TYPE SYSTEMS FOR OBJECT-ORIENTED PROGRAMMING LANGUAGES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Kathleen Shanahan Fisher
August 1996

Abstract

Object-oriented programming languages (OOPL's) provide important support for today's large-scale software development projects. Unfortunately, the typing issues arising from the object-oriented features that provide this support are substantially different from those that arise in typing procedural languages. Attempts to adapt procedural type systems to object-oriented languages have resulted in languages like Simula, C++, and Object Pascal, which have overly restrictive type systems. Among other things, the rigidity of these systems frequently force programmers to use type casts, which are a notorious source of hard-to-find bugs. These restrictive type systems also mean that many programming idioms common to untyped OOPL's such as Smalltalk are not typeable. One source of this lack of flexibility is the conflation of *subtyping* and *inheritance*. Briefly, inheritance is an implementation technique in which new object definitions may be given as incremental modifications to existing definitions. Subtyping concerns substitutivity: when can one object safely replace another? By tying subtyping to inheritance, existing OOPL's greatly reduce the number of legal substitutions in a system, and hence their degree of polymorphism. Attempts to fix this rigidity have resulted in unsound type systems, most notably Eiffel's.

This thesis develops a sound type system for a model object-oriented language that addresses this lack of flexibility. It separates the notions of subtyping and inheritance, producing a more flexible language. It also supports *method specialization*, which means that the types of methods may be specialized in certain ways during inheritance. The lack of such a mechanism is one of the key sources of type casts in languages like C++ and Object Pascal. The thesis then extends this core object calculus with abstraction primitives that support a class construct similar to the one found in languages such as C++, Eiffel, and Java. This formal study explains the link between inheritance and subtyping: object types that include implementation information are a form of abstract type, and the only way to get a subtype of an abstract type is by extension (*i.e.*, by inheritance). The study also suggests that object primitives and encapsulation are orthogonal language features that together produce object-oriented programming. Hence, adding object primitives to a language that already supports encapsulation (such as ML) should be sufficient to create an object-oriented language.

Formally, the language is presented as an object calculus and a type system with row variables, variance annotations, method absence annotations, and abstract types. The thesis proves type soundness via an operational semantics and an analysis of typing rules.

Acknowledgements

I would like to thank my advisor John Mitchell for encouraging me to look at formal models of object-oriented programming and for believing in me. His confidence inspired my own. I would also like to acknowledge the other members of my reading committee, Vaughan Pratt and Martín Abadi. While I was doing the research described here, many people provided significant moral and emotional support. In particular, I would like to thank my various officemates, including Ofer Matan, Liz Wolf, and Anna Patterson, for the time they took to listen to my troubles and to show their faith in me. Their support meant the world to me. I would also like to thank Amy Smith and Daphne Koller for being the best of friends through some very difficult times. My family, of course, also provided wonderful support. Finally, I am grateful for the external financial support I received during my tenure as a Ph.D. student, in particular, a National Science Foundation Graduate Fellowship and a Hertz Foundation Fellowship. Without their generous support, I could not have completed this work.

Contents

Chapter 1

Overview

This thesis develops a sound type system for a model object-oriented language that is more flexible than the type systems found in common practical languages. This flexibility partially stems from separating the notions of *subtyping*, a code-substitution principle, and *inheritance*, an object-definition reuse mechanism. Another source of its flexibility is its support for *method specialization*, a mechanism whereby the types of methods may be refined in certain ways during inheritance. The lack of such a mechanism is one of the key sources of type casts in languages like C++ and Object Pascal. The formal language presented in this thesis is developed in three stages: (i) an object calculus to model inheritance, (ii) an extension to this calculus that incorporates subtyping, (iii) a further extension to model classes. We prove the soundness of the resulting type system via a subject reduction theorem and an analysis of typing derivations.

1.1 An Object Calculus

There are several forms of object-oriented languages. One of the major lines of difference is between class-based and object-based languages. In class-based languages such as Smalltalk [GR83] and C++ [ES90], each object is created by a class and inheritance is determined by the class. In object-based languages such as Self [US91, CU89], an object may be created from another object, inheriting properties from the original. In this stage, we use an untyped lambda calculus of objects with a functional form of object-based inheritance as a tool for studying typing issues in object-oriented programming languages. Our main interests lie in (i) understanding how the functionality of a method may change as it is inherited, intuitively due to reinterpretation of the special symbol *self* (or *this* in C++), and (ii) providing a simple model of object-based inheritance that will serve as a building block for more complex systems.

In our calculus, the main operations on objects are to send a message `m` to an object `e`, written $\mathtt{e} \Leftarrow \mathtt{m}$, and two forms of method definition. If expression `e` denotes an object without method `m`, then $\langle \mathtt{e} \longleftrightarrow\!\!\!\!+ \; \mathtt{m} = \mathtt{e}' \rangle$ denotes an object obtained from e by adding the method body $\mathtt{e}'$ for m. When $\langle \mathtt{e} \longleftrightarrow\!\!\!\!+ \; \mathtt{m} = \mathtt{e}' \rangle$ is sent the message m, the result is obtained by applying $\mathtt{e}'$ to $\langle \mathtt{e} \longleftrightarrow\!\!\!\!+ \; \mathtt{m} = \mathtt{e}' \rangle$. This form of "self-application" allows us to model the special symbol *self* of object-oriented languages directly by lambda abstraction. Intuitively, the method body $\mathtt{e}'$ must be a function, and the first actual parameter of $\mathtt{e}'$ will always be the "object itself." To reinforce this intuition, we often write method bodies in the form $\lambda\mathtt{self}.(\ldots)$. The final method operation on objects is to replace one method body by another. This provides a functional form of update. As in the language Self, we do not distinguish instance variables from methods, since this does not seem essential. The untyped lambda calculus we use bears a strong resemblance to the T object system [RA82, AR88] (although it was originally developed without prior knowledge of T) and the untyped part of the calculus used in [Aba94] to model a fragment of Modula 3 [Nel91, CDJ+89].

The main goal of this stage is to develop a type system that allows methods to be specialized appropriately as they are inherited. Intuitively, it may be the case that the types of methods should become "more specific" when they are inherited, reflecting the fact that they belong to a more detailed object after the inheritance. This issue is perhaps best explained via an example. Briefly, suppose p is a two-dimensional point object with x and y methods returning the integer x- and y-coordinates of p, and a move method with type $\mathit{int} \times \mathit{int} \rightarrow \mathit{point}$. Method move has this type because if we send the message move to p, we obtain a function which given distances to move in the x and y directions, returns a point identical to p, but with updated $x-$ and y-coordinates. If we create a colored point cp from p by the object-extension operation, then cp inherits the x, y, and move methods from p. In an untyped object-oriented language such as Smalltalk, the inherited move method will change the position of a colored point, leaving the color unchanged. Therefore, in a typed language, we want the move method of cp to have the "specialized" type $\mathit{int} \times \mathit{int} \rightarrow \mathit{color\_point}$. If the inherited method had its original type $\mathit{int} \times \mathit{int} \rightarrow \mathit{point}$, then whenever we moved a colored point, we would obtain an ordinary point without color, making the inherited move function largely useless. While an imperative version of move could bypass this difficulty by returning type *unit* (as it is called in ML, or *void* in C++), experience with imperative object-oriented languages such as Eiffel and C++ suggest such self-returning methods are frequently convenient. Eiffel's like Current construct [Mey92], analyzed in [Coo89b], illustrates the value of specializing the type of a method in an imperative language. While C++ did not originally include such a construct, its widespread use is not counter-evidence to the usefulness of method specialization. In fact, it appears to be common for novice C++ programmers to attempt to specialize the types of methods in derived classes. More experienced C++ programmers appear to use "down casts" to approximate the effects described here. Additionally, a recent change to C++ adds a form of method specialization that allows the return types of methods to be refined during inheritance.

Formally, in this stage we present an object calculus, an operational semantics, and a type system for the calculus, although we defer the explanation of some of the technical details regarding "variance analysis" to the next stage. Subject reduction and type soundness theorems for this language follow from the corresponding theorems for the final language, which appear in Chapter 7.

1.2 Adding Subtyping

When this study was started, we initially regarded the object-inheritance-based approach, described above, as a technically simpler method for analyzing inheritance. This impression appeared correct for the study of method specialization carried out in [Mit90, FHM94], but in [FM95b] we observed that there appeared to be a fundamental trade-off between object-based inheritance and subtyping. Specifically, if an object may be extended with new methods, then it is important to know at compile-time that certain methods have *not* been defined already. This requirement conflicts with the usual motivation for subtyping, which is to allow code to operate uniformly over all objects having some minimum set of required methods. (Similar observations appear in [AC96c]; see Section 4.1.1.)

In this second stage, we present one way of resolving this conflict. Intuitively, the main idea is to distinguish between objects that may be extended with additional methods (or have existing methods redefined) and those that cannot. This distinction is achieved by giving different uses of objects different types. In other words, an object may be created and then have new methods added or existing methods redefined. At this point, only trivial subtyping may be used because the type system must keep track of exactly the set of methods associated with the object. However, such an object may be "converted" to a different kind of object, whose methods can no longer be altered. This conversion is done by changing the type of the object to a form which has the expected subtyping properties. In this way, we allow both object-based inheritance and subtyping, at the cost of some increase in the complexity of the type system.

Technically, in this stage we present an extension to the type system already developed that supports object subtyping. We also explain the details deferred above regarding the so-called "variance analysis" that tracks how type variables may vary in type expressions. Such information is crucial to determining subtyping relations between partially abstract object types, which will be very useful in modeling classes. As in the previous stage, subject reduction and type soundness theorems for this language follow from the corresponding theorems for the final language. A preliminary version of this system appeared in [FM95a].

1.3 Adding Classes

At this point, we have described an object calculus that models object-based inheritance and rich object subtyping. However, most practical object-oriented languages are class-based languages. This fact raises the question of how to appropriately model classes. Although a full discussion of the motivation for object-oriented languages is beyond the scope of this thesis, it is worth considering some desirable language characteristics before proceeding. Generally speaking, objects provide a useful encapsulation mechanism, separating internal implementation details from externally observable behavior, and provide a uniform framework for identifying and specifying the interfaces of various data and system resources. Within this context, classes serve several functions.

- *Programming methodology:* Classes provide a mechanism for declaring both a hierarchy of object types and object implementations.
- *Implementation considerations:* In class-based languages, it is possible to have class-based protection mechanisms, where the methods of an object may access the private data of another object of the same class (or of related classes). This access can be statically checked: classes specify all or part of the implementations of their objects, so all objects of the same class can be guaranteed to share certain implementation characteristics. This guarantee is useful for binary operations, such as set union, and for optimizing method lookup (as in C++).
- *Static analysis:* In comparison with prototype-based languages like Self, where delegation pointers may be set at run time, classes generally force inheritance to be a compile-time operation. This restriction greatly simplifies the static determination of the correctness of class declarations.

Based on these, and other considerations, we believe that in a typed setting, a class mechanism should have the following characteristics.

- A class provides an extensible collection of object "parts". The parts may be methods, data, specifications of communication protocols, and so on. *Extensibility* means that a derived class can use the object parts defined in a base class, possibly adding other parts to be used by subsequent derived classes.
- A class construct should include some static condition for guaranteeing that the object "parts" defined in the class are consistent with each other. For example, it should not be possible to define a class where one integer method f requires a string method g, but g is declared to be an integer method requiring a string method f.
- A class should provide the ability to specify which "parts" are private (for use within the class implementation) and which are public (for use by client programs). It should be possible to distinguish private from public parts for the current class and for all derived classes.
- A class should provide control over initialization of objects, both for the current class and all derived classes. This is essential for establishing invariants of private data structures, initializing system resources used by the object, and so on.
- A class construct should support incremental changes to its definition. In particular, if a given class is modified, all derived classes should be updated automatically.

In this third stage, we consider a class construct which resembles the form of class found in C++, Eiffel, and Java, and has all of the above properties. This class construct may be written in our object calculus extended with a form of abstract data type declaration. One appeal of this interpretation is that it clearly shows how classes may be viewed as an orthogonal combination of pure operations on objects (providing aggregation but no encapsulation) and data abstraction (providing encapsulation but no aggregation). This analysis provides some insight into the suitability of using object-inheritance-based systems to support traditional class-based programming. To the best of our knowledge, this interpretation of classes provides the first type soundness proof for the form of class construct found in Eiffel, C++, and Java.

Furthermore, this analysis sheds light on the long-standing controversy over the proper relationship between subtyping and inheritance. An early and influential paper, [Sny86], argues that the two ideas are distinct. This point is reinforced in [Coo92], which shows that the subtyping and inheritance hierarchies used in the Smalltalk collection classes are essentially unrelated. We believe that the arguments in [Sny86, Coo92] are correct for *interface types*, which are types that specify the operations of their objects but not their implementations. Such types have been the focus of recent theoretical studies of object systems, such as [AC96c, Bru93, FHM94, PT94] and the earlier papers appearing in [GM94]. However, existing object-oriented languages such as Eiffel [Mey92] and C++ [ES90, Str86] use a form of *implementation type* that constrains both the interface and the implementations of objects. We argue that there *is* a connection between subtyping and inheritance for implementation types: the only way to produce a subtype of an implementation type, without violating basic principles of data abstraction, is via inheritance. In addition, we show a connection between interface types and implementation types: every implementation type is a subtype of the interface type obtained by "forgetting" its implementation constraints. Our class construct will provide a mechanism for declaring implementation types, and the subtyping properties of our language will enable us to prove that implementation types are subtypes of the corresponding interface types (as long as a technical condition on the variance of the interface type is satisfied).

Formally, we extend the language developed in the second stage with expressions to define and use a form of abstract data type. We introduce a new type, a form of bounded existential, to type abstract data type implementations, and we extend our operational semantics to allow evaluation of abstract data type uses. Chapter 6 presents the full language in formal detail. We prove that the resulting language is sound via a subject reduction theorem and an analysis of typing derivations.

1.4 Road Map

The rest of the thesis is organized as follows. Chapter 2 summarizes some of the major issues in object-oriented programming. Chapter 3 presents the first stage described above, a simple prototype-based calculus for modeling inheritance. In Chapter 4, we introduce subtyping. Chapter 5 extends our calculus with an abstract data type mechanism to model classes of the form described above. Chapter 6 presents the full formal system needed to model this final language. Finally, Chapter 7 presents subject reduction and type soundness proofs for the final language in detail.

Chapter 2

Introduction to Object-Oriented Concepts

"Object orientation" is both a language feature and a design methodology. In general, object-oriented design is concerned with ways in which programs may be organized and constructed. Objects provide a program-structuring tool whose importance seems to increase with the size of the programs we build. Roughly speaking, an object consists of a set of operations on some hidden, or encapsulated, data. A characteristic of objects is that they provide a uniform interface to a variety of system components. For example, an object can be as small as a single integer or as large as a file system or output device. Regardless of its size, all interactions with an object occur via simple operations that are called "message sends" or "member function invocations." The use of objects to hide implementation details and provide a "black box" interface is useful for the same reasons that data and procedural abstraction are useful.

Although this chapter is about language features, not methodology, we describe object-oriented design briefly since this design paradigm is one of the reasons for the success of object-oriented programming. The following programming methodology is taken from [Boo91], one of many current books on object-oriented design.

1. Identify the objects at a given level of abstraction.
2. Identify the semantics (intended behavior) of these objects.
3. Identify the relationships among the objects.
4. Implement the objects.

This is an iterative process based on associating objects with components or concepts in a system. The process is iterative because an object is typically implemented using a number of "sub-objects," just as in top-down programming a procedure is typically implemented by a number of finer-grained procedures.

The data structures used in the early examples of top-down programming (see [Dij72]) were very simple and remained invariant under successive refinements of the program. Since these refinements involved simply replacing procedures with more detailed versions, older forms of structured programming languages, such as Algol, Pascal, and C, were adequate. When solving more complex tasks, however, it is often the case that both the procedures and the data structures of a program need to be refined in parallel. Object-oriented languages support this joint refinement of function and data.

2.1 Basic Concepts

Not surprisingly, all object-oriented languages have some notion of an "object," which is essentially some data and a collection of methods that operate on that data. There are (at least) two flavors of object-oriented languages: class-based and object-based. These flavors correspond to two different ways of defining and creating objects. In class-based languages, such as Smalltalk [GR83] and C++ [ES90], the implementation of an object is specified by its *class*. In such languages, objects are created by *instantiating* their classes. In object-based languages, such as Self, objects are defined directly from other objects by adding new methods via *method addition* and replacing old methods via *method override*. In the remainder of the chapter, we will focus on the more common class-based languages.

Although there is some debate as to what exactly constitutes an object-oriented programming language (besides merely having objects), there seems to be general agreement that such a language should provide the following features: dynamic lookup, subtyping, inheritance, and encapsulation. Briefly, a language supports dynamic lookup if when a message is sent to an object, the method body to execute is determined by the run-time type of the object, not its static type. Subtyping means that if some object ob_1 has all of the functionality of another object ob_2, then we may use ob_1 in any context expecting ob_2. Inheritance is the ability to use the definition of simpler objects in the definitions of more complex ones. Encapsulation means that access to some portion of an object's data is restricted to that object (or perhaps to its descendants). We explore these features in more detail in the following subsections.

2.1.1 Dynamic Lookup

In any object-oriented language, there is some way to invoke the methods associated with an object. In Smalltalk, this process is called "sending a message to an object," while in C++ it is "calling a member function of an object." To give a neutral syntax, we write

$$\texttt{receiver} \Leftarrow \texttt{operation}$$

for invoking `operation` on the object `receiver`. For expositional clarity, we will use the Smalltalk terminology for the remainder of this section.

Sending messages is a dynamic process: the method body corresponding to a given message is selected according to the run-time identity of the receiver object. The fact that this selection is dynamic is essential to object-oriented programming. Consider, for example, a simple graphics program that manipulates "pictures" containing many different kinds of shapes: squares, circles, triangles, *etc.* Each square object "knows" how to `draw` a square, each circle "knows" how to `draw` a circle, *etc.* When the program wants to display a given picture, it sends the `draw` message to each shape in the picture. At compile-time, the most we know about an object in the picture is that it is some kind of a shape and hence has some `draw` method. At run-time, we can find the appropriate `draw` method for each shape by querying that shape for its version of the `draw` method. If the shape is a square, it will have the square `draw` method, *etc.* [1]

There are two main views for what sending a message means operationally. In the first view, each object contains a "method table" that associates a method body with each message defined for that object. When a message is sent to an object at run-time, the corresponding method is retrieved from that object's method table. As a result, sending the same message to different objects may result in the execution of different code. In the example above, a square shape draws a square in response to the `draw` message, while a circle draws a circle. This behavior is called *dynamic lookup*, or, variously, dynamic binding, dynamic dispatch, and run-time dispatch. Both C++ and Smalltalk support this model of message sending.

The second view of message sending treats each message name as an "overloaded" function. When a message `m` is sent to an object `ob`, `ob` is treated as the first argument to an overloaded function named `m`. Unlike the traditional overloading of arithmetic operators, the appropriate code to execute when `m` is invoked is selected according to the run-time type of `ob`, not its static type. In this view, the methods of an object are not actually part of the object. Each object consists solely of its state. The methods from all the objects in a program are collected together by name. For example, the circle and square objects from above would simply contain their local state, *i.e.*, the circle might contain its center and radius, the square its corner points. The `draw` methods from each would be collected into some "method repository". If the `draw` message were sent to some object `ob`, the dynamic type of `ob` would be determined and the appropriate `draw` code selected from the repository. If `ob` were a circle, the circle `draw` method would be executed, *etc.* In this view, we again get the important characteristic that sending the same message to different objects can result in the execution of different code. Languages such as CLOS [Ste84] and Dylan [App92] support this model of message sending. A theoretical study appears in [CGL95].

In the second approach, it is possible to take more than the first argument into account in the

[1]In C++, only member functions designated *virtual* are selected dynamically. Non-virtual member functions are selected according to the static type of the receiver object. Needless to say, this distinction is the source of some confusion.

selection of the appropriate method body to execute. For example, if we write

$$\texttt{receiver} \Leftarrow \texttt{operation}(\texttt{arguments})$$

for invoking an operation with a list of arguments, then the actual code invoked can depend on the receiver alone (as explained above), or on the receiver and one or more arguments. When the selection of code depends only on the receiver, it is called *single dispatch;* when it also depends on one or more arguments, it is called *multiple dispatch.* Multiple dispatch is useful for implementing operations such as equality, where the appropriate comparisons to use depend on the dynamic type of both the receiver object and the argument object.

Although multiple dispatch is in some ways more general than the single dispatch found in C++ and Smalltalk, there seems to be some loss of encapsulation. This apparent loss arises because in order to define a function on different kinds of arguments, that function must typically have access to the internal data of each function argument. For example, suppose we wanted to define a `same_center` method that compares the centers of any two shapes and returns true if they match. Using multiple dispatch, we can write such a function by giving one version of the method for each pair of shapes we wish to consider: circle and circle, circle and square, square and circle, *etc.* Notice that this `same_center` method does not conceptually belong to any one of the shapes, and yet it must have access to the internal data of each shape object in order to do any meaningful comparisons. This external access of object internals violates the standard notions of encapsulation for object-oriented languages. It is not clear that this loss of encapsulation is inherent to multiple dispatch. However, current multiple dispatch systems do not seem to offer any reasonable encapsulation of private or local data for objects. Recent work addressing this issue appears in [CL94].

2.1.2 Subtyping

The basic principle associated with subtyping is *substitutivity:* if `A` is a subtype of `B`, then any expression of type `A` may be used without type error in any context that requires an expression of type `B`. We will write "`A` <: `B`" to indicate that `A` is a subtype of `B`.

The primary advantage of subtyping is that it permits uniform operations over various types of data. For example, subtyping makes it possible to have heterogeneous data structures containing objects that belong to different subtypes of some common base type. Consider as an example a queue containing various bank accounts to be balanced. These accounts could be savings accounts, checking accounts, investment accounts, *etc.*, but each is a subtype of `bank_account` so balancing is done in the same way for each. This uniform treatment is generally not possible in strongly typed languages without subtyping.

Subtyping in an object-oriented language also allows functionality to be added with minimal modification to the system. If objects of a type `B` lack some desired behavior, then we may wish to replace objects of type `B` with objects of another type `A` that have the desired behavior. In many cases, the type `A` will be a subtype of `B`. By designing the language so that substitutivity is allowed, one may add functionality in this way without any other modification to the original program.

An example illustrating this use of subtyping occurs in building a series of prototypes of an airport scheduling system. In an early prototype, one would define a class `airplane` with methods such as `position`, `orientation`, and `acceleration` that would allow a control tower object to affect the approach of an airplane. In a later prototype, it is likely that different types of airplanes would be modeled. If one adds classes for Boeing 757's and Beechcrafts, these would be subtypes of `airplane`, containing extra methods and fields reflecting features specific to these aircraft. By virtue of the subtyping relation, all Beechcrafts are instances of `airplane` and the general control algorithms that apply to all airplanes can be used for Beechcrafts without modification or re-compilation.

2.1.3 Inheritance

Inheritance is a language features that allows new classes to be defined as increments to existing ones. It is an implementation technique. For every object or class of objects defined using inheritance, there is an equivalent definition that does not use inheritance, obtained by expanding the definition so that inherited code is duplicated. The importance of inheritance is that it saves the effort of duplicating (or reading duplicated) code, and that when one class is implemented by inheriting from another, changes to one affect the other. This has a significant and sometimes debated impact on program maintenance and modification.

Using a neutral notation, we can illustrate by a simple example the form of inheritance that appears in most object-oriented languages. The two classes below define objects with private data `v` and public methods `f` and `g`. The class `B` is defined by inheriting the declarations of `A`, redefining the function `g`, and adding a private variable `w`.

```
class A =
    private
        val v =  ...
    public
        fun f(x)  = ... g(...) ...
        fun g(y)  = ... original definition ...
    end;

class B = extend A with
    private
        val w = ...
    public
        fun g(y)  = ... new definition ...
    end;
```

The simplest, but not most efficient, implementation of inheritance is to incorporate the relationship between classes explicitly in the run-time representation of objects, as is done in Smalltalk. For the example classes `A` and `B` above, this implementation is shown in Figure 2.1. This figure shows data structures representing the

`A` Class: stores pointers to the `A` Template and `A` Method Dictionary.

`A` Template: gives the names and order of data associated with each `A` object.

`A` Method Dictionary: contains pointers to the names and code for methods defined in the `A` Class.

`B` Class: stores pointers to the `B` Template, `B` Method Dictionary, and base class `A`.

`B` Template: gives the names and order of data associated with each `B` object.

`B` Method Dictionary: contains pointers to the names and code for methods defined in the `B` Class.

The figure also shows an `A` object `a` and a `B` object `b`. Both of these objects contain pointers to their class and storage for their data.

We can see how this data structure allows us to find the correct methods to execute at run-time by tracing the evaluation of the expression `b` $\Leftarrow$ `f()`. The sequence of events is:

1. We find the method dictionary for `B` objects by following `b`'s class pointer to the `B` Class and then accessing the class's method dictionary.
2. We search the `B` method dictionary for method name `f`.
3. Since `f` is not there, we follow the `B` Class's base class pointer to the `A` Class and then access the `A` method dictionary.
4. We find the function `f` in the `A` method dictionary.
5. When the body of `f` refers to `g`, we begin the search for the `g` method with the `b` object, guaranteeing that we find the `g` function defined in the `B` Class.

This implementation may be optimized in several ways. The first is to cache recently-found methods. Another possibility is to expand the method tables of derived classes to include the method tables of their base classes. This expansion eliminates the upward search through the method dictionaries of more than one class. Since the dictionaries contain only pointers to functions, this duplication does not involve a prohibitive space overhead. C++ makes this optimization.

A more significant optimization may be made in typed languages such as C++, where the set of possible messages to each object can be determined statically. If method dictionaries, or *virtual function tables* (vtables) in C++ terminology, can be constructed so that all subtypes of a given class `A` store pointers to the common methods in the same relative positions in their respective vtables,

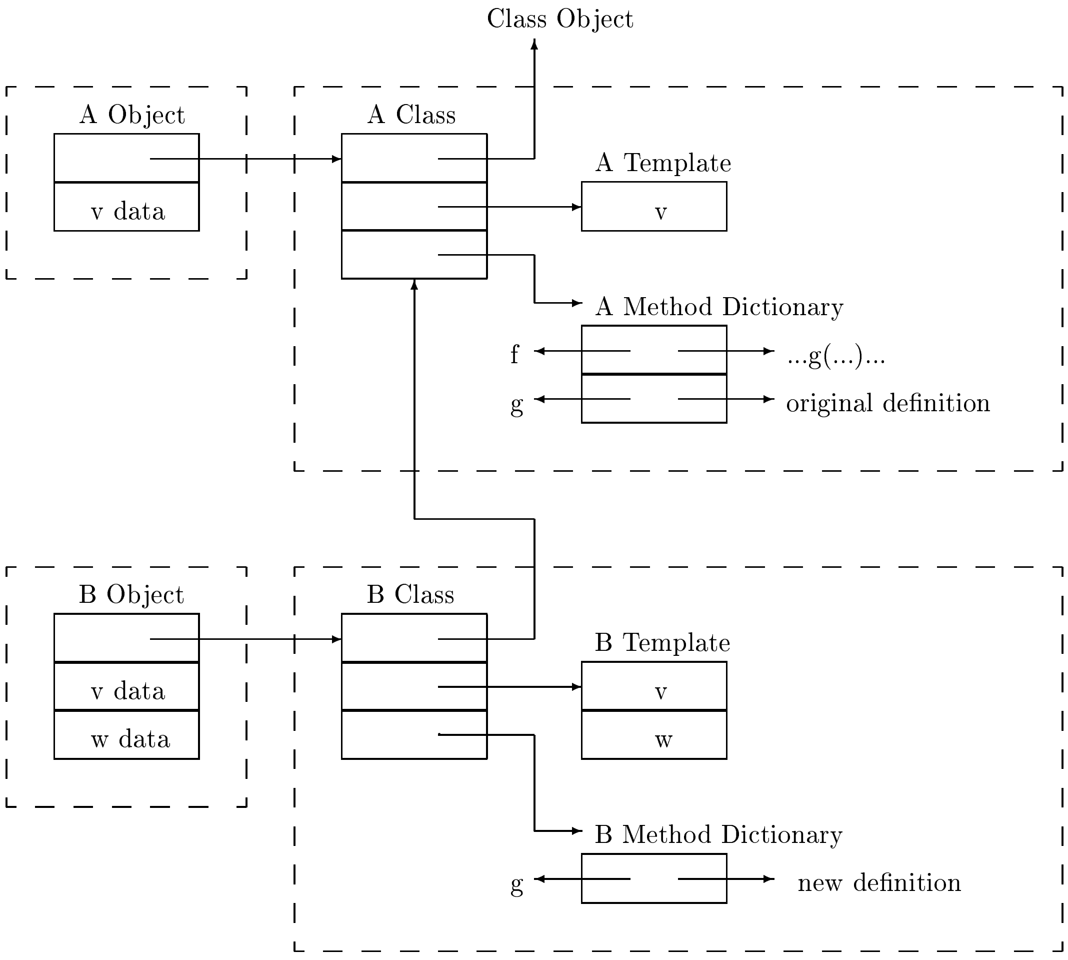


Figure 2.1: Smalltalk-style representation of `A` and `B` Classes and Objects, where Class `B` inherits from class `A`.

then the offset of a method within any vtable can be computed at compile-time. This optimization reduces the cost of method lookup to a simple indirection without search, followed by an ordinary function call. In untyped languages such as Smalltalk, this optimization is not possible because at compile-time, all we know about an object is that it *is* an object. In general, we do not know what messages it understands, let alone where the corresponding methods are stored.

Figure 2.2 shows a schematic C++ representation of the example classes `A` and `B` given above. This figure contains an `A` object `a` and a `B` object `b`. Each of these objects stores its instance variables and has a pointer to its class's vtable. The `A` vtable contains pointers to the methods defined in the `A` class, while the `B` vtable contains pointers to all the methods defined in `B` and to those defined in `A` but not redefined in `B`. (The expression `A::f` denotes the `f` function defined in the `A` Class and the "`&`" denotes C++'s address-of operator.) By duplicating the `f` method pointer in the `B` vtable, we do not have to access the `A` vtable when manipulating a `B` object.

We may see how this data structure works by tracing the evaluation of the expression $\mathtt{b} \Leftarrow \mathtt{f()}$. The sequence of events is essentially:

1. We find the vtable for `B` objects by following `b`'s vtable pointer.

2. At compile-time, we may determine that the `f` method is the first entry in the `B` vtable, so we retrieve the `f` method from the vtable without searching.

3. When the body of `f` refers to `g`, we retrieve the `g` method from `b`'s vtable, guaranteeing that we use the `g` function defined in the `B` class.

For more information, see [ES90, Section 10.7c]. The actual process is somewhat more complicated because of multiple inheritance. See [ES90, Chapter 10] for more details.

2.1.4 Encapsulation

Objects are used in most object-oriented programming languages to provide encapsulation barriers similar to those given by abstract data types (ADT's). However, because object-oriented languages have inheritance, object-oriented encapsulation can be more complex than simple data abstraction. In particular, there are two "clients" of the code in a given ADT: the implementor, who "lives" inside the encapsulation barrier, and the general client, who "lives" outside and may only interact with the ADT via its interface. A graphic representation of this relationship appears in Figure 2.3. Because of inheritance, there are three "clients" of the code in a given object definition, not two. The additional "client," the inheritor, uses the given object definition via inheritance to implement new object definitions. Because object definitions have two external clients, there are two interfaces to the "outside": the *public* interface lists what the general client may see, while the *protected* interface lists what inheritors may see. (This terminology comes from C++.) A graphic representation appears in Figure 2.4. It is typically the case that the public interface is less detailed than the protected one.

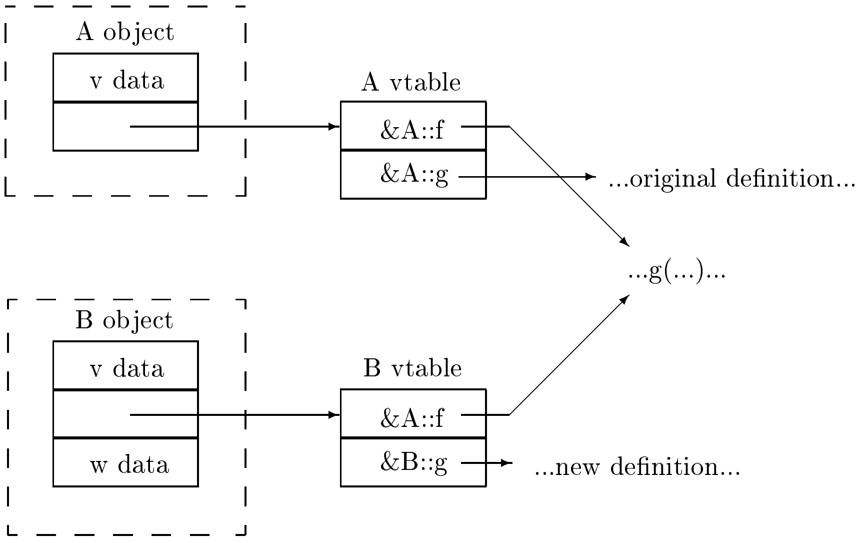


Figure 2.2: A C++-style representation of `A` and `B` objects where class `B` inherits from class `A`.

