Vlissides. Visitor in frameworks. *C++ Report*, 11(10):40–46, November/December 1999.

[104] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, April 1990.

[105] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.

[106] Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*. ACM, September 3-5 2001.

Appendix A

TYPE SOUNDNESS FOR MINI-EML

This appendix provides the complete proof of type soundness for Mini-Eml. As discussed in section 4.3.4, progress and type preservation theorems are proven. The first section below contains a proof of the progress theorem and the key technical lemmas that it depends upon. The second section does the same for the type preservation theorem. Both theorems also rely on several basic facts about Mini-Eml, which are proven in the final section.

As in the formal dynamic and static semantics, the proof assumes a fixed structure table ST satisfying the two sanity conditions given in section 4.3 and a fixed signature table $SigT$ containing the principal signatures of the structures in ST. Also as in the formal rules, it is assumed that S OK holds for each structure S in ST.

A.1 Progress

The progress theorem is straightforward except for the case when E is a function application. That case follows easily from lemma 1, which is given below.

Theorem 1 (Progress) If $\vdash E : \tau$ and E is not a value, then there exists an E' such that $E \longrightarrow E'$.

Proof By (strong) induction on the depth of the derivation of $\vdash E : \tau$. Case analysis of the last rule used in the derivation.

- Case T-Id. Then $E = I$ and $(I, \tau) \in \{\}$, so we have a contradiction. Therefore this rule could not be the last rule used in the derivation.

- Case T-New. Then $E = Ct(\overline{E})$ and $Ct = (\overline{\tau}\ Sn.Cn)$ and $\{\}; \bullet \vdash Ct(\overline{E})$ OK and concrete($Sn.Cn$). Therefore by lemma 37, there exist $\overline{V_1}$ and $\overline{E_1}$ such that rep($Ct(\overline{E})$) $= \{\overline{V_1} = \overline{E_1}\}$. Then by E-New we have $E \longrightarrow Ct\{\overline{V_1} = \overline{E_1}\}$.

- Case T-Rep. Then $E = Ct\{V_1 = E_1, \ldots, V_k = E_k\}$ and for all $1 \leq i \leq k$ we have $\vdash E_i : \tau_i$ for some τ_i. We have two subcases:
  - For all $1 \leq i \leq k$, E_i is a value. Then E is a value, contradicting our assumption.
  - There exists some j such that $1 \leq j \leq k$ and E_j is not a value. Without loss of generality, let j be the smallest integer satisfying this condition, so for all $1 \leq q < j$ we have that E_q is a value. By induction, there exists an E'_j such that $E_j \longrightarrow E'_j$. Therefore by E-Rep we have $Ct\{V_1 = E_1, \ldots, V_k = E_k\} \longrightarrow Ct\{V_1 = E_1, \ldots, V_{j-1} = E_{j-1}, V_j = E'_j, V_{j+1} = E_{j+1}, \ldots, V_k = E_k\}$.
- Case T-Fun. Then $E = \overline{\tau}\ Sn.Fn$. Then E is a value, contradicting our assumption.
- Case T-Tup. Then $E = (E_1, \ldots, E_k)$ and $\tau = \tau_1 * \cdots * \tau_k$ and for all $1 \leq i \leq k$ we have $\vdash E_i : \tau_i$. We have two subcases:
  - For all $1 \leq i \leq k$, E_i is a value. Then E is a value, contradicting our assumption.
  - There exists some j such that $1 \leq j \leq k$ and E_j is not a value. Without loss of generality, let j be the smallest integer satisfying this condition, so for all $1 \leq q < j$ we have that E_q is a value. By induction, there exists an E'_j such that $E_j \longrightarrow E'_j$. Therefore by E-Tup we have $(E_1, \ldots, E_k) \longrightarrow (E_1, \ldots, E_{j-1}, E'_j, E_{j+1}, \ldots, E_k)$.
- Case T-App. Then $E = E_1\ E_2$ and $\vdash E_1 : \tau_2 \to \tau$ and $\vdash E_2 : \tau'_2$ and $\tau'_2 \leq \tau_2$. We have three subcases:
  - E_1 is not a value. Then by induction, there exists an E'_1 such that $E_1 \longrightarrow E'_1$. Therefore by E-App1 we have $E_1\ E_2 \longrightarrow E'_1\ E_2$.
  - E_1 is a value, but E_2 is not a value. Then by induction, there exists an E'_2 such that $E_2 \longrightarrow E'_2$. Therefore by E-App2 we have $E_1\ E_2 \longrightarrow E_1\ E'_2$.
  - Both E_1 and E_2 are values. Since $\vdash E_1 : \tau_2 \to \tau$ and E_1 is a value, the last rule in the derivation of $\vdash E_1 : \tau_2 \to \tau$ must be T-Fun, so E_1 has the form

Fv. Therefore by lemma 1 we have that there exist ρ_0 and E_0 such that most-specific-case-for $(Fv,E_2) = (\rho_0, E_0)$. Let $\rho_0 = \{(\overline{I}, \overline{v})\}$. Then by E-AppRed we have $Fv\ E_2 \longrightarrow [\overline{I} \mapsto \overline{v}]E_0$.

□

This lemma says that a most-specific applicable function case exists for each type-correct function application. It follows easily from lemmas 2 and 7, which say that all functions are exhaustive and unambiguous, respectively.

Lemma 1 If $\vdash Fv : \tau_1 \rightarrow \tau_2$ and $\vdash v : \tau_1'$ and $\tau_1' \leq \tau_1$ then there exist ρ and E such that most-specific-case-for $(Fv,v) = (\rho, E)$.

Proof Let $Fv = (\overline{\tau}\ F)$. By lemma 2, there exists some $Sn \in \mathrm{dom}(SigT)$, some $(\texttt{extend fun}_{Mn}\ \overline{\alpha}\ F\ Pat) \in SigT(Sn)$, and some environment ρ_0 such that $\mathrm{match}(v, Pat) = \rho_0$. Then by lemma 7 there exists some $Sn' \in \mathrm{dom}(SigT)$, some $(\texttt{extend fun}_{Mn'}\ \overline{\alpha_1}\ F\ Pat') \in SigT(Sn')$, and some ρ' such that $\mathrm{match}(v, Pat') = \rho'$ and $\forall Sn'' \in \mathrm{dom}(SigT).\forall(\texttt{extend fun}_{Mn''}\ \overline{\alpha_2}\ F\ Pat'') \in SigT(Sn'').\forall\rho''.\ ((\mathrm{match}(v, Pat'') = \rho'' \wedge Sn'.Mn' \neq Sn''.Mn'') \Rightarrow Pat' < Pat'')$.

By the definition of $SigT$ we have that $Sn' \in \mathrm{dom}(ST)$ and there exists some E' such that $(\texttt{extend fun}_{Mn'}\ \overline{\alpha_1}\ F\ Pat') \in ST(Sn')$ and $\forall Sn'' \in \mathrm{dom}(ST).\forall(\texttt{extend fun}_{Mn''}\ \overline{\alpha_2}\ F\ Pat'' = E'') \in ST(Sn'').\forall\rho''.\ ((\mathrm{match}(v, Pat'') = \rho'' \wedge Sn'.Mn' \neq Sn''.Mn'') \Rightarrow Pat' < Pat'')$.

Since $\vdash Fv : \tau_1 \rightarrow \tau_2$, by T-Fun we have $F = Sn_0.Fn_0$ and $(\texttt{fun}\ \overline{\alpha_0}\ Fn_0 : Mt_0 \rightarrow \tau_0) \in SigT(Sn_0)$ and $|\overline{\alpha_0}| = |\overline{\tau}|$. Since $(\texttt{extend fun}_{Mn'}\ \overline{\alpha_1}\ F\ Pat') \in SigT(Sn')$, by CaseOK we have $|\overline{\alpha_1}| = |\overline{\alpha_0}|$. Therefore we have $|\overline{\alpha_1}| = |\overline{\tau}|$. Then by Lookup there exist ρ and E such that most-specific-case-for $((\overline{\tau}\ F),v) = (\rho, E)$. □

A.1.1 Exhaustiveness

These lemmas prove that all functions are exhaustive. They make use of the notion of the *owner* of a value v with respect to a marked type *Mt*. Intuitively, v's owner is the class

in v located at the owner position as specified by Mt. It is defined via the following two inference rules:

$\boxed{\text{owner}(Mt, v) = C}$

$$\frac{\text{owner}(Mt, v_i) = C}{\text{owner}(\tau_1 * \cdots * \tau_{i-1} * Mt * \tau_{i+1} * \cdots * \tau_k, (v_1, \ldots, v_k)) = C}\ \textsc{OwnerTupVal}$$

$$\frac{}{\text{owner}(\#Ct, (\overline{\tau}\ C)\{\overline{V} = \overline{v}\}) = C}\ \textsc{OwnerInstance}$$

This is the main exhaustiveness lemma, which says that every type-correct argument value for a function has at least one applicable function case. Its proof follows from the global- and local-default requirements. If the function has a global default case, then the result follows from the definition of a global default. Otherwise, the function must be internal. In that case, there must be a local default case for each type-correct argument value's owner, so the result follows from the definition of a local default.

Lemma 2 (Exhaustiveness) If $\vdash (\overline{\tau}\ F) : \tau_2 \to \tau$ and $\vdash v : \tau_2'$ and $\tau_2' \leq \tau_2$, then there exist some $Sn \in \text{dom}(SigT)$, some $(\texttt{extend fun}_{Mn}\ \overline{\alpha_1}\ F\ Pat) \in SigT(Sn)$, and some environment ρ such that $\text{match}(v, Pat) = \rho$.

Proof Since $\vdash (\overline{\tau}\ F) : \tau_2 \to \tau$, by T-Fun we have $F = Sn'.Fn$ and $(\texttt{fun}\ \overline{\alpha}\ Fn : Mt \to \tau_0) \in SigT(Sn')$ and $|\overline{\alpha}| = |\overline{\tau}|$ and $\tau_2 \to \tau = [\overline{\alpha} \mapsto \overline{\tau}](\hat{Mt} \to \tau_0)$. By the definition of $SigT$, also $(\texttt{fun}\ \overline{\alpha}\ Fn : Mt \to \tau_0) \in ST(Sn')$. Let $ST(Sn') = (\texttt{structure}\ Sn' = \texttt{struct}\ \overline{D}\ \texttt{end})$. Then by StructOK we have $\overline{Sn} \subseteq \text{dom}(SigT)$ and $\overline{Sn} \vdash (\texttt{fun}\ \overline{\alpha}\ Fn : Mt \to \tau_0)$ OK in Sn', so by FunOK we have that $\text{owner}(Mt) = Sn''.Cn$. Then by lemma 3 there exists some class C such that $\text{owner}(Mt, v) = C$ and $\text{concrete}(C)$ and $C \leq Sn''.Cn$. Also by FunOK we have either $\overline{Sn} \vdash F$ has-gdefault or $Sn' = Sn''$. We consider these cases separately.

- Case $\overline{Sn} \vdash F$ has-gdefault. By GDefault we have $\text{owner}(F) = C'$ and $\overline{Sn} \vdash F$ has-default-for C'. By OwnerFun, $C' = Sn''.Cn$. Then by lemma 4 there exists some $Sn \in \overline{Sn}$, some $(\texttt{extend fun}_{Mn}\ \overline{\alpha_1}\ F\ Pat) \in SigT(Sn)$, and some environment ρ

such that match$(v, Pat) = \rho$. Since $\overline{Sn} \subseteq$ dom$(SigT)$, we have $Sn \in$ dom$(SigT)$ and the result is shown.

- Case $Sn' = Sn''$. Let $C = Sn_0.Cn_0$. Since concrete(C), by CONCRETE we have (`class` $\overline{\alpha_0}$ Cn_0 ...) $\in ST(Sn_0)$. Let $ST(Sn_0) =$ (`structure` Sn_0 = `struct` $\overline{D_0}$ `end`). Then by STRUCTOK we have $\overline{Sn_0} \subseteq$ dom$(SigT)$ and $\overline{Sn_0} \vdash$ `class` $\overline{\alpha_0}$ Cn_0 ... OK in Sn_0, so by CLASSOK we have concrete$(C) \Rightarrow \overline{Sn_0} \vdash$ funs-have-ldefault-for C. Since concrete(C) holds, we have $\overline{Sn_0} \vdash$ funs-have-ldefault-for C.

  Also by CLASSOK we have $\overline{Sn_0} \vdash C$ ITCTransUses. Since $C \leq Sn''.Cn$ and $Sn' = Sn''$, by lemma 42 we have $Sn' \in \overline{Sn_0}$.

  Since $F = Sn'.Fn$ and $Sn' \in \overline{Sn_0}$, by FUNITCUSES we have $\overline{Sn_0} \vdash F$ ITCUses. Since (`fun` $\overline{\alpha}$ Fn : $Mt \rightarrow \tau_0$) $\in SigT(Sn')$ and owner$(Mt) = Sn'.Cn$, by OWNERFUN we have owner$(F) = Sn'.Cn$. Also, we showed above that $C \leq Sn'.Cn$. Therefore, since $\overline{Sn_0} \vdash$ funs-have-ldefault-for C, by LDEFAULT we have $\overline{Sn_0} \vdash F$ has-default-for C. By SUBREF $C \leq C$, so by lemma 4 there exists some $Sn \in \overline{Sn_0}$, some (`extend` $\texttt{fun}_{Mn}$ $\overline{\alpha_1}$ $Sn.Fn$ Pat) $\in SigT(Sn)$, and some environment ρ such that match$(v, Pat) = \rho$. Since $\overline{Sn_0} \subseteq$ dom$(SigT)$, we have $Sn \in$ dom$(SigT)$, and the result is shown.

□

This lemma says that a value conforming to a marked type has a well-defined owner (with respect to that marked type), which is a subclass of the marked type's owner.

Lemma 3 If $\vdash v : \tau'$ and $\tau' \leq \tau$ and $\tau = [\overline{\alpha} \mapsto \overline{\tau}]\hat{Mt}$ and owner$(Mt) = C'$, then there exists some class C such that owner$(Mt, v) = C$ and concrete(C) and $C \leq C'$.

Proof By induction on the depth of the derivation of $\vdash v : \tau'$. Case analysis of the last rule used in the derivation.

- Case T-REP. Then v has the form $(\overline{\tau_0}\ C)\{\overline{V} = \overline{v}\}$ and $\tau' = (\overline{\tau_0}\ C)$ and concrete(C). Since $\tau' \leq \tau$, by lemma 16 τ has the form $(\overline{\tau_1}\ C'')$. Since $\tau = [\overline{\alpha} \mapsto \overline{\tau}]\hat{Mt}$, $\hat{Mt}$ has the form $(\overline{\tau_2}\ C'')$, and by the grammar for marked types Mt must be `#`$(\overline{\tau_2}\ C'')$. Then by

OwnerInstance we have owner($\#(\overline{\tau_2}\ C''), (\overline{\tau_0}\ C)\{\overline{V} = \overline{v}\}) = C$. We're given $\tau' \leq \tau$, so by lemma 18 we have $C \leq C''$. Since owner$(Mt) = C'$, by OwnerClass we have $C' = C''$, so $C \leq C'$.

- Case T-Fun. Then v has the form $(\overline{\tau_0}\ F)$ and τ' has the form $\tau_1 \to \tau_2$. Therefore by lemma 21 τ has the form $\tau_1' \to \tau_2'$. Since $\tau = [\overline{\alpha} \mapsto \overline{\tau}]\hat{Mt}$, $\hat{Mt}$ has the form $\tau_1'' \to \tau_2''$, but this contradicts the grammar of marked types. Therefore, T-Fun cannot be the last rule in the derivation.

- Case T-Tup: Then v has the form $(v_1, \ldots, v_k)$ and τ' has the form $\tau_1' * \cdots * \tau_k'$ and for all $1 \leq j \leq k$ we have $\vdash v_j : \tau_j'$. Therefore by lemma 23 τ has the form $\tau_1 * \cdots * \tau_k$, where for all $1 \leq j \leq k$ we have $\tau_j' \leq \tau_j$. Since $\tau = [\overline{\alpha} \mapsto \overline{\tau}]\hat{Mt}$, $\hat{Mt}$ has the form $\tau_1'' * \cdots * \tau_k''$, and by the grammar for marked types Mt is $\tau_1'' * \cdots * \tau_{i-1}'' * Mt_i * \tau_{i+1}'' * \cdots * \tau_k''$, where $\hat{Mt_i} = \tau_i''$. We're given owner$(Mt) = C'$, so by OwnerTup we have owner$(Mt_i) = C'$.

  Therefore we have $\vdash v_i : \tau_i'$ and $\tau_i' \leq \tau_i$ and $\tau_i = [\overline{\alpha} \mapsto \overline{\tau}]\hat{Mt_i}$ and owner$(Mt_i) = C'$, so by induction there exists C such that owner$(Mt_i, v_i) = C$ and concrete(C) and $C \leq C'$. By OwnerTupVal we have owner$(\tau_1'' * \cdots * \tau_{i-1}'' * Mt_i * \tau_{i+1}'' * \cdots * \tau_k'', (v_1, \ldots, v_k)) = C$, so the result follows.

□

This lemma validates the has-default-for judgment: if a function F has a default for class C, then F has at least one applicable function case for each type-correct argument value whose owner is a subclass of C.