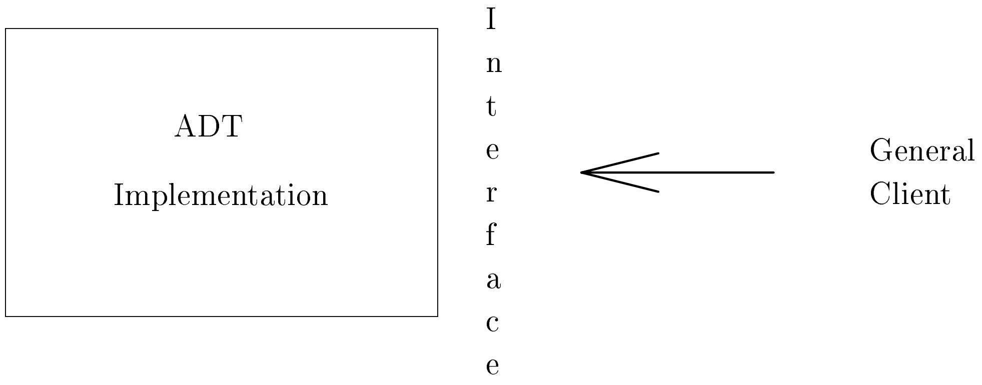


Figure 2.3: In ADT-style encapsulation, the general client interacts with the ADT implementation through a single interface.

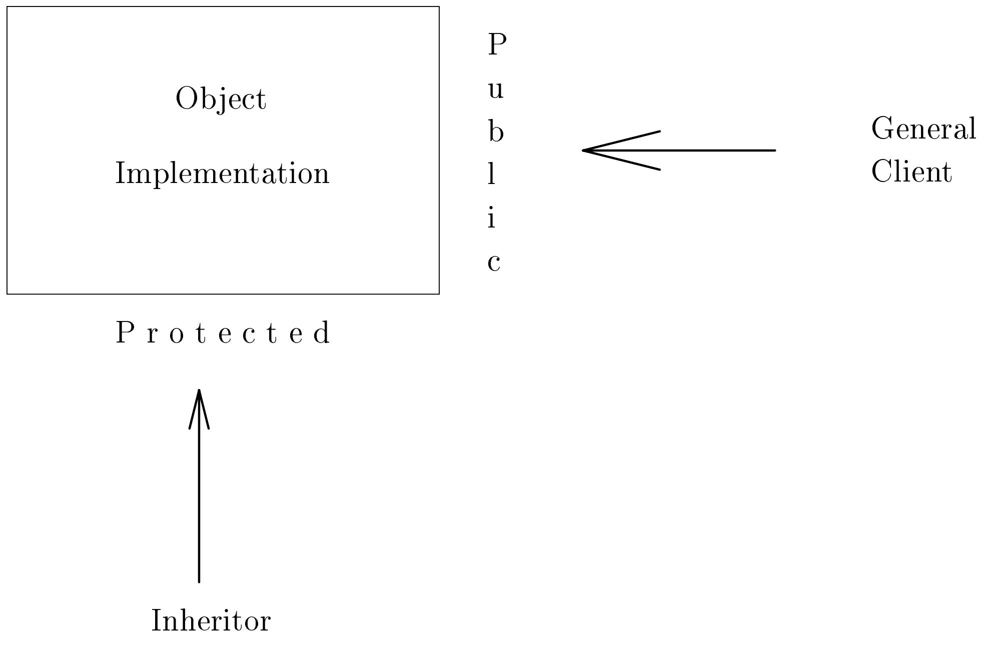


Figure 2.4: In object-oriented encapsulation, the general client interacts with the object implementation via the public interface, while the inheritors interact via the protected interface.

In Smalltalk, these interfaces are generated automatically: the public interface lists the methods of an object, while the protected interface lists its methods and its instance variables. In C++, the programmer explicitly declares which components of an object are public, which are protected, and which are *private*, visible only in the object definition itself.

The encapsulation provided by object-oriented languages helps insure that programs can be written in a modular fashion and that the implementation of an object can be changed without forcing changes in the rest of the system. In particular, as long as the public interface of an object remains unchanged, modifications to its implementation do not force general clients to change their code. Similarly, if implementation modifications preserve an object's protected interface, inheritors need not update their code, either. In both cases, however, they may have to recompile.

2.2 ADT's vs. Objects

As we saw in the previous section, the encapsulation benefits provided by objects are the same as those realized by abstract data types. Because object-oriented languages provide dynamic lookup, subtyping, and inheritance in addition to encapsulation, objects may be used more flexibly than

ADT's. The importance of these added features becomes apparent when we wish to use related data abstractions in similar ways. We illustrate this point with the following example involving queues.

A typical language construct for defining an abstract data type is the ML `abstype` declaration, which we use below to define a queue ADT.

```
exception Empty;
abstype queue = Q of int list
with
     fun mk_Queue()         = Q(nil)
     and is_empty(Q(l))     = l=nil
     and add(x,Q(l))        = Q(l @ [x])
     and first (Q(nil))     = raise Empty
     |   first (Q(x::l))    = x
     and rest (Q(nil))      = raise Empty
     |   rest (Q(x::l))     = Q(l)
     and length (Q(nil))    = 0
     |   length (Q(x::l))   = 1 + length (Q(l))
end;
```

In this example, a queue is represented by a list. However, only the functions given in the declaration may access the list. This restriction allows the invariant that list elements appear in first-in/first-out order to be maintained, regardless of how queues are used in client programs.

A drawback of the kind of abstract data types used in ML and other languages such as CLU [LSAS77, L$^+$81] and Ada [US 80] becomes apparent when we consider a program that uses both queues and priority queues. For example, suppose that we are simulating a system with several "wait queues," such as a bank or hospital. In a teller line or hospital billing department, customers are served on a first-come, first-served basis. However, in a hospital emergency room, patients are treated in an order that takes into account the severity of their injuries. Some aspects of this kind of "wait queue" are modeled by the abstract data type of priority queues, shown below:

```
exception Empty;
abstype pqueue = Q of int list
with
     fun mk_PQueue()        = Q(nil)
     and is_empty(Q(l))     = l=nil
     and add(x,Q(l))        =
         let fun insert(x,nil) = [x:int]
            insert(x,y::l) = if x < y then x::y::l else y::insert(x,l)
         in Q(insert(x,l)) end
```

```
      and first (Q(nil))   = raise Empty
      |   first (Q(x::l))  = x
      and rest (Q(nil))    = raise Empty
      |   rest (Q(x::l))   = Q(l)
      and length (Q(nil))  = 0
      |   length (Q(x::l)) = 1 + length (Q(l))
end;
```

For simplicity, like the queues above, this queue is defined only for integer data. Although the priority of a queue element may come from any ordered set, we use the integer value as the priority, with lower numbers given higher priority.

Note that the signature of priority queues, the list of available methods and their associated types, is the same as for ordinary queues: both have the same number of operations, and each operation has the same type, except for the difference between the type names `pqueue` and `queue`. However, if we declare both queues and priority queues in the same scope, the second declarations of `is_empty`, `add`, `first`, `rest`, and `length` hide the first. This name clashing requires us to rename them, say as `q_is_empty`, `q_add`, `q_first`, `q_rest`, `q_length` and `pq_is_empty`, `pq_add`, `pq_first`, `pq_rest`, `pq_length`.

In a hospital simulation (or real-time hospital management) program, we might occasionally like to treat priority queues and ordinary queues uniformly. For example, we might wish to count the total number of people waiting in any line in the hospital. To write this code, we would like to have a list of all the queues (both priority and ordinary) in the hospital and go down the list asking each queue for its length. But if the `length` operation is different for queues and priority queues, we have to decide whether to call `q_length` or `pq_length`, even though the correct operation is uniquely determined by the data. This shortcoming of ordinary abstract data types is eliminated in object-oriented programming languages by a combination of subtyping and dynamic lookup.

Another drawback of traditional abstract data types becomes apparent when considering the implementation of the priority queue above. Although the priority queue's version of the `add` function is different from the queue's version, the other five functions have identical implementations. In an object-oriented language, we may use inheritance to define `pqueue` from `queue` (or vice versa), giving only the new `add` function.

2.3 Object-Oriented vs. Conventional Organization

Because object-oriented languages have subtyping, inheritance, and dynamic lookup, programs written in an object-oriented style are organized quite differently from those written in a traditional style. In this section, we illustrate some of the differences between object-oriented and "conventional" program organizations via an extended example. We give two versions of a program that manipulates several kinds of geometric shapes. One version uses classes; the other does not.

Without classes, we use records (or `struct`'s) to represent each shape. For each operation on shapes, we have a function that tests the type of shape passed as an argument and branches accordingly. We illustrate this program structure using a C program, with each shape represented as a `struct` (analogous to a Pascal or ML record). The code appears in Appendix A. We will refer to this program as the "typecase" version, since each function is implemented by a case analysis on the types of shapes. For brevity, the only shapes are circles and rectangles.

We can see the advantage of object-oriented programming by rewriting the program so that each object has the shape-specific operations as methods. This version appears in Appendix B.

Some observations:

- We can see the difference between the two program organizations in the following matrix. For each function, `center, move, rotate` and `print`, there is code for each geometric shape, in this case `circle` and `rectangle.` Thus we have eight different pieces of code.

class	*function*			
	center	move	rotate	print
circle	`c_center`	`c_move`	`c_rotate`	`c_print`
rectangle	`r_center`	`r_move`	`r_rotate`	`r_print`

  In the "typecase" version, these functions are arranged by column, while in the class-based program, they are arranged by row. Each arrangement has some advantages when it comes to program maintenance and modification. In the object-oriented approach, adding a new shape is straightforward. The code detailing how the new shape should respond to the existing operations all goes in one place: the class definition. Adding a new operation is more complicated, since the appropriate code must be added to each of the class definitions, which could be spread throughout the system. In the "typecase" version, the reverse situation is true: adding a new operation is relatively easy, but adding a new shape is difficult.

- There is a loss of encapsulation in the typecase version, since the data manipulated by `rotate, print` and the other functions has to be publicly accessible. In contrast, the object-oriented solution encapsulates the data in the `circle` and `square` objects. Only the methods of these objects may access this data.

- The "typecase" version cannot be statically type-checked in C. It could be type-checked in a language with a built-in "typecase" statement which tests the type of an struct directly. An example of such a language feature is the Simula `inspect` statement. Adding such a statement would require that every struct be tagged with its type, a process which requires about the same amount of space overhead as making each struct into an object.

- In the typecase version, "subtyping" is used in an ad hoc manner. We coded circle and rectangle so that they have a shared field in their first location. This is a hack to implement a tagged union that could be avoided in a language providing disjoint (as opposed to C unchecked) unions.

- The complexity of the two programs is roughly the same. In the "typecase" version, there is the space cost of an extra data field (the type tag) and the time cost, in each function, of branching according to type. In the "object" version, there is a hidden class or `vtable` pointer in each object, requiring essentially the same space as a type tag. In the optimized C++ approach, there is one extra indirection in determining which method to invoke, which corresponds to the switch statement in the "typecase" version. (Although in practice a single indirection will frequently be more efficient than a switch statement.) A Smalltalk-like implementation would be less efficient in general, but for methods that are found immediately in the subclass method dictionary (or via caching), the run-time efficiency may be comparable.

A similar example appears in [Str86, Sections 7.2.7–8].

2.4 Advanced Topics

2.4.1 Inheritance Is Not Subtyping

Perhaps the most common confusion surrounding object-oriented programming is the difference between subtyping and inheritance. One reason subtyping and inheritance are often confused is that some class mechanisms combine the two. A typical example is C++, where `A` will be recognized by the compiler as a subtype of `B` only if `B` is a public parent class of `A`. Combining these mechanisms is an elective design decision, however; there seems to be no inherent reason for linking subtyping and inheritance in this way.

We may see the differences between inheritance and subtyping most clearly by considering an example. Suppose we are interested in writing a program that requires `dequeue`s, `stack`s, and `queue`s. One way to implement these three classes is first to implement `dequeue` and then to implement `stack` and `queue` by appropriately restricting (and perhaps renaming) the operations of `dequeue`. For example, `stack` may be obtained from `dequeue` by limiting access to those operations that add and remove elements from one end of the dequeue. Similarly, we may obtain `queue` from `dequeue` by restricting access to those operations that add elements at one end and remove them from the other. This method of defining `stack` and `queue` by inheriting from `dequeue` is possible in C++ through the use of `private` inheritance. (We are not recommending this style of implementation; we use this example simply to illustrate the differences between subtyping and inheritance.) Note that although `stack` and `queue` inherit from `dequeue`, they are not subtypes of `dequeue`. To see this point, consider a function `f` that takes a `dequeue d` as an argument and then adds an element to both ends of `d`. If `stack` or `queue` were a subtype of `dequeue`, then function `f` should work equally well when given a `stack s` or a `queue q`. However, adding elements to both ends of either a `stack` or a `queue` is not legal; hence, neither `stack` nor `queue` is a subtype of `dequeue`. In fact, the reverse is true. `Dequeue` is a subtype of both `stack` and `queue`, since any operation valid for either a stack or a queue would be a legal operation on a dequeue. Thus, inheritance and subtyping are different relations: we defined `stack` and `queue` by inheriting from `dequeue`, but `dequeue` is a subtype of `stack` and `queue`, not the other way around.

A more detailed comparison of the two mechanisms appears in [Coo92], which analyzes the inheritance and subtyping relationships between Smalltalk's collection classes. In general, there is little relationship between the two relations. See [Sny86] for more examples.

2.4.2 Object Types

There are two forms of types we might give to objects. The first is a type that simply gives the interface to its objects. The second is an interface plus some implementation information. In the first case, the elements of a type will be all objects that have a given interface. We call such types "interface types". In the second case, a type will contain only those elements that also have a certain representation. The type that C++ gives to an object is of the second form, since all objects of the same type are guaranteed to have the same basic implementation.

Since the first form of type is more basic, we begin by discussing it. The following example uses the syntax of Rapide, an experimental language designed for prototyping software and mixed software/hardware systems [BL90, MMM91, KLM94, KLMM94].

```
type Point is interface
    x_val : int;
    y_val : int;
    distance : Point → int;
end interface;
```

Objects of type *Point* must have two integer methods, called `x_val` and `y_val`, and a method called `distance`. This `distance` method requires only one argument, since the method belongs to a particular point and therefore may compute the distance between the point passed as an actual parameter and the particular point to which the method belongs. In other words, the intended use of the `distance` method of a point object `p` is to compute the distance between `p` and another point object `q`, by a call of the form `p⇐distance(q)`. Of course, since the interface gives only the names of methods and their types, the `distance` method is not actually forced to compute the distance between two points. If we wish to specify that `distance` must compute distance, then a more expressive form of specification must be added to the interface. One significant feature of this type interface for *Point*s is that the type name *Point* appears within it. Hence interface types often seem to be recursively-defined types.

To discuss object types in general, we introduce the syntax $\{\!|\mathtt{m_1}\colon A_1, \ldots, \mathtt{m_k}\colon A_k|\!\}$ for the interface type specifying methods $\mathtt{m_1}, \ldots, \mathtt{m_k}$ of types $A_1, \ldots, A_k$, respectively. Using this notation, we may recursively define the type *Point* as

$$Point \overset{def}{=} \{\!|\mathtt{x\_val}\colon nt, \mathtt{y\_val}\colon int, \mathtt{distance}\colon Point \rightarrow int|\!\}$$

Objects that have this interface type are guaranteed to have integer `x_val` and `y_val` methods. They are also guaranteed to have a method `distance` that returns an integer whenever it is given another object with the $\widehat{\text{P}}$oint interface. Objects with this interface are *not* required to have any particular implementation. For example, an object that stores a point in polar coordinates and implements `x_val` and `y_val` as functions that convert the stored polar coordinates into their cartesian counterparts may be given this interface type, just as the obvious cartesian implementation may. It is also the case that objects with the *Point* interface may have more methods than just those listed in the interface. For example, the polar point object described above must have some fields storing the polar coordinates of the point. These fields are not reflected in the *Point* interface.

If the type of an object is its interface, then subtyping for object types is "compatibility" or "conformance" of interfaces. More specifically, if one interface provides all of the methods of another with compatible types, then every object of the first type should be acceptable in any context expecting an object of the second type. This kind of subtyping is of the form:

$$\{\!|\mathtt{x}\colon Point,\ \mathtt{c}\colon color|\!\} <: \{\!|\mathtt{x}\colon Point|\!\}$$

which we call "width" subtyping. (We use the symbol $<:$ to denote the subtype relation between types.) This subtyping "judgment" says that we may consider any object that has the interface $\{\!|\mathtt{x}\colon Point,\ \mathtt{c}\colon color|\!\}$ to have the interface $\{\!|\mathtt{x}\colon Point|\!\}$ as well. In other words, we may put an object with interface $\{\!|\mathtt{x}\colon Point,\ \mathtt{c}\colon color|\!\}$ into any context expecting an object with interface $\{\!|\mathtt{x}\colon Point|\!\}$ and be guaranteed that no type errors will result. We may see the justification for this guarantee by considering what a context `C[ob]` may ask of its argument object `ob`. Since `C` expects to be given an object with the $\{\!|\mathtt{x}\colon Point|\!\}$ interface, all it "knows" about its argument object is that it has an `x` method that returns a *Point* object. Hence all it may do with `ob` is ask for its `x` method and then treat the result as a *Point*. Since any object with the $\{\!|\mathtt{x}\colon Point,\ \mathtt{c}\colon color|\!\}$ interface has an `x` method that returns an *Point* object, giving such an object to our context cannot result in any type errors.

It is also generally possible to specialize the type of one or more methods to a subtype. We illustrate this form of subtyping, which we call "depth" subtyping, using type *ColorPoint*, defined as follows:

$$ColorPoint \overset{def}{=} \{\!|\mathtt{x}\colon Point,\ \mathtt{c}\colon color|\!\}$$

As we saw above, $ColorPoint <: Point$. Using this fact, we get the depth subtyping relation:

$$\{\!|\mathtt{x}\!: ColorPoint|\!\} <: \{\!|\mathtt{x}\!: Point|\!\}$$

This subtyping judgment says that we may consider any object with interface $\{\!|\mathtt{x}\!: ColorPoint|\!\}$ to have interface $\{\!|\mathtt{x}\!: Point|\!\}$ as well. In other words, we may put any object that supports the more detailed interface $\{\!|\mathtt{x}\!: ColorPoint|\!\}$ into any context expecting a $\{\!|\mathtt{x}\!: Point|\!\}$ object without producing any type errors. As above, we may see the justification for this guarantee by considering what such a context might ask of its argument. Because it expects an object with interface $\{\!|\mathtt{x}\!: Point|\!\}$, all it "knows" about its argument object is that it has an `x` method that returns a *Point* object, and hence that is all it may ask for. If we give such a context some object `cp` with interface $\{\!|\mathtt{x}\!: ColorPoint|\!\}$, the context may only ask `cp` for its `x` value, at which point `cp` returns something with interface *ColorPoint*. Because we know that $ColorPoint <: Point$, we are guaranteed that this resulting object may be safely treated as a *Point* object. Hence no type errors may result from putting a $\{\!|\mathtt{x}\!: ColorPoint|\!\}$ object into a context expecting a $\{\!|\mathtt{x}\!: Point|\!\}$ object.

Combining these two forms of subtyping, we have the subtyping judgment that

$$\{\!|\mathtt{x}\!: ColorPoint,\ \mathtt{c}\!: color|\!\} <: \{\!|\mathtt{x}\!: Point|\!\}$$

An alternative form of object type is an interface type with some additional guarantees about the form of the implementations of objects given that type. The types that C++ gives to its objects have this flavor. If we know that a particular object `ob` has type B, then we know `ob` has all of the methods and the associated types defined in the class `B`, even if these methods are not included in `B`'s public interface . We are also guaranteed that the layout of `ob` is an extension (perhaps trivial) of the layout implicitly specified by class `B`.

These implementation guarantees are important for objects with binary operations (those that take another object of the same type as an argument), and they permit more efficient implementations of objects. For these types, subtyping must take into account both interface subtyping and compatibility of implementations. Since the implementation of an object is intended to be hidden, the second form of type should not give any explicit information about the implementation. Instead, it appears that "implementation types" are properly treated as a form of partially-abstract types. We explore these ideas in Chapter 5.

2.4.3 Method Specialization

It is relatively common for one or more methods of an object to take objects of the same type as parameters or return objects of the same type as results. For example, consider points with the interface shown in Figure 2.5.

(For simplicity, we work with one-dimensional points.) The `move` method of a point `p` returns a *Point*. Similarly, the `eq` method takes as a parameter an object of *Point* type. When color points

```
type Point is interface
    x : int;
    move : int → Point;
    eq : Point → bool;
end interface;
```

Figure 2.5: *Point* interface declaration.

are defined in terms of points, it is desirable that the types of the methods be specialized to return or use color points instead of points. Otherwise, we effectively lose type information about the object we are dealing with whenever we send the `move` method, and we are restricted to using only point methods when comparing color points for equality. If it is possible to inherit a `move` method defined for points in such a way that the resulting method on color points has type $int \rightarrow ColorPoint$, then we say that *method specialization* occurs. This form of method specialization is called "mytype" specialization because the type that changes is the type of the object that contains the methods [Bru92, Bru93]. It is also meaningful to specialize types other than the type of the object itself when defining a derived class.

Method specialization is generally not provided in existing typed object-oriented languages, but it is common to take advantage of method specialization (in effect) in untyped object-oriented languages. Therefore, if we are to devise typed languages to support useful untyped programming idioms, we need to devise type systems that support method specialization. We address some of these points in Chapter 3.

Chapter 3

Object Calculus

As a first step in our study of object-oriented features, we develop a simple, formal model of inheritance. In existing languages, inheritance mechanisms are either *class-based* or *object-based*. In class-based systems, such as Simula [BDMN73], Smalltalk [GR83], and C++ [ES90], objects are defined in *classes*, which are typically static constructs. Objects are *instantiated* from their defining classes at run-time and are then used in computation via message sending. In object-based languages such as Self [US91], objects are created directly from other objects via inheritance primitives such as adding a new method to an existing object (*object extension*) and replacing an existing method with a new one (*method override*). Hence class-based languages require two kinds of entities for inheritance: classes and objects, whereas object-based languages require only objects. In the interest of adopting as simple a model of inheritance as possible, we therefore adopt object-based inheritance primitives. (In Chapter 5, we will use these inheritance primitives in conjunction with an abstraction mechanism to develop a formal interpretation of classes as the combination of these two more primitive concepts.) Briefly, in this chapter our main interests lie in understanding

- how the methods of an object interact with each other, intuitively through the pseudo-variable *self* representing the host object,
- how the meanings of methods may change because of the reinterpretation of *self* that occurs during inheritance, and
- the typing issues that arise from inheritance.

We are also interested in providing a simple model of object-based inheritance that will serve as a building block in more complex systems.

In our model, objects support three operations: message sending, method override, and object extension. Syntactically, the expression $\mathtt{e} \Leftarrow \mathtt{m}$ denotes sending object $\mathtt{e}$ the message $\mathtt{m}$. Our type system insures that object $\mathtt{e}$ actually has an $\mathtt{m}$-method defined in it. Expression $\langle \mathtt{e}_1 \longleftrightarrow\!\!\!\!+ \; \mathtt{m} = \mathtt{e}_2 \rangle$

denotes the object that is just like object $\mathtt{e}_1$ except that it has a new method $\mathtt{m}$ with method body $\mathtt{e}_2$. The type system guarantees that $\mathtt{e}_1$ does not already have a method $\mathtt{m}$ and hence the new method body cannot violate any typing assumptions the older methods of $\mathtt{e}_1$ may have made. Finally, the expression $\langle \mathtt{e}_1 \leftarrow \mathtt{m} = \mathtt{e}_2 \rangle$ denotes the object just like $\mathtt{e}_1$ except that the new object's $\mathtt{m}$-method has method body $\mathtt{e}_2$. Here, the type system insures that the type of the new method body exactly matches the type of the existing method body. This guarantee is essential for type soundness: if an unrelated or weaker type were given, the new method body could violate typing assumptions made by the other methods of object $\mathtt{e}_1$. Since these methods are inherited by the new object, such violations can cause type errors. Notice that the method overriding operation provides a functional form of update.

In object-oriented languages, the methods of an object can typically refer to the other methods of their host object, for example via the pseudo variable *self* in Smalltalk and *this* in C++. We model this behavior by making method bodies functions from their host object to the actual code for the method body. The semantics for message sending then connects the "self" parameter to the actual method body by extracting the corresponding method body and applying it to the full object. For example, when we evaluate sending the message $\mathtt{m}$ to the object $\langle \mathtt{e}_1 \longleftrightarrow\!\!\!+ \ \mathtt{m} = \mathtt{e}_2 \rangle$, we first extract the method body $\mathtt{e}_2$ and then apply it to the object $\langle \mathtt{e}_1 \longleftrightarrow\!\!\!+ \ \mathtt{m} = \mathtt{e}_2 \rangle$.

For simplicity, we work in a purely functional model, and in a fashion similar to the language Self [US91], we do not distinguish fields from methods. Recent studies such as [AC96a, Bv93, Pie93, DF96] suggest that imperative features do not present any surprises, at least from a typing perspective.

3.1 Method Specialization

In typing objects, it is useful to have a type that intuitively means "the type of the host object", frequently referred to as "mytype". For example, if we have a functional point object of some type *Point*, with integer `x`- and `y`- coordinates and a `move` method, the return type of the `move` method might be *Point*, reflecting the fact that when a *Point* object is sent the message `move`, a *Point* object is the result. However, this typing is not flexible enough in the context of inheritance. For example, consider adding a color method `c` to our point object via object extension. Then when we send a *ColorPoint* object the message `move`, we will get back a *ColorPoint* object. Unfortunately however, the type system will indicate that we have a *Point*. Hence if we want to use the color of the resulting object, we will have to insert a type cast to convert the resulting type to *ColorPoint*. The solution to this problem is to give as the return type of `move` not *Point* but "mytype". Then when the object is specialized via method extension, the return type will still be "mytype". In the derived object, the "mytype" will be reinterpreted to mean the type of *ColorPoint* objects.

Imperative languages reduce the need for this mechanism, as they can work via side-effects.

However, as illustrated by the `like Current` mechanism in Eiffel [Mey92], method specialization is still useful in imperative object-oriented languages. Additionally, although C++ did not originally support such a mechanism, recent additions allow a form of method specialization in return positions (such as for the `move` method above).

The phenomenon we are concerned with is called "method specialization" in [Mit90], which describes a precursor to the calculus used here. The earlier work describes method specialization and explains its usefulness, but only presents a tentative type system by extending the already complicated record calculus of [CM91]. In addition, no analysis of the type system is given. This chapter presents a calculus of objects alone, without recourse to record calculi (although we owe a substantial debt to previous studies of record calculi), simplifies the typing rules substantially, and proves type soundness. A preliminary version of this work appeared in [MHF93].

3.2 Untyped Objects and Object-Based Inheritance

3.2.1 Untyped Calculus of Objects

We extend the untyped lambda calculus with four object-related syntactic forms,

$$\mathtt{e} ::= \quad \mathtt{x} \mid \mathtt{c} \mid \lambda\mathtt{x}.\,\mathtt{e} \mid \mathtt{e}_1\mathtt{e}_2 \mid \langle\rangle \mid \mathtt{e} \Leftarrow \mathtt{m} \mid \langle \mathtt{e}_1 \longleftrightarrow\!\!\!\!\!+ \ \mathtt{m}{=}\mathtt{e}_2\rangle \mid \langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2\rangle$$

In this grammar, $\mathtt{x}$ may be any variable, $\mathtt{c}$ is a constant symbol (such as a "built-in" function), $\lambda\mathtt{x}.\,\mathtt{e}$ is a lambda abstraction (function expression) and $\mathtt{e}_1\,\mathtt{e}_2$ is function application. The object forms are described in tabular form for easy reference:

$\langle\rangle$	the empty object
$\mathtt{e} \Leftarrow \mathtt{m}$	send message $\mathtt{m}$ to object $\mathtt{e}$
$\langle \mathtt{e}_1 \longleftrightarrow\!\!\!\!\!+ \ \mathtt{m}{=}\mathtt{e}_2\rangle$	extend object $\mathtt{e}_1$ with new method $\mathtt{m}$ having body $\mathtt{e}_2$
$\langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2\rangle$	replace $\mathtt{e}_1$'s method body for $\mathtt{m}$ by $\mathtt{e}_2$

We consider $\langle \mathtt{e}_1 \longleftrightarrow\!\!\!\!\!+ \ \mathtt{m}{=}\mathtt{e}_2\rangle$ meaningful only if $\mathtt{e}_1$ denotes an object that does not have an $\mathtt{m}$ method and $\langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2\rangle$ meaningful only if $\mathtt{e}_1$ denotes an object that already has an $\mathtt{m}$ method. These conditions will be enforced by the type system. The reason for distinguishing extension from method replacement is that these two operations will have different typing rules. If a method is new, then no other method in the object could have referred to it, so it may have any type. On the other hand, if a method is being replaced, then we must be careful not to violate any typing assumptions made by other methods of the host object. If we were not concerned with static typing, then we could use a single operation that adds a method to an object, replacing any existing method with the same name.

3.2.2 Examples of Objects, Inheritance, and Method Specialization

To provide some intuition for this calculus, we give a few short examples. The first shows how records may be encoded as objects, while the second and third illustrate method specialization. The latter examples may be regarded as the motivating examples for the work presented in this chapter; rather than try to define method specialization in general, we attempt to convey the essential properties by the examples of points and color points given below.

To simplify notation, we write $\langle \mathtt{m}_1 = \mathtt{e}_1, \ldots, \mathtt{m}_k = \mathtt{e}_k \rangle$ for $\langle \ldots \langle \langle\rangle \leftarrow\!\!+ \mathtt{m}_1 = \mathtt{e}_1 \rangle \ldots \leftarrow\!\!+ \mathtt{m}_k = \mathtt{e}_k \rangle$, where $\mathtt{m}_1, \ldots, \mathtt{m}_k$ are distinct method names. We illustrate the computational behavior of objects in this section using a simplified evaluation rule that reflects the operational semantics defined precisely below,

$$\langle \mathtt{m}_1 = \mathtt{e}_1, \ldots, \mathtt{m}_k = \mathtt{e}_k \rangle \Leftarrow \mathtt{m}_i \quad \overset{eval}{\longrightarrow} \quad \mathtt{e}_i \, \langle \mathtt{m}_1 = \mathtt{e}_1, \ldots, \mathtt{m}_k = \mathtt{e}_k \rangle$$

This rule allows us to evaluate a message send by retrieving the appropriate method body from the object and applying it to the entire object itself. Note that the relation $\overset{eval}{\longrightarrow}$ represents one evaluation step, not full evaluation of an expression.

Record with two components. The first example is a form of point object that has constant $\mathtt{x}, \mathtt{y}$-coordinates:

$$\mathtt{r} \overset{def}{=} \langle \mathtt{x} = \lambda \mathtt{self}.\,\mathtt{3},\ \mathtt{y} = \lambda \mathtt{self}.\,\mathtt{2} \rangle$$

If we send the message $\mathtt{x}$ to $\mathtt{r}$, we may calculate the result by

$$\mathtt{r} \Leftarrow \mathtt{x} \quad \overset{eval}{\longrightarrow} \quad (\lambda \mathtt{self}.\,\mathtt{3})\,\mathtt{r} \quad \overset{eval}{\longrightarrow} \quad \mathtt{3}$$

where the second evaluation step is ordinary β-reduction from lambda calculus. This example may be generalized to show how any record may be represented as an object whose methods are constant functions. In particular, we may represent the record $\langle \mathtt{l}_1 = \mathtt{e}_1, \ldots, \mathtt{l}_k = \mathtt{e}_k \rangle$ by the object $\langle \mathtt{m}_1 = \mathtt{K}\mathtt{e}_1, \ldots, \mathtt{m}_k = \mathtt{K}\mathtt{e}_k \rangle$, where $\mathtt{K} = \lambda \mathtt{self}.\,\lambda \mathtt{x}.\,\mathtt{x}$.

One-dimensional point with `move` function. A more interesting object, which we will refer to again, is the following point object with an `x`-coordinate and `move` method. We could easily give a similar two-dimensional point with `x`- and `y`-coordinates, but the one-dimensional case illustrates the same ideas more simply.

$$\begin{array}{lll} \mathtt{p} & \overset{def}{=} & \langle \quad \mathtt{x} = \lambda \mathtt{self}.\,\mathtt{3}, \\ & & \qquad \mathtt{move} = \lambda \mathtt{self}.\,\lambda \mathtt{dx}.\ \langle \mathtt{self} \leftarrow \mathtt{x} = \lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} \rangle \\ & & \rangle \end{array}$$

The `move` method, when applied to the object itself and a displacement `dx`, replaces the `x` method with one returning a coordinate incremented by `dx`. This behavior is illustrated in the following example calculation, where we send the message `move` with parameter 2 to the object `p`:

$$\begin{aligned} \mathtt{p} \Leftarrow \mathtt{move}\,2 &= (\lambda\mathtt{self}.\,\lambda\mathtt{dx}.\langle\ \ldots\ \rangle)\ \mathtt{p}\ \ 2 \\ &= \langle \mathtt{p} \leftarrow \mathtt{x} = \lambda\mathtt{s}.(\mathtt{p} \Leftarrow \mathtt{x}) + 2\rangle \\ &= \langle \mathtt{p} \leftarrow \mathtt{x} = \lambda\mathtt{s}.\,3 + 2\rangle \\ &= \langle \mathtt{p} \leftarrow \mathtt{x} = \lambda\mathtt{self}.\,5\rangle \end{aligned}$$

Using a sound rule for object equality,

$$\langle\langle \mathtt{m_1}{=}\mathtt{e_1}, \ldots, \mathtt{m_k}{=}\mathtt{e_k}\rangle \leftarrow \mathtt{m_i}{=}\mathtt{e'_i}\rangle = \langle \mathtt{m_1}{=}\mathtt{e_1}, \ldots, \mathtt{m_i}{=}\mathtt{e'_i}, \ldots, \mathtt{m_k}{=}\mathtt{e_k}\rangle$$

we may reach the conclusion

$$\begin{aligned} \mathtt{p} \Leftarrow \mathtt{move}\,2 \;=\; \langle\ \ & \mathtt{x} = \lambda\mathtt{self}.\,5, \\ & \mathtt{move} = \lambda\mathtt{self}.\,\lambda\mathtt{dx}.\langle\ldots\rangle \\ \rangle\ \ & \end{aligned}$$

showing that the result of sending a `move` message with an integer parameter is an object identical to `p`, but with an updated `x`-coordinate.

Inherit `move` from point to color point. Our third introductory example shows how `x` and `move` are inherited when a color point is defined from `p` by adding a `color` method.

$$\mathtt{cp} \overset{def}{=} \langle \mathtt{p} \leftarrow\!\!+\ \mathtt{c} = \lambda\mathtt{self}.\,\mathtt{red}\rangle$$

If we send the `move` message to `cp` with the same parameter as above, we may calculate the resulting object in exactly the same way as before:

$$\begin{aligned} \mathtt{cp} \Leftarrow \mathtt{move}\,2 &= (\lambda\mathtt{self}.\,\lambda\mathtt{dx}.\langle\ \ldots\ \rangle)\ \mathtt{cp}\ \ 2 \\ &= \langle \mathtt{cp} \leftarrow \mathtt{x} = \lambda\mathtt{s}.(\mathtt{cp} \Leftarrow \mathtt{x}) + 2\rangle \\ &= \ldots \\ &= \langle \mathtt{cp} \leftarrow \mathtt{x} = \lambda\mathtt{self}.\,5\rangle \end{aligned}$$

with the final conclusion that

$$\begin{aligned} \mathtt{cp} \Leftarrow \mathtt{move}\,2 \;=\; \langle\ \ & \mathtt{x} = \lambda\mathtt{self}.\,5, \\ & \mathtt{move} = \lambda\mathtt{self}.\,\lambda\mathtt{dx}.\langle\ldots\rangle, \\ & \mathtt{c} = \lambda\mathtt{self}.\,\mathtt{red} \\ \rangle\ \ & \end{aligned}$$

The important feature of this computation is that the color `c` of the resulting color point is the same as the original one. While `move` was defined originally for points, which only have an `x`-coordinate, the method body performs the correct computation when the method is inherited by a more complicated object with additional methods.

In many cases, it is also useful to redefine an inherited method to exhibit more specialized behavior. This may be accomplished in our calculus by a combination of inheritance and method redefinition. For example, if we want an object to change to a darker color when moved, we could first define color points from points as above, obtaining a color point with the right type of `move` method. Then, `move` could be redefined (without changing its type) to have the right behavior.

Mutually recursive methods. As a technical simplification, our system is formulated so that methods are added to an object one at a time. This approach leads us to formulate our typing rules in a manner that makes it difficult to write object expressions with mutually recursive functions. More specifically, the static type system will only allow a method body to be added if all the other methods it refers to are already available from the object. For example, we cannot type the object expression

$$\langle\langle\rangle \longleftarrow\!\!+ \texttt{x\_plus1} = \lambda\texttt{self}.(\texttt{self} \Leftarrow \texttt{x}) + 1\rangle$$

which has a method referring to `x` but does not have an `x` method. The reason this object expression is not typeable is that if we send it the message `x_plus1`, the object will then send the message `x` to itself. But since the object does not have an `x` method, this is an error; it is precisely the error we aim to prevent with our type system. On the other hand, we may type the expression

$$\langle\langle\langle\rangle \longleftarrow\!\!+ x = \lambda\texttt{self}.\,3\rangle \longleftarrow\!\!+ \texttt{x\_plus1} = \lambda\texttt{self}.(\texttt{self} \Leftarrow \texttt{x}) + 1\rangle,$$

which is formed by first extending the empty object with an `x` method, then with the `x_plus1` method that refers to `x`.

The typing restriction that no method may refer to a method that the object does not have is inconvenient if we wish to add mutually-recursive methods `m` and `n` to some object. However, there is a standard idiom for adding mutually recursive methods. Specifically, we first extend the object by giving some method body for `m` that has the correct type but does not depend on `n`. Then, the object may be extended with the desired method body for `n`, referring to `m`. Finally, we replace the "dummy" method body for `m` with the desired method body referring to `n`. While this is a programming inconvenience, it is not a limitation in expressiveness. It therefore does not seem serious enough to merit complicating the typing rules in a way that alleviates the difficulty. In any "real" programming language based on our object calculus, we would expect there to be convenient syntactic sugar for simultaneously adding several, possibly mutually recursive, methods to an object.

3.3 Operational Semantics

In defining the operational semantics of our calculus, we must give rules for extracting and applying the appropriate method of an object. A natural way to approach this is to use a permutation rule

$$\langle\langle \texttt{e}_1 \leftarrow\!\!\circ\ \texttt{n}{=}\texttt{e}_2\rangle \leftarrow\!\!\circ\ \texttt{m}{=}\texttt{e}_3\rangle = \langle\langle \texttt{e}_1 \leftarrow\!\!\circ\ \texttt{m}{=}\texttt{e}_3\rangle \leftarrow\!\!\circ\ \texttt{n}{=}\texttt{e}_2\rangle$$

where $\mathtt{m}$ and $\mathtt{n}$ are distinct and each occurrence of $\leftarrow\!\!\circleddash$ may be either $\longleftarrow\!\!+$ or $\leftarrow$. Such a rule would let us treat objects as sets of methods, rather than ordered sequences. However, this equational rule would cause typing complications, since our typing rules only allow us to type object expressions when methods are added in an appropriate order. In particular, if we permute the methods of the object expression

$$\langle\langle\langle\rangle \longleftarrow\!\!+ \mathtt{x} = \lambda\mathtt{self.}\,3\rangle \longleftarrow\!\!+ \mathtt{x\_plus1} = \lambda\mathtt{self.}(\mathtt{self} \Leftarrow \mathtt{x}) + 1\rangle$$

then the subexpression

$$\langle\langle\rangle \longleftarrow\!\!+ \mathtt{x\_plus1} = \lambda\mathtt{self.}(\mathtt{self} \Leftarrow \mathtt{x}) + 1\rangle$$

is not well-typed, as described in the previous section. Therefore, the entire expression cannot be typed.

We circumvent the problem of method order using a more complicated "standard form" for object expressions, namely,

$$\langle\langle\langle \mathtt{m_1}{=}\mathtt{e_1}, \ldots \mathtt{m_k}{=}\mathtt{e_k}\rangle \leftarrow \mathtt{m_1}{=}\mathtt{e'_1}\rangle \ldots \leftarrow \mathtt{m_k}{=}\mathtt{e'_k}\rangle$$

where each method is defined exactly once, using some arbitrary method body that does not contribute to the observable behavior of the object, and redefined exactly once by giving the desired method body. Even if the two definitions of a method are the same, this form is useful since it allows us to permute the list of method redefinitions arbitrarily. More formally, in addition to the $\xrightarrow{eval}$ relation that allows us to evaluate object and function expressions, the operational semantics includes a subsidiary "bookkeeping" relation $\xrightarrow{book}$, which allows each object to be transformed into the "standard form" indicated above. The relation $\xrightarrow{book}$ is the congruence closure of the first four clauses listed in Table 3.1. These rules also allow the method redefinitions to be permuted arbitrarily. An important property of $\xrightarrow{book}$, proved in Section 7.14, is that if $\mathtt{e} \xrightarrow{book} \mathtt{e}'$, then any type for $\mathtt{e}$ is also derivable for $\mathtt{e}'$. This property would fail if we had the more general permutation rule discussed above.

The evaluation relation is the congruence closure of the union of $\xrightarrow{book}$ and the two evaluation clauses, (β) and $(\Leftarrow)$, at the bottom of Table 3.1. In other words, $\mathtt{e} \xrightarrow{eval} \mathtt{e}'$ if we may obtain $\mathtt{e}'$ from $\mathtt{e}$ by applying a bookkeeping or basic evaluation step to one subterm.

3.4 Static Type System

3.4.1 Pro Types and Message Send

The type of an object will be called a **pro***type*, short for *prototype.* The intuition is that expressions with **pro** type will be extensible objects.

The type defined by the type expression

$$\mathbf{pro}\, t\,.\langle\!\langle \mathtt{m}_1\colon \tau_1, \ldots, \mathtt{m}_k\colon \tau_k \rangle\!\rangle$$

(*switch ext ov*)	$\langle\langle \mathtt{e}_1 \leftarrow \mathtt{n}{=}\mathtt{e}_2\rangle \leftarrow\!\!+\ \mathtt{m}{=}\mathtt{e}_3\rangle$	$\xrightarrow{book}$	$\langle\langle \mathtt{e}_1 \leftarrow\!\!+\ \mathtt{m}{=}\mathtt{e}_3\rangle \leftarrow \mathtt{n}{=}\mathtt{e}_2\rangle$
(*perm ov ov*)	$\langle\langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2\rangle \leftarrow \mathtt{n}{=}\mathtt{e}_3\rangle$	$\xrightarrow{book}$	$\langle\langle \mathtt{e}_1 \leftarrow \mathtt{n}{=}\mathtt{e}_3\rangle \leftarrow \mathtt{m}{=}\mathtt{e}_2\rangle$
(*add ov*)	$\langle \mathtt{e}_1 \leftarrow\!\!+\ \mathtt{m}{=}\mathtt{e}_3\rangle$	$\xrightarrow{book}$	$\langle\langle \mathtt{e}_1 \leftarrow\!\!+\ \mathtt{m}{=}\mathtt{e}_3\rangle \leftarrow \mathtt{m}{=}\mathtt{e}_3\rangle$
(*cancel ov ov*)	$\langle\langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2\rangle \leftarrow \mathtt{m}{=}\mathtt{e}_3\rangle$	$\xrightarrow{book}$	$\langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_3\rangle$
(β)	$(\lambda \mathtt{x}.\, \mathtt{e}_1)\mathtt{e}_2$	$\xrightarrow{eval}$	$[\mathtt{e}_2/\mathtt{x}]\mathtt{e}_1$
($\Leftarrow$)	$\langle \mathtt{e}_1 \leftarrow\!\!\ominus\ \mathtt{m}{=}\mathtt{e}_2\rangle \Leftarrow \mathtt{m}$	$\xrightarrow{eval}$	$\mathtt{e}_2\langle \mathtt{e}_1 \leftarrow\!\!\ominus\ \mathtt{m}{=}\mathtt{e}_2\rangle$ where $\leftarrow\!\!\ominus$ may be either $\leftarrow\!\!+$ or $\leftarrow$.

Table 3.1: Bookkeeping and evaluation rules.

is a type with the property that any element e of this type is an object such that for $1 \le i \le k$, the result of $\mathtt{e} \Leftarrow \mathtt{m}_i$ is a value of type τ_i. A significant aspect of this type is that the bound type variable t may appear in the types $\tau_1, \ldots, \tau_k$. Thus, when we say $\mathtt{e} \Leftarrow \mathtt{m}_i$ will have type τ_i, we mean type τ_i with any free occurrences of t in τ_i referring to the type $\mathbf{pro}\, t.\langle\!\langle \mathtt{m}_1\!:\tau_1, \ldots, \mathtt{m}_k\!:\tau_k\rangle\!\rangle$ itself. Thus, $\mathbf{pro}\, t.\langle\!\langle \ \ldots\ \rangle\!\rangle$ is a special form of recursively-defined type.

The typing rule for message send has the form

$$\frac{,\ \vdash \mathtt{e} : \mathbf{pro}\, \mathtt{t}.\langle\!\langle \ldots, \mathtt{m}\!: \tau, \ldots\rangle\!\rangle}{,\ \vdash \mathtt{e} \Leftarrow \mathtt{m} : [\mathbf{pro}\, \mathtt{t}.\langle\!\langle \ldots, \mathtt{m}\!: \tau, \ldots\rangle\!\rangle / t]\tau}$$

where the substitution for t in τ reflects the recursive nature of **pro** types. This rule may be used to give the point p with x and move methods considered in Section 3.2.2 the type

$$\mathbf{pro}\, t.\langle\!\langle\, \mathtt{x}\!: \mathit{int},\ \mathtt{move}\!: \mathit{int} \to t\rangle\!\rangle,$$

since $\mathtt{p} \Leftarrow \mathtt{x}$ returns an integer and $\mathtt{p} \Leftarrow \mathtt{move}\ \mathtt{n}$ has the same type as p.

A subtle but very important aspect of the type system is that when an object is extended with an additional method, the syntactic type of each method does not change. For example, when we extend p with a color to obtain cp, also given in Section 3.2.2, we obtain an object with type

$$\mathbf{pro}\, t.\langle\!\langle\, \mathtt{x}\!: \mathit{int},\ \mathtt{move}\!: \mathit{int} \to t,\ \mathtt{c}\!: \mathit{color}\rangle\!\rangle$$

The important change to notice here is that although the syntactic type of move is still $\mathit{int} \to t$, the meaning of the variable t has changed. Instead of referring to the type of p, as it did originally, it now refers to the type of cp. This effect is what we have called method specialization: the type of a method may change when the method is inherited. For this kind of reinterpretation of type variables to be sound, the typing rule for object extension must insure that every possible type for a new method will be correct. The rule makes this guarantee through a form of implicit higher-order polymorphism.

Another subtle aspect of the type system is that objects which behave identically when we send them any sequence of messages may have different types. This difference reflects the fact that adding and redefining methods are also considered operations on objects. A simple example is given by the following two objects.

$$\begin{aligned}\mathtt{p} \overset{def}{=} \langle\; & \mathtt{x} = \lambda\mathtt{self}.\,3,\\ & \mathtt{move} = \lambda\mathtt{self}.\,\lambda\mathtt{dx}.\\ & \quad \langle\mathtt{self} \leftarrow \mathtt{x} = \lambda\mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx}\rangle\rangle\end{aligned}$$

$$\begin{aligned}\mathtt{q} \overset{def}{=} \langle\; & \mathtt{x} = \lambda\mathtt{self}.\,3,\\ & \mathtt{move} = \lambda\mathtt{self}.\,\lambda\mathtt{dx}.\ (\mathtt{p} \Leftarrow \mathtt{move}\ \mathtt{dx})\rangle\end{aligned}$$

It is not hard to see that `p` and `q` return the same results for any sequence of message sends. (Either we send `x`, which clearly gives the same result for `p` or `q`, or we send the message `move`. But since $\mathtt{q} \Leftarrow \mathtt{move}$ uses $\mathtt{p} \Leftarrow \mathtt{move}$, any sequence of subsequent messages will produce identical results.) However, `p` and `q` are *not* equivalent if we extend them with additional methods. The reason is that the first `move` method will preserve any additional methods added by object extension, but the second will not. This distinction is reflected in the type system in the following way. We can give `p` the first type below and `q` the second, but cannot give `q` the first type. Indeed, it would be unsound to do so.

$$\begin{aligned} point &= \mathbf{pro}\, t.\langle\!\langle\, \mathtt{x}\colon int,\ \mathtt{move}\colon int \to t \rangle\!\rangle \\ q\_point &= \mathbf{pro}\, t.\langle\!\langle\, \mathtt{x}\colon int,\ \mathtt{move}\colon int \to point \rangle\!\rangle \end{aligned}$$

A similar situation arises in Smalltalk, for example, where it is possible to have two classes that generate equivalent objects, but behave differently when we inherit from them. In adapting our type system to Smalltalk, we might expect to distinguish two such classes by type. The reason is that we wish the type of a Smalltalk class to not only give information about the behavior of objects, but also about the types of methods when they are inherited by other classes.

3.4.2 Types, Rows, and Kinds

The type expressions include type variables, function types, and **pro** types. It would not change the system in any substantial way to add type constants, but we will not need them here.

Types

$$\tau ::= t \mid \tau_1 \to \tau_2 \mid \mathbf{pro}\, t.R$$

Rows

$$R ::= r \mid \langle\!\langle\rangle\!\rangle \mid \langle\!\langle R \mid m{:}\tau \rangle\!\rangle \mid \lambda t.\, R \mid R\tau$$

Kinds

$$\begin{aligned} kind &::= T \mid \kappa \\ \kappa &::= T \to M \mid M \\ M &::= \{m_1, \ldots, m_k\} \end{aligned}$$

Row expressions appear as subexpressions of type expressions. Intuitively, rows list method names and their associated types. To insure that our method bodies are appropriately polymorphic, we will need *row variables*, listed as r in the above table. The row form $\langle\!\langle\rangle\!\rangle$ represents the *empty row*, the row containing no method names. The form $\langle\!\langle R \mid m : \tau \rangle\!\rangle$ extends row R with a new method name m of type τ. The kind system will insure that R did not already list an m method. Finally, because we want our method types to be polymorphic in the type of their host object, we need to be able to write *row functions*, functions from types to rows. Intuitively, we will use such functions to map the type of an object to its list of method name/type pairs, allowing such method types to depend on the type of their host object. The form $R\,\tau$ denotes applying row R to type τ. We will use the term "flat row" (in contrast to "row function") to denote rows of the forms $\langle\!\langle\rangle\!\rangle, \langle\!\langle R \mid m : \tau \rangle\!\rangle$, and $R\,\tau$. Row variables will always range over row functions.

We distinguish rows and types by kinds. In this chapter, types will be given kind T. (We refine this kind in the next chapter.) Flat rows have kinds of the form $\{m_1, \ldots, m_k\}$. Such a kind indicates its row does *not* include method names $m_1, \ldots, m_k$. We must statically know that some method does not appear to guarantee that methods are not multiply defined. Row functions have kinds of the form $T \to \{m_1, \ldots, m_k\}$, indicating that whenever such a function is applied to a type, it will produce a flat row guaranteed not to contain methods named $m_1, \ldots, m_k$.

The environments, or contexts, of the system list term, type, and row variables.

$$, \;::=\; \epsilon \mid ,\, , x{:}\tau \mid ,\, , t{:}T \mid ,\, , (r <: R :: \kappa)$$

Note that contexts are ordered lists, not sets.

The judgment forms are:

$, \vdash *$	well-formed context
$, \vdash e{:}\tau$	term has type
$, \vdash \tau{:}T$	type is well-formed
$, \vdash R_1 <:_w R_2$	row R_1 "subtype" of R_2
$, \vdash \tau_1 <: \tau_2$	type τ_1 subtype of τ_2
$, \vdash R :: \kappa$	row has kind

Although this language does not include a subsumption rule, we list subtyping judgments here because they are used to extract the types of methods from **pro** types.

3.4.3 Typing Rules

For expository purposes, the typing rules presented in this section are a simplification of the actual typing rules. We introduce the additional complexity in the next chapter because subtyping, the topic of the next chapter, is the source of the complexity. The rules discussed below, which involve prototype formation and message sending, are largely independent of the additional complexities. The complete type system appears in Chapter 6, where it is discussed in its entirety.

The empty object $\langle\rangle$ has the type $\mathbf{pro}\, t.\langle\!\langle\rangle\!\rangle$, as specified in the rule:

$$(\textit{empty pro}) \qquad \frac{,\ \vdash *}{,\ \vdash \langle\rangle : \mathbf{pro}\, t.\langle\!\langle\rangle\!\rangle}$$

The empty object has no methods and therefore cannot respond to any messages. However, this object can be extended with methods to obtain objects of other types.

The typing rule $(\textit{pro} \Leftarrow)$ for sending messages has the form described in the previous section. More precisely, the rule is as follows:

$$(\textit{pro} \Leftarrow) \qquad \frac{\begin{array}{c} ,\ \vdash e : \mathbf{pro}\, t.R \\ ,\ , t : T \vdash R <:_w \langle\!\langle m{:}\tau\rangle\!\rangle \end{array}}{,\ \vdash e \Leftarrow m : [\mathbf{pro}\, t.R/t]\tau}$$

The substitution for t in τ reflects the recursive nature of **pro** and **obj** types. This rule differs from the ones given in [FHM94, FM95a] by requiring only that we may derive $,\ , t : T \vdash R <:_w \langle\!\langle m{:}\tau\rangle\!\rangle$ instead of the more stringent requirement that R be of the form $\langle\!\langle R'|m : \tau\rangle\!\rangle$. This relaxation permits us to type message sends to objects whose types may be partially abstract (*i.e.*, types containing row variables). Another thing to notice about this rule is the subscript w on the subtyping judgment. This symbol indicates that only *width* subtyping was used to reach this conclusion. Our system will eventually support both width and depth subtyping on object types (see Chapter 4); however, it is essential to keep track of exactly where depth subtyping occurs to insure the soundness of certain operations.

The most subtle and complicated rule of the system is the $(\textit{pro} \longleftrightarrow\!\!+)$ typing rule for adding methods to objects, which appears below. In this rule we assume e_1 is an object of some **pro** type and that e_1 does not include a method m, to be added. The final assumption is a typing for e_2, the expression to be used as the method body for m. The first thing to notice about the typing for e_2 is that it contains a row variable r, which is implicitly universally quantified. Because of this quantification, e_2 will have the indicated type for any substitution of row expression R' for r, provided R' has an appropriate kind and is a subtype of $\lambda t.\langle\!\langle R \,|\, m : \tau\rangle\!\rangle$. (See Lemma 7.13.1 in

Chapter 7.) This flexibility is essential, since it implies that e_2 will have the required functionality for every possible future extension of $\langle e_1 \longleftarrow\!\!+\, m = e_2\rangle$. The second important property of the typing for e_2 is that the type has the form $t \to \tau$, with a **pro** type substituted for t. Although t is hidden in the **pro** type of $\langle e_1 \longleftarrow\!\!+\, m = e_2\rangle$, t must appear in the hypothesis because sending the message m to $\langle e_1 \longleftarrow\!\!+\, n = e_2\rangle$ results in the application of e_2 to this object. This rule has the side condition that $r \notin FV(\tau)$, indicating that row variable r does not appear in the free variables of τ.

$$(pro\ ext)\quad \frac{\begin{array}{l} , \vdash e_1 : \mathbf{pro}\, t.R \\ , , t{:}T \vdash R :: \{m\} \\ , , r <:_w \lambda t.\langle\!\langle R \mid m : \tau\rangle\!\rangle :: T \to \emptyset \vdash e_2 : [\mathbf{pro}\, t.rt/t](t \to \tau) \qquad r \notin FV(\tau) \end{array}}{, \vdash \langle e_1 \longleftarrow\!\!+\, m{=}e_2\rangle : \mathbf{pro}\, t.\langle\!\langle R \mid m{:}\tau\rangle\!\rangle}$$

The rule for redefining, or overriding, a method body (*pro ov*) has the same form, but is slightly simpler. Rule (*pro ov*) appears in full detail in Appendix D.

3.4.4 Example Typing Derivations

We will present a sample typing derivation at the end of the next section, after the additional complexity has been introduced and we can use the full typing rules.

3.5 Related Object Models

Early models of object-oriented programming were based on recursive records. This approach, in which objects are encoded as recursive records of access functions, was developed by William Cook and others [Coo87, Coo89a, CHC90] using concepts pioneered by Luca Cardelli [Car88, CW85]. The recursion is necessary to give each of the methods of an object access to the other components. Although quite useful for developing denotational models of untyped object-oriented languages, it proved difficult to extend this model to a typed version reflecting the intuitive properties of typed object-oriented languages. An additional problem is that the operation of method override is difficult to model, as the new method is in some sense "outside" the recursion.

In [PT94], Pierce and Turner introduced an encoding of objects as elements of existential type. Each object consists of a record containing the "state" of the object and a record of "methods" that operate on the state. The existential type then hides the state component, making only the record of methods externally visible. The advantages of this approach are that the encoding makes the hiding of private "instance variables" explicit and it renders type recursion non-essential. A disadvantage is that message sending is not a uniform operation: it depends on the type of the method being invoked.

The most closely related work to the material described in this chapter is that of Abadi and Cardelli. Abadi's Baby Modula-3 [Aba94] introduced a very simple object calculus with method override and a static form of object extension. Together, Abadi and Cardelli have developed a family of object calculi, including untyped, first-order [AC96c] second-order [AC95], and imperative versions [AC96a]. Abadi and Cardelli have collected this material into the book [AC96b]. The crucial difference between the object calculi that they have developed and the one presented here is that their system supports only the operations of message sending and method override, not object extension. Because of this limitation, their system directly supports width subtyping. In contrast, adding subtyping to our system is more difficult. (We will see how it may be accomplished in the next chapter.) As this comparison suggests, the choice of whether or not to support object extension has major ramifications as to the style of the type system required. In particular, supporting object extension requires method bodies to be much more polymorphic and lends itself to row-based approaches to typing.

In contrast to the system presented in this chapter, neither the recursive record nor the existential model provides explicit support for inheritance; instead they requiring complex encodings of class-like constructs to model object-implementation reuse. The method override operation of the Abadi-Cardelli calculi provides some direct support for inheritance. In [AC96c], Abadi and Cardelli show a relatively simple encoding of inheritance without using an object extension operation. We will discuss this encoding in more detail in Section 5.6.1.

3.6 Summary

We have given a typed calculus of functions and objects with a sound type system, although the full presentation of that system is deferred to the next chapter. In this "kernel language," we define objects and their interfaces (called **pro** types here) directly instead of through some subsidiary calculus of records. This approach gives an axiomatic presentation of a simple object-oriented language and its type system in a form that we hope is conducive to further work. A feature of the calculus is method specialization: using method redefinition (expressions of the form $\langle e \leftarrow n = e' \rangle$), we may define functions whose type and behavior change in a natural and useful way as a result of inheritance. This capability seems very difficult to achieve directly with any calculus of records. While it seems too early to claim that we have captured "the essence of inheritance in a simple form," it seems that some progress has been made in this direction.

There are many technical open problems, including development of a denotational model and a proof system for equivalence that is sufficiently powerful to derive nontrivial equations between method bodies. Hopefully, the calculus presented here will provide a basis for studying these mathematical problems in a manner that is faithful to substantial uses of object-oriented programming in practice.

Chapter 4

Adding Subtypes

In languages that support pure object-based inheritance, such as the one described in the previous chapter, no subtyping is possible. We did not realize this complication in writing [FHM94]. Consider the intuitive definition of a subtype: A is a subtype of B if we may use an object of type A in any context expecting an object of type B. If objects of type A are to be used as B's, then A-objects must have all of the methods of B-objects. Because method addition is a legal operation on objects in object-based languages, objects with extra methods cannot be used in some contexts where an object with fewer methods may. As an example, a colored point object cannot be used in a context that will add color, but a point object can. For A to be a subtype of B then, A's must contain exactly the same methods as B's. It is not even possible to specialize the types of methods that appear in both A's and B's, since the so-called "depth" object subtyping is unsound when method override is a legal operation on objects. This observation is made in [AC96c]. The following example, discussed in [AC96c], illustrates the problem.

Consider the object types A and B:

$$\begin{array}{rcl} B & \overset{def}{=} & \mathbf{pro}\,\mathtt{t}.\langle\!\langle \mathtt{x}{:}\, \mathit{int},\ \mathtt{y}{:}\, \mathit{real} \rangle\!\rangle \\ A & \overset{def}{=} & \mathbf{pro}\,\mathtt{t}.\langle\!\langle \mathtt{x}{:}\mathit{posint},\ \mathtt{y}{:}\, \mathit{real} \rangle\!\rangle \end{array}$$

If we allow depth subtyping, then $A <: B$ since $\mathit{posint} <: \mathit{int}$. Now consider an object $\mathtt{a}$ defined as follows:

$$\mathtt{a} \overset{def}{=} \langle \mathtt{x} = \mathtt{1},\ \mathtt{y} = \lambda\mathtt{self.ln(self} \Leftarrow \mathtt{x)} \rangle$$

We can see that $\mathtt{a}$ has type A. By the subsumption rule, we may consider $\mathtt{a}$ to have type B. With this typing, the expression $\langle \mathtt{a} \leftarrow \mathtt{x} = -\mathtt{1} \rangle$ is legal. But then sending the message $\mathtt{y}$ to $\mathtt{a}$ produces a run-time type error.

Because of this complete elimination of subtyping for pure object-based languages, the system described in [AC96c] does not permit object extension as a run-time operation, instead supporting width subtyping on object types.

In this chapter, we present a second way of resolving this conflict. Intuitively, the main idea is to introduce a phase distinction (on a per object basis) that is enforced by the static type system. (*c.f.* Baby Modula-3 [Aba94] for a "baby" form of this phase distinction.) During the initial phase, objects are created via the inheritance primitives of method override and object extension, as described in the previous chapter. Only trivial subtyping is permitted between object types during this phase, for reasons outlined above. When the object is finished, it may be "converted" to a second form of object that no longer supports inheritance. Because of this restriction, the second form of object is free to support rich subtyping. The "conversion" corresponds to changing the type of the object from the form of object type described in the previous chapter to a form with the expected subtyping properties for object types.

By introducing this phase distinction, we support both object-based inheritance and subtyping, at the cost of some increase in the complexity of the type system. In [BL95], Bono and Liquori develop an alternate way of extending the previous chapter's calculus with subtyping. We review their approach in Section 4.3.

The two forms of object type used in this chapter highlight the distinction between two interfaces: the inheritance-interface and the client-interface of an object. This distinction is essentially familiar from object-oriented languages and databases, but often not explicitly mentioned in language documentation. If we write a C++ class such as *stack,* then there are really two separate ways of using this class. The first is by calling the constructor of the class to create *stack* objects and then calling their member functions. The second use of the *stack* class is by defining a derived class. (This "reuse" of implementation is traditionally called inheritance.) One way to see that these are very different uses of a class is to notice that they induce very different notions of class equivalence. If we just want the objects constructed by a class to behave in the same way, then we can perform a number of optimizations and program transformations. However, some transformations that preserve the behavior of constructed objects would observably change the behavior of derived classes. A simple example occurs when a class has two mutually recursive member functions, say f and g. If we replace these functions by two equivalent non-recursive functions, we do not change the behavior of constructed objects. However, this transformation may radically alter the behavior of a derived class if both are virtual functions and one is redefined in the derived class. An innovation of our object calculus, when compared with other recent work such as [AC96c, Bru93, FHM94, PT94] (summarized and compared in [FM95b]), is the type distinction between these two uses of a class.

The distinction between inheritance and client interfaces leads us to distinguish *prototypes* and *objects.* Intuitively, a prototype is a collection of methods that may be used to implement one or more objects. The operations on prototypes are (i) sending a message (which results in the invocation of a method), (ii) adding a new method (method addition), and (iii) redefining an existing method without changing its type (method override). The type of a prototype is called a **pro** type; we saw them in the previous chapter. For the reasons described above, only trivial subtyping exists between

pro types.