Lemma 4 If $\vdash v : \tau_2'$ and $\tau_2' \leq \tau_2$ and $\tau_2 = [\overline{\alpha} \mapsto \overline{\tau}]\hat{Mt}$ and $(\texttt{fun}\ \overline{\alpha}\ Fn : Mt \to \tau_0) \in SigT(Sn)$ and owner$(Mt, v) = C_0$ and $C_0 \leq C$ and $\overline{Sn} \vdash Sn.Fn$ has-default-for C, then there exists some $Sn' \in \overline{Sn}$, some $(\texttt{extend fun}_{Mn}\ \overline{\alpha_1}\ Sn.Fn\ Pat) \in SigT(Sn')$, and some environment ρ such that match$(v, Pat) = \rho$.

Proof Since $\overline{Sn} \vdash Sn.Fn$ has-default-for C, by Default we have defaultPat$(Mt, C, d) = Pat'$. Therefore by lemma 5 there exists ρ' such that match$(v, Pat') = \rho'$. Also by Default

we have ($\texttt{extend fun}_{Mn}$ $\overline{\alpha_1}$ $Sn.Fn$ Pat) $\in$ $SigT(Sn')$ and $Pat' \leq Pat$ and $Sn' \in \overline{Sn}$. By lemma 29 there exists ρ such that match$(v, Pat) = \rho$, so the result follows. $\square$

The next two lemmas validate the defaultPat judgment, by showing that the generated pattern is in fact a (local or global) default pattern: each type-correct argument value matches the generated pattern. The first lemma handles patterns generated with respect to marked types, and the second lemma handles patterns generated with respect to ordinary (unmarked) types.

Lemma 5 If $\vdash v : \tau'$ and $\tau' \leq \tau$ and $\tau = [\overline{\alpha} \mapsto \overline{\tau}]\hat{Mt}$ and owner$(Mt, v) = C_0$ and $C_0 \leq C$ and defaultPat$(Mt, C, d) = Pat$, then there exists ρ such that match$(v, Pat) = \rho$.

Proof By strong induction on the depth of the derivation of defaultPat$(Mt, C, d) = Pat$. Case analysis of the last rule in the derivation.

- Case DefZero. Then Pat has the form _, so by E-MatchWild we have match$(v, _)$ $= \{\}$.

- Case DefOwnerClassType. Then Mt has the form $\#(\overline{\tau_1}\ C')$ and Pat has the form $(C\{\overline{V} = \overline{Pat}\})$ and repType$(\overline{\tau_1}\ C) = \{\overline{V} : \overline{\tau'}\}$ and defaultPat$(\overline{\tau'}, C, d-1) = \overline{Pat}$ and $d > 0$. By lemma 27 we have repType$([\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_1}\ C) = [\overline{\alpha} \mapsto \overline{\tau}]\{\overline{V} : \overline{\tau'}\}$. Since owner$(\#(\overline{\tau_1}\ C'), v) = C_0$, by OwnerInstance we have that v is of the form $(\overline{\tau_0}\ C_0)\{\overline{V_2} = \overline{v_2}\}$.

  Since we're given that $\vdash v : \tau'$, by T-Rep we have that $\tau' = (\overline{\tau_0}\ C_0)$ and $\bullet \vdash (\overline{\tau_0}\ C_0)$ OK and repType$(\overline{\tau_0}\ C_0) = \{\overline{V_2} : \overline{\tau_2}\}$ and $\vdash \overline{v_1} : \overline{\tau_2'}$ and $\overline{\tau_2'} \leq \overline{\tau_2}$. We're given that $\tau' \leq \tau$, so that means $(\overline{\tau_0}\ C_0) \leq ([\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_1}\ C')$, and by lemma 17 we have $\overline{\tau_0} = [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_1}$. Since $C_0 \leq C$ and $\bullet \vdash (\overline{\tau_0}\ C_0)$ OK, by lemma 19 we have $(\overline{\tau_0}\ C_0) \leq (\overline{\tau_0}\ C)$. Therefore by lemma 41 we have $\{\overline{V_2} : \overline{\tau_2}\} = \{\overline{V} : [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau'}, \overline{V_3} : \overline{\tau_3}\}$.

  Therefore there is some prefix $\overline{v_4}$ of $\overline{v_2}$ and some prefix $\overline{\tau_4}$ of $\overline{\tau_2'}$ such that $\vdash \overline{v_4} : \overline{\tau_4}$ and $\overline{\tau_4} \leq [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau'}$ and defaultPat$(\overline{\tau'}, C, d-1) = \overline{Pat}$, so by lemma 6, there exists $\overline{\rho}$ such that match$(\overline{v_4}, \overline{Pat}) = \bigcup \overline{\rho}$. Finally, we're given $C_0 \leq C$, so by E-MatchClass we have match$((\overline{\tau_0}\ C_0)\{\overline{V_2} = \overline{v_2}\}, (C\{\overline{V} = \overline{Pat}\})) = \bigcup \overline{\rho}$.

- Case DefTupType. Then Mt has the form $\tau_1 * \cdots * \tau_{i-1} * Mt_i * \tau_{i+1} * \cdots * \tau_k$ and Pat has the form $(Pat_1, \ldots, Pat_k)$ and for all $1 \leq j \leq k$ such that $j \neq i$ we have $\text{defaultPat}(\tau_j, C, d-1) = Pat_j$ and we have $\text{defaultPat}(Mt_i, C, d-1) = Pat_i$. Let $\tau_i = \hat{Mt_i}$. Since $\tau' \leq [\overline{\alpha} \mapsto \overline{\tau}](\tau_1 * \cdots * \tau_k)$, by lemma 22 we have that τ' has the form $\tau'_1 * \cdots * \tau'_k$, where for all $1 \leq j \leq k$ we have $\tau'_j \leq [\overline{\alpha} \mapsto \overline{\tau}]\tau_j$. Since $\vdash v : \tau'$, by T-Tup we have that v has the form $(v_1, \ldots, v_k)$ and for all $1 \leq j \leq k$ we have $\vdash v_j : \tau'_j$. Therefore by lemma 6, for all $1 \leq j \leq k$ such that $j \neq i$ we have that there exists some ρ_j such that $\text{match}(v_j, Pat_j) = \rho_j$. We're given that $\text{owner}(Mt, v) = C_0$, so by OwnerTupVal we have $\text{owner}(Mt_i, v_i) = C_0$. Therefore by induction we have that there exists some ρ_i such that $\text{match}(v_i, Pat_i) = \rho_i$. Then by E-MatchTup we have $\text{match}(v, Pat) = \rho_1 \cup \cdots \cup \rho_k$.

□

Lemma 6 If $\vdash v : \tau'$ and $\tau' \leq \tau$ and $\tau = [\overline{\alpha} \mapsto \overline{\tau}]\tau_0$ and $\text{defaultPat}(\tau_0, C_0, d) = Pat$, then there exists ρ such that $\text{match}(v, Pat) = \rho$.

Proof By strong induction on the depth of the derivation of $\text{defaultPat}(\tau_0, C_0, d) = Pat$. Case analysis of the last rule in the derivation.

- Case DefZero or DefTypeVar or DefFunType. Then Pat has the form $\_$, so by E-MatchWild we have $\text{match}(v, \_) = \{\}$.

- Case DefClassType. Then τ_0 has the form $(\overline{\tau_0}\ C)$ and Pat has the form $(C\{\overline{V} = \overline{Pat}\})$ and $\text{repType}(\overline{\tau_0}\ C) = \{\overline{V} : \overline{\tau'}\}$ and $\text{defaultPat}(\overline{\tau'}, C_0, d-1) = \overline{Pat}$ and $d > 0$. Since $\tau = [\overline{\alpha} \mapsto \overline{\tau}]\tau_0$, by lemma 27 we have $\text{repType}(\tau) = [\overline{\alpha} \mapsto \overline{\tau}]\{\overline{V} : \overline{\tau'}\}$. Further, $\tau = [\overline{\alpha} \mapsto \overline{\tau}](\overline{\tau_0}\ C) = ([\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_0}\ C)$. Since $\tau' \leq \tau$, by lemma 15 τ' has the form $(\overline{\tau_1}\ C')$. Since $\vdash v : \tau'$, by T-Rep v has the form $(\overline{\tau_1}\ C')\{\overline{V_1} = \overline{v_1}\}$ and $\bullet \vdash (\overline{\tau_1}\ C')$ OK and $\text{repType}(\overline{\tau_1}\ C') = \{\overline{V_1} : \overline{\tau_1}\}$ and $\vdash \overline{v_1} : \overline{\tau'_1}$ and $\overline{\tau'_1} \leq \overline{\tau_1}$.

  Since $(\overline{\tau_1}\ C') \leq ([\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_0}\ C)$, by lemma 18 we have $C' \leq C$. Further, by lemma 41 we have that $\{\overline{V_1} : \overline{\tau_1}\} = \{\overline{V} : [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau'}, \overline{V_2} : \overline{\tau_2}\}$. Therefore there is some prefix $\overline{v_3}$ of $\overline{v_1}$ and some prefix $\overline{\tau_3}$ of $\overline{\tau'_1}$ such that $\vdash \overline{v_3} : \overline{\tau_3}$ and $\overline{\tau_3} \leq [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau'}$ and

defaultPat($\overline{\tau'}, C_0, d-1$) = $\overline{Pat}$. Therefore by induction, match($\overline{v_3}, \overline{Pat}$) = $\overline{\rho}$. Therefore by E-MatchClass we have match(($\overline{\tau_1}$ C')$\{\overline{V_1} = \overline{v_1}\}$, ($C\{\overline{V} = \overline{Pat}\}$)) = $\bigcup \overline{\rho}$.

- Case DefTupType. Then τ_0 has the form $\tau_1 * \cdots * \tau_k$ and Pat has the form $(Pat_1, \ldots, Pat_k)$ and for all $1 \leq i \leq k$ we have defaultPat($\tau_i, C_0, d-1$) = Pat_i and $d > 0$. Since $\tau' \leq [\overline{\alpha} \mapsto \overline{\tau}](\tau_1 * \cdots * \tau_k)$, by lemma 22 we have that τ' has the form $\tau'_1 * \cdots * \tau'_k$, where for all $1 \leq i \leq k$ we have $\tau'_i \leq [\overline{\alpha} \mapsto \overline{\tau}]\tau_i$. Since $\vdash v : \tau'$, by T-Tup we have that v has the form $(v_1, \ldots, v_k)$ and for all $1 \leq i \leq k$ we have $\vdash v_i : \tau'_i$. Therefore by induction, for all $1 \leq i \leq k$ we have that there exists some ρ_i such that match(v_i, Pat_i) = ρ_i. Then by E-MatchTup we have match(v, Pat) = $\rho_1 \cup \cdots \cup \rho_k$.

□

A.1.2 Ambiguity

The following lemma says that if a value has at least one applicable function case then it has a most-specific applicable case. The lemma thereby validates the modular ambiguity checking algorithm and requirement M1. The lemma is proven by induction on the number of applicable function cases that are not overridden by the given one. If there are none (other than the given case itself), then the given case is the most-specific applicable case. Otherwise, some applicable function case is not overridden by the given one. In that case I show that, by modular ambiguity checking, there must exist a resolving case, and the result follows by induction on the resolving case.

Lemma 7 (Unambiguity) If $\vdash v : \tau$ and $Sn \in \text{dom}(SigT)$ and (`extend fun`$_{Mn}$ $\overline{\alpha}$ F Pat) $\in SigT(Sn)$ and match(v, Pat) = ρ, then there exists some $Sn' \in \text{dom}(SigT)$, some (`extend fun`$_{Mn'}$ $\overline{\alpha_1}$ F Pat') $\in SigT(Sn')$, and some ρ' such that match(v, Pat') = ρ' and $\forall Sn'' \in \text{dom}(SigT).\forall$(`extend fun`$_{Mn''}$ $\overline{\alpha_2}$ F Pat'') $\in SigT(Sn'')$. $\forall \rho''.((\text{match}(v, Pat'') = \rho'' \wedge Sn'.Mn' \neq Sn''.Mn'') \Rightarrow Pat' < Pat'')$.

Proof By (strong) induction on the number of function cases of F that are applicable to v but are not overridden by $Sn.Mn$. These are the cases of the form (`extend fun`$_{Mn_0}$ $\overline{\alpha_0}$ F

Pat_0) such that ($\texttt{extend fun}_{Mn_0}$ $\overline{\alpha_0}$ F Pat_0) $\in SigT(Sn_0)$ for some $Sn_0 \in \text{dom}(SigT)$, and $\text{match}(v, Pat_0) = \rho_0$ for some ρ_0, and $Pat \not< Pat_0$.

- Case there are zero function cases of F that are applicable to v but are not overridden by *Sn.Mn*. We're given that $Sn \in \text{dom}(SigT)$ and ($\texttt{extend fun}_{Mn}$ $\overline{\alpha}$ F Pat) $\in SigT(Sn)$ and $\text{match}(v, Pat) = \rho$. Further, since it cannot both be the case that $Pat \leq Pat$ and $Pat \not\leq Pat$, we have $Pat \not< Pat$. Therefore, *Sn.Mn* itself is applicable to v but is not overridden by *Sn.Mn*, so we have a contradiction.

- Case there is exactly one function case of F that is applicable to v but is not overridden by *Sn.Mn*. Then from the previous case we know that *Sn.Mn* must itself be that function case. Therefore it follows that $\forall Sn'' \in \text{dom}(SigT).\forall(\texttt{extend fun}_{Mn''}$ $\overline{\alpha_2}$ F $Pat'') \in SigT(Sn'').\forall\rho''.((\text{match}(v, Pat'') = \rho'' \wedge Sn.Mn \neq Sn''.Mn'') \Rightarrow Pat < Pat'')$. Then the result follows.

- There are $k > 1$ function cases of F that are applicable to v but are not overridden by *Sn.Mn*. Let ($\texttt{extend fun}_{Mn_1}$ $\overline{\alpha_3}$ F Pat_1) be one such function case, so ($\texttt{extend fun}_{Mn_1}$ $\overline{\alpha_3}$ F Pat_1) $\in SigT(Sn_1)$ for some $Sn_1 \in \text{dom}(SigT)$, and $\text{match}(v, Pat_1) = \rho_1$ for some ρ_1, and $Pat \not< Pat_1$. Since $k > 1$, at least one of the function cases satisfying the conditions is not *Sn.Mn*, so assume without loss of generality that $Sn.Mn \neq Sn_1.Mn_1$.

  By CaseOK we have $\text{matchType}(\tau_0, Pat) = (\Gamma_0, \tau_0')$ and $\text{matchType}(\tau_1, Pat_1) = (\Gamma_1, \tau_1')$. We're given that $\vdash v : \tau$. Finally, we saw above that $\text{match}(v, Pat) = \rho$ and $\text{match}(v, Pat_1) = \rho_1$. Therefore by lemma 35 there exists some Pat_{int} such that $Pat \cap Pat_1 = Pat_{int}$. Further, by lemma 8 we have $\text{dom}(SigT) \vdash (Pat, Pat_1)$ unambiguous. Therefore by PairAmb we have that $Pat \ncong Pat_1$ and there exists some $Sn_2 \in \text{dom}(SigT)$ and some ($\texttt{extend fun}_{Mn_2}$ $\overline{\alpha_4}$ F Pat_2) $\in SigT(Sn_2)$ such that $Pat_{int} \cong Pat_2$. Since $\text{match}(v, Pat) = \rho$ and $\text{match}(v, Pat_1) = \rho_1$ and $Pat \cap Pat_1 = Pat_{int}$, by lemma 36 there exists some ρ_{int} such that $\text{match}(v, Pat_{int}) = \rho_{int}$. Then since $Pat_{int} \leq Pat_2$, by lemma 29 there exists ρ_2 such that $\text{match}(v, Pat_2) = \rho_2$.

So we have shown there exists some $Sn_2 \in \mathrm{dom}(SigT)$ and some ($\texttt{extend fun}_{Mn_2}\ \overline{\alpha_4}\ F\ Pat_2$) $\in SigT(Sn_2)$ and some ρ_2 such that $\mathrm{match}(v, Pat_2) = \rho_2$. Let l be the number of function cases of F that are applicable to v but are not overridden by $Sn_2.Mn_2$. This case is finished by showing that $l < k$, so that the result follows by induction with respect to $Sn_2.Mn_2$.

First we show that $l \leq k$. Consider some $Sn_0 \in \mathrm{dom}(SigT)$, some ($\texttt{extend fun}_{Mn_0}\ \overline{\alpha_0}\ F\ Pat_0$) $\in SigT(Sn_0)$, and some ρ_0 such that $\mathrm{match}(v, Pat_0) = \rho_0$ and $Pat_2 \not< Pat_0$. We will show that also $Pat \not< Pat_0$.

Since $Pat \cap Pat_1 = Pat_{int}$, by lemma 33 we have that $Pat_{int} \leq Pat$ and $Pat_{int} \leq Pat_1$. Since $Pat_2 \leq Pat_{int}$, by lemma 31 also $Pat_2 \leq Pat$ and $Pat_2 \leq Pat_1$. Since $Pat_2 \not< Pat_0$, either $Pat_2 \not\leq Pat_0$ or $Pat_0 \leq Pat_2$. We consider these cases in turn.

- Case $Pat_2 \not\leq Pat_0$. Suppose $Pat \leq Pat_0$. Since $Pat_2 \leq Pat$, by lemma 31 we have $Pat_2 \leq Pat_0$, contradicting the assumption of this case. Therefore $Pat \not\leq Pat_0$, so also $Pat \not< Pat_0$.
- Case $Pat_0 \leq Pat_2$. We showed above that $Pat_2 \leq Pat$. Then by lemma 31 $Pat_0 \leq Pat$, so $Pat \not< Pat_0$.