Intuitively, an object is created by "packaging" a prototype, an operation that does nothing except "seal" the prototype so that no methods can be added or redefined. In our calculus, the "sealing" operation only changes the type from a **pro** type to an object, or **obj** type. Although the only operations allowed on objects from the "outside" are message sends, the methods within an object can override the methods of their host object. This ability to override methods internally permits methods such as a point object's `move` method to replace its host object's `x` method with a new location. Preventing external method override and method extension by sealing prototypes is sufficient, however, to make interesting subtyping sound for objects. The internal redefinitions do not cause an unsoundness in the type system because they are type-checked with precise knowledge of the **pro** type of the expression to which they were added.

In addition to subtyping, the distinction between prototypes and objects provides some useful insight into the two uses of classes mentioned above. Inheritance on classes will be modeled in our system (in the next chapter) by operations on prototypes. Similarly, creating an object from a class will be modeled by "sealing" a prototype to an object. The distinction between **pro** and **obj** types clearly illustrates the difference between a derived class's interface to a class and a client program's view of the objects created from that class. In particular, inheritance depends on the presence *and* absence of the methods in a prototype, whereas a client is only concerned with the presence of the methods it uses.

4.1 Static Type System

4.1.1 Subtyping and Delegation

In this section, we give a type system for the object calculus described in Section 3.2.1 that is an extension of the one given in the previous chapter. The major change to the earlier system is the addition of **obj** types, which permit subtyping in both width and depth. In both the earlier system and the one we describe here, expressions with **pro** type may have new methods added via method addition ($\leftarrow\!\!+$) and old methods replaced via method override($\leftarrow$). As described above, only trivial subtyping is sound for such types. In the system presented here, we may decide that we are no longer interested in modifying the methods of an expression with **pro** type. At this point, we may "seal" the expression by converting it to a value with an **obj** type. Since $\leftarrow$ and $\leftarrow\!\!+$ are not valid operations on **obj** types, width and depth subtyping are sound for these types. We call expressions given **pro** type *prototypes* because they may either be sent messages or modified to create new kinds of prototypes. Expressions with **obj** type are called *objects* since such expressions may be sent messages or subtyped, but not extended or modified. In the following, we will use the meta-variable **probj** to denote either **obj** or **pro**.

4.1.2 Adding Subtyping

To insure that subtyping is sound for **obj** expressions, we disable object extension and object override on them, permitting them only to receive messages. Since the methods handling such messages are defined when the expression has a **pro** type, however, it is possible for these methods to redefine themselves or other methods of the host expression. (The polymorphism requirement for method bodies prevents them from adding new methods.) These redefinitions do not cause an unsoundness in the type system, however, because they were type-checked with precise knowledge of the **pro** type of the expression to which they were added. The unsoundness discussed above arises when the type of an expression is promoted via subsumption and then a method is replaced by another method with higher type. This scenario cannot occur in our system because once we seal an expression to a potentially imprecise **obj** type, methods cannot be overridden from the outside, where only the imprecise sealed type is available. Internal methods, type-checked with precise **pro** type information about the expression to which they were added, are type-safe.

Since sealing a prototype e requires no change to e, we would have liked to be able to use the following rule in conjunction with a subsumption rule to "seal" prototypes:

$$(\textit{seal-unsound}) \qquad \frac{, \vdash \mathbf{obj}\, t.R : T}{, \vdash \mathbf{pro}\, t.R <: \mathbf{obj}\, t.R}$$

where the judgment $, \vdash \mathbf{obj}\, t.R : T$ denotes that type $\mathbf{obj}\, t.R$ is well formed. Unfortunately, this rule is unsound when the variable t appears contravariantly (or invariantly) in R. Hence we need a more complicated rule, which happens to have the same form as the rule for deriving subtyping judgments on **obj** types. Because of this similarity, we may combine the sealing rule and the **obj** subtyping rule to produce a typing rule of the form:

$$(<: \textit{obj}) \qquad \frac{\begin{array}{c}, , t : T \vdash R_1 <: R_2 \\ t \text{ covariant in } R_2\end{array}}{, \vdash \mathbf{probj}\, t.R_1 <: \mathbf{obj}\, t.R_2}$$

We show how to formalize this rule and the condition"t covariant in R_2" in the next section. The meta-variable **probj** is **pro** when this rule is used to seal prototypes. When used simply for subtyping between object types, **probj** is **obj**. Since this rule allows subtyping in both width and depth between the row expressions R_1 and R_2, this rule gives us subtyping in both width and depth for **obj** types. This rule has the somewhat unfortunate consequence that we cannot seal prototypes to **obj** types containing methods that are contravariant (or invariant) in the bound type variable. However, this appears to be a fundamental trade-off and limitation.

4.1.3 Rows, Types, and Kinds

More formally, the type expressions are given by:

Types

$$\tau ::= \quad t \,|\, \tau_1 \rightarrow \tau_2 \,|\, \mathbf{pro}\, t.R \,|\, \mathbf{obj}\, t.R$$

Rows

$$R ::= \quad r \mid \langle\!\langle\rangle\!\rangle \mid \langle\!\langle R \mid m{:}\, \tau \rangle\!\rangle \mid \lambda t.\, R \mid R\tau$$

Variance Annotations

$$b ::= \quad + \mid - \mid o$$
$$a ::= \quad b \,|\, ?$$

Variance Sets

$$V ::= \quad \{\vec{t^b}\}$$

Method Annotations

$$M ::= \quad \{\vec{m}\}$$

Kinds

$$k ::= \quad V \,|\, \kappa$$
$$S ::= \quad T^a \qquad \text{Symbol } T \text{ is a terminal.}$$
$$\kappa ::= \quad S \rightarrow \nu \mid \nu$$
$$\nu ::= \quad (M;\, V)$$

The type expressions are just those of Chapter 3 extended with **obj** types. To avoid unnecessary repetition in our presentation, we use the meta-variable **probj** to denote either **obj** or **pro**. As we saw in the previous chapter, a *row* is either a finite list of *method name, type* pairs or a function from a type to such a list. Row expressions appear as subexpressions of type expressions, with rows and types distinguished by kinds.

The kinds described here are more detailed than those in the previous chapter; in addition to the method absence annotations, they include *variance information.* This information, which tells whether a variable appears covariantly, contravariantly, or invariantly in type and row expressions, is necessary for subtyping judgments involving types of the form $\mathbf{pro}\, t.R$ or $\mathbf{obj}\, t.R$. Such information is required because R may contain row variables, making it impossible to infer subtyping relationships by simply inspecting the types in question. Intuitively, the elements of kind V are types whose free type variables appear with the variance indicated in V. For example, if t^+ appears in the variance set of a type τ, then t appears covariantly in τ. Similarly, if t^- is in V, then t appears contravariantly. Finally, an annotation of o indicates the variable appears invariantly in τ. Similarly, the elements of kind $(\{\vec{m}\};\, V)$ are flat rows whose the free type variables appear with variance indicated in variance set V. Row functions have kinds of the form $T^a \rightarrow (\{\vec{m}\};\, V)$. The annotation a specifies the variance of the abstracted variable. A value of ? for a indicates that the given row function ignores its argument. Variance set V reflects the variances of the free type variables in the body of the row function.

The environments, or contexts, of the system list term, type, and row variables.

$$, \ ::= \ \epsilon \,|\, , \ ,x{:}\tau \,|\, , \ ,t{:}V \,|\, , \ ,r <:_w R :: \kappa$$

Note that contexts are ordered lists, not sets.

The judgment forms are:

$, \vdash *$	well-formed context
$, \vdash \tau{:}V$	well-formed type with variance V
$, \vdash R :: \kappa$	row has kind
$, \vdash \tau_1 <: \tau_2$	type τ_1 subtype of τ_2
$, \vdash R_1 <:_B R_2$	row R_1 is a B-subtype of R_2
	B gives width vs. depth information
$, \vdash e{:}\tau$	term e has type τ

The only unusual judgment in the above list is the form for writing "subtyping" relations between row expression, namely $, \vdash R_1 <:_B R_2$. (We use the terminology "subtyping", instead of the more correct term "subrowing," for convenience.) Because width and depth subtyping have very different properties with respect to our object-inheritance operations, it is crucial that we be able to track exactly when depth subtyping occurs. To that end, we use the subscript B on the subtype symbol $<:$ to indicate whether or not any depth subtyping rules were used to infer the subtyping relationship between R_1 and R_2. In particular, if B is w, then only width subtyping rules were used. If B is w, d, then both width and depth subtyping rules may have been used.

4.1.4 Typing Rules

The key rule in this type system is the rule that allows us to convert from **pro** to **obj** types, which we described informally in the previous section. In full detail, the rule is:

$$(<:\ obj) \qquad \frac{\begin{array}{c} , \ , t : \{t^+\} \vdash R_1 <:_B R_2 \\ , \vdash \mathbf{probj}\, t.R_1 : V_1 \\ , \ , t : \{t^+\} \vdash R_2 :: (M;\, V_2) \qquad Var(t, V_2) \in \{?, +\} \end{array}}{, \vdash \mathbf{probj}\, t.R_1 <: \mathbf{obj}\, t.R_2}$$

The first assumption requires that the list of methods for the lower row, R_1, be a subtype of the list of methods for the upper row, R_2. For purposes of this rule, it does not matter if R_1 is a width-only or a width-and-depth subtype of R_2; hence the subscript on the subtyping symbol is left unspecified. The second assumption is a well-formedness condition that helps insure that any type appearing in a subtype judgment is itself well-formed. The final two assumptions assert that the type variable t appears covariantly in R_2. The variance of t in R_1 is irrelevant for the soundness of this subtyping.

4.1.5 Variance Analysis

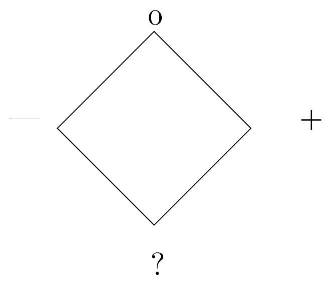
To get a feel for our variance analysis, we present a few of the typing rules related to tracking variance. The first of these, *(type arrow)*, calculates the variance of $\tau_1 \rightarrow \tau_2$ from the variances of τ_1 and τ_2:

$$(\textit{type arrow}) \qquad \frac{, \vdash \tau_1 : V_1 \qquad , \vdash \tau_2 : V_2}{, \vdash \tau_1 \rightarrow \tau_2 : \mathit{Merge}(\overline{V_1}, V_2)}$$

The function $\mathit{Merge}(V_1, V_2)$ combines variance sets V_1 and V_2 so that the variance of each type variable in the resulting set is the least upper bound of the variance of that type variable in V_1 and V_2, where variance annotations are ordered as follows:

$$+ \leq o, \; - \leq o, \; ? \leq -, \; ? \leq +$$

Pictorially, we have the ordering:



This definition implies, for example, that if t^+ is in V_1 and t^- is in V_2, then t^o will be in $\mathit{Merge}(V_1, V_2)$. The overline function $\overline{V_1}$ inverts the variance of each type variable in V_1, changing $+$'s to $-$'s and vice versa while leaving $?$'s and o's unchanged. The definitions of Merge and the inversion function appear in Appendix C.

The typing rule for analyzing the variance of **pro** types, *(pro)*, calculates the variance of $\mathbf{pro}\, t.R$ by finding the variance of R, removing t (since t becomes a bound variable in $\mathbf{pro}\, t.R$), and then making all remaining type variables invariant because **pro** types support only trivial subtyping:

$$(\textit{pro}) \qquad \frac{, , t : \{t^+\} \vdash R :: (M;\, V)}{, \vdash \mathbf{pro}\, t.R : \mathit{Invar}(V \setminus t)}$$

The function $\mathit{Invar}(V)$ returns the variance set that contains the same type variables as V, each annotated with o. The notation $V \backslash t$ indicates the variance set just like V except without type variable t. The formal definitions of these two functions also appear in Appendix C.

Finally, there are four different versions of the rule for analyzing the variance of applying a row function to a type argument, one for each possible variance of the abstracted type variable. The general form of these rules is:

$$(row\ fn\ app\ a) \qquad \frac{\Gamma \vdash R :: T^a \rightarrow (M;\, V_1) \qquad \Gamma \vdash \tau : V_2}{\Gamma \vdash R\tau :: (M;\, f_a(V_1, V_2))}$$

How the two variance sets V_1 and V_2 are combined depends on the value of a, the variance of the abstracted variable. In particular,

$$\begin{array}{rcl} f_+(V_1, V_2) &=& Merge(V_1, V_2) \\ f_-(V_1, V_2) &=& Merge(V_1, \overline{V_2}) \\ f_o(V_1, V_2) &=& Merge(V_1, Invar(V_2)) \\ f_?(V_1, V_2) &=& V_1 \end{array}$$

The other non-routine rules in the type system either generalize those from Chapter 3, keep track of the variance of type variables in row and type expressions, or show subtyping relations between rows and types based on the variance analysis. A complete list of the typing rules for the language fragment described so far appears in Appendix D. This appendix contains all of the typing rules for the full system; to obtain the system we have described so far, we need only omit the $(exist)$, $(\exists <: intro)$, and $(\exists <: elim)$ typing rules.

4.1.6 Soundness

Type soundness for this system follows from the soundness of the next system, whose proof is presented in Chapter 7.

4.2 Examples

4.2.1 Example Typing Derivation

We claimed earlier that we should be able to prove the following two typings:

$$\begin{array}{lcl} \texttt{p} &:& \mathbf{pro}\, t.\langle\!\langle \texttt{x}\colon int,\ \texttt{move}\colon int \rightarrow t \rangle\!\rangle \\ \texttt{q} &:& \mathbf{pro}\, t.\langle\!\langle \texttt{x}\colon int,\ \texttt{move}\colon int \rightarrow point \rangle\!\rangle, \end{array}$$

and argued that it is unsound to give `q` the first type. To illustrate the use of the typing rules, we prove the first of these in Table 4.1. The second is proved similarly. In addition, we can prove that the object expression

$$\texttt{cp} \;=\; \langle \texttt{p} \leftarrow\!\!+ \texttt{c} = \lambda\texttt{self.red} \rangle$$

Contexts

$$\begin{array}{rcl} ,_1 & = & r <:_w \lambda t.\langle\!\langle \texttt{x}{:}\, int, \texttt{move}{:}\, int \to t\rangle\!\rangle :: \kappa_{min(\Gamma)} \\ & & \texttt{self} : \textbf{pro}\, t\textbf{.}r\, t \\ & & \texttt{dx} : int \\ ,_2 & = & ,_1,\ r <:_w \lambda t.\langle\!\langle \texttt{x}{:}\, int, \texttt{move}{:}\, int \to t\rangle\!\rangle :: \kappa_{min(\Gamma)} \\ & & \texttt{p} : \textbf{pro}\, t\textbf{.}r'\, t \end{array}$$

Derivation (assuming $\epsilon \vdash \langle \texttt{x} = \lambda\texttt{self}.\, 3\rangle : \textbf{pro}\, t\textbf{.}\langle\!\langle \texttt{x}{:}\, int\rangle\!\rangle$)

$$\begin{array}{rcl} ,_2 & \vdash & (\texttt{self} \Leftarrow \texttt{x}) + \texttt{dx} : int \\ ,_2 - \texttt{p} & \vdash & \lambda\texttt{p}.(\texttt{self} \Leftarrow \texttt{x}) + \texttt{dx} : \textbf{pro}\, t\textbf{.}r' t \to int \\ ,_1 & \vdash & \langle \texttt{self} \leftarrow \texttt{x} = \lambda\texttt{p}.(\texttt{self} \Leftarrow \texttt{x}) + \texttt{dx}\rangle : \textbf{pro}\, t\textbf{.}r\, t \\ ,_1 - \texttt{dx} - \texttt{self} & \vdash & \lambda\texttt{self}.\, \lambda\texttt{dx}.\langle \texttt{self} \leftarrow \texttt{x} = \lambda\texttt{p}.(\texttt{self} \Leftarrow \texttt{x}) + \texttt{dx}\rangle \\ & & : [\textbf{pro}\, t\textbf{.}r\, t/t](t \to int \to t) \\ \epsilon & \vdash & \langle\langle \texttt{x} = \lambda\texttt{self}.\, 3\rangle \leftarrow\!\!+ \texttt{move} = \lambda\texttt{self}.\, \lambda\texttt{dx}.\langle \texttt{self} \leftarrow \texttt{x} = \lambda\texttt{p}.(\texttt{self} \Leftarrow \texttt{x}) + \texttt{dx}\rangle\rangle \\ & & : \textbf{pro}\, t\textbf{.}\langle\!\langle \texttt{x}{:}\, int, \texttt{move}{:}\, int \to t\rangle\!\rangle \end{array}$$

Table 4.1: Example typing derivation.

has type

$$\textbf{pro}\, t\textbf{.}\langle\!\langle\, \texttt{x}{:}\, int, \texttt{move}{:}\, int \to t, \texttt{c}{:}\, color\rangle\!\rangle$$

by similar means. These examples are intended to demonstrate that the type system captures the desired form of method specialization.

4.2.2 Subtyping Examples

To give some intuition for this calculus, we present two examples, neither of which is appropriately typeable in the system described in Chapter 3. The first example below illustrates the use of subtyping for code reuse, while the second demonstrates its use for encapsulation.

4.2.2.1 Subtyping for Code Reuse

Consider the function

$$\texttt{average} \overset{def}{=} \lambda\texttt{p}_1.\, \lambda\texttt{p}_2.((\texttt{p}_1 \Leftarrow \texttt{x}) + (\texttt{p}_2 \Leftarrow \texttt{x}))/2$$

We may give this expression type:

$$\texttt{average} \ :\ \textbf{obj}\, \texttt{t}\textbf{.}\langle\!\langle \texttt{x}{:}\, int\rangle\!\rangle \to \textbf{obj}\, \texttt{t}\textbf{.}\langle\!\langle \texttt{x}{:}\, int\rangle\!\rangle \to int$$

(assuming integer division). With this type, we may apply the average function to any two objects that both support integer x-method. As an example of this flexibility, consider the following two expressions:

$$\begin{aligned}
\texttt{p} &\overset{def}{=} \langle\ \texttt{x} = \lambda\texttt{self}.\,3, \\
&\qquad \texttt{move} = \lambda\texttt{self}.\,\lambda\texttt{dx}.\ \langle\texttt{self} \leftarrow \texttt{x} = \lambda\texttt{s}.(\texttt{self} \Leftarrow \texttt{x}) + \texttt{dx}\rangle\rangle \\
\texttt{cp} &\overset{def}{=} \langle\ \texttt{p} \leftarrow\!+\ \texttt{c} = \lambda\texttt{self}.\texttt{red}\ \rangle
\end{aligned}$$

As in the system from the previous chapter, we may give `p` and `cp` the following types:

$$\begin{aligned}
\texttt{p} &: \textbf{pro}\,\texttt{t}.\langle\!\langle\, \texttt{x}\colon \mathit{int},\ \texttt{move}\colon \mathit{int} \to t \rangle\!\rangle \\
\texttt{cp} &: \textbf{pro}\,\texttt{t}.\langle\!\langle\, \texttt{x}\colon \mathit{int},\ \texttt{move}\colon \mathit{int} \to t,\ \texttt{c}\colon \mathit{color} \rangle\!\rangle
\end{aligned}$$

In the new system, we may seal these expressions and give them both the following common supertype:

$$\texttt{p}, \texttt{cp}\ :\ \textbf{obj}\,\texttt{t}.\langle\!\langle \texttt{x}\colon \mathit{int} \rangle\!\rangle$$

Hence we may give the expression `average p cp` type *int*. Without retyping `average`, we could also give each of the expressions `average p p`, `average cp p`, and `average cp cp` type *int*. Thus this type system allows us to use the function `average` to calculate the average of color points and points interchangeably.

4.2.2.2 Subtyping for Encapsulation

Once we change the type of an expression from **pro** type to **obj** type, the methods of that prototype become "read-only," including those methods that are effectively data fields. Hence if the designers of a prototype wish its users to be able to change the values of some of its methods after it is sealed, they must provide methods to change those values. This restriction provides a mechanism whereby prototype designers can guarantee invariants for the objects created from their prototypes.

Suppose, for example, we were interested in color points whose color was guaranteed to be either blue or green. Then consider the expression:

$$\begin{aligned}
\texttt{cp}_2 \overset{def}{=} \langle\ & \texttt{x} = \ldots, \\
& \texttt{move} = \ldots, \\
& \texttt{c} = \lambda\texttt{self}.\,\texttt{blue}, \\
& \texttt{makeBlue} = \lambda\texttt{self}.\langle\texttt{self} \leftarrow \texttt{c} = \texttt{blue}\rangle, \\
& \texttt{makeGreen} = \lambda\texttt{self}.\langle\texttt{self} \leftarrow \texttt{c} = \texttt{green}\rangle, \\
\rangle\ &
\end{aligned}$$

As in the original system, we may give this expression type:

$$\texttt{cp}_2 : \textbf{pro}\,\texttt{t}.\langle\!\langle\, \texttt{x}\colon \mathit{int},\ \texttt{move}\colon \mathit{int} \to t,\ \texttt{c}\colon \mathit{color},\ \texttt{makeBlue}\colon t,\ \texttt{makeGreen}\colon t \rangle\!\rangle$$

With this type, the user may override $\texttt{cp}_2$'s `c` method to give $\texttt{cp}_2$ any color. However, we may prevent this undesired change by sealing $\texttt{cp}_2$. We may use the rule $(<\colon \mathit{obj})$ and subsumption to give $\mathtt{cp}_2$ the type:

$$\mathtt{cp}_2 : \mathbf{obj}\, \mathtt{t}.\langle\!\langle\, \mathtt{x}\colon \mathit{int},\ \mathtt{move}\colon \mathit{int} \to t,\ \mathtt{c}\colon \mathit{color},\ \mathtt{makeBlue}\colon \mathtt{t},\ \mathtt{makeGreen}\colon t \rangle\!\rangle$$

With this type, users may no longer set the color of $\mathtt{cp}_2$ by overriding its value for `c`. Now they may only change its color by sending the messages `makeBlue` and `makeGreen`, a fact which guarantees that the object only takes on legal colors.

4.3 Related Work: Adding Subtyping to a Language with Object Extension

The authors of [BL95] approach the trade-off between delegation and subtyping in a very different way. Their system, which is an extension of [FHM94], permits a limited form of width subtyping on expressions with (what we call) **pro** type. In particular, they consider $\mathbf{pro}\, t.R$ a subtype of $\mathbf{pro}\, t.R'$ so long as R contains all the methods of R' (with matching types) and none of the "forgotten" methods are referred to in the methods listed in R'. This second condition guarantees that if we forget about the presence of some method `m` via subsumption and then re-add `m` with a potentially unrelated type, we cannot violate any typing assumptions the other methods of the object may have made about `m`. This guarantee is essential to the soundness of this approach to adding subtyping; however, its inability to "forget" methods referred to by "remembered" methods is somewhat unfortunate. In particular, it cannot type examples such as that given in Section 4.2.2.2, where desired invariants about an object are maintained by restricting access to certain methods via subsumption.

4.4 Conclusion

In this chapter, we presented a type system for an object calculus that extends previous work in [Mit90, FHM94]. The main additions to our previous object calculus are the distinction between prototypes and objects and the addition of subtyping. The distinction between prototypes, which allow method addition and override, and objects, which only receive messages, is essential for subtyping, since previous studies show an incompatibility between object extension, method override, and non-trivial subtyping [FM95b].

Chapter 5

Classes

There are several forms of object-oriented languages, including class-based languages such as Simula [BDMN73], Smalltalk [GR83], C++ [ES90], Java [AG96], and Eiffel [Mey92], prototype-based languages such as Self [US91] and Obliq [Car95], and multi-method-based approaches such as CommonLisp [Ste84] and Cecil [Cha95]. This chapter is concerned with the study of class-based languages and the relationship between three language constructs: classes, prototype-based object primitives, and traditional data abstraction of the form found in languages such as Clu [L$^+$81], Ada [US 80] and ML [MTH90]. Specifically, we consider a class construct which resembles the form of class found in C++, Eiffel and Java.

This class construct may be written in our object calculus extended with a form of abstract data type declaration. One appeal of this interpretation is that it clearly shows how classes may be viewed as an orthogonal combination of pure operations on objects (providing aggregation but no encapsulation) and data abstraction (providing encapsulation but no aggregation). Furthermore, this interpretation of classes sheds some light on a long-standing conflict in the literature on object-oriented programming: the connection between subtyping and inheritance.

An early and influential paper, [Sny86], argues that subtyping and inheritance are distinct. This point is reinforced in [Coo92], which shows that the subtyping and inheritance hierarchies used in the Smalltalk collection classes are essentially unrelated. We believe that the arguments in [Sny86, Coo92] are correct for *interface types*, which are types that specify the operations of their objects but not their implementations. Such types have been the focus of recent theoretical studies of object systems, such as [AC96c, Bru93, FHM94, PT94] and the earlier papers appearing in [GM94]. However, existing object-oriented languages such as Eiffel and C++ use a form of *implementation type* that constrains both the interface and the implementations of objects. We argue that there *is* a connection between subtyping and inheritance for implementation types: the only way to produce a subtype of an implementation type, without violating basic principles of data abstraction, is via inheritance. In addition, we show a connection between interface types and implementation types: every implementation type is a subtype of the interface type obtained by "forgetting" its implementation constraints.

This chapter describes a sound type system, based on accepted type-theoretic principles, that allows us to write both interface types and implementation types in the same framework. The system is relatively simple in outline, since it may be viewed as a straightforward combination of basic constructs that have been studied previously. The main technical idea is that we may model implementation types by adding an abstract data type mechanism to the object calculus developed in the previous chapter. While there is a folkloric belief that C++ and Eiffel classes provide a form of data abstraction, we believe that this study is the first technical account giving a precise correspondence between class constructs and a standard non-object-oriented form of data abstraction.

The rest of this chapter is organized as follows. Section 5.1 explores in more detail the role of classes in object-oriented languages and identifies what properties we believe a class construct should possess. Section 5.2 considers the connections between classes and the long-standing controversy surrounding the relationship between subtyping and inheritance. Then Section 5.3 shows how we may translate a class construct into a combination of our object calculus and a data abstraction mechanism. We show how this encoding may be generalized in Section 5.4 to capture C++'s *protected* level of visibility. Section 5.5 briefly describes the changes to our formal language necessary to model classes. (We formally present the full system in Chapter 6; the type soundness proof appears in Chapter 7). In Section 5.6 we describe other approaches people have taken to model classes. Finally, we conclude in Section 5.7 with some observations.

5.1 Properties of Classes

Although space considerations preclude a full discussion of the motivations for classes in object-oriented languages, it is worth briefly considering their role in existing languages. Typically, classes serve several functions. One of the most important of these is their role in defining both a hierarchy of object implementations and a hierarchy of object types via inheritance. Commonly, these two hierarchies are closely related to each other, but need not exactly coincide. In C++, for example, *private inheritance* defines a relationship in the implementation hierarchy, but not in the subtyping hierarchy, while *abstract base classes* may be used to declare a subtyping relationship with only minimal connections in implementation. As both hierarchies support code reuse, classes provide two distinct mechanisms for reusing code: the implementation hierarchy supports object-definition reuse while the subtyping hierarchy allows subtyping polymorphism. Such code reuse mechanisms are one of the reasons object-oriented languages support large-scale programming.

Other roles of classes arise from the fact that classes typically constrain the implementations of their objects. One ramification of these constraints is that *class-based* protection mechanisms are possible. Such protection mechanisms allow the objects of a class to access the hidden components of other objects of the same class. The implementation constraints specified by the class make it possible to statically check such accesses. In particular, placing access restrictions at the class level, instead of at the object level, make it easier to write useful binary methods, which are methods that take an object of the same type as the receiver object as an argument. Typical examples of binary methods are equality methods, set union operations, and matrix multiplication. Additionally, because classes provide a useful and intuitive boundary for access restrictions, they help divide code into separately maintainable entities. Such separations make it easier to write and maintain large bodies of code. Another consequence of the implementation constraints imposed by classes is that compilers can use this information to optimize method lookup (as in C++).

This brief discussion reveals that in class-based object-oriented languages, classes play a fundamental role in structuring code, providing both reuse (subtyping and inheritance) and code separation (protection mechanisms). Both aspects support large-scale programming. Hence any class construct should be evaluated with large-scale programming in mind and should support both inheritance and encapsulation. Based on these, and other considerations, we believe that a class mechanism should have the characteristics outlined in the following paragraphs.

Coherent, extensible collection. A class construct should provide an extensible collection of object "parts." Such parts may include methods, data, local constants, specifications of communication protocols, *etc.* These parts should be *coherent*, in the sense that if a class is well-formed, then it should be guaranteed that all objects instantiated from the class will themselves be well-formed. For example, if one of the methods of the class requires an integer `f` method, then the class must contain an integer `f` method. The consistency of the parts should be checked when the class is formed, not when objects are actually instantiated from the class, a timeliness which facilitates debugging. The *extensibility* of the collection means that derived classes may reuse object parts from other classes, possibly adding additional coherent parts.

Access Restrictions. A class construct should allow programmers to specify the level of visibility for each of the "parts" defined in the class. Some common visibilities are *private* (for use within the class implementation) and *public* (for use by client programs). Such visibilities help insure that separate blocks of code have no hidden dependencies and hence may be separately maintained. These specifications should be enforced by the language. For example, if a programmer specifies that a given method is private, the type system should insure that only the implementation of the given class may access the method in any way. This restriction guarantees that the implementor of a class may change its private implementation without fear of introducing type errors into clients of the class. Access controls should clearly indicate the visibility of each of the parts for both the current class and all derived classes, so if a method is private in a given class, that method is guaranteed to be private in all derived classes as well.

Initialization. Classes should provide code to initialize their collection of parts when objects are instantiated from them. A class-based language should insure that this initialization code is executed both for objects instantiated directly from a given class and for those instantiated from its descendents. Running this initialization code is essential for establishing invariants for the various parts, particularly in light of the private level of visibility. Since derived classes cannot see private components and are not supposed to be aware of their parents' implementation details, they should not (and indeed cannot) be responsible for initializing and setting up invariants for inherited components.

Support for Incremental Development. Because classes are intended to support large scale programs and because such programs inevitably include program modifications, classes should support code maintenance. In particular, incremental changes to a base class should automatically be propagated to all derived classes. If any type errors are introduced in derived classes because of such changes, these errors should be reported immediately. For example, if a programmer adds a new method to a parent class, that method should then be automatically added to all derived classes. If the new method conflicts with a method defined in a descendent, the conflict should be reported immediately, not when objects are instantiated from the given derived class.

The interpretation of classes described in this chapter possesses all of the above desired properties.

5.2 The Connection between Subtyping and Inheritance

5.2.1 Subtyping and Inheritance

As we saw in Sections 2.1.2 and 2.1.3, object-oriented programming languages have two distinct, powerful mechanisms that support code reuse, namely inheritance and subtyping. We briefly review these concepts here. Inheritance is a language feature that allows new objects to be defined as incremental modifications to the definitions of existing ones. It is an implementation technique. For every object or class of objects defined via inheritance, there is in principle an equivalent definition that does not use inheritance, obtained by expanding the definition so that inherited code is duplicated. The importance of inheritance is that it saves the effort of duplicating (or reading duplicated) code, and that when one class is implemented by inheriting from another, changes to one affect the other. This has a significant and sometimes debated impact on program maintenance and modification.

Subtyping is a relation on types. The basic principle associated with subtyping is *substitutivity:* if A is a subtype of B, then any expression of type A may be used without type error in any context that requires an expression of type B. We write "$A <: B$" to indicate that A is a subtype of B. The primary advantage of subtyping is that it permits uniform operations over various types of data that share some basic structure. For example, subtyping makes it possible to have heterogeneous data structures containing objects that belong to different subtypes of some common base type.

As their definitions suggest, and as pointed out in [Sny86] and [Coo92], subtyping and inheritance are distinct notions. However, in the history of object-oriented programming, they have often been

```
Class Point{
             private          x  :  int
             public        move  :  int → Point
                       newPoint  :  int → Point
          }

Class ColorPoint1 :  public Point {
             private          c  :  color
             public     turnRed  :  ColorPoint1
                 newColorPoint1  :  int → color → ColorPoint1
          }

Class ColorPoint2 {
             private          x  :  int
                              c  :  color
             public c      move  :  int → ColorPoint2
                        turnRed  :  ColorPoint2
                 newColorPoint2  :  int → color → ColorPoint2
          }
```

Figure 5.1: C++-style declarations for `Point` and `ColorPoint` classes.

confused. One source of this confusion is that existing languages like C++ [Str86, ES90] and Eiffel [Mey92] conflate the two concepts by making inheritance the only way to produce subtypes. For example, consider the class declarations in Figure 5.1, written in a C++-like syntax. In C++ or Eiffel, $ColorPoint1$ is a subtype of $Point$, but $ColorPoint2$ is not, despite the fact that $ColorPoint2$ objects can respond to exactly the same messages as $ColorPoint1$ objects. To distinguish between classes and types, throughout this section we will typeset class names in teletype font (`ClassName`) and type names in italics (*TypeName*).