Therefore we have shown that every function case of F that is applicable to v and is not overridden by $Sn_2.Mn_2$ is also not overridden by $Sn.Mn$, so $l \leq k$. To finish the proof, we show that there exists a function case of F that is applicable to v and is not overridden by $Sn.Mn$ but *is* overridden by $Sn_2.Mn_2$. We showed in the first case above that $Sn.Mn$ is not overridden by itself. We will show that $Sn.Mn$ is overridden by $Sn_2.Mn_2$. We do this by proving that $Pat_2 < Pat$. We showed above that $Pat_2 \leq Pat$, so we simply need to prove that $Pat \not\leq Pat_2$. Suppose $Pat \leq Pat_2$. We're given that $Pat \not< Pat_1$ and $Pat \not\cong Pat_1$. Therefore, $Pat \not\leq Pat_1$. On the other hand, since $Pat \leq Pat_2$ and $Pat_2 \leq Pat_1$, by lemma 31 we have $Pat \leq Pat_1$, and we have a contradiction.

□

This lemma says that every pair of function cases belonging to the same generic function is unambiguous. If the two cases are both available during some structure's modular ITC, then the result follows from that structure's ambiguity checks. Otherwise, the modular ambiguity requirement ensures that the cases are disjoint, so they are also unambiguous.

Lemma 8 If $(\texttt{extend fun}_{Mn}\ \overline{\alpha}\ F\ Pat) \in SigT(Sn)$ and $(\texttt{extend fun}_{Mn'}\ \overline{\alpha'}\ F\ Pat') \in SigT(Sn')$ and $Sn.Mn \neq Sn'.Mn'$, then $\mathrm{dom}(SigT) \vdash (Pat, Pat')$ unambiguous.

Proof Since $(\texttt{extend fun}_{Mn}\ \overline{\alpha}\ F\ Pat) \in SigT(Sn)$, by StructOK we have $\overline{Sn} \subseteq \mathrm{dom}(SigT)$ and $\overline{Sn} \vdash (\texttt{extend fun}_{Mn}\ \overline{\alpha}\ F\ Pat = E)$ OK in Sn for some $\overline{Sn}$ and E, so by CaseOK we have $Sn; \overline{Sn} \vdash \texttt{extend fun}_{Mn}\ \overline{\alpha}\ F\ Pat$ unambiguous. Similarly, $\overline{Sn'} \subseteq \mathrm{dom}(SigT)$ and $Sn'; \overline{Sn'} \vdash \texttt{extend fun}_{Mn'}\ \overline{\alpha'}\ F\ Pat'$ unambiguous, for some $\overline{Sn'}$. There are several cases.

- Case $Sn' \in \overline{Sn}$. Since $Sn.Mn \neq Sn'.Mn'$ and $Sn; \overline{Sn} \vdash \texttt{extend fun}_{Mn}\ \overline{\alpha}\ F\ Pat$ unambiguous, by Amb we have $\overline{Sn} \vdash (pat, pat')$ unambiguous. Since $\overline{Sn} \subseteq \mathrm{dom}(SigT)$, by Amb also $\mathrm{dom}(SigT) \vdash (Pat, Pat')$ unambiguous.

- Case $Sn \in \overline{Sn}$. Symmetric to the above case.

- Case $Sn' \notin \overline{Sn}$ and $Sn \notin \overline{Sn'}$. Since $Sn; \overline{Sn} \vdash \texttt{extend fun}_{Mn}\ \overline{\alpha}\ F\ Pat$ unambiguous, by Amb we have $F = Sn_1.Fn$ and $(\texttt{fun}\ \overline{\alpha_1}\ Fn : Mt \to \tau) \in SigT(Sn_1)$ and $Sn = Sn_1 \vee \mathrm{owner}(Mt, Pat) = Sn.Cn$. Similarly, $Sn' = Sn_1 \vee (\mathrm{owner}(Mt, Pat') = Sn'.Cn'$. We have three sub-cases.

  - Case $Sn' = Sn_1$. Since $\overline{Sn} \vdash (\texttt{extend fun}_{Mn}\ \overline{\alpha}\ F\ Pat = E)$ OK in Sn, by CaseOK we have $\overline{Sn} \vdash Sn'.Fn$ ITCUses. Then by FunITCUses $Sn' \in \overline{Sn}$, so we have a contradiction.

  - Case $Sn = Sn_1$. Symmetric to the above case.

  - Case $\mathrm{owner}(Mt, Pat) = Sn.Cn$ and $\mathrm{owner}(Mt, Pat') = Sn'.Cn'$. First we show that Pat and Pat' are disjoint: there does not exist Pat_0 such that $Pat \cap Pat' = Pat_0$. Suppose not. Then by lemma 34 either $Sn.Cn \leq Sn'.Cn'$ or $Sn'.Cn' \leq Sn.Cn$. There are two subcases.

* Case $Sn.Cn \leq Sn'.Cn'$. By StructOK $\overline{Sn} \vdash$ (<`abstract`> `class` $\overline{\alpha_4}$ Cn...) OK in Sn, so by ClassOK $\overline{Sn} \vdash Sn.Cn$ ITCTransUses. Since $Sn.Cn \leq Sn'.Cn'$, by lemma 42 we have $Sn' \in \overline{Sn}$, which is a contradiction.
* Case $Sn'.Cn' \leq Sn.Cn$. Symmetric to the above case.

So we have shown that there does not exist Pat_0 such that $Pat \cap Pat' = Pat_0$. Then by the contrapositive of lemma 32, $Pat \not\leq Pat'$, so also $Pat \not\cong Pat'$. Then by PairAmb we have $\text{dom}(SigT) \vdash (Pat, Pat')$ unambiguous.

□

A.2 Type Preservation

The type preservation theorem is straightforward except for the case when E is a function application. That case follows easily from lemmas 9 and 10, which are provided below.

Theorem 2 (Type Preservation) If $\vdash E : \tau$ and $E \longrightarrow E'$ then $\vdash E' : \tau'$, for some τ' such that $\tau' \leq \tau$.

Proof By (strong) induction on the depth of the derivation of $E \longrightarrow E'$. Case analysis of the last rule used in the derivation.

- Case E-New. Then E has the form $Ct(\overline{E})$ and E' has the form $Ct\{\overline{V_0} = \overline{E_0}\}$ and $Ct = (\overline{\tau}\ C)$ and concrete(C) and $\text{rep}(Ct(\overline{E})) = \{\overline{V_0} = \overline{E_0}\}$. Since $\vdash E : \tau$, by T-New we have $\tau = Ct$ and $\{\}; \bullet \vdash Ct(\overline{E})$ OK. Then by T-Constr we have $\bullet \vdash Ct$ OK. Therefore by lemmas 38 and 39 there exists $\overline{\tau_0}$ such that $\text{repType}(Ct) = \{\overline{V_0} : \overline{\tau_0}\}$, and by lemma 40 we have $\vdash \overline{E_0} : \overline{\tau_0'}$, where $\overline{\tau_0'} \leq \overline{\tau_0}$. Then by T-Rep we have $\vdash Ct\{\overline{V_0} = \overline{E_0}\} : Ct$, and by SubTRef we have $Ct \leq Ct$.

- Case E-Rep. Then E has the form $Ct\{\overline{V_0} = \overline{v_0}, V_0 = E_0, \overline{V_1} = \overline{E_1}\}$ and E' has the form $Ct\{\overline{V_0} = \overline{v_0}, V_0 = E_0', \overline{V_1} = \overline{E_1}\}$ and $E_0 \longrightarrow E_0'$. Since $\vdash E : \tau$, by T-Rep we have $\tau = Ct = (\overline{\tau}\ C)$ and concrete(C) and $\bullet \vdash Ct$ OK and $\text{repType}(Ct) = \{\overline{V_0} : \overline{\tau_0}, V_0 : \tau_0, \overline{V_1} : \overline{\tau_1}\}$ and $\vdash \overline{v_0} : \overline{\tau_0'}$ and $\overline{\tau_0'} \leq \overline{\tau_0}$ and $\vdash E_0 : \tau_0'$ and $\tau_0' \leq \tau_0$ and $\vdash \overline{E_1} : \overline{\tau_1'}$ and $\overline{\tau_1'} \leq \overline{\tau_1}$. By induction we have $\vdash E_0' : \tau_0''$, for some τ_0'' such that $\tau_0'' \leq \tau_0'$. Therefore by

SubTTrans $\tau_0'' \leq \tau_0$. Then by T-Rep $\vdash Ct\{\overline{V_0} = \overline{v_0}, V_0 = E_0', \overline{V_1} = \overline{E_1}\} : Ct$, and by SubTRef $Ct \leq Ct$.

- Case E-Tup. Then E has the form $(v_1, \ldots, v_{i-1}, E_i, \ldots, E_k)$ and E' has the form $(v_1, \ldots, v_{i-1}, E_i', E_{i+1}, \ldots, E_k)$ and $E_i \longrightarrow E_i'$, where $1 \leq i \leq k$. Since $\vdash E : \tau$, by T-Tup τ has the form $\tau_1 * \cdots * \tau_k$ and $\vdash v_j : \tau_j$ for all $1 \leq j < i$ and $\vdash E_j : \tau_j$ for all $i \leq j \leq k$. By induction we have $\vdash E_i' : \tau_i'$ for some τ_i' such that $\tau_i' \leq \tau_i$. Then by T-Tup we have $\vdash (v_1, \ldots, v_{i-1}, E_i', E_{i+1}, \ldots, E_k) : \tau_1 * \cdots * \tau_{i-1} * \tau_i' * \tau_{i+1} * \cdots * \tau_k$. By SubTRef we have $\tau_j \leq \tau_j$ for all $1 \leq j \leq k$, so by SubTTup we have $\tau_1 * \cdots * \tau_{i-1} * \tau_i' * \tau_{i+1} * \cdots * \tau_k \leq \tau_1 * \cdots * \tau_k$.

- Case E-App1. Then E has the form $E_1\ E_2$ and E' has the form $E_1'\ E_2$ and $E_1 \longrightarrow E_1'$. Since $\vdash E : \tau$, by (T-App) we have $\vdash E_1 : \tau_2 \to \tau$ and $\vdash E_2 : \tau_2'$ and $\tau_2' \leq \tau_2$. By induction $\vdash E_1' : \tau'$, for some τ' such that $\tau' \leq \tau_2 \to \tau$. By lemma 20 τ' has the form $\tau_2'' \to \tau''$, where $\tau_2 \leq \tau_2''$ and $\tau'' \leq \tau$. Therefore by SubTTrans we have $\tau_2' \leq \tau_2''$, so by T-App we have $\vdash E_1'\ E_2 : \tau''$, where $\tau'' \leq \tau$.

- Case E-App2. Then E has the form $v_1\ E_2$ and E' has the form $v_1\ E_2'$ and $E_2 \longrightarrow E_2'$. Since $\vdash E : \tau$, by T-App we have $\vdash v_1 : \tau_2 \to \tau$ and $\vdash E_2 : \tau_2'$ and $\tau_2' \leq \tau_2$. By induction $\vdash E_2' : \tau_2''$, for some τ_2'' such that $\tau_2'' \leq \tau_2'$. By SubTTrans we have $\tau_2'' \leq \tau_2$, so by T-App we have $\vdash v_1\ E_2' : \tau$ and by SubTRef we have $\tau \leq \tau$.

- Case E-AppRed. Then $E = (\overline{\tau}\ F)\ v$ and $E' = [\overline{I_0} \mapsto \overline{v_0}]E_0$ and most-specific-case-for$((\overline{\tau}\ F), v) = (\{(\overline{I_0}, \overline{v_0})\}, E_0)$. Since $\vdash E : \tau$, by T-App we have $\vdash (\overline{\tau}\ F) : \tau_2 \to \tau$ and $\vdash v : \tau_2'$ and $\tau_2' \leq \tau_2$. Then by T-Fun we have and $F = Sn.Fn$ and $\tau_2 \to \tau = [\overline{\alpha} \mapsto \overline{\tau}](\hat{Mt} \to \tau_0)$ and $(\texttt{fun}\ \overline{\alpha}\ Fn : Mt \to \tau_0) \in SigT(Sn)$ and $\bullet \vdash \overline{\tau}$ OK. By Lookup we have $E_0 = [\overline{\alpha_0} \mapsto \overline{\tau}]E_0'$ and $(\texttt{extend fun}_{Mn}\ \overline{\alpha_0}\ F\ Pat = E_0') \in ST(Sn')$ and match$(v, Pat) = \{(\overline{I_0}, \overline{v_0})\}$. Then by CaseOK we have matchType$([\overline{\alpha} \mapsto \overline{\alpha_0}]\hat{Mt}, Pat)$ $= (\Gamma, \tau'')$ and $\Gamma; \overline{\alpha_0} \vdash E_0' : \tau_0'$ and $\tau_0' \leq [\overline{\alpha} \mapsto \overline{\alpha_0}]\tau_0$.

  By lemma 28 we have matchType$([\overline{\alpha_0} \mapsto \overline{\tau}][\overline{\alpha} \mapsto \overline{\alpha_0}]\hat{Mt}, Pat) = ([\overline{\alpha_0} \mapsto \overline{\tau}]\Gamma, [\overline{\alpha_0} \mapsto \overline{\tau}]\tau'')$. By FunOK we have $\overline{\alpha} \vdash \hat{Mt}$ OK, so by lemma 11 all type variables in $\hat{Mt}$ are

in $\overline{\alpha}$. Therefore $[\overline{\alpha_0} \mapsto \overline{\tau}][\overline{\alpha} \mapsto \overline{\alpha_0}]\hat{M}t$ is equivalent to $[\overline{\alpha} \mapsto \overline{\tau}]\hat{M}t = \tau_2$, so we have matchType$(\tau_2, Pat) = ([\overline{\alpha_0} \mapsto \overline{\tau}]\Gamma, [\overline{\alpha_0} \mapsto \overline{\tau}]\tau'')$. Then by lemma 10 we have $\tau_2' \leq [\overline{\alpha_0} \mapsto \overline{\tau}]\tau''$ and $\text{dom}([\overline{\alpha_0} \mapsto \overline{\tau}]\Gamma) = \text{dom}(\{(\overline{I_0}, \overline{v_0})\})$ and for each $(I_x, \tau_x) \in [\overline{\alpha_0} \mapsto \overline{\tau}]\Gamma$, there exists $(I_x, v_x) \in \{(\overline{I_0}, \overline{v_0})\}$ such that $\vdash v_x : \tau_x'$, where $\tau_x' \leq \tau_x$.

By lemma 26 we have $[\overline{\alpha_0} \mapsto \overline{\tau}]\Gamma; \bullet \vdash [\overline{\alpha_0} \mapsto \overline{\tau}]E_0' : [\overline{\alpha_0} \mapsto \overline{\tau}]\tau_0'$. Then by lemma 9 $\vdash [\overline{I_0} \mapsto \overline{v_0}][\overline{\alpha_0} \mapsto \overline{\tau}]E_0' : \tau_{sub}$ and $\tau_{sub} \leq [\overline{\alpha_0} \mapsto \overline{\tau}]\tau_0'$. By lemma 25 we have $[\overline{\alpha_0} \mapsto \overline{\tau}]\tau_0' \leq [\overline{\alpha_0} \mapsto \overline{\tau}][\overline{\alpha} \mapsto \overline{\alpha_0}]\tau_0$. By FunOK we have $\overline{\alpha} \vdash \tau_0$ OK, so by lemma 11 all type variables in τ_0 are in $\overline{\alpha}$. Therefore $[\overline{\alpha_0} \mapsto \overline{\tau}][\overline{\alpha} \mapsto \overline{\alpha_0}]\tau_0$ is equivalent to $[\overline{\alpha} \mapsto \overline{\tau}]\tau_0 = \tau$, so we have $[\overline{\alpha_0} \mapsto \overline{\tau}]\tau_0' \leq \tau$. Then by SubTTrans we have $\tau_{sub} \leq \tau$. Therefore we have shown $\vdash E' : \tau_{sub}$ and $\tau_{sub} \leq \tau$.

□

This is a standard lemma showing that type-correct substitution preserves the well-typedness of an expression.

Lemma 9 (Substitution) If $\Gamma; \overline{\alpha_0} \vdash E : \tau$ and $\Gamma = \{(\overline{I_0}, \overline{\tau_0})\}$ and $\Gamma_0; \overline{\alpha_0} \vdash \overline{E_0} : \overline{\tau_0}'$ and $\overline{\tau_0'} \leq \overline{\tau_0}$, then $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]E : \tau'$, for some τ' such that $\tau' \leq \tau$.
Proof By (strong) induction on the depth of the derivation of $\Gamma; \overline{\alpha_0} \vdash E : \tau$. Case analysis of the last rule used in the derivation.

- Case T-Id. Then $E = I$ and $(I, \tau) \in \Gamma$, so $I = I_j$ and $\tau = \tau_j$ for some $1 \leq j \leq k$, where $\overline{I_0} = I_1, \ldots, I_k$ and $\overline{\tau_0} = \tau_1, \ldots, \tau_k$ and $\overline{E_0} = E_1, \ldots, E_k$. Therefore $[\overline{I_0} \mapsto \overline{E_0}]E = E_j$. Since we're given that $\Gamma_0; \overline{\alpha_0} \vdash E_j : \tau_j'$ and $\tau_j' \leq \tau_j$, the result is shown.