5.2.2 Interface Types

An *interface type* specifies a list of operations, generally as method names and return types. If an object is declared to implement all of the operations, it is considered an element (or member) of an interface type. As a result, objects with the same interface type may have significantly different internal representations. Some aspects of interface types may be illustrated using the pseudo-code in Figure 5.1. Each of the three classes described in Figure 5.1 has an associated interface type, giving the operations that may be invoked by client programs. We may write the *Point* interface type as follows, using a type notation defined precisely in Section 4.1.

$$Point_{Inter} \overset{def}{=} \mathbf{obj}\, u . \langle\!\langle \mathtt{move} : int \rightarrow u \rangle\!\rangle$$

Expressions with this type are guaranteed to be objects that have at least a `move` method that takes an integer as an argument and returns an object with the same type as that of the receiver.

We omit the `x` method from the interface type, because `x` is a private method and therefore not visible to clients of the class. Also, function `newPoint` does not appear in this type because it is a special *constructor* function, distinguished by the syntactic form `newClassname`. Since all objects of a class are created by calling a class constructor, it is important to be able to call the constructor function before any objects have been created. Therefore, unlike the other functions listed in the class declarations, the class constructor is not a method; it does not belong to objects of the class. Constructors are included in class declarations primarily as a syntactic convenience. Because the `Point` class is recursive, (the return type of `Point`'s `move` method is *Point*), we use a form of recursive type for the `Point` class interface type. To that end, we have replaced the *Point* return type of the `move` method in the class declaration with a type variable u.

We may similarly write the interface type for the `ColorPoint1` class as follows:

$$ColorPoint1_{Inter} \stackrel{def}{=} \mathbf{obj}\, u.\langle\!\langle \texttt{move}\colon int \to u,\ \texttt{turnRed}\colon u \rangle\!\rangle$$

This interface type includes the `move` method because the `ColorPoint1` class inherits it from the `Point` class. As above, the `c` method is omitted because it is private. Finally, the interface type for the `ColorPoint2` class is:

$$ColorPoint2_{Inter} \stackrel{def}{=} \mathbf{obj}\, u.\langle\!\langle \texttt{move}\colon int \to u,\ \texttt{turnRed}\colon u \rangle\!\rangle$$

The fact that $ColorPoint1_{Inter}$ and $ColorPoint2_{Inter}$ are identical indicates that the objects instantiated from these two classes will respond to exactly the same messages.

From the point of view of these interfaces, it is straightforward to see that:

$$ColorPoint1_{Inter} \;<:\; Point_{Inter} \quad \text{and} \quad ColorPoint2_{Inter} \;<:\; Point_{Inter}$$

These subtyping relationships reflect the fact, pointed out by [Sny86] and [Coo92], that interface subtyping has nothing to do with how objects are implemented. Specifically, the `ColorPoint2` class produces an interface subtype of $Point_{Inter}$ without inheriting from the `Point` class.

5.2.3 Implementation Types

A basic principle in statically typed, class-based object systems is that certain aspects of the implementation of an object may be hidden from client code. Typically, this may include private data, or private methods that may only be invoked by other methods of the object. Like traditional data abstraction in languages such as Ada, Clu, and ML, encapsulation in object-oriented languages makes it possible to enforce representation invariants, such maintaining a sorted array in the implementation of a priority queue. In addition, encapsulation allows the implementation of an object (or class of objects) to be changed in various ways, without adversely affecting the correctness of client code. For example, it is possible to construct a geometry library so that the representation of point objects may be changed from rectangular to polar coordinates, say, without invalidating any essential properties of the subtype or inheritance hierarchies.

An *implementation type* is a form of object type that guarantees both a specific public interface and some aspects of the implementation of objects which are typically hidden from client code. For example, the type $ColorPoint2_{Imp}$ that would be associated with a C++ class declaration of the form shown in Figure 5.1 guarantees the public interface written as $ColorPoint2_{Inter}$ and the implementation property that colored points of this type have private integer and color data, with `x` occurring before `c` in the representation of an object. The private information is not accessible in client code, but it is an important part of the implementation of C++ that is used in the calling sequence for public methods. In particular, a C++ program would not work properly if objects with the same public interface, but different internal representation, were cast to type $ColorPoint2_{Imp}$.

A fundamental tenet of this thesis is that C++ and Eiffel are based on a sensible and meaningful notion of implementation type which differs from the form of interface type considered in other studies. These implementation types specify not only the interfaces of objects but also provide control over how objects of a certain type may be created or represented. This extra implementation information is useful for several reasons. If we know that all point objects inherit a specific representation of `x` and `y` coordinates, for example, then a program may be optimized to take advantage of this static guarantee. The usual implementations of C++, for example, use type information to statically calculate the offset of member data relative to the starting address of the object. A similar calculation is used to find the offset of virtual member functions in the virtual function table (vtable) at compile time [ES90, Section 10.7c]. Such optimizations are not possible in an untyped language such as Smalltalk [GR83] and would not be possible in a typed language where objects of a single type could have arbitrarily dissimilar implementations.

A second, more methodological reason that programmers may be interested in implementation types is that there are greater guarantees of behavioral similarity across subtype hierarchies. More specifically, traditional type systems generally give useful information about the signature (or domain and range) of operations. This information is a very weak form of specification and, in many programming situations, it is desirable to have more detailed guarantees. While behavioral specifications are difficult to manipulate effectively, we have a crude but useful approximation when types fix part of the implementation of an object. To return to points, for example, if we know that all subtypes of point share a common implementation of a basic function like `move`, then the type system, in effect, guarantees a behavioral property of points. (This may be achieved in our framework if `move` is private or if we add the straightforward capability of restricting redefinition of protected or public methods.)

A more subtle reason to use types that restrict the implementations of objects has to do with the implementation of binary operations. In an object-oriented context, a binary operation on type A is realized as a member function that requires another A object as a parameter. In a language where all objects of type A share some common representation, it is possible for an A member function to safely access part of the private internal representation of another A object. A simple example of this access arises with *set* objects that have only a membership test and a union operation in their public interfaces. With interface types, some objects of type *set* might be represented internally using bit vectors, while others might use linked lists. In this case, there is no type-safe way to implement union, since no single operation will access both a bit vector and a linked list correctly. With only interface types, it is necessary to extend the public interface of both kinds of sets to make this operation possible. In contrast, if the type of an object conveys implementation information, then a less flexible but type-safe implementation of set union is possible. In this case, all *set* objects would have one representation and a union operation could be implemented by taking advantage of this uniformity.

5.2.4 Interface Types vs. Implementation Types

From a programming point of view, interface types are often more flexible than implementation types. Using interface types, for example, we can define a single type of matrix objects and then represent dense matrices with one form of object and sparse matrices with another. If the type only gives the interface of an object, then both matrix representations could have the same type and therefore be used interchangeably in any program. This kind of flexibility is particularly useful when we write library operations on matrices without assuming any particular implementation. Such library functions may be written using a standard interface type, without concern for how matrices might be implemented in later (or earlier) development of a software system.

From this discussion, it is clear that interface types and implementation types have opposite strengths. To summarize:

	Implementation types	Interface types
Flexibility	-	+
Efficiency	+	-
Binary Methods	+	-

5.2.5 Inheritance Necessary for Subtyping Implementation Types

As mentioned above, the main principle associated with subtyping is substitutivity. Many have observed that this principle gives solid guidance on the definition of subtyping for interface types: type A is a subtype of interface type B if every public operation guaranteed by B is also a guarantee of A. We can also apply this principle to implementation types, suggesting the following condition: type A is a subtype of implementation type B if every public operation and every private implementation property guaranteed by B is also guaranteed by A. However, this condition is not the whole story since it does not take principles of data abstraction into account.

The relationship between data abstraction and subtyping may be illustrated by again returning to the example classes in Figure 5.1. As noted above, the public interfaces of the `ColorPoint1` and `ColorPoint2` classes are identical. In addition, under the usual implementations of C++, objects of the two classes could be implemented identically. Therefore, one might consider making the corresponding implementation types (which we will write $ColorPoint1_{Imp}$ and $ColorPoint2_{Imp}$) each a subtype of $Point$, the implementation type for the `Point` class. However, such a relationship would violate a basic principle of data abstraction. Specifically, if we modify the program by changing the implementation of class `Point`, say by adding private `y` data, then $ColorPoint1_{Imp}$ will remain an implementation subtype of $Point_{Imp}$, since `ColorPoint1` will inherit the added private data. However, $ColorPoint2_{Imp}$ will cease to be an implementation subtype. With this in mind, we propose the following provisional condition for subtyping with implementation types:

> Type A is a subtype of implementation type B if
> (i) every public operation guaranteed by B is also guaranteed by A
> (ii) every private implementation property guaranteed by B is also guaranteed by A
> (iii) these relationships are preserved by modifications to the private implementation of B, excluding those modifications that might invalidate the implementation of A.

The exclusion in clause (iii) is awkward and we would prefer that it not be necessary. However, the condition is needed for existing languages, as well as our foundational system, to account for the special case where a extending a parent (or base) class conflicts with the implementation of the child (or derived) class. This would happen to `ColorPoint1` if we extended `Point` in Figure 5.1 with a `c` field of type integer.

In the following sections, we present a sound type system that allows us to write both interface types and implementation types. To our knowledge, this framework is the first theoretical study to incorporate implementation types. An insight provided by this analysis is that there is a connection between implementation and interface types. In particular, it turns out that a given implementation type is a subtype of the interface type obtained by "forgetting" the implementation constraints. For example, the various types arising from the class declarations in Figure 5.1 are related via the subtyping relations shown in Figure 5.2.

5.3 Overview of Formal System by Example

We introduce our formal system by showing how the `Point`, `ColorPoint1`, and `ColorPoint2` classes presented in Figure 5.1 may be formalized in our language.

5.3.1 Interface Type Expressions

Notice that each method in the classes in Figure 5.1 is declared to be either `public` or `private`. *Private* methods are only accessible within the implementation of the class, whereas *public* methods may be accessible, through objects of the class, in any module of the program. In translating the pseudo-code in Figure 5.1 above into a more precise form, we write a type expression for both the

$$\begin{tikzcd}
 & Point_{inter} & \\
Point_{imp} \arrow[ur] & & \\
 & & ColorPoint1_{inter}, ColorPoint2_{inter} \arrow[uul] \\
ColorPoint1_{imp} \arrow[uu] \arrow[urr] & & ColorPoint2_{imp} \arrow[u]
\end{tikzcd}$$

Figure 5.2: Subtyping relations between types arising from class declarations in Figure 5.3.

public and private interface of each class, resulting in six distinct but related types. In hopes that this practice will provide useful mnemonics, we follow a systematic naming convention where, for class A, we call the type expressions for the public and private interfaces A_{pub} and A_{priv}, respectively.

Although each interface is essentially a list of method names and their types, it is necessary to use a type function instead of a type for each interface. The reason is that the type associated with the objects instantiated from a given class is recursively defined; this type is a fixed-point of a type function. Pointwise subtyping between such type functions is the critical relation between interfaces for type-checking inheritance. We may obtain these type functions from the corresponding class declarations by listing the methods and their types available at each level of visibility. Occurrences of the class name are replaced by a type variable, which is then lambda abstracted to form a type function.

For `Point`, this methodology gives us the following type functions, using the notation from the previous two chapters:

$$\begin{array}{lcl} P_{pub} & \stackrel{def}{=} & \lambda\, u\,.\langle\!\langle \texttt{move} : int \to u \rangle\!\rangle \\ P_{priv} & \stackrel{def}{=} & \lambda\, u\,.\langle\!\langle \texttt{x} : int,\, \texttt{move} : int \to u \rangle\!\rangle \end{array}$$

These interface functions are formed from the class declaration for `Point` by replacing occurrences of `Point` by a type variable u and lambda-abstracting to obtain a type function. We use row variables to range over such type functions.

Because the `ColorPoint1` class is defined by inheritance from the `Point` class, the analogous interfaces for `ColorPoint1` are written using a free row variable p. This row variable will be bound to the private interface for points P_{priv} in the scope where the `ColorPoint1` interfaces appear.

$$\begin{array}{lcl} CP1_{pub} & \overset{def}{=} & \lambda\, u \,.\langle\!\langle p\, u \mid \mathtt{turnRed} \colon u \rangle\!\rangle \\ CP1_{priv} & \overset{def}{=} & \lambda\, u \,.\langle\!\langle p\, u \mid \mathtt{c} \colon \mathit{color},\, \mathtt{turnRed} \colon u \rangle\!\rangle \end{array}$$

A subtype bound on identifier p will provide access to the public `Point` methods, so there is no need to list the methods inherited from `Point`. Consequently the `ColorPoint1` interfaces list exactly the same methods as our pseudo-code `ColorPoint1` class. Since the variable p will be "existentially bound" in an abstract data type declaration, the occurrence of p in each expression will guarantee that the private methods of `Point` objects are present in `ColorPoint1` objects, without exposing any other information about private `Point` methods. Kind information in the declaration of p guarantees that methods named `c` and `turnRed` are not present, making it type-safe to extend p objects with these new methods.

Because the `ColorPoint2` class is not defined to inherit from the `Point` class, its interfaces are not written in terms of p:

$$\begin{array}{lcl} CP2_{pub} & \overset{def}{=} & \lambda\, u \,.\langle\!\langle \mathtt{move} \colon \mathit{int} \to u,\, \mathtt{turnRed} \colon u \rangle\!\rangle \\ CP2_{priv} & \overset{def}{=} & \lambda\, u \,.\langle\!\langle \mathtt{x} \colon \mathit{int},\, \mathtt{c} \colon \mathit{color},\, \mathtt{move} \colon \mathit{int} \to u,\, \mathtt{turnRed} \colon u \rangle\!\rangle \end{array}$$

5.3.2 Implementations

A class implementation specifies an object layout and set of method bodies (code for the methods of the objects). In our approach, the object layout will be given by a type expression and the method bodies will be part of the constructor function. Following general principles of data abstraction, the method bodies may rely on aspects of the representation that are hidden from other parts of the program.

We use a subtype-bounded form of data abstraction based on existential types [MP88, CW85]. Using this formalism, the implementation of points will be given by a pair with subtype-bounded existential type of the form:

$$\{\!\mid p <:_w P_{pub} :: \kappa = P_{priv},\ \mathtt{ConsImp_P} \mid\!\}$$

consisting of the private interface P_{priv} for points and a constructor function $\mathtt{ConsImp_P}$ that might use an initial value for the `x`-coordinate to return a new point object. (Our framework allows any number of constructor functions, or other "non-virtual" operations to be provided here. However, for simplicity, we discuss only the special case of one constructor per class.) The kind κ will indicate that we are declaring an abstract type function, list methods that are guaranteed not to be present in the implementation of points, and describe the variance of point objects. As in the previous chapter, the variance information is needed to infer subtyping relationships for object types containing row variable p.

The implementation of the `ColorPoint1` class has the form:

$$\{\!|cp1 <:_w CP1_{pub} :: \kappa' = CP1_{priv},\ \mathtt{ConsImp}_{\mathtt{cp1}}|\!\}$$

Similarly, the implementation of `ColorPoint2` has the form:

$$\{\!|cp2 <:_w CP2_{pub} :: \kappa' = CP2_{priv},\ \mathtt{ConsImp}_{\mathtt{cp2}}|\!\}$$

Next, we give definitions of appropriate constructor functions.

5.3.3 Constructor Implementations

The `Point` class constructor $\mathtt{Imp}_{\mathtt{p}}$ may be written as follows:

$$\begin{array}{lrcl} \lambda\mathtt{initX}.\langle & \mathtt{x} & = & \lambda\mathtt{self}.\,\mathtt{initX}, \\ & \mathtt{move} & = & \lambda\mathtt{self}.\,\lambda\mathtt{dx}.\langle\mathtt{self} \leftarrow \mathtt{x} = \lambda\mathtt{s}.\,\mathtt{dx} + \mathtt{self} \Leftarrow \mathtt{x}\rangle\rangle \end{array}$$

The `ColorPoint1` constructor $\mathtt{Imp}_{\mathtt{cp2}}$ first invokes the `Point` class constructor `newPoint`, (which is implemented by the above $\mathtt{Imp}_{\mathtt{p}}$ function) then extends the resulting prototype with the new methods `c` and `turnRed`:

$$\begin{array}{llcl} \lambda\mathtt{iX}.\,\lambda\mathtt{iC}.\ \langle\langle\langle\langle\mathtt{newPoint(iX)} & \leftarrow\!\!+\ \mathtt{c} & = & \lambda\mathtt{self}.\,\mathtt{iC}\rangle \\ & \leftarrow\!\!+\ \mathtt{turnRed} & = & \lambda\mathtt{self}.\langle\mathtt{self} \leftarrow \mathtt{c} = \lambda\mathtt{self}'.\,\mathtt{red}\rangle\rangle\rangle \end{array}$$

By first calling the constructor for the `Point` class, the `ColorPoint1` class permits the parent `Point` class to build the parts of the `ColorPoint1` objects inherited from `Point`, including the private components that the `ColorPoint1` class cannot directly access.

The `ColorPoint2` constructor $\mathtt{Imp}_{\mathtt{cp2}}$ does not inherit from the `Point` class, and so is defined independently of `newPoint`. The constructor for this class may be written as follows:

$$\begin{array}{lrcl} \lambda\mathtt{initX}.\,\lambda\mathtt{initC}.\langle & \mathtt{x} & = & \lambda\mathtt{self}.\,\mathtt{initX}, \\ & \mathtt{c} & = & \lambda\mathtt{self}.\,\mathtt{initC}, \\ & \mathtt{move} & = & \lambda\mathtt{self}.\,\lambda\mathtt{dx}.\langle\mathtt{self} \leftarrow \mathtt{x} = \lambda\mathtt{s}.\,\mathtt{dx} + \mathtt{self} \Leftarrow \mathtt{x}\rangle \\ & \mathtt{turnRed} & = & \lambda\mathtt{self}.\langle\mathtt{self} \leftarrow \mathtt{c} = \mathtt{red}\rangle\rangle \end{array}$$

5.3.4 Class Declarations

In this formalism, a "class declaration" intuitively corresponds to an abstract data type declaration of the form:

$$\mathtt{Abstype}\ r <:_w R :: \kappa\ \mathtt{with}\ \mathtt{newClassName} : \tau\ \mathtt{is}\ \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2$$

Here r is intuitively the "name" of the class. Function `newClassName` of type τ is the class's constructor. Expression $\mathtt{e}_1$ is the implementation of the class, and $\mathtt{e}_2$ contains the program that uses this class declaration.

```
Abstype  $p <:_w P_{pub} :: T^+ \to (\{\texttt{c}, \texttt{turnRed}\}; \emptyset)$
   with  $\texttt{newPoint} : int \to \textbf{pro}\, u \,\texttt{.}\, p\ u$
     is  $\{\!|p <:_w P_{pub} :: (T^+ \to (\{\texttt{c}, \texttt{turnRed}\}; \emptyset)) = P_{priv},\ \texttt{Imp}_{\texttt{p}}|\!\}$
     in
         Abstype  $cp1 <:_w CP1_{pub} :: T^+ \to (\emptyset; \emptyset)$
            with  $\texttt{newColorPoint1} : int \to color \to \textbf{pro}\, u \,\texttt{.}\, cp1\ u$
              is  $\{\!|cp1 <:_w CP1_{pub} :: (T^+ \to (\emptyset; \emptyset)) = CP1_{priv},\ \texttt{Imp}_{\texttt{cp1}}|\!\}$
              in
                  Abstype  $cp2 <:_w CP2_{pub} :: T^+ \to (\emptyset; \emptyset)$
                     with  $\texttt{newColorPoint2} : int \to \textbf{pro}\, u \,\texttt{.}\, cp2\ u$
                       is  $\{\!|cp2 <:_w CP2_{pub} :: (T^+ \to (\emptyset; \emptyset)) = CP2_{priv},\ \texttt{Imp}_{\texttt{cp2}}|\!\}$
                       in
                           ⟨Program⟩
```

Figure 5.3: Nested abstract data types for point and colored point classes.

5.3.5 Class Hierarchies as Nested Abstract Types

Our basic view of classes and implementation types is that the class-based pseudo-code in Figure 5.1 may be regarded as sugar for the sequence of nested abstract data type (`Abstype`) declarations given in Figure 5.3. In order, the three abstract type declarations correspond to the class `Point`, the class `ColorPoint1`, and the class `ColorPoint2`. In this example, each constructor returns an object with a **pro** type. Such types support the operations of method override and object extension, which are used to model inheritance. For example, the `ColorPoint1` constructor will use these operations to define `ColorPoint1` objects as extensions of `Point` objects. As explained in the previous chapter, such extensible object types do not directly support subtyping, however. When programmers wish to manipulate the objects defined in a class as traditional non-extensible objects that support subtyping, they must "instantiate" the objects returned from the constructors. This "instantiation" seals the objects so that they no longer support method override or object extension, but now have **obj** types that provide rich subtyping. As we saw in Chapter 4, it turns out that this "instantiation" operation is just subsumption; the object expressions themselves are unaffected.

In the example above, the **obj** types associated with objects returned from the three constructors `newPoint`, `newColorPoint1`, and `newColorPoint2` are $\textbf{obj}\, u\texttt{.}(p\ u)$, $\textbf{obj}\, u\texttt{.}(cp1\ u)$, and $\textbf{obj}\, u\texttt{.}(cp2\ u)$, respectively. Our subtyping rules allow us to verify that we get the subtyping hierarchy shown in Figure 5.4, which exactly matches the conceptual hierarchy shown in Figure 5.2. It is possible for our rules to verify these relationships because the ambient environment includes the assumption that $cp1 <:_w \lambda u.\langle\!\langle p\, u \mid \texttt{turnRed} \colon u\rangle\!\rangle$.

$$\begin{tikzcd}
 & \mathbf{obj}\, u . \langle\!\langle \texttt{move} : int \rightarrow u \rangle\!\rangle & \\
\mathbf{obj}\, u . (p\ u) \arrow[ur] & & \\
 & & \mathbf{obj}\, u . \langle\!\langle \texttt{move} : int \rightarrow u,\ \texttt{turnRed} : u \rangle\!\rangle \arrow[uul] \\
\mathbf{obj}\, u . (cp1\ u) \arrow[uu] \arrow[urr] & & \mathbf{obj}\, u . (cp2\ u) \arrow[u]
\end{tikzcd}$$

Figure 5.4: Subtyping relations between formal types arising from class declarations in Figure 5.3.

5.4 The Protected Visibility Level

In this section, we show how to model a third level of visibility, the *protected* level.

5.4.1 Protection Levels

Up to this point, we have only considered two levels of protection: public and private. However, there is an intermediate protection level: the protected level. Each class in an object-oriented program has two kinds of external clients: sections of the program that use objects created from the class and classes that derive new classes from the original. Since the methods of a class may refer to each other, the class also has an internal "client," namely itself. It is therefore sensible to associate *three* interfaces with each class. Using C++ terminology, these may be distinguished as follows:

> *Private* methods are only accessible within the implementation of the class,
>
> *Protected* methods are only accessible in the implementation of the class and derived classes,
>
> *Public* methods may be accessible, through objects of the class, in any module of the program.

One goal of this work is to show how we can associate a different type with each interface and use essentially standard type-theoretic constructs to restrict visibility in each part of a program appropriately. In doing so, we pay particular attention to the fact that although the private or protected methods may not be accessible in certain contexts, it is important for the type system to guarantee their existence.

5.4.2 Point and Color Point Classes

We introduce a simpler class hierarchy in this section to simplify the exposition. Again, using a C++-like syntax, we might declare classes of one-dimensional points and color points as shown below.

```
Class Point{
              private          x  :  int
              protected     setX  :  int → Point
              public        getX  :  int
                        newPoint  :  int → Point
            }

Class ColorPoint :  public Point {
              private          c  :  color
              protected     setC  :  color → ColorPoint
              public        getC  :  color
                   newColorPoint  :  int → color → ColorPoint
            }
```

Intuitively, the `set_` methods are used to assign to private `x`-coordinate or color data `c`, and `get_` methods are used to read the values of the data. These classes reflect a common idiom of C++ programming, where the basic data fields are kept private so that the class implementor may change the internal representation without invalidating client code. Protected methods make it possible for derived classes to change the values of private data without providing the same capability outside of derived classes.

5.4.3 Interface Type Expressions

In translating the pseudo-code above into a more precise form, we write a type expression for each interface of each class, as we did in the previous section. Because we have a third protection level, we need a third interface: the protected interface. Following our naming convention, we call the type expression for the protected interface of class `A` A_{prot}.

For the `Point` class, the public, protected, and private interfaces are:

$$\begin{aligned} P_{pub} &\overset{def}{=} \lambda t.\langle\!\langle \texttt{getX} : int \rangle\!\rangle \\ P_{prot} &\overset{def}{=} \lambda t.\langle\!\langle \texttt{setX} : int \rightarrow t,\ \texttt{getX} : int \rangle\!\rangle \\ P_{priv} &\overset{def}{=} \lambda t.\langle\!\langle \texttt{x} : int,\ \texttt{setX} : int \rightarrow t,\ \texttt{getX} : int \rangle\!\rangle \end{aligned}$$

As in the previous section, the analogous interfaces for the `ColorPoint` class are written using a free row variable p, which will be bound to the abstract type-function for points in the scope where

```
Abstype  $p <:_w P_{prot} :: T^+ \to (\{\texttt{c}, \texttt{getC}, \texttt{setC}\}; \emptyset)$
   with  $\texttt{newPoint} : \mathit{int} \to \mathbf{pro}\ u\,.\,p\ u$
     is  $\{\!|p <:_w P_{prot} :: (T^+ \to (\{\texttt{c}, \texttt{getC}, \texttt{setC}\}; \emptyset)) = P_{priv},\ \texttt{Imp}_{\texttt{P}}|\!\}$
     in
         Abstype  $cp <:_w CP_{prot} :: T^+ \to (\emptyset; \emptyset)$
            with  $\texttt{newColorPoint} : \mathit{int} \to \mathit{color} \to \mathbf{pro}\ u\,.\,cp\ u$
              is  $\{\!|cp <:_w CP_{prot} :: (T^+ \to (\emptyset; \emptyset)) = CP_{priv},\ \texttt{Imp}_{\texttt{CP}}|\!\}$
              in
                  Abstype  $p <:_w P_{pub} :: T^+ \to (\emptyset; \emptyset)$
                     with  $\texttt{newPoint} : \mathit{int} \to \mathbf{obj}\ t\,.\,p\ t$
                       is  $\{\!|p <:_w P_{pub} :: (T^+ \to (\emptyset; \emptyset)) = p,\ \texttt{newPoint}|\!\}$
                       in
                           Abstype  $cp <:_w CP_{pub} :: T^+ \to (\emptyset; \emptyset)$
                              with  $\texttt{newColorPoint} : \mathit{int} \to \mathit{color} \to \mathbf{obj}\ t\,.\,cp\ t$
                                is  $\{\!|cp <:_w P_{pub} :: (T^+ \to (\emptyset; \emptyset)) = cp,\ \texttt{newColorPoint}|\!\}$
                                in
                                    $\langle$Program$\rangle$
```

Figure 5.5: Nested abstract data types for point and colored point classes, illustrating the protected visibility level.

the `ColorPoint` interfaces appear:

$$
\begin{aligned}
CP_{pub} &\overset{def}{=} \lambda t.\langle\!\langle p\,t \mid \texttt{getC} : \mathit{color}\rangle\!\rangle \\
CP_{prot} &\overset{def}{=} \lambda t.\langle\!\langle p\,t \mid \texttt{setC} : \mathit{color} \to t,\ \texttt{getC} : \mathit{color}\rangle\!\rangle \\
CP_{prot} &\overset{def}{=} \lambda t.\langle\!\langle p\,t \mid \texttt{c} : \mathit{color},\ \texttt{setC} : \mathit{color} \to t,\ \texttt{getC} : \mathit{color}\rangle\!\rangle
\end{aligned}
$$

Because the subtyping bounds on identifier p give all of the relevant (protected or public) `Point` methods, there is no need to list the methods inherited from `Point`. Consequently the `ColorPoint` interfaces list exactly the same methods as our pseudo-code `ColorPoint` class. Figure 5.4.3 shows the interpretation of the above pseudo-code.

In order, the four abstract type declarations give the protected view of `Point`, the protected view of `ColorPoint`, the public view of `Point`, and the public view of `ColorPoint`. The nesting structure allows the implementation of `ColorPoint` to refer to the protected view of `Point` and allows the program to refer to public views of both classes. The two inner declarations hide the protected view of a class by redeclaring the same type name and constructor and exposing the public view (with a different type, as discussed below). Since the implementation of the public view, in each case, is exactly the same as the implementation of the protected view, hiding the protected methods is the only function of the two inner declarations. We admit reusing bound variables is a bit of a "hack."

One distinction between the protected and public views is that the constructors for the protected view return an object with a **pro** type while the public view constructors return objects with **obj** types. This difference reflects the fact that derived clients extend their parents and hence need the operations of method override and object extension, but do not require subtyping. In contrast, object clients need rich subtyping but not method override or object extension.

5.4.4 Constructor Implementations

The $\texttt{Imp}_{\texttt{P}}$ and $\texttt{Imp}_{\texttt{CP}}$ expressions, which implement the constructors for the `Point` and `ColorPoint` classes, respectively, may be defined in a fashion similar to the constructor implementations given in Section 5.3.3.

5.5 Formal Language and Type System

The changes to the formal system described in this section are relatively minor additions to provide an abstract data type mechanism. To a large extent, these changes are minimal because the previous two languages were designed to support the constructs added here.

5.5.1 The Language

We extend the calculus from the previous chapter with two encapsulation primitives:

$$\begin{array}{l} \mathit{Abstype}\ r <:_w R :: \kappa\ \mathit{with}\ x : \tau\ \mathit{is}\ e_1\ \mathit{in}\ e_2 \\ \{\!|r <:_w R :: \kappa = R',\ e|\!\} \end{array}$$

The first form provides limited access to implementation e_1 in client e_2. Type expression $r <:_w R :: \kappa$ and assumption $x : \tau$ provide the interface for this access. The type system will require expression e_1 to have the form $\{\!|r <:_w R :: \kappa = R',\ e|\!\}$, which is the implementation of the abstraction.

5.5.2 Operational Semantics

We extend the operational semantics of the previous chapters with a reduction rule (*Abstype*) for evaluating abstract data type uses:

$$\mathit{Abstype}\ r <:_w R :: \kappa\ \mathit{with}\ x{:}\tau\ \mathit{is}\ \{\!|r <:_w R :: \kappa = R',\ e_1|\!\}\ \mathit{in}\ e_2 \ \xrightarrow{eval}\ [R'/r, e_1/x]e_2$$

5.5.3 Static Type System

The type systems is just that of the previous chapter, extended with existential types:

Types

$$\begin{array}{lcl} \sigma & ::= & \tau \,|\, \exists(r <:_w R :: \kappa)\tau \\ \tau & ::= & t \,|\, \tau_1 \to \tau_2 \,|\, \mathbf{pro}\, t.R \,|\, \mathbf{obj}\, t.R \end{array}$$

To reduce the complexity of the type system, these types are divided into two categories. The unquantified, monotypes are indicated using metavariables $\tau, \tau', \tau_1, \ldots$. The quantified types, indicated using metavariables $\sigma, \sigma', \sigma_1, \ldots$, may contain existential quantifiers. The row and kind expressions are the same as in the previous chapter.

The contexts of the system are just as in the previous section. The judgment forms are:

$, \vdash *$	well-formed context
$, \vdash \sigma : V$	well-formed type with variance V
$, \vdash R :: \kappa$	row has kind
$, \vdash \tau_1 <: \tau_2$	type τ_1 subtype of τ_2
$, \vdash R_1 <:_B R_2$	row R_1 is a B-subtype of R_2
$, \vdash e : \sigma$	term e has type τ

As in the previous section, the B annotation on the row subtyping judgment tracks whether or not depth subtyping rules were used to infer the subtyping relation. An annotation of w indicates only width rules were used, while a w, d indicates depth rules may have been used.

5.5.4 Typing Rules

There are only three new typing rules required by the extension presented in this system: a well-formedness rule for existential types (*exists*), and introduction ($\exists <:$ *intro*) and elimination rules ($\exists <:$ *elim*). The rule for forming an expression with existential type is the standard rule for existential introduction, extended to address kinding and subtyping constraints.

$$(\exists <:\ intro) \qquad \frac{\begin{array}{c} , \vdash R_1 :: \kappa \\ , \vdash R_1 <:_w R \\ , \vdash e : [R_1/r]\tau \end{array}}{, \vdash \{\!|r <:_w R :: \kappa = R_1,\ e|\!\} : \exists(r <:_w R :: \kappa)\tau}$$

The rule for existential elimination is standard as well. The full type system appears in the next chapter.