- Case T-New. Then $E = Ct(\overline{E})$ and $\tau = Ct$ and $\Gamma; \overline{\alpha_0} \vdash Ct(\overline{E})$ OK and $Ct = (\overline{\tau_1}\ Sn.Cn)$ and concrete$(Sn.Cn)$. Then by T-Constr we have $\overline{\alpha_0} \vdash Ct$ OK and (<`abstract`> `class` $\overline{\alpha_1}\ Cn(\overline{I} : \overline{\tau})\ \ldots) \in SigT(Sn)$ and $\Gamma; \overline{\alpha_0} \vdash \overline{E} : \overline{\tau'}$ and $\overline{\tau'} \leq [\overline{\alpha_1} \mapsto \overline{\tau_1}]\overline{\tau}$. Since $[\overline{I_0} \mapsto \overline{E_0}]Ct = Ct$ and $[\overline{I_0} \mapsto \overline{E_0}]Sn.Cn = Sn.Cn$, we have $\overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]Ct$ OK and concrete$([\overline{I_0} \mapsto \overline{E_0}]Sn.Cn)$. By induction we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]\overline{E} : \overline{\tau''}$ and $\overline{\tau''} \leq \overline{\tau'}$. Then by SubTTrans we have $\overline{\tau''} \leq [\overline{\alpha_1} \mapsto \overline{\tau_1}]\overline{\tau}$.

Therefore by T-Constr we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]E$ OK, so by T-New we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]E : \tau$. By SubTRef we have $\tau \leq \tau$, so the result is shown.

- Case T-Rep. Then $E = Ct\{\overline{V} = \overline{E}\}$ and $\tau = Ct$ and $\overline{\alpha_0} \vdash Ct$ OK and $Ct = (\overline{\tau_1}\ Sn.Cn)$ and concrete($Sn.Cn$) and repType(Ct) $= \{\overline{V} : \overline{\tau}\}$ and $\Gamma; \overline{\alpha_0} \vdash \overline{E} : \overline{\tau'}$ and $\overline{\tau'} \leq \overline{\tau}$. Since $[\overline{I_0} \mapsto \overline{E_0}]Ct = Ct$ and $[\overline{I_0} \mapsto \overline{E_0}]Sn.Cn = Sn.Cn$, we have $\overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]Ct$ OK and concrete($[\overline{I_0} \mapsto \overline{E_0}]Sn.Cn$) and repType($[\overline{I_0} \mapsto \overline{E_0}]Ct$) $= \{\overline{V} : \overline{\tau}\}$. By induction we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]\overline{E} : \overline{\tau''}$ and $\overline{\tau''} \leq \overline{\tau'}$. Then by SubTTrans we have $\overline{\tau''} \leq \overline{\tau}$, so by T-Rep we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]E : \tau$. By SubTRef we have $\tau \leq \tau$, so the result is shown.

- Case T-Fun. Then since Γ is not used in T-Fun and $\Gamma; \overline{\alpha_0} \vdash E : \tau$, also $\Gamma_0; \overline{\alpha_0} \vdash E : \tau$. Further, we have $E = Fv$, so $[\overline{I_0} \mapsto \overline{E_0}]E = E$. Therefore $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]E : \tau$, and by SubTRef $\tau \leq \tau$, so the result is shown.

- Case T-Tup. Then $E = (E_1, \ldots, E_k)$ and $\tau = \tau_1 * \cdots * \tau_k$ and for all $1 \leq j \leq k$ we have $\Gamma; \overline{\alpha_0} \vdash E_j : \tau_j$. Then by induction, for all $1 \leq j \leq k$ we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]E_j : \tau'_j$ and $\tau'_j \leq \tau_j$. Then by T-Tup we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}](E_1, \ldots, E_k) : \tau'_1 * \cdots * \tau'_k$. Finally, by SubTTup we have $\tau'_1 * \cdots * \tau'_k \leq \tau_1 * \cdots * \tau_k$.

- Case T-App. Then $E = E_1\ E_2$ and $\Gamma; \overline{\alpha_0} \vdash E_1 : \tau_2 \to \tau$ and $\Gamma; \overline{\alpha_0} \vdash E_2 : \tau'_2$ and $\tau'_2 \leq \tau_2$. By induction we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]E_1 : \tau_0$ and $\tau_0 \leq \tau_2 \to \tau$. Also by induction we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}]E_2 : \tau''_2$ and $\tau''_2 \leq \tau'_2$. Then by SubTTrans we have $\tau''_2 \leq \tau_2$. By lemma 20 τ_0 has the form $\tau_{arg} \to \tau_{res}$, where $\tau_2 \leq \tau_{arg}$ and $\tau_{res} \leq \tau$. Therefore by SubTTrans we have $\tau''_2 \leq \tau_{arg}$. Therefore by T-App we have $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_0} \mapsto \overline{E_0}](E'_1\ E'_2) : \tau_{res}$. We saw above that $\tau_{res} \leq \tau$, so the result is shown.

□

This lemma relates the results of pattern matching to its static approximation. In particular, the type of any value matching a pattern is a subtype of the pattern's type, and similarly for values bound to identifiers in the pattern. The proof is straightforward.

Lemma 10 If $\vdash v : \tau''$ and $\tau'' \leq \tau$ and $\text{match}(v, Pat) = \rho$ and $\text{matchType}(\tau, Pat) = (\Gamma, \tau')$, then (1) $\tau'' \leq \tau'$; and (2) $\text{dom}(\Gamma) = \text{dom}(\rho)$ and (3) for each $(I_0, \tau_0) \in \Gamma$, there exists $(I_0, v_0) \in \rho$ such that $\vdash v_0 : \tau_0'$, for some τ_0' such that $\tau_0' \leq \tau_0$.

Proof By (strong) induction on the length of the derivation of $\text{match}(v, Pat) = \rho$. Case analysis of the last rule used in the derivation:

- Case E-MatchWild. Then Pat has the form $\_$ and $\rho = \{\}$. By T-MatchWild we have $\Gamma = \{\}$ and $\tau' = \tau$. Therefore, conditions 1 and 2 are shown, and condition 3 holds vacuously.

- Case E-MatchBind. Then Pat has the form I `as` Pat' and $\rho = \rho' \cup \{(I, v)\}$ and $\text{match}(v, Pat') = \rho'$. By T-MatchBind we have $\Gamma = \Gamma' \cup \{(I, \tau')\}$ and $\text{matchType}(\tau, Pat') = (\Gamma', \tau')$. By induction we have $\tau'' \leq \tau'$ and $\text{dom}(\Gamma') = \text{dom}(\rho')$ and for each $(I_0, \tau_0) \in \Gamma'$, there exists $(I_0, v_0) \in \rho'$ such that $\vdash v_0 : \tau_0'$, where $\tau_0' \leq \tau_0$. Therefore, we have $\tau'' \leq \tau'$ and $\text{dom}(\Gamma' \cup \{(I, \tau')\}) = \text{dom}(\rho' \cup \{(I, v)\})$ and for each $(I_0, \tau_0) \in \Gamma' \cup \{(I, \tau')\}$, there exists $(I_0, v_0) \in \rho' \cup \{(I, v)\}$ such that $\vdash v_0 : \tau_0'$, where $\tau_0' \leq \tau_0$.

- Case E-MatchTup. Then $v = (v_1, \ldots, v_k)$ and Pat has the form $(Pat_1, \ldots, Pat_k)$ and $\rho = \rho_1 \cup \cdots \cup \rho_k$ and for all $1 \leq i \leq k$ we have $\text{match}(v_i, Pat_i) = \rho_i$. By T-MatchTup we have $\tau = \tau_1 * \cdots * \tau_k$ and $\Gamma = \Gamma_1 \cup \ldots \cup \Gamma_k$ and $\tau' = \tau_1' \cdots * \tau_k'$ and for all $1 \leq i \leq k$ we have $\text{match}(\tau_i, Pat_i) = (\Gamma_i, \tau_i')$.

  Since $\vdash v : \tau''$, by T-Tup we have that $\tau'' = \tau_1'' * \cdots * \tau_k''$ and for all $1 \leq i \leq k$ we have $\vdash v_i : \tau_i''$. Since we're given that $\tau'' \leq \tau$, by lemma 22 we have $\tau_i'' \leq \tau_i$ for all $1 \leq i \leq k$. Then by induction, for all $1 \leq i \leq k$ we have $\tau_i'' \leq \tau_i'$. Then by SubTTup we have $\tau_1'' * \cdots * \tau_k'' \leq \tau_1' * \ldots * \tau_k'$, proving condition 1. Also by induction, for all $1 \leq i \leq k$ we have $\text{dom}(\Gamma_i) = \text{dom}(\rho_i)$ and for each $(I_0, \tau_0) \in \Gamma_i$, there exists $(I_0, v_0) \in \rho_i$ such that $\vdash v_0 : \tau_0'$, where $\tau_0' \leq \tau_0$. Therefore conditions 2 and 3 follow.

- Case E-MatchClass. Then $v = ((\overline{\tau}\ C)\{\overline{V_1} = \overline{v_1}, \overline{V_2} = \overline{v_2}\})$ and Pat has the form $(C'\{\overline{V_1} = \overline{Pat_1})$ and $C \leq C'$ and $\rho = \bigcup \overline{\rho_1}$ and $\text{match}(\overline{v_1}, \overline{Pat_1}) = \overline{\rho_1}$. By T-

MatchClass we have $\tau = (\overline{\tau'}\ C'')$ and $\tau' = (\overline{\tau'}\ C')$ and $\Gamma = \bigcup \overline{\Gamma_1}$ and $C' \leq C''$ and $\text{repType}(\overline{\tau'}\ C') = \{\overline{V_1} : \overline{\tau_1}\}$ and $\text{matchType}(\overline{\tau_1}, \overline{Pat_1}) = (\overline{\Gamma_1}, \overline{\tau_1'})$.

Since $\vdash v : \tau''$ and $v = ((\overline{\tau}\ C)\{\overline{V_1} = \overline{v_1}, \overline{V_2} = \overline{v_2}\})$, by T-Rep we have that $\tau'' = (\overline{\tau}\ C)$ and $\bullet \vdash (\overline{\tau}\ C)$ OK and and $\text{repType}(\overline{\tau}\ C) = \{\overline{V_1} : \overline{\tau_1''}, \overline{V_2} : \overline{\tau_2''}\}$ and $\vdash \overline{v_1} : \overline{\tau_1'''}$ and $\overline{\tau_1'''} \leq \overline{\tau_1''}$. Since $\tau'' \leq \tau$, we have $(\overline{\tau}\ C) \leq (\overline{\tau'}\ C'')$, so by lemma 17 we have $\overline{\tau} = \overline{\tau'}$. Since $C \leq C'$ and $\bullet \vdash (\overline{\tau}\ C)$ OK, by lemma 19 we have $(\overline{\tau}\ C) \leq (\overline{\tau}\ C')$, and since $\overline{\tau} = \overline{\tau'}$, condition 1 is shown. By lemma 41 we have $\overline{\tau_1''} = \overline{\tau_1}$. Therefore $\vdash \overline{v_1} : \overline{\tau_1'''}$ and $\overline{\tau_1'''} \leq \overline{\tau_1}$ and $\text{match}(\overline{v_1}, \overline{Pat_1}) = \overline{\rho_1}$ and $\text{matchType}(\overline{\tau_1}, \overline{Pat_1}) = (\overline{\Gamma_1}, \overline{\tau_1'})$, so by induction it follows that $\text{dom}(\bigcup \overline{\Gamma_1}) = \text{dom}(\bigcup \overline{\rho_1})$ and for each $(I_0, \tau_0) \in \bigcup \overline{\Gamma_1}$, there exists $(I_0, v_0) \in \bigcup \overline{\rho_1}$ such that $\vdash v_0 : \tau_0'$, where $\tau_0' \leq \tau_0$. Therefore, conditions 2 and 3 are shown.

□

A.3 Basic Lemmas

A.3.1 Type Well-formedness

Lemma 11 If $\overline{\alpha} \vdash \tau$ OK, then all type variables in τ are in $\overline{\alpha}$.

Proof By (strong) induction on the depth of the derivation of $\overline{\alpha} \vdash \tau$ OK. Case analysis on the last rule used in the derivation. For TVarOK, τ has the form α and the premise ensures that $\alpha \in \overline{\alpha}$. All other cases are proven by induction. □

A.3.2 Subclassing and Subtyping

Lemma 12 If $C_1 \leq C_2$, then there is a path in the declared inheritance graph from C_1 to C_2.

Proof By induction on the depth of the derivation of $C_1 \leq C_2$. Case analysis of the last rule used in the derivation.

- Case SubRef. Then $C_1 = C_2$, so there is a trivial path in the inheritance graph from C_1 to C_2.

- Case SubTrans. Then $C_1 \leq C_3$ and $C_3 \leq C_2$. By induction, there is a path in the inheritance graph from C_1 to C_3 and from C_3 to C_2, so the concatenation of these paths is a path from C_1 to C_2.

- Case SubExt. Then $C_1 = Sn_1.Cn_1$ and <`abstract`> `class` $\overline{\alpha_1}$ $Cn_1(\overline{I_0} : \overline{\tau_0})$ `extends` $\overline{\tau}$ C_2 $\ldots) \in ST(Sn_1)$. Therefore there is an edge from C_1 to C_2 in the declared inheritance graph, so there is also a path from C_1 to C_2.

□

Lemma 13 If $C_1 \leq C_2$ and $C_1 \leq C_3$, then either $C_2 \leq C_3$ or $C_3 \leq C_2$.

Proof By induction on the depth of the derivation of $C_1 \leq C_2$. Case analysis of the last rule used in the derivation.

- Case SubRef. Then $C_1 = C_2$. Since $C_1 \leq C_3$, also $C_2 \leq C_3$.

- Case SubTrans. Then $C_1 \leq C_4$ and $C_4 \leq C_2$. So we have $C_1 \leq C_4$ and $C_1 \leq C_3$, and by induction either $C_4 \leq C_3$ or $C_3 \leq C_4$.

  - Case $C_4 \leq C_3$. Then we have $C_4 \leq C_2$ and $C_4 \leq C_3$, so by induction either $C_2 \leq C_3$ or $C_3 \leq C_2$.
  - Case $C_3 \leq C_4$. Then we have $C_3 \leq C_4$ and $C_4 \leq C_2$, so by SubTrans $C_3 \leq C_2$.

- Case SubExt. Then $C_1 = Sn_1.Cn_1$ and (<`abstract`> `class` $\overline{\alpha}$ $Cn_1(\overline{I_0} : \overline{\tau_0})$ `extends` $\overline{\tau}$ C_2 $\ldots) \in ST(Sn_1)$. Case analysis of the last rule used in the derivation of $C_1 \leq C_3$.

  - Case SubRef. Then $C_1 = C_3$. Since $C_1 \leq C_2$, also $C_3 \leq C_2$.
  - Case SubTrans. Then $C_1 \leq C_4$ and $C_4 \leq C_3$. Assume without loss of generality that the derivation of $C_1 \leq C_4$ ends with a use of SubExt (any nontrivial use of SubTrans can be made to satisfy this assumption). Then (<`abstract`> `class` $\overline{\alpha}$ $Cn_1(\overline{I_0} : \overline{\tau_0})$ `extends` $\overline{\tau}$ C_4 $\ldots) \in ST(Sn_1)$, so $C_2 = C_4$. Since $C_4 \leq C_3$, also $C_2 \leq C_3$.

– Case SubExt. Then (<abstract> `class` $\overline{\alpha}$ $Cn_1(\overline{I_0}$: $\overline{\tau_0})$ `extends` $\overline{\tau}$ C_3 ...) $\in ST(Sn_1)$, so $C_2 = C_3$. Then by SubRef $C_2 \leq C_3$.

□

Lemma 14 If $C_1 \leq C_2$ and $C_2 \leq C_1$, then $C_1 = C_2$.

Proof By lemma 12, there is a path in the declared inheritance graph from C_1 to C_2 and a path from C_2 to C_1. By assumption, the declared inheritance graph is acyclic, so it must be the case that $C_1 = C_2$. □

Lemma 15 If $\tau \leq (\overline{\tau}\ C)$, then τ has the form $(\overline{\tau_1}\ C')$.

Proof By (strong) induction on the depth of the derivation of $\tau \leq (\overline{\tau}\ C)$. Case analysis of the last rule used in the derivation.

- Case SubTRef. Then $\tau = (\overline{\tau}\ C)$.
- Case SubTTrans. Then $\tau \leq \tau'$ and $\tau' \leq (\overline{\tau}\ C)$. By induction τ' has the form $(\overline{\tau_2}\ C'')$. Then by induction again, τ has the form $(\overline{\tau_1}\ C')$.
- Case SubTExt. Then τ has the form $(\overline{\tau_1} Sn.Cn)$, which is also of the form $(\overline{\tau_1}\ C')$.

□

Lemma 16 If $(\overline{\tau}\ C) \leq \tau$, then τ has the form $(\overline{\tau_1}\ C')$.

Proof By (strong) induction on the depth of the derivation of $(\overline{\tau}\ C) \leq \tau$. Case analysis of the last rule used in the derivation.

- Case SubTRef. Then $\tau = (\overline{\tau}\ C)$.
- Case SubTTrans. Then $(\overline{\tau}\ C) \leq \tau'$ and $\tau' \leq \tau$. By induction τ' has the form $(\overline{\tau_2}\ C'')$. Then by induction again, τ has the form $(\overline{\tau_1}\ C')$.
- Case SubTExt. Then τ has the form $[\overline{\alpha} \mapsto \overline{\tau}]Ct$, which is also of the form $(\overline{\tau_1}\ C')$.

□

Lemma 17 If $(\overline{\tau}\ C) \leq (\overline{\tau_1}\ C')$, then $\overline{\tau} = \overline{\tau_1}$.

Proof By (strong) induction on the depth of the derivation of $(\overline{\tau}\ C) \leq (\overline{\tau_1}\ C')$. Case analysis of the last rule used in the derivation.

- Case SubTRef. Then $(\overline{\tau}\ C) = (\overline{\tau_1}\ C')$, so $\overline{\tau} = \overline{\tau_1}$.

- Case SubTTrans. Then $(\overline{\tau}\ C) \leq \tau$ and $\tau \leq (\overline{\tau_1}\ C')$. By lemma 16, τ has the form $(\overline{\tau_2}\ C'')$. Then by induction we have $\overline{\tau} = \overline{\tau_2}$ and $\overline{\tau_2} = \overline{\tau_1}$, so $\overline{\tau} = \overline{\tau_1}$.

- Case SubTExt. Then $C = Sn.Cn$ and $(\overline{\tau_1}\ C') = [\overline{\alpha} \mapsto \overline{\tau}](\overline{\tau_2}\ C')$ and (<`abstract`> `class` $\overline{\alpha}$ $Cn(I_1 : \tau_1, \ldots, I_m : \tau_m)$ `extends` $(\overline{\tau_2}\ C')\ \ldots) \in SigT(Sn)$. By ClassOK, we have $\overline{\tau_2} = \overline{\alpha}$. Therefore $(\overline{\tau_1}\ C') = [\overline{\alpha} \mapsto \overline{\tau}](\overline{\alpha}\ C') = (\overline{\tau}\ C')$. Therefore $\overline{\tau} = \overline{\tau_1}$.

□

Lemma 18 If $(\overline{\tau}\ C) \leq (\overline{\tau_1}\ C')$ then $C \leq C'$.

Proof By (strong) induction on the depth of the derivation of $(\overline{\tau}\ C) \leq (\overline{\tau_1}\ C')$. Case analysis of the last rule used in the derivation.

- Case SubTRef. Then $(\overline{\tau}\ C) = (\overline{\tau_1}\ C')$, so $C = C'$. Then the result holds by SubRef.

- Case SubTTrans. Then $(\overline{\tau}\ C) \leq \tau$ and $\tau \leq (\overline{\tau_1}\ C')$. By lemma 16 τ has the form $(\overline{\tau_2}\ C'')$. Then by induction we have that $C \leq C''$ and $C'' \leq C'$. Therefore the result follows by SubTrans.

- Case SubTExt. Then $C = Sn.Cn$ and (<`abstract`> `class` $\overline{\alpha}$ $Cn(\overline{I_0} : \overline{\tau_0})$ `extends` $(\overline{\tau_2}\ C')\ \ldots) \in SigT(Sn)$. Then the result follows by SubExt.

□

Lemma 19 If $C_1 \leq C_2$ and $\overline{\alpha} \vdash (\overline{\tau}\ C_1)$ OK then (1) $(\overline{\tau}\ C_1) \leq (\overline{\tau}\ C_2)$; and (2) $\overline{\alpha} \vdash (\overline{\tau}\ C_2)$ OK.

Proof By (strong) induction on the depth of the derivation of $C_1 \leq C_2$. Case analysis of the last rule used in the derivation.

- Case SubRef. Then $C_1 = C_2$. Then condition 1 follows from SubTRef, and condition 2 follows by assumption.

- Case SubTrans. Then $C_1 \leq C_3$ and $C_3 \leq C_2$. By induction we have $(\overline{\tau}\ C_1) \leq (\overline{\tau}\ C_3)$ and $\overline{\alpha} \vdash (\overline{\tau}\ C_3)$ OK. Then by induction again we have $(\overline{\tau}\ C_3) \leq (\overline{\tau}\ C_2)$ and $\overline{\alpha} \vdash (\overline{\tau}\ C_2)$ OK. Therefore condition 2 is shown, and condition 1 follows from SubTTrans.

- Case SubExt. Then $C_1 = Sn.Cn$ and (<`abstract`> `class` $\overline{\alpha_0}$ $Cn(\overline{I_0} : \overline{\tau_0})$ `extends` $(\overline{\tau_2}\ C_2) \ldots) \in ST(Sn)$. Then by ClassOK we have $\overline{\tau_2} = \overline{\alpha_0}$. Since $\overline{\alpha} \vdash (\overline{\tau}\ C_1)$ OK, by ClassTypeOK we have $|\overline{\alpha_0}| = |\overline{\tau}|$ and $\overline{\alpha} \vdash \overline{\tau}$ OK. Therefore by SubTExt we have $(\overline{\tau}\ C_1) \leq [\overline{\alpha_0} \mapsto \overline{\tau}](\overline{\alpha_0}\ C_2)$. Since $[\overline{\alpha_0} \mapsto \overline{\tau}](\overline{\alpha_0}\ C_2) = (\overline{\tau}\ C_2)$, condition 1 is shown. Also by ClassOK $\Gamma; \overline{\alpha_0} \vdash (\overline{\alpha_0}\ C_2)(\overline{E})$ OK, for some Γ and $\overline{E}$, so by T-Constr we have $\overline{\alpha_0} \vdash (\overline{\alpha_0}\ C_2)$ OK. Therefore by lemma 24 we have $\overline{\alpha} \vdash (\overline{\tau}\ C_2)$ OK, so condition 2 is shown.

□

Lemma 20 If $\tau \leq \tau_1 \to \tau_2$, then τ has the form $\tau_1' \to \tau_2'$, where $\tau_1 \leq \tau_1'$ and $\tau_2' \leq \tau_2$.

Proof By (strong) induction on the depth of the derivation of $\tau \leq \tau_1 \to \tau_2$. Case analysis on the last rule used in the derivation.

- Case SubTRef. Therefore $\tau = \tau_1 \to \tau_2$, and by SubTRef we have $\tau_1 \leq \tau_1$ and $\tau_2 \leq \tau_2$.

- Case SubTTrans. Therefore $\tau \leq \tau'$ and $\tau' \leq \tau_1 \to \tau_2$. By induction τ' has the form $\tau_1'' \to \tau_2''$, where $\tau_1 \leq \tau_1''$ and $\tau_2'' \leq \tau_2$. Therefore, again by induction τ has the form $\tau_1' \to \tau_2'$, where $\tau_1'' \leq \tau_1'$ and $\tau_2' \leq \tau_2''$. By SubTTrans we have $\tau_1 \leq \tau_1'$ and $\tau_2' \leq \tau_2$.

- Case SubTFun. Then τ has the form $\tau_1' \to \tau_2'$, where $\tau_1 \leq \tau_1'$ and $\tau_2' \leq \tau_2$.

□

Lemma 21 If $\tau_1 \to \tau_2 \leq \tau$, then τ has the form $\tau_1' \to \tau_2'$.

Proof By (strong) induction on the depth of the derivation of $\tau_1 \to \tau_2 \leq \tau$. Case analysis of the last rule used in the derivation.

- Case SubTRef. Then $\tau = \tau_1 \to \tau_2$.
- Case SubTTrans. Then $\tau_1 \to \tau_2 \leq \tau'$ and $\tau' \leq \tau$. By induction τ' has the form $\tau_1'' \to \tau_2''$. Then by induction again, τ has the form $\tau_1' \to \tau_2'$.
- Case SubTFun. Then τ has the form $\tau_1' \to \tau_2'$.

□

Lemma 22 If $\tau \leq \tau_1 * \cdots * \tau_k$, then τ has the form $\tau_1' * \cdots * \tau_k'$, where for all $1 \leq i \leq k$ we have $\tau_i' \leq \tau_i$.

Proof By (strong) induction on the depth of the derivation of $\tau \leq \tau_1 * \cdots * \tau_k$. Case analysis of the last rule used in the derivation.

- Case SubTRef. Then $\tau = \tau_1 * \cdots * \tau_k$. By SubTRef, for all $1 \leq i \leq k$ we have $\tau_i \leq \tau_i$, so the result follows.
- Case SubTTrans. Then $\tau \leq \tau'$ and $\tau' \leq \tau_1 * \cdots * \tau_k$. By induction τ' has the form $\tau_1'' * \cdots * \tau_k''$, where for all $1 \leq i \leq k$ we have $\tau_i'' \leq \tau_i$. Then by induction again, τ has the form $\tau_1' * \cdots * \tau_k'$, where for all $1 \leq i \leq k$ we have $\tau_i' \leq \tau_i''$. Then by SubTTrans, for all $1 \leq i \leq k$ we have $\tau_i' \leq \tau_i$.
- Case SubTTup. Then τ has the form $\tau_1' * \cdots * \tau_k'$, where for all $1 \leq i \leq k$ we have $\tau_i' \leq \tau_i$.

□

Lemma 23 If $\tau_1 * \cdots * \tau_k \leq \tau$, then τ has the form $\tau_1' * \cdots * \tau_k'$, where for all $1 \leq i \leq k$ we have $\tau_i \leq \tau_i'$.

Proof By (strong) induction on the depth of the derivation of $\tau_1 * \cdots * \tau_k \leq \tau$. Case analysis of the last rule used in the derivation.

- Case SubTRef. Then $\tau = \tau_1 * \cdots * \tau_k$. By SubTRef, for all $1 \leq i \leq k$ we have $\tau_i \leq \tau_i$.

- Case SubTTrans. Then $\tau_1 * \cdots * \tau_k \leq \tau'$ and $\tau' \leq \tau$. By induction τ' has the form $\tau''_1 * \cdots * \tau''_k$, where for all $1 \leq i \leq k$ we have $\tau_i \leq \tau''_i$. Then by induction again, τ has the form $\tau'_1 * \cdots * \tau'_k$, where for all $1 \leq i \leq k$ we have $\tau''_i \leq \tau'_i$. By SubTTrans, for all $1 \leq i \leq k$ we have $\tau_i \leq \tau'_i$.

- Case SubTTup. Then τ has the form $\tau'_1 * \cdots * \tau'_k$, where for all $1 \leq i \leq k$ we have $\tau_i \leq \tau'_i$.

□

A.3.3 Type Substitution

Lemma 24 If $\overline{\alpha} \vdash \tau$ OK and $|\overline{\alpha}| = |\overline{\tau}|$ and $\overline{\alpha'} \vdash \overline{\tau}$ OK, then $\overline{\alpha'} \vdash [\overline{\alpha} \mapsto \overline{\tau}]\tau$ OK.

Proof By (strong) induction on the depth of the derivation of $\overline{\alpha} \vdash \tau$ OK. Case analysis on the last rule used in the derivation. For TVarOK, τ has the form α and the premise ensures that $\alpha \in \overline{\alpha}$. Therefore $[\overline{\alpha} \mapsto \overline{\tau}]\tau$ is some τ_0 in $\overline{\tau}$. By assumption $\overline{\alpha'} \vdash \tau_0$ OK so the result follows. All other cases are proven by induction. □

Lemma 25 If $\tau \leq \tau'$ and $|\overline{\alpha}| = |\overline{\tau}|$, then $[\overline{\alpha} \mapsto \overline{\tau}]\tau \leq [\overline{\alpha} \mapsto \overline{\tau}]\tau'$.

Proof By (strong) induction on the depth of the derivation of $\tau \leq \tau'$. Case analysis of the last rule used in the derivation.

- Case SubTRef. Then $\tau = \tau'$, so $[\overline{\alpha} \mapsto \overline{\tau}]\tau = [\overline{\alpha} \mapsto \overline{\tau}]\tau'$ and the result follows by SubTRef.

- Case SubTTrans. Then $\tau \leq \tau''$ and $\tau'' \leq \tau'$. By induction we have $[\overline{\alpha} \mapsto \overline{\tau}]\tau \leq [\overline{\alpha} \mapsto \overline{\tau}]\tau''$ and $[\overline{\alpha} \mapsto \overline{\tau}]\tau'' \leq [\overline{\alpha} \mapsto \overline{\tau}]\tau'$, and the result follows by SubTTrans.

- Case SubTExt. Then τ has the form $\overline{\tau_0}\ Sn.Cn$ and τ' has the form $[\overline{\alpha_0} \mapsto \overline{\tau_0}]Ct$ and (<`abstract`> `class` $\overline{\alpha_0}$ $Cn(\ldots)$ `extends` Ct $\ldots$) $\in SigT(Sn)$. Then by SubTExt

we have $([\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_0})\ Sn.Cn \leq [\overline{\alpha_0} \mapsto [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_0}]Ct$. By ClassOK we have $\Gamma; \overline{\alpha_0} \vdash Ct(\overline{E})$ OK for some Γ and $\overline{E}$, so by T-Constr $\overline{\alpha_0} \vdash Ct$ OK. Therefore, by lemma 11 all type variables in Ct are in $\overline{\alpha_0}$. Therefore $[\overline{\alpha_0} \mapsto [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_0}]Ct$ is equivalent to $[\overline{\alpha} \mapsto \overline{\tau}][\overline{\alpha_0} \mapsto \overline{\tau_0}]Ct$. Since $([\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_0})\ Sn.Cn$ is equivalent to $[\overline{\alpha} \mapsto \overline{\tau}](\overline{\tau_0}\ Sn.Cn)$, the result follows.

- Case SubTFun. Then τ has the form $\tau_1 \to \tau_2$ and τ' has the form $\tau_1' \to \tau_2'$ and $\tau_1' \leq \tau_1$ and $\tau_2 \leq \tau_2'$. By induction we have $[\overline{\alpha} \mapsto \overline{\tau}]\tau_1' \leq [\overline{\alpha} \mapsto \overline{\tau}]\tau_1$ and $[\overline{\alpha} \mapsto \overline{\tau}]\tau_2 \leq [\overline{\alpha} \mapsto \overline{\tau}]\tau_2'$, and the result follows by SubTFun.

- Case SubTTup. Then τ has the form $\tau_1 * \cdots * \tau_k$ and τ' has the form $\tau_1' * \cdots * \tau_k'$ and for all $1 \leq i \leq k$ we have $\tau_i \leq \tau_i'$. By induction, for all $1 \leq i \leq k$ we have $[\overline{\alpha} \mapsto \overline{\tau}]\tau_i \leq [\overline{\alpha} \mapsto \overline{\tau}]\tau_i'$, and the result follows by SubTTup.

□

Lemma 26 If $\Gamma; \overline{\alpha} \vdash E : \tau$ and $|\overline{\alpha}| = |\overline{\tau}|$ and $\overline{\alpha_0} \vdash \overline{\tau}$ OK, then $[\overline{\alpha} \mapsto \overline{\tau}]\Gamma; \overline{\alpha_0} \vdash [\overline{\alpha} \mapsto \overline{\tau}]E : [\overline{\alpha} \mapsto \overline{\tau}]\tau$.

Proof By (strong) induction on the depth of the derivation of $\Gamma; \overline{\alpha} \vdash E : \tau$. Case analysis of the last rule used in the derivation.

- Case T-Id. Then $E = I$ and $(I, \tau) \in \Gamma$. Therefore, $(I, [\overline{\alpha} \mapsto \overline{\tau}]\tau) \in [\overline{\alpha} \mapsto \overline{\tau}]\Gamma$. Also, $I = [\overline{\alpha} \mapsto \overline{\tau}]I$. So by T-Id we have $[\overline{\alpha} \mapsto \overline{\tau}]\Gamma; \overline{\alpha_0} \vdash [\overline{\alpha} \mapsto \overline{\tau}]E : [\overline{\alpha} \mapsto \overline{\tau}]\tau$.

- Case T-New. Then $E = Ct(\overline{E})$ and $\tau = Ct$ and $\Gamma; \overline{\alpha} \vdash Ct(\overline{E})$ OK and $Ct = (\overline{\tau_1}\ Sn.Cn)$ and concrete($Sn.Cn$). By T-Constr we have $\overline{\alpha} \vdash Ct$ OK and (<`abstract`> `class` $\overline{\alpha_1}\ Cn(\overline{I_0} : \overline{\tau_0})\ \ldots) \in SigT(Sn)$ and $\Gamma; \overline{\alpha} \vdash \overline{E} : \overline{\tau_0'}$ and $\overline{\tau_0'} \leq [\overline{\alpha_1} \mapsto \overline{\tau_1}]\overline{\tau_0}$. By lemma 24 we have $\overline{\alpha_0} \vdash [\overline{\alpha} \mapsto \overline{\tau}]Ct$ OK. Since $Ct = (\overline{\tau_1}\ Sn.Cn)$ we have $[\overline{\alpha} \mapsto \overline{\tau}]Ct = [\overline{\alpha} \mapsto \overline{\tau}](\overline{\tau_1}\ Sn.Cn) = ([\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_1}\ Sn.Cn)$. By induction we have $[\overline{\alpha} \mapsto \overline{\tau}]\Gamma; \overline{\alpha_0} \vdash [\overline{\alpha} \mapsto \overline{\tau}]\overline{E} : [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_0'}$. By lemma 25 we have $[\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_0'} \leq [\overline{\alpha} \mapsto \overline{\tau}][\overline{\alpha_1} \mapsto \overline{\tau_1}]\overline{\tau_0}$. By ClassOK we have $\overline{\alpha_1} \vdash \overline{\tau_0}$ OK, so by lemma 11 all type variables in $\overline{\tau_0}$ are in $\overline{\alpha_1}$. Therefore $[\overline{\alpha} \mapsto \overline{\tau}][\overline{\alpha_1} \mapsto \overline{\tau_1}]\overline{\tau_0}$ is equivalent to $[\overline{\alpha_1} \mapsto [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_1}]\overline{\tau_0}$. Therefore by T-Constr we have $[\overline{\alpha} \mapsto \overline{\tau}]\Gamma; \overline{\alpha_0} \vdash [\overline{\alpha} \mapsto \overline{\tau}]Ct(\overline{E})$ OK, and the result follows by T-New.