5.5.5 Type Soundness

The soundness proof for this type system appears in Chapter 7.

```
         type p   =  obj ⟪getX : int, setX : int → Point⟫
   val PointImpl  :  record ... end
                  =  record getX = λ (self) 0 end, setX = λ (self, nwX) ... end end
    fun newPoint  :  int → p
                  =  λ xi: int.⟨getX = PointImpl.getX, setX = PointImpl.setX⟩ ⇐ setX xi
            in
                              type cp   =  obj ⟪getX : int, setX : int → Point,
                                           getC : color, setC : color → CPoint⟫
                    val ColorPointImpl  :  record ... end
                                        =  record getX = PointImpl.getX,  ...  end
                     fun newColorPoint  :  int → color → cp
                                        =  λ xinit: int. λ cinit: color.
                                           ⟨getX = ColorPointImpl.getX,  ...  ⟩ ...
                                  in
                                           ⟨Program⟩
```

Figure 5.6: Type and constructor declarations for points and color point classes.

5.6 Related Work: Modeling Classes

5.6.1 Classes as Records of Pre-Methods

In Cook's important early work on the foundations of object-oriented languages [Coo87, Coo89a], a class was represented by a function, called an *object generator.* The fixed point of a generator is a recursively defined record that represents an object, with recursion used to resolve references to *self*. The reason for object generators is that inheritance cannot work directly on objects that encapsulate (or hide) the dependence of one method on another. However, inheritance may be formulated as an ordinary operation on method bodies if we explicitly treat methods as functions of *self*.

In the context of object calculi, it seems natural to define inheritance using pre-methods, functions that are written with the intent of being object methods, but which are not yet installed in an object. Presumably based on this intuition, Abadi and Cardelli have proposed encoding classes in a pure object system using records of pre-methods [AC96c]. This is illustrated by example in the next section. The record of pre-methods can be grouped with a constructor; the object containing both of these is used as the encoding of a class. In a typed system, we would also associate this object with the type of objects of the class. A similar approach is used in [RR96].

5.6.2 Example Classes

Figure 5.6 shows six declarations. The first three give the object type associated with the point class **obj** ⟪`getX`: *int*, `setX`: *int* → *Point*⟫, a record of point pre-methods `PointImpl`, and a point constructor `newPoint`. The next three declarations give a corresponding representation of the color point class. Note that the implementation of color points, `ColorPointImpl`, is defined from the implementation of points, `PointImpl`. However, this dependency appears to be the extent to which inheritance can be realized. It does not seem to be possible to reuse the point constructor because objects are not extensible, and it is therefore not possible to use a point object to create a color point object (except trivially).

There appear to be two ways to incorporate private fields into this approach, each with its advantages and disadvantages. The declared type p, for point objects, lists the public operations on points. The constructor, on the other hand, must return an object that contains the public and private operations. (For example, we could have a separate `x` field whose value is returned by `getX`.) However, by subtyping, we are able to give the constructor a static type that hides the private operations, exposing only the public ones. There is a choice, however, regarding whether the private fields are including in the record `PointImpl.` If they are, then private methods may be inherited. But their use is not restricted – they are as public from the point of view of any derived class as any of the fields that are intended to be public. The alternative is to define private parts in the constructor only, as part of a closure, but not include them in the class (record `PointImpl` of pre-methods). This would not allow private fields to be inherited since inheritance cannot be based on the object constructor. On the other hand, it would preserve them as strictly private.

While there is a technical detail that makes it difficult to type this example in the system described in the previous two chapters, it could be done if explicit universal quantification were added so that we can write a record of polymorphic pre-methods. It appears simpler to use the system of [AC96c, AC95], since less polymorphism is required for these methods.

5.6.3 Analysis

The primary advantage of the record-of-premethods encoding of classes is that it does not require a complicated form of basic objects. All that is needed is a way of forming an object from a list of field definitions. However, looking back at the list of desiderata in Section 5.1, it appears that only the first of four important criteria are satisfied. While reasonable people may disagree about the potential for records of pre-methods in large-scale object-oriented system design, it seems clear to us that this approach involves a significant loss of language support, in favor of very simple object primitives. While we readily admit that simplicity is a virtue, it also seems worthwhile to carefully examine the features that are lost.

The general shortcoming in this approach is the lack of support for inheritance. For example, if a derived class `D` is defined from a base class `B` in C++ or related languages, then adding a method to `B` will result in an additional method of `D`, and similarly for every other class derived from `B` (and there may be many). This update is important, since one of the goals of object-oriented programming is to support incremental changes in programs. In the approach shown here, the constructor for the derived class is written by explicitly naming each of the fields “inherited” from the base class. This part of the program needs to be rewritten explicitly whenever the base class is changed. Another way of stating this point is to say that the translation from classes to object operations, in this approach, is not local – the definition of the constructor of a class depends on all of the definitions of classes from which fields are inherited. In summary, the points related to inheritance are:

- There is no way to inherit the constructor from the parent class, only its list of object parts. Hence the constructor of each derived class must be written explicitly. As a result, base classes do not have control over the way private parts are to be initialized.
- A base class appears to provide only public (and protected) fields to derived classes; it does not seem possible to inherit private methods except by treating methods that are publicly visible as private.
- Additions to a base class do not result in additions to derived classes, unless the derived classes are rewritten.

Another issue that we believe is equally significant is that lack of explicit control over the class hierarchy. This point is largely orthogonal to the ones listed above, according to our analysis below, since it is an encapsulation issue that may be attributed to the lack of data abstraction mechanism, not a choice of basic object primitives. (On the other hand, successful use of data abstraction appears to require extensible objects; perhaps this is a useful area for future research.) One issue here is that in many existing class-based languages, it is possible to restrict the subtypes of an object type to classes that inherit all or part of the class implementation. As we discussed in the beginning of this chapter, this restriction may be useful for optimizing operations on objects, allowing object access in binary methods and guaranteeing semantic consistency beyond type considerations (some discussion of these points appears in [KLM94]). A special case of this capabilities the ability to define *final* classes, as recognized in work on Rapide [KLM94] and incorporated (presumably independently) as a language feature in Java. This ability is lacking in the record-of-premethods approach since any object whose type is a structural subtype can be used as an object of that type.

Extensible objects, such as those described in the previous two chapters, provide an alternative to generators or pre-methods. An advantage of extensible objects is that the methods in question remain part of an object, and it is therefore possible to impose static constraints on the way one method may be combined with others. For example, if an object contains two mutually recursive methods, then we cannot replace one with another of a different type. (In the record-of-premethods approach, it would be possible to form a class implementation with inconsistent methods, but it would not be possible to write a type-correct constructor function.) Another advantage arises with private (or protected) methods, which remain in the object when it is extended but cannot be accessed except by original methods that were defined before the private method became hidden. Of course, a complication is that extensible objects are generally not as useful to client programs, since the usual forms of subtyping fail, as explained in detail above. However, as we saw previously, this problem may be circumvented by using two forms of objects, an extensible form for inheritance and a non-extensible form that supports subtyping. A weakness of our current formulation of this idea is that subclasses must be declared within the scope of their parent classes. We believe this shortcoming may be overcome by replacing our block-structured construct with modules and a dot notation in the style of [CL90, HL94]. A further weakness of our interpretation is the need for negative information. In effect, parent classes must currently "guess" what methods their descendants will want to add. Some form of α-binding to hide private method names and allow descendants to add new method names freely may be necessary to solve this problem.

5.6.4 Other Approaches

Existential Model. In [PT94], Pierce and Turner interpret classes as object-generating functions. (Recall that in their model, objects are interpreted as elements of existential type, *c.f.* Section 3.5). In this setting, inheritance is interpreted as an extension to the object-generating functions that model classes. This encoding is somewhat cumbersome, since it requires programmers to explicitly manipulate `get` and `put` functions, which intuitively convert between the hidden state of parent class objects and derived class objects. Also, because this model provides protection at the object-level (as opposed to the class-level), binary methods require extra machinery. One such solution appears in [PT93].

In [HP95], Hofmann and Pierce introduce a refined version of $F_{<:}$ that permits only positive subtyping. With this restriction, `get` and `put` functions are both guaranteed to exist and hence may be handled in a more automatic fashion in class encodings. A disadvantage of this approach is that no subtyping is possible between existential types because the `put` functions for existential types are not meaningful. Since objects have existential type, this model does not currently support subtyping between object types; some form of explicit coercions are necessary. Extensions that combine positive subtyping with normal subtyping, addressing this lack of object subtyping, seem possible.

Direct Models. Kim Bruce has developed a family of type-safe formal languages that model classes directly instead of via an interpretation as the combination of more basic primitives. In [Bru93], Bruce describes TOOPL, a functional object-oriented language. PolyTOIL, presented in [BSv95], incorporates imperative features and introduces the notion of *matching*, a relationship between object types that holds whenever the first is an extension of the second, regardless of the variance of the *mytype* type variable. In these languages, the type of an object reflects only its public interface; it cannot convey implementation information.

Scott Smith and the Hopkins Object Group have designed a type-safe class-based object-oriented language with a rich feature set called I-LOOP, [EST95]. Their type system is based on polymorphic recursively constrained types, for which they have a sound type inferencing algorithm. The main advantage of this approach is the extreme flexibility afforded by recursively constrained types. Currently, the main problem is that it returns large, difficult-to-read types. Some form of simplification may be required. Work in this area is in progress.

5.7 Conclusion

Pure object systems are simpler than class-based systems; however, most of the empirical evidence for the utility of object-oriented languages is based on class-based languages, and certain features of classes seem critical to this success. These features include:

- Explicit support for inheritance (derived classes).
- The ability to specify which methods/fields are private and which are public (in the current class and all derived classes).
- Control over initialization of objects (in the current class and all derived classes). This control is essential for establishing invariants, for example.
- Static checking that guarantees that the object "parts" defined in the class are consistent with each other (for example, all pre-methods must have consistent assumptions about the type of "self").
- Preservation of relationships between classes when private or public fields or methods are added.

We believe it is important to evaluate pure object systems in light of their ability to support class-based programming styles. The fundamental constructs that seem necessary are (i) an extensible form of object, or perhaps pre-object, with the characteristics listed above, (ii) a non-extensible form of object that can be created easily from an extensible form, and (iii) subtyping on the types of proper objects. Given structural subtyping on object types, we have shown in this chapter how to gain more precise control over the class hierarchy using standard data abstraction, enhanced with subtype constraints as described in [CW85].

A further insight from the analysis in this chapter is that the Abadi/Cardelli and Fisher/Mitchell calculi are incomparable. The Fisher/Mitchell calculi were originally formulated with method specialization in mind, which has led to specific rules for object formation that enforce a strong degree of polymorphism. This degree of polymorphism is necessary to make each method that is added to an object suitable for inclusion in any future extension (via inheritance or object extension). However, it also precludes formation of certain objects that would behave sensibly only if not extended. On the other hand, the Abadi/Cardelli chain of calculi do not provide any form of extensible object. This lack makes it difficult to support class-based programming. A key technical issue for further exploration is to devise a system with two forms of objects, extensible objects with extension (like our prototypes, for representing classes) and objects with overriding and width subtyping (like the Abadi/Cardelli objects), with a natural form of conversion between the first and the second. We do not see any fundamental impediment to completing a study of this additional part of the design space. Recent work by Luigi Liquori [Liq96] addresses this issue.

In this chapter, we described a type-theoretic model of various levels of encapsulation and visibility in object-oriented systems. More specifically, we showed that classes, of the form found in C++, Eiffel, and related languages, may be regarded as the combination of two orthogonal language features: a form of objects without encapsulation and a standard form of data abstraction mechanism (albeit higher-order and including subtype constraints).

Here we briefly note several implications of our approach.

5.7.1 Representation Independence for Classes

One advantage of our decomposition of classes into object operations and standard data abstraction mechanism is that a number of properties developed in the analysis of traditional data abstraction without objects may be applied to object-oriented languages. For example, the results in [Rey83, MM85, Mit86, Mit91] give various sufficient conditions on interchangeability of implementations. Put briefly, we may replace one implementation of a class with another, in any program, as long as the protect and public interfaces of the new implementation conform to the old ones and the observable behaviors correspond.

5.7.2 Subtyping and Inheritance

A basic issue in the literature on object-oriented programming is the relation between subtyping and inheritance. We believe that the arguments in [Sny86, Coo92], discussed earlier in this chapter, are essentially correct for interface types: subtyping between interface types has nothing to do with the way objects are implemented. However, the analysis in the present chapter shows that for implementation types, inheritance may be necessary (although not always sufficient) to produce a subtype. In short, if a type t is abstract, in the sense that all or part of its implementation is hidden, then the only safe way to define a subtype of t is by extending the hidden implementation.

Furthermore, we show that implementation types are subtypes of the corresponding interface types. In fact, this work suggests that implementation types and interface types are actually the endpoints of a spectrum of types that convey *partial* implementation types. For example, the type $\mathbf{obj}\, u.\langle\!\langle p\ u \mid \mathtt{turnRed}\colon int \to u\rangle\!\rangle$ specifies that the initial segment of each of its objects must have been defined in the `Point` class, but it contains only interface information about the remaining structure of its objects. Thus this type lies in the middle of a spectrum of types, anchored at one end with the pure implementation type $\mathbf{obj}\, u.\langle\!\langle cp1\ u\rangle\!\rangle$ and at the other with pure interface type $\mathbf{obj}\, u.\langle\!\langle \mathtt{move}\colon int \to u,\ \mathtt{turnRed}\colon int \to u\rangle\!\rangle$. We can give a single object many types along this spectrum, since we may pass from its implementation side to its interface side by "forgetting" implementation information via subtyping. This spectrum suggests the possibility of a language in which a type inference system infers for each context in a program the type that maximizes the implementation information available for optimization while providing the polymorphism necessary to cover all of the objects that could appear in the context.

Chapter 6

Formal System

In this chapter, we give an annotated version of the full type system for the language described in Chapter 5. Appendices C and D include the same formal system, without the annotations.

6.1 Formal System

Expressions

$$\begin{array}{ll} e ::= & x \mid c \mid \lambda x.\, e \mid e_1 e_2 \mid \\ & \langle\rangle \mid e \Leftarrow m \mid \langle e_1 \leftarrow m = e_2 \rangle \mid \langle e_1 \leftarrow\!\!+\; m = e_2 \rangle \mid \\ & \{\!|\, r <:_w R :: \kappa = R',\ e \,|\!\} \mid \\ & \mathit{Abstype}\ r <:_w R :: \kappa\ \mathit{with}\ x : \tau\ \mathit{is}\ e_1\ \mathit{in}\ e_2 \end{array}$$

Types

$$\begin{array}{ll} \sigma ::= & \tau \mid \exists (r <:_w R :: \kappa)\tau \\ \tau ::= & t \mid \tau_1 \rightarrow \tau_2 \mid \mathbf{pro}\, t\mathbf{.}R \mid \mathbf{obj}\, t\mathbf{.}R \end{array}$$

Rows

$$\begin{array}{ll} R ::= & r \mid \langle\!\langle\rangle\!\rangle \mid \langle\!\langle R \mid m{:}\,\tau \rangle\!\rangle \mid \lambda t.\, R \mid R\tau \end{array}$$

Variance Annotations

$$\begin{array}{ll} b ::= & + \mid - \mid o \\ a ::= & b \mid ? \end{array}$$

Variance Sets

$$\begin{array}{ll} V ::= & \{\vec{t^b}\} \end{array}$$

Method Absence Annotations

$$\begin{array}{ll} M ::= & \{\vec{m}\} \end{array}$$

Kinds

$$\begin{array}{lll} k ::= & V \mid \kappa & \\ S ::= & T^a & \text{Symbol } T \text{ is a terminal.} \\ \kappa ::= & S \to \nu \mid \nu & \\ \nu ::= & (M; V) & \end{array}$$

Contexts

$$, \ ::= \quad \epsilon \mid , \ , x{:}\tau \mid , \ , t{:}V \mid , \ , r <:_w R :: \kappa$$

6.1.1 Subtyping Annotation

As discussed in the previous chapters, we use an annotated version of the symbol $<:$ in row subtyping judgments (defined below) to track where depth subtyping occurs. The annotations are:

$$B ::= w \mid w, d \mid B_1 + B_2 \qquad aqB \text{ indicates row subtyping forms.}$$

Intuitively, an annotation of w indicates that only width subtyping is used in the given subtyping judgment. The annotation w, d indicates that both width and depth subtyping may have been used. Formally, $w < w, d$ and $+$ denotes the least upper bound of B_1 and B_2 with respect to this ordering.

6.1.2 Judgment Forms

$, \vdash *$	well-formed context
$, \vdash \sigma{:}V$	well-formed type with variance V
$, \vdash R :: \kappa$	row has kind
$, \vdash \tau_1 <: \tau_2$	type τ_1 subtype of τ_2
$, \vdash R_1 <:_B R_2$	row R_1 is a B-subtype of R_2
$, \vdash e{:}\sigma$	term e has type σ

6.1.3 Judgment Shorthands

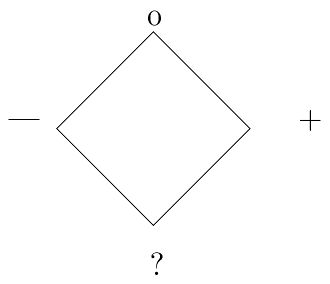
We will use the meta-judgment $, \vdash A$ to range over all of the above judgments. In addition, we use the meta-variable U to range over types τ and rows R. The meta-judgment $, \vdash U{:}-\gamma$ represents judgments of the form $, \vdash \tau{:}V$ and $, \vdash R :: \kappa$. Similarly, meta-judgment $, \vdash U_1 <:_{(B)} U_2$ represents the judgments $, \vdash \tau_1 <: \tau_2$ and $, \vdash R_1 <:_B R_2$. Finally, meta-judgment $, \vdash U_1 \cong_{(B)} U_2$ is short for the two judgments $, \vdash U_1 <:_{(B)} U_2$ and $, \vdash U_2 <:_{(B)} U_1$. We will also use the syntax **probj** $t.R$ as shorthand for either **pro** $t.R$ or **obj** $t.R$.

6.1.4 Context Access Functions

- We use the function $dom(,)$ to denote the set of row, type, and expression variables that are listed in context , .
- We use the function $TVar(,)$ to denote the set of type variables t such that the type assumptions $t\colon \{t^+\} \in$, .

6.1.5 Ordering on Variance Annotations

The ? variance indicates that the given type variable does not appear in the given expression, so the type variable is both vacuously covariant and contravariant. Intuitively, this type variable is less constrained than those with variances of $+$, $-$, or o. Hence the ? variance is considered "least". Variance $+$ indicates the given type variable appears (non-vacuously) covariantly; $-$ indicates (non-vacuous) contravariance; and o indicates invariance. Variables with o-variance are the most constrained; hence they are considered "greatest." The $+$ and $-$ variances are incomparable; however, both are more constrained than ? and less than o. Hence we get the following ordering on variance annotations, as shown in Chapter 4:



More formally, we have the ordering

$$+ \leq o,\ \ - \leq o,\ \ ? \leq -,\ \ ? \leq +$$

We will use the notation $\overline{b}$ to indicate the complement of annotation b with respect to the above ordering. In other words, $\overline{o} = o$, $\overline{+} = -$, $\overline{-} = +$, and $\overline{?} = ?$,

6.1.6 Operations on Variance Sets

$Var(t,V)$. This first operation looks up the variance annotation of type variable t in variance set V:

$$Var(t,V) = \begin{cases} b & \text{if } t^b \in V \\ ? & \text{if } t^{b'} \notin V \text{ for all } b'. \end{cases}$$

$D(V_1,\ldots,V_n)$. The function D extracts the type variables from variance sets $V_1,\ldots,V_n$:

$$D(V_1,\ldots,V_n) = \{t \mid t^b \in V_i \text{ for some } b, i \in 1,\ldots,n\}$$

$GVar(t, V_1, \ldots, V_n)$. The first argument to $GVar$ is a type variable; the remaining arguments are variance sets. $GVar$ collects the variance annotations of the given type variable in the variance sets. It then returns the least upper bound of these annotations, computed with respect to the ordering on variance annotations given above:

$$GVar(t, V_1, \ldots, V_n) = lub\{Var(t, V_1), \ldots, Var(t, V_n)\}$$

$Merge(V_1, \ldots, V_n)$. Intuitively, the $Merge$ function "merges" the variance annotations from a collection of variance sets into a single set. In more detail, it takes a collection of variance sets $V_1, \ldots, V_n$ as arguments and produces a new variance whose domain is the union of the domains of the argument sets. In the resulting set, each variable's annotation is the least upper bound of its annotations in the argument sets. For example, if t has $-$ variance in V_1 and $+$ variance in V_2 and does not appear in $V_3, \ldots, V_n$, then t will have variance o in the resulting set:

$$Merge(V_1, \ldots, V_n) = \{t^{GVar(t, V_1, \ldots, V_n)} \mid t \in D(V_1, \ldots, V_n)\}$$

$V \backslash t$. The backslash operator $\backslash$ takes a variance set V and a type variable t and returns the variance set just like V except it does not contain t:

$$V \backslash t = \{t_1^b \mid t \neq t_1 \text{ and } t_1^b \in V\}$$

The backslash operation is useful when a variable becomes bound in a type expression; since it no longer appears free in the given type, it no longer appears in the type's variance set.

$\overline{V}$. The inversion operation $\overline{V}$ takes a variance set V and returns the variance set with the same set of type variables, but with complementary annotations:

$$\overline{V} = \{t^{\overline{b}} \mid t^b \in V\}$$

This operation is useful when a type is placed in a contravariant position; for example, the left-hand side of a function type. In this case, covariant positions become contravariant and vice versa. The inversion operation reflects this change in the variance sets.

$Invar(V)$. The $Invar$ operation returns a variance set with the same type variables as its argument; however, all the annotations in the new set are o's, making them all invariant:

$$Invar(V) = \{t^o \mid t \in D(V)\}$$

This operation is used when types are placed in invariant positions; for example, within **pro** types.

6.1.7 Generalized Variance

These functions simply extend the variance access function described above to more complex kinds. They are included as a notational convenience for proofs. The first two functions extract the variance of a particular type variable from row kinds. The final three functions extract complete variance sets from kinds, throwing away any row absence annotations.

$$\begin{array}{lcl} Var(t,(M;\,V)) & = & Var(t,V) \\ Var(t,T^a \to \nu) & = & Var(t,\nu) \\ Var(V) & = & V \\ Var(M;\,V) & = & V \\ Var(T^a \to \nu) & = & T^a \to Var(\nu) \end{array}$$

6.1.8 Variance Shorthand

We will use the notation $\kappa_{min(\Gamma)}$ as a shorthand for the kind $T^o \to (\emptyset;\, Invar(TVar(,)))$. This kind is useful in typing method bodies, as it is the most restrictive with respect to subtyping.

6.1.9 Variance Substitutions

The substitution function given below describes the merging that happens when a type variable in a given variance set is replaced by a variance set. Intuitively, this arises when a type τ_2 with variance V_2 is substituted for a type variable t in a second type τ_1 with variance V_1. If t appears covariantly in V_1, then the first clause below applies. Essentially, t is removed from V_1 and the resulting variance set is merged with V_2. The second clause covers the case when t appears contravariantly in τ_1. In this case, t is removed from V_1 and the inversion of V_2 is merged with the result. This inversion corresponds to the fact that the type τ_2 is being placed in a contravariant position. The third clause corresponds to the case when t appears invariantly in τ_1. In this case, τ_2 is being placed in an invariant position. Hence $Invar(V_2)$ is merged with the result of removing t from V_1. Finally, if t does not appear in V_1, then t appears vacuously in τ_1. Therefore, the variance of the result is just V_1.

$$[V_2/t]V_1 = \begin{cases} Merge(V_1', V_2) & \text{if } V_1 = V_1', t^+ \\ Merge(V_1', \overline{V_2}) & \text{if } V_1 = V_1', t^- \\ Merge(V_1', Invar(V_2)) & \text{if } V_1 = V_1', t^o \\ V_1 & \text{if } t \notin D(V_1) \end{cases}$$

6.1.10 Ordering on Kinds

We must provide an ordering on kinds (given below) so that we can use expressions with a given variance in contexts with weaker constraints. Roughly, this relationship on kinds is the analog of subtyping on types. In more detail, variance set V_1 is less than V_2 if V_1 more tightly constrains the variances of its type variables with respect to the ordering on variance annotations given above. For example, variance set $\{t^o\}$ is less than $\{t^+\}$, because the first set requires t to be invariant, whereas the second allows t to be covariant. This relationship is extended to flat kinds by ignoring the method absence annotations. It then lifts to row function kinds by comparing the annotations on the kind arguments contravariantly and the flat kinds covariantly.

$$\begin{array}{rcl} V_1 \leq V_2 & \text{iff} & \forall t, Var(t, V_2) \leq Var(t, V_1) \\ (M_1;\, V_1) \leq (M_2;\, V_2) & \text{iff} & V_1 \leq V_2 \\ T^{b'} \rightarrow \nu_1 \leq T^b \rightarrow \nu_2 & \text{iff} & b \leq a \text{ and } \nu_1 \leq \nu_2 \end{array}$$

6.2 Typing Rules

This section presents all of the typing rules of the formal system. They are presented in groups, according to their function. The first group includes the rules for context formation. The second and third describe how to compute the variance of type and row expressions, respectively. The fourth contains the rules describing the subtyping relationships between type expressions, while the fifth does the same for row expressions. The sixth group contains various rules for type equality, while the seventh (and last) group describes how to type term expressions.

6.2.1 Context Rules

The context rules are the typing rules that govern the formation of contexts. The first of these rules, the only axiom of the system, says that the empty context is well-formed.

$(start)$

$$\frac{}{\epsilon \vdash *}$$

The $(type\ var)$ rule says that if a given type variable does not appear in a well-formed context , , then extending , with $t\colon \{t^+\}$ produces a well-formed context.

$(type\ var)$

$$\frac{\begin{array}{c} , \vdash * \\ t \notin dom(,) \end{array}}{, , t\colon \{t^+\} \vdash *}$$

The typing rule for adding row variables to contexts (given below) is one of the most complex in the system. The first hypothesis asserts that the proposed upper bound is well-formed and has functional kind $S_1 \rightarrow (M_1;\, V_1)$. The second hypothesis asserts that the proposed kind for the new row variable r is more constrained than the kind of its bound. The next hypothesis is a well-formedness condition for the kind of r, requiring that all the type variables in its variance set V_0 occur in the domain of , . The constraint $M_0 \subseteq M_1$ again serves to connect the kind of r to that of its bound; in this case requiring that any methods promised to be absent from the bound must also be absent from r. Finally, the last condition insures that r is a fresh variable.

$$(\mathit{row\ var}) \qquad \frac{\begin{array}{c} ,\ \vdash R_1 :: S_1 \rightarrow (M_1; V_1) \\ S_0 \rightarrow (M_0; V_0) \leq S_1 \rightarrow (M_1; V_1) \\ D(V_0) \subseteq \mathit{dom}(,\ ) \qquad M_0 \subseteq M_1 \\ r \notin \mathit{dom}(,\ ) \end{array}}{,\ , (r <:_w R_1 :: S_0 \rightarrow (M_0; V_0)) \vdash *}$$

The typing rule for adding expression variables to a context , is standard. It requires simply that the given type is well-formed in , , and that the variable does not already appear in , .

$$(\mathit{exp\ var}) \qquad \frac{\begin{array}{c} ,\ \vdash \tau : V \\ x \notin \mathit{dom}(,\ ) \end{array}}{,\ , x{:}\tau \vdash *}$$

The (*weakening*) rule is standard. It asserts that if we may derive a judgment A from a small context $,_1, ,_2$, and an expansion to that context $,_1, a, ,_2$ is well-formed, then we may derive A from the longer context.

$$(\mathit{weakening}) \qquad \frac{\begin{array}{c} ,_1, ,_2 \vdash A \\ ,_1, a, ,_2 \vdash * \end{array}}{,_1, a, ,_2 \vdash A}$$

$$\text{where } a ::= x{:}\tau \mid t{:}V \mid r <:_w R :: \kappa$$

We will see in Lemma 7.2.3 in Chapter 7 that this rule is derivable, and hence need not have been formally included.

6.2.2 Rules for Type Expressions

This subsection collects together the typing rules used for showing that type expressions are well-formed. In the process, these typing rules compute the variance of the type expressions. The first rule in this section asserts that any type variable in a well-formed context is well-formed and has the given variance. (Note that we can only add assumptions of the form $t : \{t^+\}$ to contexts.)

$$(\mathit{type\ proj}) \qquad \frac{\begin{array}{c} ,\ \vdash * \\ t : \{t^+\} \in\ , \end{array}}{,\ \vdash t : \{t^+\}}$$

The (*type arrow*) typing rule calculates the variance of a function type by merging the variance of its components. Since the left-hand side of function types is a contravariant position, we first invert the variance set for type τ_1.

$$(\textit{type arrow}) \qquad \frac{\begin{array}{c}, \vdash \tau_1 : V_1 \\ , \vdash \tau_2 : V_2\end{array}}{, \vdash \tau_1 \rightarrow \tau_2 : \mathit{Merge}(\overline{V_1}, V_2)}$$

To compute the variance a **pro** type **pro** $t.R$, the (*pro*) typing rule first computes the variance V for row R in a context extended with type variable t. Because **pro** is a type binding operator, it removes t from V. Finally it makes the resulting set invariant, intuitively because **pro** types are invariant.

$$(\textit{pro}) \qquad \frac{, , t{:}\,\{t^+\} \vdash R :: (M;\, V)}{, \vdash \mathbf{pro}\, t.R : \mathit{Invar}(V \backslash t)}$$

The rule for computing the variance of types of the form **obj** $t.R$ is similar to the previous rule. In this case, however, we must also know the variance of the bound variable t in R. Intuitively, if t appears either vacuously or (non-vacuously) covariantly then **obj** $t.R$ is covariant. In this case, the variance annotations for **obj** $t.R$ are just those for R without the t, which is reflected in the rule below.

$$(\textit{cov object}) \qquad \frac{\begin{array}{c}, , t{:}\,\{t^+\} \vdash R :: (M;\, V) \\ \mathit{Var}(t, V) \in \{+, ?\}\end{array}}{, \vdash \mathbf{obj}\, t.R : V \backslash t}$$

When t appears contravariantly or invariantly, then **obj** $t.R$ is invariant. However, since technical restrictions prevent us from forming such objects, we omit the corresponding rule as it is not necessary.