- Case T-Rep. Then $E = Ct\{\overline{V} = \overline{E}\}$ and $\tau = Ct$ and $\overline{\alpha} \vdash Ct$ OK and $Ct = (\overline{\tau_1}\ Sn.Cn)$ and concrete($Sn.Cn$) and repType(Ct) = $\{\overline{V_0} : \overline{\tau_0}\}$ and $\Gamma; \overline{\alpha} \vdash \overline{E} : \overline{\tau'_0}$ and $\overline{\tau'_0} \leq \overline{\tau_0}$. By lemma 24 we have $\overline{\alpha_0} \vdash [\overline{\alpha} \mapsto \overline{\tau}]Ct$ OK. Since $Ct = (\overline{\tau_1}\ Sn.Cn)$ we have $[\overline{\alpha} \mapsto \overline{\tau}]Ct = [\overline{\alpha} \mapsto \overline{\tau}](\overline{\tau_1}\ Sn.Cn) = ([\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_1}\ Sn.Cn)$. By lemma 27 we have repType($[\overline{\alpha} \mapsto \overline{\tau}]Ct$) = $[\overline{\alpha} \mapsto \overline{\tau}]\{\overline{V_0} : \overline{\tau_0}\}$. By induction we have $[\overline{\alpha} \mapsto \overline{\tau}]\Gamma; \overline{\alpha_0} \vdash [\overline{\alpha} \mapsto \overline{\tau}]\overline{E} : [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau'_0}$. By lemma 25 we have $[\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau'_0} \leq [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_0}$. Therefore by T-Rep the result follows.

- Case T-Fun. Then $E = \overline{\tau_1}\ Sn.Fn$ and $\tau = [\overline{\alpha_1} \mapsto \overline{\tau_1}](\hat{Mt} \to \tau')$ and $\overline{\alpha} \vdash \overline{\tau_1}$ OK and (`fun` $\overline{\alpha_1}\ Fn : Mt \to \tau'$) $\in SigT(Sn)$. By lemma 24 we have $\overline{\alpha_0} \vdash [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_1}$ OK. Therefore by T-Fun we have $[\overline{\alpha} \mapsto \overline{\tau}]\Gamma; \overline{\alpha_0} \vdash [\overline{\alpha} \mapsto \overline{\tau}](\overline{\tau_1}\ Sn.Fn) : [\overline{\alpha} \mapsto \overline{\tau}][\overline{\alpha_1} \mapsto \overline{\tau_1}](\hat{Mt} \to \tau')$. By FunOK we have $\overline{\alpha_1} \vdash \hat{Mt}$ OK and $\overline{\alpha_1} \vdash \tau'$ OK. Therefore by lemma 11 all type variables in $\hat{Mt}$ and τ' are in $\overline{\alpha_1}$. Therefore, $[\overline{\alpha} \mapsto \overline{\tau}][\overline{\alpha_1} \mapsto \overline{\tau_1}](\hat{Mt} \to \tau')$ is equivalent to $[\overline{\alpha_1} \mapsto [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_1}](\hat{Mt} \to \tau')$, so the result follows.

- Case T-Tup. Then $E = (E_1, \ldots, E_k)$ and $\tau = \tau_1 * \cdots * \tau_k$ and for all $1 \leq i \leq k$ we have $\Gamma; \overline{\alpha} \vdash E_i : \tau_i$. Therefore by induction, for all $1 \leq i \leq k$ we have $[\overline{\alpha} \mapsto \overline{\tau}]\Gamma; \overline{\alpha_0} \vdash [\overline{\alpha} \mapsto \overline{\tau}]E_i : [\overline{\alpha} \mapsto \overline{\tau}]\tau_i$, and the result follows by T-Tup.

- Case T-App. Then $E = E_1\ E_2$ and $\Gamma; \overline{\alpha} \vdash E_1 : \tau_2 \to \tau$ and $\Gamma; \overline{\alpha} \vdash E_2 : \tau'_2$ and $\tau'_2 \leq \tau_2$. By induction we have $[\overline{\alpha} \mapsto \overline{\tau}]\Gamma; \overline{\alpha_0} \vdash [\overline{\alpha} \mapsto \overline{\tau}]E_1 : [\overline{\alpha} \mapsto \overline{\tau}](\tau_2 \to \tau)$ and $[\overline{\alpha} \mapsto \overline{\tau}]\Gamma; \overline{\alpha_0} \vdash [\overline{\alpha} \mapsto \overline{\tau}]E_2 : [\overline{\alpha} \mapsto \overline{\tau}]\tau'_2$. By lemma 25 we have $[\overline{\alpha} \mapsto \overline{\tau}]\tau'_2 \leq [\overline{\alpha} \mapsto \overline{\tau}]\tau_2$, so the result follows by T-App.

□

Lemma 27 If repType(Ct) = $\{\overline{V} : \overline{\tau}\}$ and $|\overline{\alpha}| = |\overline{\tau}|$, then repType($[\overline{\alpha} \mapsto \overline{\tau}]Ct$) = $[\overline{\alpha} \mapsto \overline{\tau}]\{\overline{V} : \overline{\tau}\}$.

Proof By induction on the depth of the derivation of repType(Ct) = $\{\overline{V} : \overline{\tau}\}$. By RepType, $Ct = (\overline{\tau_0}\ Sn.Cn)$ and $\{\overline{V} : \overline{\tau}\} = [\overline{\alpha_0} \mapsto \overline{\tau_0}]\{ < \overline{V_1} : \overline{\tau_1}, > Sn.\overline{Vn} : \overline{\tau_2}\}$ and (<<`abstract`>> `class` $\overline{\alpha_0}\ Cn(\ldots)$ <`extends` Ct' > `of` $\{\overline{Vn} : \overline{\tau_2}\}$) $\in SigT(Sn)$ and < repType(Ct') = $\{\overline{V_1} : \overline{\tau_1}\}$ >. Therefore by RepType we have repType($[\overline{\alpha} \mapsto \overline{\tau}](\overline{\tau_0}\ Sn.Cn)$) = $[\overline{\alpha_0} \mapsto [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_0}]\{ < \overline{V_1} : \overline{\tau_1}, > Sn.\overline{Vn} : \overline{\tau_2}\}$. By ClassOK we have

$< \Gamma; \overline{\alpha_0} \vdash C t'(\overline{E})$ OK $>$ for some Γ and $\overline{E}$, so by T-ConstrOK we have $< \overline{\alpha_0} \vdash C t'$ OK $>$. Then by lemma 38 we have $< \overline{\alpha_0} \vdash \overline{\tau_1}$ OK $>$, so by lemma 11 all type variables $\overline{\tau_1}$ are in $\overline{\alpha_0}$. Also by ClassOK we have $\overline{\alpha_0} \vdash \overline{\tau_2}$ OK, so by lemma 11 all type variables in $\overline{\tau_2}$ are in $\overline{\alpha_0}$. Therefore $[\overline{\alpha_0} \mapsto [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_0}]\{\overline{V_1} : \overline{\tau_1}, Sn.\overline{Vn} : \overline{\tau_2}\}$ is equivalent to $[\overline{\alpha} \mapsto \overline{\tau}][\overline{\alpha_0} \mapsto \overline{\tau_0}]\{\overline{V_1} : \overline{\tau_1}, Sn.\overline{Vn} : \overline{\tau_2}\}$, so the result follows. □

Lemma 28 If $\text{matchType}(\tau, Pat) = (\Gamma, \tau')$ and $|\overline{\alpha}| = |\overline{\tau}|$, then $\text{matchType}([\overline{\alpha} \mapsto \overline{\tau}]\tau, Pat) = ([\overline{\alpha} \mapsto \overline{\tau}]\Gamma, [\overline{\alpha} \mapsto \overline{\tau}]\tau')$.

Proof By (strong) induction on the depth of the derivation of $\text{matchType}(\tau, Pat) = (\Gamma, \tau')$. Case analysis of the last rule used in the derivation.

- Case T-MatchWild. Then Pat has the form $\_$ and $\Gamma = \{\}$ and $\tau' = \tau$. Then $[\overline{\alpha} \mapsto \overline{\tau}]\tau = [\overline{\alpha} \mapsto \overline{\tau}]\tau'$ and $[\overline{\alpha} \mapsto \overline{\tau}]\Gamma = \{\}$, so the result follows by T-MatchWild.

- Case T-MatchBind. Then Pat has the form I `as` Pat' and $\Gamma = \Gamma' \cup \{(I, \tau')\}$ and $\text{matchType}(\tau, Pat') = (\Gamma', \tau')$. By induction we have $\text{matchType}([\overline{\alpha} \mapsto \overline{\tau}]\tau, Pat') = ([\overline{\alpha} \mapsto \overline{\tau}]\Gamma', [\overline{\alpha} \mapsto \overline{\tau}]\tau')$. Therefore by T-MatchBind we have $\text{matchType}([\overline{\alpha} \mapsto \overline{\tau}]\tau, (I \texttt{ as } Pat') = [\overline{\alpha} \mapsto \overline{\tau}]\Gamma' \cup \{(I, [\overline{\alpha} \mapsto \overline{\tau}]\tau')\}, [\overline{\alpha} \mapsto \overline{\tau}]\tau')$, and the result follows.

- Case T-MatchTup. Then $\tau = \tau_1 * \cdots * \tau_k$ and Pat has the form $(Pat_1, \ldots, Pat_k)$ and $\Gamma = \Gamma_1 \cup \ldots \cup \Gamma_k$ and $\tau' = \tau'_1 * \cdots * \tau'_k$ and for all $1 \leq i \leq k$ we have $\text{matchType}(\tau_i, Pat_i) = (\Gamma_i, \tau'_i)$. By induction, for all $1 \leq i \leq k$ we have $\text{matchType}([\overline{\alpha} \mapsto \overline{\tau}]\tau_i, Pat_i) = ([\overline{\alpha} \mapsto \overline{\tau}]\Gamma_i, [\overline{\alpha} \mapsto \overline{\tau}]\tau'_i)$. Therefore, the result follows by T-MatchTup.

- Case T-MatchClass. Then Pat has the form $C\{\overline{V} = \overline{Pat}\}$ and $\tau = (\overline{\tau_1}\ C')$ and $\tau' = (\overline{\tau_1}\ C)$ and $\Gamma = \bigcup \overline{\Gamma}$ and $C \leq C'$ and $\text{repType}(\overline{\tau_1}\ C) = \{\overline{V} : \overline{\tau}\}$ and $\text{matchType}(\overline{\tau}, \overline{Pat}) = (\overline{\Gamma}, \overline{\tau'})$. By lemma 27 we have $\text{repType}([\overline{\alpha} \mapsto \overline{\tau}](\overline{\tau_1}\ C)) = [\overline{\alpha} \mapsto \overline{\tau}]\{\overline{V} : \overline{\tau}\}$. By induction we have $\text{matchType}([\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau}, \overline{Pat}) = ([\overline{\alpha} \mapsto \overline{\tau}]\overline{\Gamma}, [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau'})$. Therefore the result follows by T-MatchClass.

□

A.3.4 Pattern Matching, Specificity, and Intersection

Lemma 29 If $\text{match}(v, Pat) = \rho$ and $Pat \leq Pat'$, then there exists ρ' such that $\text{match}(v, Pat') = \rho'$.

Proof By induction on the depth of the derivation of $Pat \leq Pat'$. Case analysis of the last rule used in the derivation:

- Case SpecWild. Then Pat' has the form _, so by E-MatchWild we have match(v, _) = {}.

- Case SpecBind1.: Then Pat has the form (I `as` Pat_1) and we have $Pat_1 \leq Pat'$. Since we're given that match(v, I `as` Pat_1) = ρ, by E-MatchBind we also have that match(v, Pat_1) = $\rho - \{(I, v)\}$. Therefore by induction there exists ρ' such that $\text{match}(v, Pat') = \rho'$.

- Case SpecBind2.: Then Pat' has the form (I `as` Pat_2) and we have $Pat \leq Pat_2$. Therefore by induction there exists ρ'' such that $\text{match}(v, Pat_2) = \rho''$. Then by E-MatchBind we have match(v, I `as` Pat_2) = $\rho'' \cup \{I, v\}$.

- Case SpecTup. Then Pat has the form $(\overline{Pat})$ and Pat' has the form $(\overline{Pat'})$ and $\overline{Pat} \leq \overline{Pat'}$. Since we're given that match(v,$(\overline{Pat})$) = ρ, by E-MatchTup we have that $v = (\overline{v})$ and $\text{match}(\overline{v}, \overline{Pat}) = \overline{\rho}$. Therefore by induction there exists $\overline{\rho'}$ such that $\text{match}(\overline{v}, \overline{Pat'}) = \overline{\rho'}$. Then by E-MatchTup we have $\text{match}((\overline{v}), (\overline{Pat})) = \bigcup \overline{\rho'}$.

- Case SpecClass. Then Pat has the form $(C_1\{\overline{V} = \overline{Pat_1}, \overline{V_3} = \overline{Pat_3}\})$ and Pat' has the form $(C_2\{\overline{V} = \overline{Pat_2}\})$ and $C_1 \leq C_2$ and $\overline{Pat_1} \leq \overline{Pat_2}$. Since we're given that match(v,$C_1\{\overline{V} = \overline{Pat_1}, \overline{V_3} = \overline{Pat_3}\}$) = ρ, by E-MatchClass we have that $v = ((\overline{\tau}\ C_0)\{\overline{V} = \overline{v}, \overline{V_3} = \overline{v_3}, \overline{V_4} = \overline{v_4}\})$ and $C_0 \leq C_1$ and $\text{match}(\overline{v}, \overline{Pat_1}) = \overline{\rho_1}$, where $\rho = \bigcup \overline{\rho_1}$. Since $C_0 \leq C_1$ and $C_1 \leq C_2$, by SubTrans we have $C_0 \leq C_2$. By induction there exists $\overline{\rho_2}$ such that $\text{match}(\overline{v}, \overline{Pat_2}) = \overline{\rho_2}$. Therefore by E-MatchClass we have $\text{match}((\overline{\tau}\ C_0)\{\overline{V} = \overline{v}, \overline{V_3} = \overline{v_3}, \overline{V_4} = \overline{v_4}\}), C_2\{\overline{V} = \overline{Pat_2}\}) = \bigcup \overline{\rho_2}$.

□

Lemma 30 $Pat \leq Pat$

Proof By (strong) induction on the "depth" of Pat (i.e. the depth of its AST representation). Case analysis of the form of Pat.

- Case Pat has the form $\_$. Then by SpecWild, $Pat \leq Pat$.
- Case Pat has the form I `as` Pat'. By induction $Pat' \leq Pat'$. Then by SpecBind1 $Pat \leq Pat'$, and by SpecBind2 $Pat \leq Pat$.
- Case Pat has the form $C\{\overline{V} = \overline{Pat}\}$. By induction we have $\overline{Pat} \leq \overline{Pat}$ and by SubRef we have $C \leq C$, so the result follows from SpecClass.
- Case Pat has the form $(\overline{Pat})$. By induction we have $\overline{Pat} \leq \overline{Pat}$, so the result follows from SpecTup.

□

Lemma 31 If $Pat \leq Pat'$ and $Pat' \leq Pat''$ then $Pat \leq Pat''$.

Proof By induction on the sum of the depths of the derivations of $Pat \leq Pat'$ and $Pat' \leq Pat''$. There are several cases.

- Case SpecBind1 is the last rule used in the derivation of $Pat \leq Pat'$. Then Pat has the form (I `as` Pat_0) and we have $Pat_0 \leq Pat'$. By induction we have $Pat_0 \leq Pat''$, and by SpecBind1 also $Pat \leq Pat''$.
- Case SpecWild is the last rule used in the derivation of $Pat' \leq Pat''$. Then Pat'' has the form $\_$, and by SpecWild we have $Pat \leq Pat''$.
- Case SpecBind1 is the last rule used in the derivation of $Pat' \leq Pat''$ and SpecBind1 is not the last rule used in the derivation of $Pat \leq Pat'$. Then Pat' has the form (I' `as` Pat'_0) and we have $Pat'_0 \leq Pat''$. Since SpecBind1 is not the last rule used in the derivation of $Pat \leq Pat'$, the last rule in that derivation must be SpecBind2, so $Pat \leq Pat'_0$. Then by induction we have $Pat \leq Pat''$.

- Case SpecBind2 is the last rule used in the derivation of $Pat' \leq Pat''$. Then Pat'' has the form $(I''$ `as` $Pat''_0)$ and we have $Pat' \leq Pat''_0$. By induction $Pat \leq Pat''_0$, and by SpecBind2 $Pat \leq Pat''$.

- Case SpecTup is the last rule used in the derivation of $Pat' \leq Pat''$ and SpecBind1 is not the last rule used in the derivation of $Pat \leq Pat'$. Then Pat' has the form $(\overline{Pat'})$ and Pat'' has the form $(\overline{Pat''})$ and $\overline{Pat'} \leq \overline{Pat''}$. Since SpecBind1 is not the last rule used in the derivation of $Pat \leq Pat'$, the last rule in that derivation must be SpecTup, so Pat has the form $(\overline{Pat})$ and $\overline{Pat} \leq \overline{Pat'}$. By induction $\overline{Pat} \leq \overline{Pat''}$, and by SpecTup $Pat \leq Pat''$.

- Case SpecClass is the last rule used in the derivation of $Pat' \leq Pat''$ and SpecBind1 is not the last rule used in the derivation of $Pat \leq Pat'$. Then Pat' has the form $C'\{\overline{V_1} = \overline{Pat'_1}, \overline{V_2} = \overline{Pat'_2}\}$ and Pat'' has the form $C''\{\overline{V_1} = \overline{Pat''_1}\}$ and $C' \leq C''$ and $\overline{Pat'_1} \leq \overline{Pat''_1}$. Since SpecBind1 is not the last rule used in the derivation of $Pat \leq Pat'$, the last rule in that derivation must be SpecClass, so Pat has the form $C\{\overline{V_1} = \overline{Pat_1}, \overline{V_2} = \overline{Pat_2}, \overline{V_3} = \overline{Pat_3}\}$ and $C \leq C'$ and $\overline{Pat_1} \leq \overline{Pat'_1}$. By Subtrans $C \leq C''$, and by induction $\overline{Pat} \leq \overline{Pat''}$, so by SpecClass $Pat \leq Pat''$.

□

Lemma 32 If $Pat \leq Pat'$ then there exists some Pat_0 such that $Pat \cap Pat' = Pat_0$.

Proof By induction on the depth of the derivation of $Pat \leq Pat'$. Case analysis of the last rule used in the derivation.

- Case SpecWild. Then Pat' has the form _, so by PatIntWild we have $Pat' \cap Pat = Pat$, and the result follows from PatIntRev.

- Case SpecBind1. Then Pat has the form $(I$ `as` $Pat'')$ and $Pat'' \leq Pat'$. By induction there exists some Pat_0 such that $Pat'' \cap Pat' = Pat_0$, and the result follows from PatIntBind.

- Case SpecBind2. Then Pat' has the form $(I \texttt{ as } Pat'')$ and $Pat \leq Pat''$. By induction there exists some Pat_0 such that $Pat \cap Pat'' = Pat_0$. Then by PatIntRev we have $Pat'' \cap Pat = Pat_0$, by PatIntBind we have $Pat' \cap Pat = Pat_0$, and the result follows from PatIntRev.

- Case SpecTup. Then Pat has the form $(\overline{Pat})$ and Pat' has the form $(\overline{Pat'})$ and $\overline{Pat} \leq \overline{Pat'}$. By induction we have that there exists $\overline{Pat_0}$ such that $\overline{Pat} \cap \overline{Pat'} = \overline{Pat_0}$, and the result follows from PatIntTup.

- Case SpecClass. Then Pat has the form $C\{\overline{V_1} = \overline{Pat_1}, \overline{V_2} = \overline{Pat_2}\}$ and Pat' has the form $C'\{\overline{V_1} = \overline{Pat'_1}\}$ and $C \leq C'$ and $\overline{Pat_1} \leq \overline{Pat'_1}$. By induction there exists some $\overline{Pat_0}$ such that $\overline{Pat_1} \cap \overline{Pat'_1} = \overline{Pat_0}$, and the result follows from PatIntClass.

□

Lemma 33 If $Pat \cap Pat' = Pat_0$ then $Pat_0 \leq Pat$ and $Pat_0 \leq Pat'$.

Proof By induction on the depth of the derivation of $Pat \cap Pat' = Pat_0$. Case analysis of the last rule used in the derivation.

- Case PatIntWild. Then $Pat = \_$ and $Pat_0 = Pat'$. By SpecWild we have $Pat_0 \leq Pat$ and by lemma 30 we have $Pat_0 \leq Pat'$.

- Case PatIntBind. Then Pat has the form $(I \texttt{ as } Pat'')$ and $Pat'' \cap Pat' = Pat_0$. By induction we have that $Pat_0 \leq Pat''$ and $Pat_0 \leq Pat'$, so by SpecBind2 we also have $Pat_0 \leq Pat$.

- Case PatIntTup. Then Pat has the form $(\overline{Pat})$ and Pat' has the form $(\overline{Pat'})$ and Pat_0 has the form $(\overline{Pat_0})$ and $\overline{Pat} \cap \overline{Pat'} = \overline{Pat_0}$. By induction we have $\overline{Pat_0} \leq \overline{Pat}$ and $\overline{Pat_0} \leq \overline{Pat'}$, and the result follows from SpecTup.

- Case PatIntClass. Then Pat has the form $C\{\overline{V_1} = \overline{Pat_1}, \overline{V_2} = \overline{Pat_2}\}$ and Pat' has the form $C'\{\overline{V_1} = \overline{Pat'_1}\}$ and $C \leq C'$ and Pat_0 has the form $C\{\overline{V_1} = \overline{Pat_0}, \overline{V_2} = \overline{Pat_2}$ and $\overline{Pat_1} \cap \overline{Pat'_1} = \overline{Pat_0}$. By induction we have $\overline{Pat_0} \leq \overline{Pat_1}$ and $\overline{Pat_0} \leq \overline{Pat'_1}$. By

SpecClass we have $Pat_0 \leq Pat'$. By SubRef we have $C \leq C$, and by lemma 30 we have $\overline{Pat_2} \leq \overline{Pat_2}$, so by SpecClass we have $Pat_0 \leq Pat$.

- Case PatIntRev. Then $Pat' \cap Pat = Pat_0$, so by induction the result follows.

□

Lemma 34 If $\text{owner}(Mt, Pat') = C'$ and $\text{owner}(Mt, Pat'') = C''$ and $Pat' \cap Pat'' = Pat$, then either $C' \leq C''$ or $C'' \leq C'$.

Proof By induction on the depth of the derivation of $Pat' \cap Pat'' = Pat$. Case analysis of the last rule used in the derivation.

- Case PatIntWild. Then Pat' has the form _. But then it cannot be the case that $\text{owner}(Mt, Pat') = C'$, because none of the three associated rules applies to a wildcard pattern.

- Case PatIntBind. Then Pat' has the form I `as` Pat_0 and $Pat_0 \cap Pat'' = Pat$. Since $\text{owner}(Mt, Pat') = C'$, by OwnerBindPat we have $\text{owner}(Mt, Pat_0) = C'$. Therefore by induction we have that either $C' \leq C''$ or $C'' \leq C'$.

- Case PatIntTup. Then Pat' has the form $(Pat'_1, \ldots, Pat'_k)$ and Pat'' has the form $(Pat''_1, \ldots, Pat''_k)$ and for all $1 \leq j \leq k$ we have $Pat'_j \cap Pat''_j = Pat_j$. Since $\text{owner}(Mt, Pat') = C'$, by OwnerTupPat we have $Mt = \tau_1 * \cdots * \tau_{i-1} * Mt_i * \tau_{i+1} * \cdots * \tau_k$ and $\text{owner}(Mt_i, Pat'_i) = C'$. Since $\text{owner}(Mt, Pat'') = C''$, by OwnerTupPat we have $\text{owner}(Mt_i, Pat''_i) = C''$. Therefore by induction we have that either $C' \leq C''$ or $C'' \leq C'$.

- Case PatIntClass. Then Pat' has the form $(C_1\{\overline{V} = \overline{Pat'}, \overline{V_2} = \overline{Pat_2}\})$ and Pat'' has the form $(C_2\{\overline{V} = \overline{Pat''}\})$ and $C_1 \leq C_2$. Since $\text{owner}(Mt, Pat') = C'$, by OwnerClassPat $C' = C_1$. Since $\text{owner}(Mt, Pat'') = C''$, by OwnerClassPat $C'' = C_2$. Therefore $C' \leq C''$.

- Case PatIntRev. Then $Pat'' \cap Pat' = Pat$, so the result follows by induction.

□

Lemma 35 If $\vdash v : \tau$ and $\text{match}(v, Pat') = \rho'$ and $\text{match}(v, Pat'') = \rho''$ and $\text{matchType}(\tau', Pat') = (\Gamma', \tau'_0)$ and $\text{matchType}(\tau'', Pat'') = (\Gamma'', \tau''_0)$, then there exists some Pat such that $Pat' \cap Pat'' = Pat$.

Proof By induction on the sum of the depths of the derivations of $\text{match}(v, Pat') = \rho'$ and $\text{match}(v, Pat'') = \rho''$. There are several cases.

- Case E-MatchWild is the last rule used in the derivation of $\text{match}(v, Pat') = \rho'$. Then Pat' has the form $\_$, so by PatIntWild we have $Pat' \cap Pat'' = Pat''$.

- Case E-MatchWild is the last rule used in the derivation of $\text{match}(v, Pat'') = \rho''$. Then Pat'' has the form $\_$, so by PatIntWild we have $Pat'' \cap Pat' = Pat'$, and by PatIntRev $Pat' \cap Pat'' = Pat'$.

- Case E-MatchBind is the last rule used in the derivation of $\text{match}(v, Pat') = \rho'$. Then Pat' has the form I `as` Pat'_0 and $\text{match}(v, Pat'_0) = \rho'_0$, for some ρ'_0. Since $\text{matchType}(\tau', Pat') = (\Gamma', \tau'_0)$, by T-MatchBind we have $\text{matchType}(\tau', Pat'_0) = (\Gamma'_0, \tau'_0)$, for some Γ'_0. Then by induction there exists some Pat such that $Pat'_0 \cap Pat'' = Pat$, so by PatIntBind we have $Pat' \cap Pat'' = Pat$.

- Case E-MatchBind is the last rule used in the derivation of $\text{match}(v, Pat'') = \rho''$. Then Pat'' has the form I `as` Pat''_0 and $\text{match}(v, Pat''_0) = \rho''_0$, for some ρ''_0. Since $\text{matchType}(\tau'', Pat'') = (\Gamma'', \tau''_0)$, by T-MatchBind we have $\text{matchType}(\tau'', Pat''_0) = (\Gamma''_0, \tau''_0)$, for some Γ''_0. Then by induction there exists some Pat such that $Pat''_0 \cap Pat' = Pat$. By PatIntBind we have $Pat'' \cap Pat' = Pat$ and by PatIntRev we have $Pat' \cap Pat'' = Pat$.

- Case E-MatchTup is the last rule used in the derivation of $\text{match}(v, Pat') = \rho'$, and neither E-MatchWild nor E-MatchBind is the last rule used in the derivation of $\text{match}(v, Pat'') = \rho''$. Then $v = (v_1, \ldots, v_k)$ and Pat' has the form $(Pat'_1, \ldots, Pat'_k)$ and for all $1 \leq i \leq k$ we have $\text{match}(v_i, Pat'_i) = \rho'_i$, for some ρ'_i. Further, E-MatchTup

must be the last rule used in the derivation of match$(v, Pat'') = \rho''$. Then Pat'' has the form $(Pat''_1, \ldots, Pat''_k)$ and for all $1 \leq i \leq k$ we have match$(v_i, Pat''_i) = \rho''_i$, for some ρ''_i. Since $\vdash v : \tau$, by T-Tup we have $\tau = \tau_1 * \cdots * \tau_k$ and $\vdash v_i : \tau_i$ for all $1 \leq i \leq k$. Since matchType$(\tau', Pat') = (\Gamma', \tau'_0)$ and matchType$(\tau'', Pat'') = (\Gamma'', \tau''_0)$, by T-MatchTup we have $\tau' = \tau'_1 * \cdots * \tau'_k$ and $\tau'' = \tau''_1 * \cdots * \tau''_k$ and for all $1 \leq i \leq k$ matchType(τ'_i, Pat') $= (\Gamma'_i, \tau'''_i)$ and matchType$(\tau''_i, Pat'') = (\Gamma''_i, \tau''''_i)$. Then by induction, for all $1 \leq i \leq k$ there exists Pat_i such that $Pat'_i \cap Pat''_i = Pat_i$. Then by PatIntTup there exists Pat such that $Pat' \cap Pat'' = Pat$.

- Case E-MatchClass is the last rule used in the derivation of match$(v, Pat') = \rho'$, and neither E-MatchWild nor E-MatchBind is the last rule used in the derivation of match$(v, Pat'') = \rho''$. Then $v = ((\overline{\tau}\ C)\{V_1 = v_1, \ldots, V_k = v_k\})$ and Pat' has the form $(C'\{V_1 = Pat'_1, \ldots, V_m = Pat'_m\})$ and $C \leq C'$ and $m \leq k$ and for all $1 \leq i \leq m$ we have match$(v_i, Pat'_i) = \rho'_i$. Further, E-MatchClass must be the last rule used in the derivation of match$(v, Pat'') = \rho''$. Then Pat'' has the form $(C''\{V_1 = Pat''_1, \ldots, V_p = Pat''_p\})$ and $C \leq C''$ and $p \leq k$ and for all $1 \leq i \leq p$ we have match$(v_i, Pat''_i) = \rho''_i$.

  Since $\vdash v : \tau$, by T-Rep we have $\bullet \vdash (\overline{\tau}\ C)$ OK and for all $1 \leq i \leq k$, $\vdash v_i : \tau_i$. Since $C \leq C'$ and $C \leq C''$, by lemma 19 we have $\bullet \vdash (\overline{\tau}\ C')$ OK and $\bullet \vdash (\overline{\tau}\ C'')$ OK. Since matchType$(\tau', Pat') = (\Gamma', \tau'_0)$ and matchType$(\tau'', Pat'') = (\Gamma'', \tau''_0)$, by T-MatchClass we have repType$(\overline{\tau_0}\ C')$ has the form $\{V_1 : \tau'_1, \ldots, V_m : \tau'_m\}$ and repType$(\overline{\tau_1}\ C'')$ has the form $\{V_1 : \tau''_1, \ldots, V_p : \tau''_p\}$, for some $\overline{\tau_0}$ and $\overline{\tau_1}$. Therefore by RepType, repType$(\overline{\tau}\ C')$ has the form $\{V_1 : \tau'''_1, \ldots, V_m : \tau'''_m\}$ and repType$(\overline{\tau}\ C'')$ has the form $\{V_1 : \tau''''_1, \ldots, V_p : \tau''''_p\}$. Also by T-MatchClass, for all $1 \leq i \leq m$ we have matchType$(\tau'_i, Pat') = (\Gamma'_i, \tau'_{i,0})$ and for all $1 \leq i \leq p$ we have matchType$(\tau''_i, Pat'') = (\Gamma''_i, \tau''_{i,0})$. Since $C \leq C'$ and $C \leq C''$, by lemma 13 either $C' \leq C''$ or $C'' \leq C'$.

  - Case $C' \leq C''$. Since $\bullet \vdash (\overline{\tau}\ C')$ OK, by lemma 19 we have $(\overline{\tau}\ C') \leq (\overline{\tau}\ C'')$. By lemma 41 $p \leq m$. Then by induction, for all $1 \leq i \leq p$ there exists Pat_i such that $Pat'_i \cap Pat''_i = Pat_i$. Then by PatIntClass there exists Pat such that

$Pat' \cap Pat'' = Pat$.

– Case $C'' \leq C'$. By a symmetric argument as in the above case, there exists Pat such that $Pat'' \cap Pat' = Pat$, and the result follows by PatIntRev.

□

Lemma 36 If $\text{match}(v, Pat') = \rho'$ and $\text{match}(v, Pat'') = \rho''$ and $Pat' \cap Pat'' = Pat$, then there exists some ρ such that $\text{match}(v, Pat) = \rho$.

Proof By induction on the depth of the derivation of $Pat' \cap Pat'' = Pat$. Case analysis of the last rule used in the derivation.

- Case PatIntWild. Then $Pat = Pat''$, so $\text{match}(v, Pat) = \rho''$.

- Case PatIntBind. Then Pat' has the form I `as` Pat'_0 and $Pat'_0 \cap Pat'' = Pat$. Since $\text{match}(v, Pat') = \rho'$, by E-MatchBind there exists some ρ'_0 such that $\text{match}(v, Pat'_0)$ $= \rho'_0$. Therefore by induction there exists some ρ such that $\text{match}(v, Pat) = \rho$.

- Case PatIntTup. Then Pat' has the form $(\overline{Pat'})$ and Pat'' has the form $(\overline{Pat''})$ and Pat has the form $(\overline{Pat})$ and $\overline{Pat'} \cap \overline{Pat''} = \overline{Pat}$. Since $\text{match}(v, Pat') = \rho'$, by E-MatchTup $v = (\overline{v})$ and $\text{match}(\overline{v}, \overline{Pat'}) = \overline{\rho'}$. Since $\text{match}(v, Pat'') = \rho''$, by E-MatchTup $\text{match}(\overline{v}, \overline{Pat''}) = \overline{\rho''}$. Therefore by induction there exists $\overline{\rho}$ such that $\text{match}(\overline{v}, \overline{Pat}) = \overline{\rho}$. Then by E-MatchTup there exists ρ such that $\text{match}(v, Pat) = \rho$.