The final typing rule in this section shows that an existential type is well-formed if its modified type is well-formed under the appropriate assumptions about the bound row variable.

$$(\textit{exist}) \qquad \frac{, , r <:_w :: S \rightarrow (M;\, V_2) \vdash \tau : V_1}{, \vdash \exists (r <:_w :: S \rightarrow (M;\, V_2))\tau : \mathit{Merge}(V_1, V_2)}$$

6.2.3 Rules for Row Expressions

The rules in this section compute the variance sets and method absence annotations for row expressions. The first rule asserts that if we have assumed a given row variable has a given kind, then we may use that fact.

$$(\mathit{row\ proj}) \qquad \frac{\begin{array}{c} ,\vdash * \\ r <:_w R :: \kappa \in \mathit{dom}(,) \end{array}}{,\vdash r :: \kappa}$$

The (*empty row*) typing rule allows us to conclude the empty row is well-formed in any well-formed context. In addition, the empty row has the empty variance set (since its normal form contains no type variables) and is guaranteed *not* to have any set of method names M.

$$(\mathit{empty\ row}) \qquad \frac{,\vdash *}{,\vdash \langle\!\langle\rangle\!\rangle :: (M;\, \emptyset)}$$

The next typing rule is a form of subsumption for method absence annotations. In particular, if row R is guaranteed not to have methods M, and N is a subset of M, then R is guaranteed not to have any methods in N either. Here the notation $S^i \to (M;\, V)$ is short for $S \to (M;\, V)$ when i is 1 and $(M;\, V)$ when it is 0.

$$(\mathit{row\ label}) \qquad \frac{\begin{array}{c} ,\vdash R :: S^i \to (M;\, V) \\ N \subseteq M \qquad i \in \{0,1\} \end{array}}{,\vdash R :: S^i \to (N;\, V)}$$

The (*row fn abs*) typing rule calculates the variance of a row function from the variance of its body. Essentially, it looks up the variance of the newly bound variable in the body and annotates the argument kind with that variance. The flat portion of the variance is just the variance of the body minus the newly bound variable. The method absence annotations are unaffected.

$$(\mathit{row\ fn\ abs}) \qquad \frac{,\, t{:}\, \{t^+\} \vdash R :: (M;\, V)}{,\vdash \lambda t.\, R :: T^{Var(t,V)} \to (M;\, V \backslash t)}$$

The next four typing rules calculate the variance of a row function applied to a type from the variances of the function and the argument. Because the proper combination of these variance sets depends on the variance annotation of the argument kind, there are four different cases: one for each possible argument kind annotation. The first rule below (*row fn app cov*) corresponds to the case where this annotation is $+$, indicating the given function is monotonic in its argument. In this case, the variance of the function body and of the type are simply merged.

$$(\mathit{row\ fn\ app\ cov}) \qquad \frac{\begin{array}{c} ,\vdash R :: T^+ \to (M;\, V_1) \\ ,\vdash \tau : V_2 \end{array}}{,\vdash R\tau :: (M;\, Merge(V_1, V_2))}$$

The (*row fn app contra*) rule corresponds to the case where the row function is anti-monotonic in its argument. Hence the variance of the function body is merged with the inversion of the variance of the type argument.

$$(row\ fn\ app\ contra) \qquad \frac{\begin{array}{c}, \vdash R :: T^{-} \to (M;\, V_1) \\ , \vdash \tau : V_2\end{array}}{, \vdash R\tau :: (M;\, Merge(V_1, \overline{V_2}))}$$

This next rule applies when the row function is invariant in its argument. In this case, the variance of the application is the combination of the variance of the row function body and the variance of the type argument made invariant via the function *Invar*.

$$(row\ fn\ app\ inv) \qquad \frac{\begin{array}{c}, \vdash R :: T^{o} \to (M;\, V_1) \\ , \vdash \tau : V_2\end{array}}{, \vdash R\tau :: (M;\, Merge(V_1, Invar(V_2)))}$$

Finally, the rule (*row fn app vac*) corresponds to the case when the given row function ignores its argument. The variance of the application is just the variance of the row function body in this case.

$$(row\ fn\ app\ vac) \qquad \frac{\begin{array}{c}, \vdash R :: T^{?} \to (M;\, V_1) \\ , \vdash \tau : V_2\end{array}}{, \vdash R\tau :: (M;\, V_1)}$$

The typing rule (*row ext*) shows how to give a kind to an extended row of the form $\langle\!\langle R \mid m : \tau \rangle\!\rangle$ based on the kinds of its constituent parts. In particular, we must show that row R does not already have a method m by giving it a kind with m in the method absence annotations. Once that fact is established, we may compute the variance of the extended row by merging the variances of R and τ, the type for the new method. The method absence annotation for the new row is just that of the old one, minus the new method name m.

$$(row\ ext) \qquad \frac{\begin{array}{c}, \vdash R :: (\{\vec{m}, m\}, V_1) \\ , \vdash \tau : V_2\end{array}}{, \vdash \langle\!\langle R \mid m{:}\tau \rangle\!\rangle :: (\{\vec{m}\};\, Merge(V_1, V_2))}$$

6.2.4 Subtyping Rules for Types

The typing rules given in this section define the subtyping relationship between types. The first rule in this section is the standard rule for reflexivity: any well-formed type is a subtype of itself.

$$(<:\ type\ refl) \qquad \frac{,\ \vdash \tau : V}{,\ \vdash \tau <: \tau}$$

The $(<:\rightarrow)$ rule is the standard rule for subtyping between function types.

$$(<:\rightarrow) \qquad \frac{\begin{array}{c},\ \vdash \tau_1' <: \tau_1 \\ ,\ \vdash \tau_2 <: \tau_2'\end{array}}{,\ \vdash \tau_1 \rightarrow \tau_2 <: \tau_1' \rightarrow \tau_2'}$$

The typing rule $(<:\ obj)$ plays two roles. In the first of these, it defines when **pro** types may be converted to **obj** types. The second role defines when two **obj** types are in the subtyping relationship. This rule is explained in detail in Section 4.1.4.

$$(<:\ obj) \qquad \frac{\begin{array}{c},\ ,\ t : \{t^+\} \vdash R_1 <:_B R_2 \\ ,\ \vdash \mathbf{probj}\, t\boldsymbol{.}R_1 : V_1 \\ ,\ ,\ t : \{t^+\} \vdash R_2 :: (M; V_2) \qquad Var(t, V_2) \in \{?, +\}\end{array}}{,\ \vdash \mathbf{probj}\, t\boldsymbol{.}R_1 <: \mathbf{obj}\, t\boldsymbol{.}R_2}$$

The next typing rule is included for technical reasons; it asserts that two **pro** types $\mathbf{pro}\, t\boldsymbol{.}R_1$ and $\mathbf{pro}\, t\boldsymbol{.}R_2$ are in the subtyping relation if their rows R_1 and R_2 are mutual width subtypes (and are well-formed). This rule is intuitively necessary because an important property of the type system is that if type variable t appears in a type or row expression with ? variance, then regardless of what we substitute for t, we will get equivalent types. Unfortunately, because variance annotations describe the variances of the *normal forms* of type and row expressions, the types resulting from such substitutions need *not* be syntactically identical. (Consider, for example, the type expression $\mathbf{pro}\, t\boldsymbol{.}r\, t$, where r is a free row-variable that is guaranteed to throw away its argument.) This typing rule helps insure that the resulting types will, however, be mutual subtypes.

$$(<:\ convert) \qquad \frac{\begin{array}{c},\ ,\ t : \{t^+\} \vdash R_1 \cong_w R_2 \\ ,\ ,\ t : \{t^+\} \vdash R_i :: (M_i; V_i) \qquad i \in \{1, 2\}\end{array}}{,\ \vdash \mathbf{pro}\, t\boldsymbol{.}R_1 <: \mathbf{pro}\, t\boldsymbol{.}R_2}$$

The $(<:\ type\ trans)$ rule is the standard transitivity rule for type expressions.

$$(<:\ type\ trans) \qquad \frac{\begin{array}{c},\ \vdash \tau_1 <: \tau_2 \\ ,\ \vdash \tau_2 <: \tau_3\end{array}}{,\ \vdash \tau_1 <: \tau_3}$$

We do not conclude any subtyping rules for existential types for the sake of simplicity.

6.2.5 Subtyping Rules for Rows

The rules collected in this section describe the subtyping relationships between row expressions. Because width and depth subtyping for row expressions have different variance properties and different ramifications for type soundness, it is necessary to closely track where depth subtyping occurs.

The first rule in this section is a reflexivity rule for row expressions. It asserts that any well-formed row expression is a subtype of itself. Furthermore, it may be considered either a width subtype or a width-and-depth subtype.

$$(<:\ \mathit{row\ refl}) \qquad \frac{,\ \vdash R :: \kappa}{,\ \vdash R <:_B R}$$

The (*row proj bound*) typing rule allows us to conclude that a row variable is a subtype of the bound given for it in any well-formed context.

$$(\mathit{row\ proj\ bound}) \qquad \frac{\begin{array}{c},\ \vdash * \\ r <:_w R :: \kappa \in\ ,\end{array}}{,\ \vdash r <:_w R}$$

The next typing rule allows us to extend the subtyping relationship between flat rows point-wise to row functions.

$$(<:\ \lambda) \qquad \frac{\begin{array}{c},\ ,\ t\colon \{t^+\} \vdash R_1 <:_B R_2 \\ ,\ ,\ t\colon \{t^+\} \vdash R_2 :: \nu\end{array}}{,\ \vdash \lambda t.\, R_1 <:_B \lambda t.\, R_2}$$

The next four typing rules allow us to show subtyping relationships between row expressions of the forms $R_1\,\tau_1$ and $R_2\,\tau_2$. There are four cases, roughly corresponding to the variance of the argument kind for R_2. Lemma 7.6.6 in Chapter 7 establishes the soundness of the first of these rules, while Lemma 7.6.7 shows the remaining three are sound.

The first of these rules roughly corresponds to the case where R_2's argument kind is o, invariant. In this case, the necessary relationship between the argument types τ_1 and τ_2 is that they are mutual subtypes, a relationship which implies they are essentially the same type. This essential-sameness guarantees that we do not need to use depth subtyping to conclude $R_1\,\tau_1$ is a subtype of $R_2\,\tau_2$. Hence, these row applications are in the same flavor subtyping relationship as R_1 and R_2. Because this relationship between τ_1 and τ_2 allows us to show a width subtyping relationship between row applications, this rule is generalized to allow any variance for R_2's argument kind. (The more specialized rules for $+$ and $-$ variance, which appear below, can only conclude depth subtyping judgments.) This rule may be viewed as generalizing reflexivity.

$$(<:\ \mathit{app\ cong}) \qquad \frac{\begin{array}{c} ,\ \vdash R_1 <:_B R_2 \\ ,\ \vdash R_2 :: T^a \to \nu \\ ,\ \vdash \tau_1 \cong \tau_2 \end{array}}{,\ \vdash R_1\tau_1 <:_B R_2\tau_2}$$

In the second of these rules, we must show that the row function R_2 is monotonic in its argument and that $\tau_1 <: \tau_2$. Under these conditions, we may conclude that $R_1\,\tau_1$ is a subtype of $R_2\,\tau_2$. However, since depth-subtyping may have been involved, we must add the depth annotation to the subtyping relation.

$$(<:\ \mathit{app\ cov}) \qquad \frac{\begin{array}{c} ,\ \vdash R_1 <:_B R_2 \\ ,\ \vdash R_2 :: T^+ \to \nu \\ ,\ \vdash \tau_1 <: \tau_2 \end{array}}{,\ \vdash R_1\tau_1 <:_{B+d} R_2\tau_2}$$

The rule $(<:\ \mathit{app\ contra})$ is dual to $(<:\ \mathit{app\ cov})$.

$$(<:\ \mathit{app\ contra}) \qquad \frac{\begin{array}{c} ,\ \vdash R_1 <:_B R_2 \\ ,\ \vdash R_2 :: T^- \to \nu \\ ,\ \vdash \tau_2 <: \tau_1 \end{array}}{,\ \vdash R_1\tau_1 <:_{B+d} R_2\tau_2}$$

The final rule in this group applies when R_2 ignores its argument, *i.e.*, it has ? variance. In this case, we need not establish any relationship between τ_1 and τ_2; we require only that they are well-formed. Additionally, because R_2 ignores its argument, we need not use depth subtyping to establish the desired subtyping relationship.

$$(<:\ \mathit{app\ vac}) \qquad \frac{\begin{array}{c} ,\ \vdash R_1 <:_B R_2 \\ ,\ \vdash R_2 :: T^? \to \nu \\ ,\ \vdash \tau_1 : V_1 \qquad ,\ \vdash \tau_2 : V_2 \end{array}}{,\ \vdash R_1\tau_1 <:_B R_2\tau_2}$$

The next three typing rules show subtyping relationships between flat row expressions. Intuitively, the $(<:\ d)$ rule corresponds to depth subtyping and $(<:\ w)$ to width subtyping. The third rule $(<:\ \mathit{cong})$ (which is the first one given) allows us to conclude width subtyping judgments between rows that have components that are not syntactically identical, but *are* mutual subtypes. This rule may be viewed as a generalization of reflexivity. In this sense, it is analogous to the ($<:$ *app cong*) rule above. The condition that $, \vdash \langle\!\langle R_i \,|\, m\colon\tau_i \rangle\!\rangle :: \nu_i$ for $i \in \{1,2\}$ helps us conclude that any types related via subtyping are well-formed.

$$(<:\ \mathit{cong}) \qquad \frac{\begin{array}{c} ,\vdash R_1 <:_B R_2 \\ ,\vdash \tau_1 \cong \tau_2 \\ ,\vdash \langle\!\langle R_i \,|\, m\colon\tau_i \rangle\!\rangle :: \nu_i \qquad i \in \{1,2\} \end{array}}{,\vdash \langle\!\langle R_1 \,|\, m\colon\tau_1 \rangle\!\rangle <:_B \langle\!\langle R_2 \,|\, m\colon\tau_2 \rangle\!\rangle}$$

$$(<:\ d) \qquad \frac{\begin{array}{c} ,\vdash R_1 <:_B R_2 \qquad ,\vdash \tau_1 <: \tau_2 \\ ,\vdash \langle\!\langle R_i \,|\, m\colon\tau_i \rangle\!\rangle :: \nu_i \qquad i \in \{1,2\} \end{array}}{,\vdash \langle\!\langle R_1 \,|\, m\colon\tau_1 \rangle\!\rangle <:_{B+d} \langle\!\langle R_2 \,|\, m\colon\tau_2 \rangle\!\rangle}$$

$$(<:\ w) \qquad \frac{\begin{array}{c} ,\vdash R_1 <:_B R_2 \\ ,\vdash \langle\!\langle R_1 \,|\, m\colon\tau \rangle\!\rangle :: \nu \end{array}}{,\vdash \langle\!\langle R_1 \,|\, m\colon\tau \rangle\!\rangle <:_{B+w} R_2}$$

The ($<:$ *row trans*) rule is essentially the standard rule for transitivity between row expressions; the only detail is in the subtyping annotations. Intuitively, if depth subtyping were used in either hypothesis, then the conclusion judgment passes that information on by making the annotation in the result the sum of the hypothesis annotations.

$$(<:\ \mathit{row\ trans}) \qquad \frac{\begin{array}{c} ,\vdash R_1 <:_B R_2 \\ ,\vdash R_2 <:_{B'} R_3 \end{array}}{,\vdash R_1 <:_{B+B'} R_3}$$

6.2.6 Type and Row Equality Rules

Type or row expressions that differ only in names of bound variables are considered identical. Additional equations between types and rows arise as a result of β-reduction, written $\rightarrow_\beta$, or β-conversion, written $\leftrightarrow_\beta$. The two equality rules for subtyping judgments are written in a restricted form to simplify soundness proofs.

$$(\mathit{row}\ \beta) \qquad \frac{,\vdash R :: \kappa \quad R \rightarrow_\beta R'}{,\vdash R' :: \kappa}$$

$$(\mathit{type}\ \beta) \qquad \frac{,\vdash \tau : V \quad \tau \rightarrow_\beta \tau'}{,\vdash \tau' : V}$$

(*type eq*)
$$\frac{,\vdash e : \sigma \quad \sigma \leftrightarrow_{\beta} \sigma' \quad ,\vdash \sigma' : V}{,\vdash e : \sigma'}$$

(<: *β right*)
$$\frac{,\vdash R_1 <:_B (\lambda t.\, R_2)\tau_2}{,\vdash R_1 <:_B [\tau_2/t]R_2}$$

(<: *β left*)
$$\frac{,\vdash (\lambda t.\, R_1)\tau_1 <:_B R_2}{,\vdash [\tau/t]R_1 <:_B R_2}$$

6.2.7 Rules for Assigning Types to Terms

The first four rules are standard rules for assigning types to terms. The first allows us to use expression variables with their assumed type. The second is the standard (*subsumption*) rule, while the third and fourth are standard lambda abstraction and application rules, respectively.

(*exp proj*)
$$\frac{\begin{array}{c},\vdash * \\ x : \tau \in ,\end{array}}{,\vdash x : \tau}$$

(*subsumption*)
$$\frac{,\vdash e : \tau_1 \quad ,\vdash \tau_1 <: \tau_2}{,\vdash e : \tau_2}$$

(*exp abs*)
$$\frac{,\ , x{:}\tau_1 \vdash e : \tau_2}{,\vdash \lambda x.\, e : \tau_1 \rightarrow \tau_2}$$

(*exp app*)
$$\frac{,\vdash e_1 : \tau_1 \rightarrow \tau_2 \quad ,\vdash e_2 : \tau_1}{,\vdash e_1\, e_2 : \tau_2}$$

Typing rule (*empty* **pro**) allows us to conclude that the empty object $\langle\rangle$ has the empty **pro** type **pro** $t.\langle\!\langle\rangle\!\rangle$ in any well-formed context.

(*empty pro*)
$$\frac{,\vdash *}{,\vdash \langle\rangle : \mathbf{pro}\, t.\langle\!\langle\rangle\!\rangle}$$

The next typing rule, $(pro\ ext)$, shows how to type prototype (or object) extension, *i.e.*, expressions of the form $\langle e_1 \longleftarrow\!\!\!+\; m = e_2\rangle$.

$$(pro\ ext) \qquad \frac{\begin{array}{l} ,\ \vdash e_1 : \mathbf{pro}\, t\,\textbf{.}R \\ ,\ ,\ t\colon\{t^+\} \vdash R :: (\{m\};\, V) \\ ,\ ,\ I_r \vdash e_2 : [\mathbf{pro}\, t\,\textbf{.}rt/t](t \to \tau) \qquad r \notin V(\tau) \end{array}}{,\ \vdash \langle e_1 \longleftarrow\!\!\!+\; m = e_2\rangle : \mathbf{pro}\, t\,\textbf{.}\langle\!\langle R \mid m\colon\tau\rangle\!\rangle}$$

$$\text{where } I_r = r <:_w \lambda t.\langle\!\langle R \mid m : \tau\rangle\!\rangle :: \kappa_{min(\Gamma)}\,.$$

The first hypothesis requires e_1 to be an extensible object with some list of available methods R. The second hypothesis insures that R does not already have an m method. Finally, the last hypothesis guarantees that the proposed method body e_2 has the appropriate type for all future extensions of the current host object, $\langle e_1 \longleftarrow\!\!\!+\; m = e_2\rangle$. To that end, we must type e_2 under the assumption that its host object has type $\mathbf{pro}\, t\,\textbf{.}r\, t$, where all we know about r is that it is a subtype of the current row of methods $\lambda t.\langle\!\langle R \mid m : \tau\rangle\!\rangle$. (It is more convenient to work with row functions than flat rows because we need these lists of method-type pairs to be parametric in the type of the host object.) Intuitively, type $\mathbf{pro}\, t\,\textbf{.}r\, t$ captures the notion of "all future extensions of the current host object." The final hypothesis also requires e_2 to be a function from its host object to the actual code for the method body.

The kind assumed for r, $\kappa_{min(\Gamma)}$, which abbreviates $T^o \to (\emptyset;\, Invar(TVar(,\ )))$, is the "smallest" kind we can write in context , . This kind guarantees that method bodies cannot add new method bodies to their host object (a nonsensical operation) by having an empty method absence annotation set. It also gives the most invariant variance possible for r. In some sense, this variance makes the minimum commitment to r's variance, since Lemma 7.13.1 reveals that we may substitute rows with a given variance into contexts expecting lower variance.

The rule for typing prototype (or object) override is similar to $(pro\ ext)$.

$$(pro\ over) \qquad \frac{\begin{array}{l} ,\ \vdash e_1 : \mathbf{pro}\, t\,\textbf{.}R \\ ,\ ,\ t\colon\{t^+\} \vdash R <:_w \langle\!\langle m\colon\tau\rangle\!\rangle \\ ,\ ,\ I_r \vdash e_2 : [\mathbf{pro}\, t\,\textbf{.}rt/t](t \to \tau) \end{array}}{,\ \vdash \langle e_1 \leftarrow m{=}e_2\rangle : \mathbf{pro}\, t\,\textbf{.}R}$$

$$\text{where } I_r = r <:_w \lambda t.\, R :: \kappa_{min(\Gamma)}\,.$$

The first hypothesis requires that e_1, the object to be overridden, is overridable and has some row of available methods, R. The second hypothesis determines that e_1 already has an m method, with type τ. The use of width subtyping insures that the new method body has a type that is equivalent (up to mutual subtyping) to the original type. This equivalence is essential for the soundness of method override. The final hypothesis guarantees that the new method body e_2 has the appropriate type for all possible future extensions of the current host object, in the same manner as the (*pro ext*) typing rule.

The ($\mathit{probj} \Leftarrow$) typing rule allows us to type-check message sending to expressions with both **pro** and **obj** type, depending on whether **probj** is **pro** or **obj**.

$$(\mathit{probj} \Leftarrow) \qquad \frac{\begin{array}{c} , \vdash e : \mathbf{probj}\, t.R \\ , , t{:}\{t^+\} \vdash R <:_w \langle\!\langle m{:}\tau \rangle\!\rangle \end{array}}{, \vdash e \Leftarrow m : [\mathbf{probj}\, t.R/t]\tau}$$

The first hypothesis requires that expression e, the receiver of the m message, be some form of object and have a list of methods R. The second hypothesis insures that R contains an m method and that this method has type τ. The message send then has type τ, with all occurrences of the "mytype" type variable t replaced by the type of the current host object, $\mathbf{probj}\, t.R$.

The next typing rule is a fairly standard bounded existential introduction rule. It allows us to form a term of bounded existential type as long as we can show the proper relationships between the various components.

$$(\exists <:\ \mathit{intro}) \qquad \frac{\begin{array}{c} , \vdash R_1 :: \kappa \\ , \vdash R_1 <:_w R \\ , \vdash e : [R_1/r]\tau \end{array}}{, \vdash \{\!|r <:_w R :: \kappa = R_1,\ e|\!\} : \exists(r <:_w R :: \kappa)\tau}$$

In particular, we must show that R_1, the actual implementation row, has the kind asserted for the variable r. The second hypothesis insures that R_1 is a subtype of the bound assumed for r. Finally, we must show that the implementation of the abstraction, e, has type τ, with all occurrences of r replaced by R_1.

The final typing rule allows us to use expressions with bounded existential type. To do so, we must show that expression e_1 has such a type and that the client expression e_2 makes the appropriate assumptions about the abstract type; namely, that the implementation type is a subtype of the asserted bound and has the appropriate kind and that the implementation expression has the indicated type, τ. The final hypothesis essentially shows that the client expression cannot leak any of the components of the abstract data type to external views.

$$(\exists <:\ \mathit{elim}) \qquad \frac{\begin{array}{c} ,\ \vdash e_1 : \exists(r <:_w R :: \kappa)\tau \\ ,\ , r <:_w R :: \kappa,\ x{:}\tau \vdash e_2 : \tau_2 \\ ,\ \vdash \tau_2 : V \end{array}}{,\ \vdash \mathit{Abstype}\ r <:_w R :: \kappa\ \mathit{with}\ x : \tau\ \mathit{is}\ e_1\ \mathit{in}\ e_2 : \tau_2}$$

Chapter 7

Proofs

We prove the soundness of our type system with respect to the operational semantics given in Appendix E. We first prove that evaluation preserves type; this property is traditionally called subject reduction. We then show that no typeable expression evaluates to *error* using the evaluation rules given in Appendix F. This fact guarantees that we have no message-not-understood errors for expressions with either **pro** or **obj** types. The structure of the proof is similar to that in [Com94]. In particular, Compagnoni's thesis provided the insight that we need not eliminate transitivity to prove subject reduction. Most of the work in proving type soundness lies in showing the subject reduction theorem.

The lemmas needed to show subject reduction and type soundness may be grouped as follows:

Variance Lemmas: Section 7.1

The lemmas in this section prove properties about the operations on variance sets.

Context Properties: Sections 7.2 and 7.4

The context lemmas allow us consider (*weakening*) as a derived rule and to remove extraneous assumptions from contexts. Some of these lemmas depend upon definitions given in the Type Normal Form section (Section 7.3); hence these lemmas appear in Section 7.4.

Type Normal Form: Section 7.3

The equality rules introduce many non-essential derivations. These extraneous derivations unnecessarily complicate derivation analysis. We therefore restrict our attention to derivations of a special form, which we call normal-form derivations. Section 7.3 defines normal-form derivations and proves that the notion is well-defined.

Type Substitution: Section 7.5

The lemmas in this section describe various ways of substituting types for type variables. This section includes the standard type substitution lemma, as well as more general ones involving connections between subtyping, variance analysis, and type substitution.

Derived Equality Rules: Section 7.6

To eliminate the equality rules in the manner proscribed by normal-form derivations, we must show we can derive all the necessary equality rules in the weakened system. This section collects together such lemmas. To establish these derived rules, we must use Lemma 7.6.2, which intuitively is a Subtyping Characterization Lemma, and Lemmas 7.6.3 and 7.6.4, which are Subtyping Properties Lemmas. However, because these lemmas are needed to prove the Derived Equality Lemmas in this section, they appear in this section as well. This slight disruption to the conceptual grouping allows us to preserve the property that every lemma is given before it is used.

Normal Form Lemma: Section 7.7

The single lemma in this section reveals that although not all judgments derivable in our full type system are derivable by normal-form derivations, all judgments whose row and type expressions are in a particular form (which we call τnf for type normal form) *are* derivable via normal-form derivations. Since every expression that has a type must have a type in τnf, this lemma shows that we may prove type soundness using only normal-form derivations.

Kinding Properties: Section 7.8

The lemmas in this section show that our kind analysis is sound. In particular, these lemmas show that each row and type expression has a unique variance in a given context and that the methods contained in a row cannot be listed in the method absence annotations for that row.

Subtyping Characterization: Sections 7.9 and 7.6

This group of lemmas show which types and rows can be related via subtyping.

Subtyping Implies Component Subtyping: Section 7.10

The lemmas collected in this section show that if two rows (or types) are related via subtyping, then their constituent row and type expressions must also be related via subtyping.

Subtyping Properties: Sections 7.11 and 7.6

This section groups proofs that connect the subtyping relation to other parts of the type system. Although Lemmas 7.6.3 and 7.6.4 fit this description, they appear in earlier sections because they are needed earlier in the proof. The first lemma listed in this section, Lemma 7.11.1, is the most difficult lemma in the entire proof of type soundness.

Method Extraction: Section 7.12

The lemmas in this section are needed to prove that method bodies in well-typed objects are themselves well-typed. Essentially, these lemmas show that width supertyping only forgets about the existence of methods, not any information about the remembered methods.

Row Substitution: Section 7.13

This section contains a fairly standard row-substitution lemma, used to show that polymorphic method bodies may be instantiated as necessary. The other lemma in this section is a technical lemma that allows us to strengthen assumptions about row variables in judgments.

Expression Lemmas: Section 7.14

We finally reach proofs specific to the expression portion of our language. This section includes a standard expression substitution lemma (Lemma 7.14.10) as well as lemmas that show that the method bodies inside of well-typed prototypes and objects are themselves well-typed and appropriately polymorphic (Lemmas 7.14.2,7.14.3, and 7.14.4). These lemmas are then used to show that each reduction axiom exhibits the subject reduction property (Lemmas 7.14.5, 7.14.6, 7.14.7, 7.14.8, 7.14.9, 7.14.12, and 7.14.13). Lemma 7.14.16 then reveals that a derivation from a judgment , $\vdash_N e : \sigma$, asserting that expression e has type σ, can depend only on the form of σ, not on the form of e. Combining this result with the subject reduction results for the reduction axioms gives us Theorem 7.14.17 (Subject Reduction) for the full operational semantics.

Type Soundness: Section 7.15

The final section formalizes the notion of message-not-understood errors and shows via the subject reduction theorem that no well-typed expression encounters such an error during its evaluation.

7.1 Variance Lemmas

For reference, the definitions used in this section all appear in Appendix C. First some observations that are immediate from the definition of $Merge$:

Observation 7.1.1 *If $V_1, \ldots, V_n$ are variance sets and p is any permutation, then*

- $Merge(V) = V$
- $Merge(V_1, \ldots, V_n) = Merge(p(V_1, \ldots, V_n))$
- $Merge(V_1, V_1, \ldots, V_n) = Merge(V_1, \ldots, V_n)$
- $D(Merge(V_1, \ldots, V_n)) = D(V_1, \ldots, V_n)$

Lemma 7.1.2 $Merge(Merge(V_1, V_2), V_3, \ldots, V_n) = Merge(V_1, V_2, \ldots, V_n)$

Proof

$$\begin{aligned}
&Merge(Merge(V_1, V_2), V_3, \ldots, V_n)\\
&= \{t^{GVar(t, Merge(V_1,V_2), V_3, \ldots, V_n)} \mid t \in D(Merge(V_1, V_2), V_3, \ldots, V_n)\}\\
&= \{t^{lub(Var(t, Merge(V_1,V_2)), Var(t,V_3), \ldots, Var(t,V_n))} \mid t \in D(V_1, \ldots, V_n)\}
\end{aligned}$$

Now

$$Var(t, Merge(V_1, V_2)) = \begin{cases} lub\{Var(t, V_1), Var(t, V_2)\} & \text{if } t \in D(V_1, V_2) \\ ? & \text{otherwise} \end{cases}$$

If $t \notin D(V_1, V_2)$, then since $t \in D(V_1, \ldots, V_n)$, there must be some $i \in 3 \ldots n$ such that $t^b \in V_i$. Hence

$$\begin{aligned}
&lub\{Var(t, Merge(V_1, V_2), Var(t, V_3), \ldots, Var(t, V_n)\}\\
&= lub\{lub\{Var(t, V_1), Var(t, V_2)\}, Var(t, V_3), \ldots, Var(t, V_n)\}\\
&= lub\{Var(t, V_1), Var(t, V_2), Var(t, V_3), \ldots, Var(t, V_n)\}\\
&= GVar(t, V_1, V_2, V_3, \ldots, V_n)
\end{aligned}$$

Thus

$$\begin{aligned}
&Merge(Merge(V_1, V_2), V_3, \ldots, V_n)\\
&= \{t^{GVar(t, V_1, V_2, V_3, \ldots, V_n)} \mid t \in D(V_1, \ldots, V_n)\}\\
&= Merge(V_1, V_2, V_3, \ldots V_n)
\end{aligned}$$ ■

Lemma 7.1.3 $Merge(V, \overline{V}) = Invar(V)$

Proof

$$\begin{aligned}
Merge(V,\overline{V}) &= \{t^{GVar(t,V,\overline{V})} \mid t \in D(V,\overline{V})\} \\
&= \{t^{lub\{Var(t,V),Var(t,\overline{V})\}} \mid t \in D(V)\} \\
&= \{t^{o} \mid t \in D(V)\} \\
&= Invar(V)
\end{aligned}$$

■

Lemma 7.1.4 $Merge(V, Invar(V)) = Invar(V)$

Proof

$$\begin{aligned}
Merge(V,Invar(V)) &= \{t^{GVar(t,V,Invar(V))} \mid t \in D(V,Invar(V))\} \\
&= \{t^{lub\{Var(t,V),Var(t,Invar(V))\}} \mid t \in D(V)\} \\
&= \{t^{o} \mid t \in D(V)\} \\
&= Invar(V)
\end{aligned}$$

■

Lemma 7.1.5 $\overline{Merge(V_1, V_2)} = Merge(\overline{V_1}, \overline{V_2})$

Proof

$$\begin{aligned}
\overline{Merge(V_1,V_2)} &= \overline{\{t^{GVar(t,V_1,V_2)} \mid t \in D(V_1,V_2)\}} \\
&= \{t^{\overline{GVar(t,V_1,V_2)}} \mid t \in D(V_1,V_2)\} \\
&= \{t^{\overline{lub\{Var(t,V_1),Var(t,V_2)\}}} \mid t \in D(V_1,V_2)\} \\
&= \{t^{lub\{\overline{Var(t,V_1)},\overline{Var(t,V_2)}\}} \mid t \in D(V_1,V_2)\} \\
&= \{t^{lub\{Var(t,\overline{V_1}),Var(t,\overline{V_2})\}} \mid t \in D(V_1,V_2)\} \\
&= \{t^{GVar(t,\overline{V_1},\overline{V_2})} \mid t \in D(V_1,V_2)\} \\
&= Merge(\overline{V_1},\overline{V_2})
\end{aligned}$$

■

Lemma 7.1.6 $Invar(Merge(V_1, V_2)) = Merge(Invar(V_1), Invar(V_2))$

Proof

$$\begin{aligned}
Invar(Merge(V_1,V_2)) &= Invar(\{t^{GVar(t,V_1,V_2)} \mid t \in D(V_1,V_2)\}) \\
&= \{t^{Invar(GVar(t,V_1,V_2))} \mid t \in D(V_1,V_2)\} \\
&= \{t^{Invar(lub\{Var(t,V_1),Var(t,V_2)\})} \mid t \in D(V_1,V_2)\} \\
&= \{t^{lub\{Invar(Var(t,V_1)),Invar(Var(t,V_2))\}} \mid t \in D(V_1,V_2)\} \\
&= \{t^{lub\{Var(t,Invar(V_1)),Var(t,Invar(V_2))\}} \mid t \in D(V_1,V_2)\} \\
&= \{t^{GVar(t,Invar(V_1),Invar(V_2))} \mid t \in D(V_1,V_2)\} \\
&= Merge(Invar(V_1),Invar(V_2))
\end{aligned}$$

■

Lemma 7.1.7 $Invar(V) = \overline{Invar(V)} = Invar(\overline{V}) = Invar(Invar(V))$

Proof Immediate from the definition of *Invar*. ■

Lemma 7.1.8 $\overline{[V/t]V_1} = [V/t]\overline{V_1}$

Proof Since

$$[V/t]V_1 = \begin{cases} Merge(V_1', V) & \text{if } V_1 = V_1', t^+ \\ Merge(V_1', \overline{V}) & \text{if } V_1 = V_1', t^- \\ Merge(V_1', Invar(V)) & \text{if } V_1 = V_1', t^o \\ V_1 & \text{otherwise} \end{cases}$$

it follows that

$$\overline{[V/t]V_1} \quad = \quad \begin{cases} \overline{Merge(V_1', V)} & \text{if } V_1 = V_1', t^+ \\ \overline{Merge(V_1', \overline{V})} & \text{if } V_1 = V_1', t^- \\ \overline{Merge(V_1', Invar(V))} & \text{if } V_1 = V_1', t^o \\ \overline{V_1} & \text{otherwise} \end{cases}$$

By Lemmas 7.1.5 and 7.1.7, we then get:

$$\begin{aligned} \overline{[V/t]V_1} \quad &= \quad \begin{cases} Merge(\overline{V_1'}, \overline{V}) & \text{if } V_1 = V_1', t^+ \\ Merge(\overline{V_1'}, V) & \text{if } V_1 = V_1', t^- \\ Merge(\overline{V_1'}, Invar(V)) & \text{if } V_1 = V_1', t^o \\ \overline{V_1} & \text{otherwise} \end{cases} \\ &= \quad \begin{cases} Merge(\overline{V_1'}, \overline{V}) & \text{if } \overline{V_1} = \overline{V_1'}, t^- \\ Merge(\overline{V_1'}, V) & \text{if } \overline{V_1} = \overline{V_1'}, t^+ \\ Merge(\overline{V_1'}, Invar(V)) & \text{if } \overline{V_1} = \overline{V_1'}, t^o \\ \overline{V_1} & \text{otherwise} \end{cases} \\ &= \quad [V/t]\overline{V_1} \end{aligned}$$

■

Lemma 7.1.9 $Invar([V/t]V_1) = [V/t]Invar(V_1)$

Proof Since

$$[V/t]V_1 = \begin{cases} Merge(V_1', V) & \text{if } V_1 = V_1', t^+ \\ Merge(V_1', \overline{V}) & \text{if } V_1 = V_1', t^- \\ Merge(V_1', Invar(V)) & \text{if } V_1 = V_1', t^o \\ V_1 & \text{otherwise} \end{cases}$$

it follows that

$$\begin{aligned}
Invar([V/t]V_1) &= \begin{cases} Invar(Merge(V_1', V)) & \text{if } V_1 = V_1', t^+ \\ Invar(Merge(V_1', \overline{V})) & \text{if } V_1 = V_1', t^- \\ Invar(Merge(V_1', Invar(V))) & \text{if } V_1 = V_1', t^o \\ Invar(V_1) & \text{otherwise} \end{cases} \\
&= \begin{cases} Merge(Invar(V_1'), Invar(V)) & \text{if } V_1 = V_1', t^+ \\ Merge(Invar(V_1'), Invar(V)) & \text{if } V_1 = V_1', t^- \\ Merge(Invar(V_1'), Invar(V)) & \text{if } V_1 = V_1', t^o \\ Invar(V_1) & \text{otherwise} \end{cases} \\
&= \begin{cases} Merge(Invar(V_1'), Invar(V)) & \text{if } Invar(V_1) = Invar(V_1'), t^o \\ Invar(V_1) & \text{otherwise} \end{cases} \\
&= [V/t]Invar(V_1)
\end{aligned}$$

■

Lemma 7.1.10 $Merge([V/t]V_1, [V/t]V_2) = [V/t]Merge(V_1, V_2)$

Proof Since

$$[V/t]V_i = \begin{cases} Merge(V_i', V) & \text{if } V_i = V_i', t^+ \\ Merge(V_i', \overline{V}) & \text{if } V_i = V_i', t^- \\ Merge(V_i', Invar(V)) & \text{if } V_i = V_i', t^o \\ V_i & \text{otherwise} \end{cases}$$

there are sixteen cases to consider.