- Case PatIntClass. Then Pat' has the form $(C'\{\overline{V} = \overline{Pat_1}, \overline{V_3} = \overline{Pat_3}\})$ and Pat'' has the form $(C''\{\overline{V} = \overline{Pat_2}\})$ and Pat has the form $(C'\{\overline{V} = \overline{Pat}, \overline{V_3} = \overline{Pat_3}\})$ and $C' \leq C''$ and $\overline{Pat_1} \cap \overline{Pat_2} = \overline{Pat}$. Since $\text{match}(v, Pat') = \rho'$, by E-MatchClass $v = ((\overline{\tau}\ C)\{\overline{V} = \overline{v}, \overline{V_3} = \overline{v_3}, \overline{V_4} = \overline{v_4}\})$ and $C \leq C'$ and there exists $\overline{\rho_1}$ such that $\text{match}(\overline{v}, \overline{Pat_1}) = \overline{\rho_1}$ and there exists $\overline{\rho_3}$ such that $\text{match}(\overline{v_3}, \overline{Pat_3}) = \overline{\rho_3}$. Since $\text{match}(v, Pat'') = \rho''$, by E-MatchClass there exists $\overline{\rho_2}$ such that $\text{match}(\overline{v}, \overline{Pat_2})$ $= \overline{\rho_2}$. By induction, there exists $\overline{\rho}$ such that $\text{match}(\overline{v}, \overline{Pat}) = \overline{\rho}$. Then by E-MatchClass there exists ρ such that $\text{match}(v, Pat) = \rho$.

- Case PatIntRev. Then $Pat'' \cap Pat' = Pat$. Then by induction there exists ρ such that $\text{match}(v, Pat) = \rho$.

□

A.3.5 Representations and Representation Types

Lemma 37 If $\Gamma; \overline{\alpha} \vdash Ct(\overline{E_0})$ OK then there exist $\overline{V}$ and $\overline{E}$ such that $\text{rep}(Ct(\overline{E_0})) = \{\overline{V} = \overline{E}\}$.

Proof Since $\Gamma; \overline{\alpha} \vdash Ct(\overline{E_0})$ OK, by T-Constr we have $\overline{\alpha} \vdash Ct$ OK and $Ct = (\overline{\tau}\ Sn.Cn)$ and $(\texttt{<<abstract>>}\ \texttt{class}\ \overline{\alpha_0}\ Cn(\overline{I_0} : \overline{\tau_0})\ \texttt{<extends}\ Ct' > \texttt{of}\ \overline{Vn} : \overline{\tau_2}\}) \in SigT(Sn)$ and $\Gamma; \overline{\alpha} \vdash \overline{E_0} : \overline{\tau_0'}$ and $\overline{\tau_0'} \leq [\overline{\alpha_0} \mapsto \overline{\tau}]\overline{\tau_0}$. Therefore $|\overline{I_0}| = |\overline{\tau_0}|$ and $|\overline{E_0}| = |\overline{\tau_0'}|$ and $|\overline{\tau_0'}| = |\overline{\tau_0}|$, so also $|\overline{I_0}| = |\overline{E_0}|$. Since $\overline{\alpha} \vdash Ct$ OK, by ClassTypeOK we have $|\overline{\alpha_0}| = |\overline{\tau}|$.

This lemma is proven by induction on the number of nontrivial superclasses of $Sn.Cn$). There are two cases.

- $Sn.Cn$ has no nontrivial superclasses, so the `extends` clause in the declaration of $Sn.Cn$ is absent. Since $|\overline{I_0}| = |\overline{E_0}|$ and $|\overline{\alpha_0}| = |\overline{\tau}|$, the result follows by Rep.

- $Sn.Cn$ has $i > 0$ nontrivial superclasses, so the `extends` clause in the declaration of $Sn.Cn$ is present. Then by ClassOK we have $\Gamma'; \overline{\alpha_0} \vdash Ct'(\overline{E_0'})$ OK for some Γ' and $\overline{E_0'}$. Since the inheritance graph is assumed to be acyclic, Ct' has the form $(\overline{\tau_1}\ Sn'.Cn')$, where $Sn'.Cn'$ has $i-1$ nontrivial superclasses, so by induction there exist $\overline{V'}$ and $\overline{E'}$ such that $\text{rep}(Ct'(\overline{E'})) = \{\overline{V'} = \overline{E'}\}$. Since also $|\overline{I_0}| = |\overline{E_0}|$ and $|\overline{\alpha_0}| = |\overline{\tau}|$, the result follows by Rep.

□

Lemma 38 If $\overline{\alpha} \vdash Ct$ OK then there exist $\overline{V_0}$ and $\overline{\tau_0}$ such that $\text{repType}(Ct) = \{\overline{V_0} : \overline{\tau_0}\}$ and $\overline{\alpha} \vdash \overline{\tau_0}$ OK.

Proof Let $Ct = (\overline{\tau}\ Sn.Cn)$. By ClassTypeOK we have $\overline{\alpha} \vdash \overline{\tau}$ OK and $(\texttt{<<abstract>>}\ \texttt{class}\ \overline{\alpha_0}\ Cn(\ldots)\ \texttt{<extends}\ Ct' > \texttt{of}\ \{\overline{Vn} : \overline{\tau_2}\}) \in SigT(Sn)$ and $|\overline{\alpha_0}| = |\overline{\tau}|$. By ClassOK we have $\overline{\alpha_0} \vdash \overline{\tau_2}$ OK. Then by lemma 24 we have $\overline{\alpha} \vdash [\overline{\alpha_0} \mapsto \overline{\tau_0}]\overline{\tau_2}$ OK >.

This lemma is proven by induction on the number of nontrivial superclasses of $Sn.Cn$. There are two cases.

- $Sn.Cn$ has no nontrivial superclasses, so the `extends` clause in the declaration of $Sn.Cn$ is absent. Since $|\overline{\alpha_0}| = |\overline{\tau}|$ and $\overline{\alpha} \vdash [\overline{\alpha_0} \mapsto \overline{\tau_0}]\overline{\tau_2}$ OK, the result follows by RepType.

- $Sn.Cn$ has $i > 0$ nontrivial superclasses, so the `extends` clause in the declaration of $Sn.Cn$ is present. Then by ClassOK we have $\Gamma; \overline{\alpha_0} \vdash Ct'(\overline{E})$ OK for some Γ and $\overline{E}$, so by T-Constr we have $\overline{\alpha_0} \vdash Ct'$ OK. Since the inheritance graph is assumed to be acyclic, Ct' has the form $(\overline{\tau_1}\ Sn'.Cn')$, where $Sn'.Cn'$ has $i-1$ nontrivial superclasses, so by induction we have that repType(Ct') has the form $\{\overline{V_1} : \overline{\tau_1}\}$ and $\overline{\alpha_0} \vdash \overline{\tau_1}$ OK. Then by lemma 24 we have $\overline{\alpha} \vdash [\overline{\alpha_0} \mapsto \overline{\tau_0}]\overline{\tau_1}$ OK. Since $|\overline{\alpha_0}| = |\overline{\tau}|$ and $\overline{\alpha} \vdash [\overline{\alpha_0} \mapsto \overline{\tau_0}]\overline{\tau_2}$ OK and $\overline{\alpha} \vdash [\overline{\alpha_0} \mapsto \overline{\tau_0}]\overline{\tau_1}$ OK and repType(Ct') $= \{\overline{V_1} : \overline{\tau_1}\}$, the result follows by RepType.

□

Lemma 39 If rep($Ct(\overline{E})$) $= \{\overline{V_1} = \overline{E_1}\}$ and repType(Ct) $= \{\overline{V_2} : \overline{\tau_2}\}$ then $\overline{V_1} = \overline{V_2}$.

Proof By induction on the depth of the derivation of rep($Ct(\overline{E})$) $= \{\overline{V_1} = \overline{E_1}\}$. By Rep we have $Ct = (\overline{\tau}\ Sn.Cn)$ and (<<`abstract`>> `class` $\overline{\alpha}$ $Cn(\ldots)$ <`extends` $Ct'(\overline{E_0})$ > `of` $\{\overline{Vn} : \overline{\tau_2} = \overline{E_2}\}$) $\in ST(Sn)$ and <rep($Ct'(\overline{E_0})$) $= \{\overline{V_3} = \overline{E_3}\}$ > and $\overline{V_1}$ is equivalent to $< \overline{V_3}, > Sn.\overline{Vn}$. Since repType($Ct$) $= \{\overline{V_2} : \overline{\tau_2}\}$, by RepType we have <repType(Ct') $= \{\overline{V_4} : \overline{\tau_4}\}$ >, so by induction $< \overline{V_3} = \overline{V_4} >$. Then by RepType, $\overline{V_2}$ is equivalent to $< \overline{V_3}, > Sn.\overline{Vn}$. □

Lemma 40 If $\Gamma_0; \overline{\alpha_0} \vdash Ct(\overline{E})$ OK and rep($Ct(\overline{E})$) $= \{\overline{V_0} = \overline{E_0}\}$ and repType(Ct) $= \{\overline{V_0} : \overline{\tau_0}\}$, then $\Gamma_0; \overline{\alpha_0} \vdash \overline{E_0} : \overline{\tau_0'}$, for some $\overline{\tau_0'}$ such that $\overline{\tau_0'} \leq \overline{\tau_0}$.

Proof Since $\Gamma_0; \overline{\alpha_0} \vdash Ct(\overline{E})$ OK, by T-Constr we have $\overline{\alpha_0} \vdash Ct$ OK and $Ct = (\overline{\tau}\ Sn.Cn)$ and (<`abstract`> `class` $\overline{\alpha}$ $Cn(\overline{I_1} : \overline{\tau_1})$ $\ldots$) $\in SigT(Sn)$ and $\Gamma_0; \overline{\alpha_0} \vdash \overline{E} : \overline{\tau_1'}$ and $\overline{\tau_1'} \leq [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_1}$. Since $\overline{\alpha_0} \vdash Ct$ OK, by ClassTypeOK we have $\overline{\alpha_0} \vdash \overline{\tau}$ OK and $|\overline{\tau}| = |\overline{\alpha}|$. The lemma is proven by induction on the depth of the derivation of rep($Ct(\overline{E})$) $= \{\overline{V_0} = \overline{E_0}\}$.

By Rep we have (<<`abstract`>> `class` $\overline{\alpha}$ $Cn(\overline{I_1} : \overline{\tau_1})$ <`extends` $Ct'(\overline{E_1})$ > `of` $\{\overline{Vn} : \overline{\tau_2} = \overline{E_2}\}$) $\in ST(Sn)$ and <rep($Ct'(\overline{E_1})$) $= \{\overline{V_3} = \overline{E_3}\}$ > and $\{\overline{V_0} = \overline{E_0}\}$ is equivalent

to $[\overline{I_1} \mapsto \overline{E}][\overline{\alpha} \mapsto \overline{\tau}]\{ < \overline{V_3} = \overline{E_3}, > Sn.\overline{Vn} = \overline{E_2}\}$. Since repType($Ct$) = $\{\overline{V_0} : \overline{\tau_0}\}$, by RepType we have that <repType(Ct') = $\{\overline{V_3} : \overline{\tau_3}\}$ > and $\{\overline{V_0} : \overline{\tau_0}\}$ is equivalent to $[\overline{\alpha} \mapsto \overline{\tau}]\{ < \overline{V_3} : \overline{\tau_3}, > Sn.\overline{Vn} : \overline{\tau_2}\}$.

Let $\Gamma = \{(\overline{I_1}, \overline{\tau_1})\}$. By ClassOK we have $< \Gamma; \overline{\alpha} \vdash Ct'(\overline{E_1})$ OK >. Therefore by induction we have $< \Gamma; \overline{\alpha} \vdash \overline{E_3} : \overline{\tau_3'} >$ and $< \overline{\tau_3'} \leq \overline{\tau_3} >$. Also by ClassOK we have $\Gamma; \overline{\alpha} \vdash \overline{E_2} : \overline{\tau_2'}$ and $\overline{\tau_2'} \leq \overline{\tau_2}$. Then by lemmas 26 and 25 we have $< [\overline{\alpha} \mapsto \overline{\tau}]\Gamma; \overline{\alpha_0} \vdash [\overline{\alpha} \mapsto \overline{\tau}]\overline{E_3} : [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_3'} >$ and $< [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_3'} \leq [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_3} >$ and $[\overline{\alpha} \mapsto \overline{\tau}]\Gamma; \overline{\alpha_0} \vdash [\overline{\alpha} \mapsto \overline{\tau}]\overline{E_2} : [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_2'}$ and $[\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_2'} \leq [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_2}$. Then by lemma 9 we have $< \Gamma_0; \overline{\alpha_0} \vdash [\overline{I_1} \mapsto \overline{E}][\overline{\alpha} \mapsto \overline{\tau}]\overline{E_3} : \overline{\tau_3''} >$ and $< \overline{\tau_3''} \leq [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_3'} >$ and $\Gamma_0; \overline{\alpha_0} \vdash [\overline{I_1} \mapsto \overline{E}][\overline{\alpha} \mapsto \overline{\tau}]\overline{E_2} : \overline{\tau_2''}$ and $\overline{\tau_2''} \leq [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_2'}$. By SubTrans we have $< \overline{\tau_3''} \leq [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_3} >$ and $\overline{\tau_2''} \leq [\overline{\alpha} \mapsto \overline{\tau}]\overline{\tau_2}$. Therefore we have shown $\Gamma_0; \overline{\alpha_0} \vdash \overline{E_0} : \overline{\tau_0'}$ and $\overline{\tau_0'} \leq \overline{\tau_0}$. □

Lemma 41 If $\bullet \vdash Ct$ OK and $Ct \leq Ct'$ then repType(Ct) has the form $\{\overline{V_1} : \overline{\tau_1}, \overline{V_2} : \overline{\tau_2}\}$, where repType($Ct'$) = $\{\overline{V_1} : \overline{\tau_1}\}$.

Proof By induction on the depth of the derivation of $Ct \leq Ct'$. Case analysis of the last rule used in the derivation.

- Case SubTRef. Then $Ct = Ct'$. Since $\bullet \vdash Ct$ OK, by lemma 38 there exist $\overline{V}$ and $\overline{\tau}$ such that repType(Ct) = $\{\overline{V} : \overline{\tau}\}$. Therefore repType($Ct'$) = $\{\overline{V} : \overline{\tau}\}$ as well, so the result follows.

- Case SubTTrans. Then $Ct \leq \tau$ and $\tau \leq Ct'$. By lemma 16 τ has the form Ct''. Therefore by induction, repType(Ct) has the form $\{\overline{V_1'} : \overline{\tau_1'}, \overline{V_2'} : \overline{\tau_2'}\}$, where repType($Ct''$) = $\{\overline{V_1'} : \overline{\tau_1'}\}$. Let Ct and Ct'' respectively be $(\overline{\tau}\ C)$ and $(\overline{\tau''}\ C'')$. By lemma 18 we have $C \leq C''$, so by lemma 19 also $\bullet \vdash Ct''$ OK. Then by induction again, repType(Ct'') has the form $\{\overline{V_1} : \overline{\tau_1}, \overline{V_2} : \overline{\tau_2}\}$, where repType($Ct'$) = $\{\overline{V_1} : \overline{\tau_1}\}$. Therefore the result follows.

- Case SubTExt. Then $Ct = (\overline{\tau}\ Sn.Cn)$ and $Ct' = [\overline{\alpha} \mapsto \overline{\tau}]Ct''$ and (<`abstract`> `class` $\overline{\alpha}$ $Cn(\overline{I_0} : \overline{\tau_0})$ `extends` Ct'' `of` $\{\overline{Vn} : \overline{\tau_2}\}$) $\in SigT(Sn)$. Since $\bullet \vdash Ct$ OK, by lemma 38 there exist $\overline{V_3}$ and $\overline{\tau_3}$ such that repType(Ct) = $\{\overline{V_3} : \overline{\tau_3}\}$. By RepType,

$\{\overline{V_3} : \overline{\tau_3}\} = [\overline{\alpha} \mapsto \overline{\tau}]\{\overline{V_1} : \overline{\tau_1}, Sn.\overline{Vn} : \overline{\tau_2}\}$ and $\text{repType}(Ct'') = \{\overline{V_1} : \overline{\tau_1}\}$. Then by lemma 27, $\text{repType}(Ct') = [\overline{\alpha} \mapsto \overline{\tau}]\{\overline{V_1} : \overline{\tau_1}\}$, so the result follows.

□

A.3.6 Module Dependency Relation

Lemma 42 If $\overline{Sn} \vdash C$ ITCTransUses and $C \leq C'$, then $\overline{Sn} \vdash C'$ ITCTransUses.

Proof By induction on the depth of the derivation of $C \leq C'$. Case analysis of the last rule in the derivation.

- Case SubRef. Then $C = C'$, and the result follows by assumption.
- Case SubTrans. Then $C \leq C''$ and $C'' \leq C'$. By induction $\overline{Sn} \vdash C''$ ITCTransUses, and by induction again $\overline{Sn} \vdash C'$ ITCTransUses.
- Case SubExt. Then $C = Sn.Cn$ and (<`abstract`> `class` $\overline{\alpha}$ $Cn(\overline{I_0} : \overline{\tau_0})$ `extends` $\overline{\tau_1}$ C' ...) $\in ST(Sn)$. Since $\overline{Sn} \vdash C$ ITCTransUses, by ClassITCTransUses also $\overline{Sn} \vdash C'$ ITCTransUses.

□

VITA

Todd Millstein was born and raised in the Washington, D.C. area. He received an A.B. from Brown University in 1996, an M.S. from the University of Washington in 1998, and a Ph.D. from the University of Washington in 2003, all in computer science. In January 2004 he will be an assistant professor in the computer science department at UCLA.