Case 1: $V_1 = V_1', t^+$ **and** $V_2 = V_2', t^+$

In this case,

$$\begin{aligned} [V/t]V_1 &= Merge(V_1', V) \\ [V/t]V_2 &= Merge(V_2', V) \end{aligned}$$

so

$$\begin{aligned}
Merge([V/t]V_1, [V/t]V_2) &= Merge(Merge(V_1', V), Merge(V_2', V)) \\
&= Merge(V_1', V_2', V, V) \\
&= Merge(Merge(V_1', V_2'), V) \\
&= Merge(Merge(V_1, V_2)\backslash t, V) \\
&= [V/t]Merge(V_1, V_2)
\end{aligned}$$

Case 2: $V_1 = V_1', t^+$ **and** $V_2 = V_2', t^-$

Here,

$$\begin{aligned} [V/t]V_1 &= Merge(V_1', V) \\ [V/t]V_2 &= Merge(V_2', \overline{V}) \end{aligned}$$

so

$$\begin{aligned} Merge([V/t]V_1, [V/t]V_2) &= Merge(Merge(V_1', V), Merge(V_2', \overline{V})) \\ &= Merge(V_1', V_2', V, \overline{V}) \\ &= Merge(Merge(V_1', V_2'), Merge(V, \overline{V})) \\ &= Merge(Merge(V_1', V_2'), Invar(V)) \\ &= Merge(Merge(V_1, V_2)\backslash t, Invar(V)) \\ &= [V/t]Merge(V_1, V_2) \end{aligned}$$

The last equation follows from the fact that $Var(t, Merge(V_1, V_2)) = o$ when $Var(t, V_1) = +$ and $Var(t, V_2) = -$.

Case 3: $V_1 = V_1', t^+$ **and** $V_2 = V_2', t^o$

In this case,

$$\begin{aligned} [V/t]V_1 &= Merge(V_1', V) \\ [V/t]V_2 &= Merge(V_2', Invar(V)) \end{aligned}$$

so

$$\begin{aligned} Merge([V/t]V_1, [V/t]V_2) &= Merge(Merge(V_1', V), Merge(V_2', Invar(V))) \\ &= Merge(V_1', V_2', V, Invar(V)) \\ &= Merge(Merge(V_1', V_2'), Merge(V, Invar(V))) \\ &= Merge(Merge(V_1', V_2'), Invar(V)) \\ &= Merge(Merge(V_1, V_2)\backslash t, Invar(V)) \\ &= [V/t]Merge(V_1, V_2) \end{aligned}$$

The last equation follows from the fact that $Var(t, Merge(V_1, V_2)) = o$ when $Var(t, V_1) = +$ and $Var(t, V_2) = o$.

Case 4: $V_1 = V_1', t^+$ **and** $t \notin D(V_2)$

Here

$$\begin{aligned}[V/t]V_1 &= Merge(V_1', V)\\ [V/t]V_2 &= V_2\end{aligned}$$

so

$$\begin{aligned}Merge([V/t]V_1, [V/t]V_2) &= Merge(Merge(V_1', V), V_2)\\ &= Merge(V_1', V_2, V)\\ &= Merge(Merge(V_1', V_2), V)\\ &= Merge(Merge(V_1, V_2)\backslash t, V)\\ &= [V/t]Merge(V_1, V_2)\end{aligned}$$

The last equation follows from the fact that $Var(t, Merge(V_1, V_2)) = +$ when $Var(t, V_1) = +$ and $t \notin D(V_2)$.

Case 5: $V_1 = V_1', t^-$ **and** $V_2 = V_2', t^-$

In this case,

$$\begin{aligned}[V/t]V_1 &= Merge(V_1', \overline{V})\\ [V/t]V_2 &= Merge(V_2', \overline{V})\end{aligned}$$

so

$$\begin{aligned}Merge([V/t]V_1, [V/t]V_2) &= Merge(Merge(V_1', \overline{V}), Merge(V_2', \overline{V}))\\ &= Merge(V_1', V_2', \overline{V}, \overline{V})\\ &= Merge(Merge(V_1', V_2'), \overline{V})\\ &= Merge(Merge(V_1, V_2)\backslash t, \overline{V})\\ &= [V/t]Merge(V_1, V_2)\end{aligned}$$

The last equation follows from the fact that $Var(t, Merge(V_1, V_2)) = -$ when $Var(t, V_1) = -$ and $Var(t, V_2) = -$.

Case 6: $V_1 = V_1', t^-$ **and** $V_2 = V_2', t^o$

In this case,

$$\begin{aligned}[V/t]V_1 &= Merge(V_1', \overline{V})\\ [V/t]V_2 &= Merge(V_2', Invar(V))\end{aligned}$$

so

$$\begin{aligned}
Merge([V/t]V_1, [V/t]V_2) &= Merge(Merge(V_1', \overline{V}), Merge(V_2', Invar(V))) \\
&= Merge(V_1', V_2', V, \overline{V}, Invar(V)) \\
&= Merge(Merge(V_1', V_2'), Merge(\overline{V}, Invar(V))) \\
&= Merge(Merge(V_1', V_2'), Invar(V)) \\
&= Merge(Merge(V_1, V_2)\backslash t, Invar(V)) \\
&= [V/t]Merge(V_1, V_2)
\end{aligned}$$

The last equation follows from the fact that $Var(t, Merge(V_1, V_2)) = o$ when $Var(t, V_1) = -$ and $Var(t, V_2) = o$.

Case 7: $V_1 = V_1', t^-$ **and** $t \notin D(V_2)$

Here

$$\begin{aligned}
[V/t]V_1 &= Merge(V_1', \overline{V}) \\
[V/t]V_2 &= V_2
\end{aligned}$$

so

$$\begin{aligned}
Merge([V/t]V_1, [V/t]V_2) &= Merge(Merge(V_1', \overline{V}), V_2) \\
&= Merge(V_1', V_2, \overline{V}) \\
&= Merge(Merge(V_1', V_2), \overline{V}) \\
&= Merge(Merge(V_1, V_2)\backslash t, \overline{V}) \\
&= [V/t]Merge(V_1, V_2)
\end{aligned}$$

The last equation follows from the fact that $Var(t, Merge(V_1, V_2)) = -$ when $Var(t, V_1) = -$ and $t \notin D(V_2)$.

Case 8: $V_1 = V_1', t^o$ **and** $V_2 = V_2', t^o$

In this case,

$$\begin{aligned}
[V/t]V_1 &= Merge(V_1', Invar(V)) \\
[V/t]V_2 &= Merge(V_2', Invar(V))
\end{aligned}$$

so

$$\begin{aligned}
Merge([V/t]V_1, [V/t]V_2) &= Merge(Merge(V_1', Invar(V)), Merge(V_2', Invar(V))) \\
&= Merge(V_1', V_2', Invar(V), Invar(V))
\end{aligned}$$

$$\begin{aligned} &= Merge(Merge(V_1', V_2'), Invar(V)) \\ &= Merge(Merge(V_1, V_2)\backslash t, Invar(V)) \\ &= [V/t]Merge(V_1, V_2) \end{aligned}$$

The last equation follows from the fact that $Var(t, Merge(V_1, V_2)) = o$ when $Var(t, V_1) = o$ and $Var(t, V_2) = o$.

Case 9: $V_1 = V_1', t^o$ **and** $t \notin D(V_2)$

Here

$$\begin{aligned} [V/t]V_1 &= Merge(V_1', Invar(V)) \\ [V/t]V_2 &= V_2 \end{aligned}$$

so

$$\begin{aligned} Merge([V/t]V_1, [V/t]V_2) &= Merge(Merge(V_1', Invar(V)), V_2) \\ &= Merge(V_1', V_2, Invar(V)) \\ &= Merge(Merge(V_1', V_2), Invar(V)) \\ &= Merge(Merge(V_1, V_2)\backslash t, Invar(V)) \\ &= [V/t]Merge(V_1, V_2) \end{aligned}$$

The last equation follows from the fact that $Var(t, Merge(V_1, V_2)) = o$ when $Var(t, V_1) = o$ and $t \notin D(V_2)$.

Case 10: $t \notin D(V_1)$ **and** $t \notin D(V_2)$

In this case,

$$\begin{aligned} [V/t]V_1 &= V_1 \\ [V/t]V_2 &= V_2 \end{aligned}$$

so

$$Merge([V/t]V_1, [V/t]V_2) = Merge(V_1, V_2)$$

Since $t \notin D(V_1) \cup D(V_2)$, $t \notin D(Merge(V_1, V_2))$. Hence

$$[V/t]Merge(V_1, V_2) = Merge(V_1, V_2)$$

which implies

$$Merge([V/t]V_1, [V/t]V_2) = [V/t]Merge(V_1, V_2)$$

Cases 11 to 16 These cases follow from the symmetry of $Merge$ and cases already presented. ∎

As a notational convenience, it is useful to introduce the following function.

Definition 7.1.11 (Variance Combination Function)

$$\begin{array}{rcl} f_{+}(V_1, V_2) & = & Merge(V_1, V_2) \\ f_{-}(V_1, V_2) & = & Merge(V_1, \overline{V_2}) \\ f_{o}(V_1, V_2) & = & Merge(V_1, Invar(V_2)) \\ f_{?}(V_1, V_2) & = & V_1 \end{array}$$

Lemma 7.1.12 $[V/t]f_a(V_1, V_2) = f_a([V/t]V_1, [V/t]V_2)$.

Proof The proof is a case analysis on the value of a.

Case 1: $a = ?$ Follows immediately from the definition of $f_?$.

Case 2: $a = +$ Follows from Lemma 7.1.10 and the definition of f_+.

Case 3: $a = -$ Follows from Lemmas 7.1.10 and 7.1.8, and the definition of f_-.

Case 4: $a = o$ Follows from Lemmas 7.1.10 and 7.1.7, and the definition of f_o. ∎

Lemma 7.1.13 $D([V_1/t]V_2) \subseteq (D(V_2 \backslash t)) \cup D(V_1)$.

Proof Immediate from the definition of variance substitution. ∎

Lemma 7.1.14 *If* $t \notin D(V_1)$, *then* $Merge(V_1, V_2) \backslash t = Merge(V_1, V_2 \backslash t)$

Proof

$$\begin{array}{rl} & Merge(V_1, V_2) \backslash t_1 \\ = & \{t^{GVar(t, V_1, V_2)} \mid t \in D(V_1, V_2)\} - \{t_1^{Var(t_1, Merge(V_1, V_2))}\} \\ = & \{t^{lub\{Var(t, V_1), Var(t, V_2)\}} \mid t \in D(V_1) \cup D(V_2)\} - \{t_1^{Var(t_1, Merge(V_1, V_2))}\} \\ = & \{t^{lub\{Var(t, V_1), Var(t, V_2)\}} \mid t \in (D(V_1) \cup D(V_2) - \{t_1\})\} \cup \{t_1^{lub\{Var(t_1, V_1), Var(t_1, V_2\}}\} \\ & -\{t_1^{Var(t_1, Merge(V_1, V_2))}\} \end{array}$$

Since $Var(t_1, Merge(V_1, V_2)) = lub\{Var(t_1, V_1), Var(t_1, V_2)\}$, we get

$$\begin{array}{rl} = & \{t^{lub\{Var(t, V_1), Var(t, V_2)\}} \mid t \in D(V_1) \cup (D(V_2) - \{t_1\})\} \\ = & \{t^{lub\{Var(t, V_1), Var(t, V_2)\}} \mid t \in D(V_1) \cup D(V_2 \backslash t_1)\} \\ = & \{t^{lub\{Var(t, V_1), Var(t, V_2)\}} \mid t \in D(V_1, V_2 \backslash t_1)\} \end{array}$$

Since $t_1 \notin D(V_1, V_2 \backslash t_1)$, for all $t \in D(V_1, V_2 \backslash t_1), Var(t, V_2) = Var(t, V_2 \backslash t_1)$. Hence, we get

$$\begin{array}{rl} = & \{t^{lub\{Var(t, V_1), Var(t, V_2 \backslash t_1)\}} \mid t \in D(V_1, V_2 \backslash t_1)\} \\ = & \{t^{GVar(t, V_1, V_2 \backslash t_1)} \mid t \in D(V_1, V_2 \backslash t_1)\} \\ = & Merge(V_1, V_2 \backslash t_1) \end{array}$$

∎

Lemma 7.1.15 *If $t_1 \notin D(V) \cup \{t\}$, then*

$$[V/t](V_1 \backslash t_1) = ([V/t]V_1)\backslash t_1$$

Proof

Case 1: $t_1 \notin D(V_1)$

By Lemma 7.1.13, the fact that $t_1 \notin D(V) \cup D(V_1)$ implies that $t_1 \notin D([V/t]V_1)$. Hence $([V/t]V_1)\backslash t_1 = [V/t]V_1$. Since $t_1 \notin D(V_1)$, $V_1\backslash t_1 = V_1$, so $[V/t](V_1\backslash t_1) = [V/t]V_1$. Thus $([V/t]V_1)\backslash t_1 = [V/t](V_1\backslash t_1)$

Case 2: $t_1 \in D(V_1)$

$$\begin{aligned}
[V/t](V_1\backslash t_1) &= \begin{cases} Merge(V_1'', V) & \text{if } V_1\backslash t_1 = V_1'', t^+ \\ Merge(V_1'', \overline{V}) & \text{if } V_1\backslash t_1 = V_1'', t^- \\ Merge(V_1'', Invar(V)) & \text{if } V_1\backslash t_1 = V_1'', t^o \\ V_1\backslash t_1 & \text{if } t \notin D(V_1\backslash t_1) \end{cases} \\
&= \begin{cases} Merge(V_1'', V) & \text{if } V_1 = V_1'', t_1^{Var(t_1,V_1)}, t^+ \\ Merge(V_1'', \overline{V}) & \text{if } V_1 = V_1'', t_1^{Var(t_1,V_1)}, t^- \\ Merge(V_1'', Invar(V)) & \text{if } V_1 = V_1'', t_1^{Var(t_1,V_1)}, t^o \\ V_1\backslash t_1 & \text{if } t \notin D(V_1) \end{cases}
\end{aligned}$$

If we let $V_1' = V_1'', t_1^{Var(t_1,V_1)}$, then $Var(t, V_1') = Var(t_1, V_1)$ and $V_1'' = V_1'\backslash t_1$. Hence

$$\begin{aligned}
[V/t](V_1\backslash t_1) &= \begin{cases} Merge(V_1'\backslash t_1, V) & \text{if } V_1 = V_1', t^+ \\ Merge(V_1'\backslash t_1, \overline{V}) & \text{if } V_1 = V_1', t^- \\ Merge(V_1'\backslash t_1, Invar(V)) & \text{if } V_1 = V_1', t^o \\ V_1\backslash t_1 & \text{if } t \notin D(V_1) \end{cases} \\
&= \begin{cases} Merge(V_1', V)\backslash t_1 & \text{if } V_1 = V_1', t^+ \\ Merge(V_1', \overline{V})\backslash t_1 & \text{if } V_1 = V_1', t^- \\ Merge(V_1', Invar(V))\backslash t_1 & \text{if } V_1 = V_1', t^o \\ V_1\backslash t_1 & \text{if } t \notin D(V_1) \end{cases} \\
&= ([V/t]V_1)\backslash t_1
\end{aligned}$$

■

Lemma 7.1.16 *If $t_1 \notin D(V) \cup \{t\}$, then*

$$Invar(([V/t]V_1)\backslash t_1) = [V/t]Invar(V_1\backslash t_1)$$

Proof By Lemma 7.1.15,

$$([V/t]V_1)\backslash t_1 = [V/t](V_1\backslash t_1)$$

Hence,

$$Invar(([V/t]V_1)\backslash t_1) = Invar([V/t](V_1\backslash t_1))$$

By Lemma 7.1.9,

$$Invar([V/t](V_1\backslash t_1)) = [V/t]Invar(V_1\backslash t_1)$$

Thus,

$$Invar(([V/t]V_1)\backslash t_1) = [V/t]Invar(V_1\backslash t_1)$$ ■

Lemma 7.1.17 *If $V_1 \leq V_2$, then $D(V_1) \subseteq D(V_2)$.*

Proof If $t \in D(V_1)$, then $Var(t, V_1)$ is strictly greater ?. Since $V_0 \leq V_1$, $Var(t, V_1) \leq Var(t, V_0)$. Hence, $Var(t, V_0)$ is strictly greater than ? and $t \in D(V_0)$. ■

Lemma 7.1.18 *For any variance set V, $Invar(V) \leq V$.*

Proof This inequality holds if and only if

$$\forall t. Var(t, V) \leq Var(t, Invar(V))$$

Let t be some type variable. If $t \in D(V)$, then $Var(t, Invar(V)) = o$. Since o is the greatest value in the variance ordering, the inequality $\forall t. Var(t, V) \leq Var(t, Invar(V))$ must be satisfied. If $t \notin D(V)$, then $Var(t, V) = ?$. Since ? is the least value in the variance ordering, the inequality $\forall t. Var(t, V) \leq Var(t, Invar(V))$ must again be satisfied. ■

Lemma 7.1.19 *If $V_1 \leq V_2$ then $V_1\backslash t \leq V_2\backslash t$.*

Proof If $V_1 \leq V_2$, then by definition, $\forall t'. Var(t', V_2) \leq Var(t', V_1)$. Let T denote the set of all type variables not equal to t. Then since the above inequality holds for all type variables, it must hold for all type variables in T:

$$\forall t' \in T. Var(t', V_2) \leq Var(t', V_1)$$

In this range for t', $Var(t', V_i) = Var(t', V_i\backslash t)$. Hence we have

$$\forall t' \in T. Var(t', V_2\backslash t) \leq Var(t', V_1\backslash t)$$

Since we know that $t \notin D(V_i\backslash t)$, $Var(t, V_i\backslash t) = ?$. Thus in this case we have the inequality

$$Var(t, V_2\backslash t) \leq Var(t, V_1\backslash t)$$

We may combine these last two inequalities to show that

$$\forall t'. Var(t', V_2\backslash t) \leq Var(t', V_1\backslash t)$$

Hence we have $V_1\backslash t \leq V_2\backslash t$. ■

Lemma 7.1.20 *If $V_1 \leq V_2$, then $Invar(V_1) \leq Invar(V_2)$.*

Proof If $V_1 \leq V_2$, then by definition, $\forall t.Var(t, V_2) \leq Var(t, V_1)$. Let t be some type variable. Then there are two cases to consider: either t is in the domain of $D(V_2)$ or it is not. If $t \in D(V_2)$, then $Var(t, V_2) > ?$. The fact that $Var(t, V_1) > ?$ follows from the definition of $\leq$ for variance sets. Hence $Var(t, Invar(V_2)) \leq Var(t, Invar(V_1))$ as they are both equal to o. In the other case, $Var(t, V_2) = Var(t, Invar(V_2)) = ?$, so $Var(t, Invar(V_2)) \leq Var(t, Invar(V_1))$ as $?$ is the smallest element in the ordering on variance annotations. Hence $\forall t.Var(t, Invar(V_2)) \leq Var(t, Invar(V_1))$, which is just $Invar(V_1) \leq Invar(V_2)$. ■

Lemma 7.1.21 *If $V_1 \leq V_2$, then $\overline{V_1} \leq \overline{V_2}$.*

Proof If $V_1 \leq V_2$, then by definition, $\forall t.Var(t, V_2) \leq Var(t, V_1)$. It then follows from the definition of inversion for variance sets that $\forall t.Var(t, \overline{V_2}) \leq Var(t, \overline{V_1})$. Hence we have that $\overline{V_1} \leq \overline{V_2}$. ■

Lemma 7.1.22 *If $\vec{V} \leq \vec{V'}$, where the ordering on variance vectors is the point-wise extension of the variance ordering, then $Merge(\vec{V}) \leq Merge(\vec{V'})$.*

Proof

$$\begin{array}{lll}
V_i \leq V_i' & \textit{implies} & \forall t.Var(t, V_i') \leq Var(t, V_i) \\
& \textit{implies} & \forall t.lub\{Var(t, \vec{V'})\} \leq lub\{Var(t, \vec{V})\} \\
& \textit{implies} & \forall t.Var(t, Merge(\vec{V'})) \leq Var(t, Merge(\vec{V})) \\
& \textit{implies} & Merge(\vec{V}) \leq Merge(\vec{V'})
\end{array}$$

where $Var(t, \vec{V})$ is short for $\{Var(t, V_i) \mid V_i \in \vec{V}\}$. ■

Corollary 7.1.23 $Merge(\vec{V}, V) \leq Merge(\vec{V})$.

Proof This lemma follows from Lemma 7.1.22 and the fact that for all variance sets V, $V \leq \emptyset$ and $Merge(V, \emptyset) = Merge(V)$. ■

Lemma 7.1.24 *If $V_1 \leq V_2$, then for all type variables t and variance sets V, $[V/t]V_1 \leq [V/t]V_2$.*

Proof The proof proceeds by a case analysis on how t appears in V_1 and V_2. All cases follow easily from the definition of variance substitution. ■

Lemma 7.1.25 *If $a \leq a'$, $V_1 \leq V_2$, and $V_1' \leq V_2'$, then $f_a(V_1, V_1') \leq f_{a'}(V_2, V_2')$.*

Proof

The proof is a case analysis on the value of a.

Case 1: $a = ?$

Since $a' \leq a$, the fact that $?$ is the least element in the variance ordering means that $a' = ?$. Then since $f_?(V_i, V_i') = V_i$, the fact that $V_1 \leq V_2$ implies that $f_?(V_1, V_1') \leq f_?(V_2, V_2')$.

Case 2: $a = -$

The fact that $a' \leq a$ implies that a' must be either $?$ or $-$. Then it follows that

$$\begin{aligned} f_a(V_1, V_1') &= Merge(V_1, \overline{V_1'}) \\ f_{a'}(V_2, V_2') &= \begin{cases} Merge(V_2, \overline{V_2'}) & \text{if } a' = - \\ V_2 & \text{if } a' = ? \end{cases} \end{aligned}$$

Lemmas 7.1.22 and 7.1.21 then reveals that $Merge(V_1, \overline{V_1'}) \leq Merge(V_2, \overline{V_2'})$. By Lemma 7.1.23, we get the equation $Merge(V_1, \overline{V_1'}) \leq V_2$. Hence $f_-(V_1, V_1') \leq f_{a'}(V_2, V_2')$

Case 3: $a = +$ This case is dual to Case 2.

Case 4: $a = o$ Here

$$\begin{aligned} f_o(V_1, V_1') &= Merge(V_1, Invar(V_1')) \\ f_{a'}(V_2, V_2') &= \begin{cases} Merge(V_2, Invar(V_2')) & \text{if } a' = o \\ Merge(V_2, V_2') & \text{if } a' = + \\ Merge(V_2, \overline{V_2'}) & \text{if } a' = - \\ V_2 & \text{if } a' = ? \end{cases} \end{aligned}$$

Lemma 7.1.22 implies that $Merge(V_1, Invar(V_1')) \leq Merge(V_2, Invar(V_2'))$ (Case $a' = o$).

Lemmas 7.1.18 and 7.1.22 imply that $Merge(V_1, Invar(V_1')) \leq Merge(V_2, V_2')$ (Case $a' = +$).

Lemmas 7.1.7, 7.1.18, and 7.1.22 imply that $Merge(V_1, Invar(V_1')) \leq Merge(V_2, \overline{V_2'})$ (Case $a' = -$).

Lemma 7.1.23 and Observation 7.1.1 imply that $Merge(V_1, Invar(V_1')) \leq V_2$ (Case $a' = ?$). ■

Lemma 7.1.26 $[V_1/t]V_2 = f_a(V_2 \backslash t, V_1)$, *where* $a = Var(t, V_2)$.

Proof The proof is a case analysis on the value of a. Each case follows by routine calculation. ■

7.2 Context Lemmas

Lemma 7.2.1 (Well Formed Context) *If the judgment* $, \vdash A$ *is derivable, then the judgment* $, \vdash *$ *is also derivable, and in fewer steps if* A *is not* $*$*.*

Proof Proof by induction on the derivation of $, \vdash *$. ■

The (*weakening*) rule in the proof system introduces extraneous judgment derivations, which unnecessarily complicate derivation analysis. We therefore restrict our attention to derivations that do not contain any occurrences of (*weakening*).

Definition 7.2.2 (Weakening-Free Derivation) *A* $\vdash_W$*-derivation is a derivation that does not contain any occurrences of the typing rule* (*weakening*)*.*

The following two lemmas establish that all judgments derivable in the full system are derivable in the restricted one.

Lemma 7.2.3 (Derived Rule Weakening) *If the judgments* $,_1,,_2 \vdash_W A$ *and* $,_1, a,,_2 \vdash_W *$ *are both derivable, then* $,_1, a,,_2 \vdash_W A$ *is also derivable.*

Proof Proof by induction on the derivation of $,_1,,_2 \vdash A$. ■

A consequence of this lemma is that we may treat (*weakening*) as a derived rule.

Lemma 7.2.4 (Weakening-Free Judgments) *If* $, \vdash A$ *is derivable, then* $, \vdash_W A$ *is also derivable.*

Proof Proof is by induction on the derivation of $, \vdash A$. All cases follow immediately from the inductive hypothesis except for (*weakening*), which follows from Lemma 7.2.3. ■
Future analyses of derivations will consider only $\vdash_W$ derivations, since removing (*weakening*) simplifies later proofs.

Lemma 7.2.5 (Free Variables Appear in Contexts) *If the judgment* $, \vdash_W A$ *is derivable, then* $FV(A) \subseteq dom(,)$*. Furthermore,*

- *if* $, \equiv ,_1, t\colon \{t^+\},,_2$, *then* $t \notin dom(,_1,,_2)$,
- *if* $, \equiv ,_1, r <:_w R :: S \to (M; V),,_2$, *then* $FV(R) \cup D(V) \subseteq dom(,_1)$ *and* $r \notin dom(,_1,,_2)$,
- *if* $, \equiv ,_1, x\colon \tau,,_2$ *then* $FV(\tau) \subseteq dom(,_1)$ *and* $x \notin dom(,_1,,_2)$.

Proof The proof is by induction on the derivation of $, \vdash A$. The equality rules follow from the fact that β-reduction does not introduce free variables. The cases for (*type arrow*), (*exist*), (*row fn app cov*), (*row fn app contra*), (*row fn app inv*), and (*row ext*) follow from the fact that $D(Merge(V_1, V_2)) = D(V_1) \cup D(V_2)$. The other cases follow immediately from the inductive hypothesis. ■

7.3 Type Normal Forms

The equality rules in the proof system, namely, (*row* β), (*type* β), (*type eq*), ($<:$ β *right*), and ($<:$ β *left*), introduce many non-essential judgment derivations, which unnecessarily complicate derivation analysis. We therefore restrict our attention to derivations of the following form.

Definition 7.3.1 (Normal Form Derivations) *A $\vdash_N$-derivation is a $\vdash_W$-derivation in which the only appearance of an equality rule is as:*

1. *($<:$ β right) immediately following an occurrence of a ($<:$ app) rule where the left hand row function is a row variable.*
2. *(type eq) immediately before an occurrence of ($\exists <:$ intro)*

Although not all judgments derivable in the full system are derivable by $\vdash_N$-derivations, we will see below that all judgments whose row and type expressions are in a particular form, which we will call τnf (for type normal form), are derivable via $\vdash_N$-derivations. Since every expression that has a type at all will have a type in τnf, we may prove soundness using only $\vdash_N$-derivations.

Definition 7.3.2 (Type Normal Form) *The τnf of a row or type expression is its normal form with respect to β-reduction, applied to the row function application redexes within it. Because of our existential types, we need to extend this definition to include term expressions as well. In detail,*

$$
\begin{aligned}
&\tau nf(t) = t\\
&\tau nf(\tau_1 \to \tau_2) = \tau nf(\tau_1) \to \tau nf(\tau_2)\\
&\tau nf(\mathbf{probj}\, t\mathbf{.}R) = \mathbf{probj}\, t\mathbf{.}\tau nf(R)\\
&\tau nf(\exists(r <:_w R :: \kappa)\tau) = \exists(r <:_w \tau nf(R) :: \kappa)\tau nf(\tau)\\
&\tau nf(r) = r\\
&\tau nf(\langle\!\langle\rangle\!\rangle) = \langle\!\langle\rangle\!\rangle\\
&\tau nf(\langle\!\langle R \,|\, m : \tau\rangle\!\rangle) = \langle\!\langle \tau nf(R) \,|\, m : \tau nf(\tau)\rangle\!\rangle\\
&\tau nf(\lambda t.R) = \lambda t.\tau nf(R)\\
&\tau nf(R\tau) = \begin{cases} \tau nf([\tau nf(\tau)/t]R_1) & \textit{if } \tau nf(R) \textit{ is of the form } \lambda t.\, R_1\\ \tau nf(R)\, \tau nf(\tau) & \textit{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&\tau nf(x) = x\\
&\tau nf(c) = c\\
&\tau nf(\lambda x.\, e) = \lambda x.\, e\\
&\tau nf(e_1 e_2) = e_1 e_2\\
&\tau nf(\langle\rangle) = \langle\rangle\\
&\tau nf(e \Leftarrow m) = e \Leftarrow m
\end{aligned}
$$

$$\begin{aligned}
&\tau nf(\langle e_1 \leftarrow\!\circ\, m = e_2\rangle) = \langle e_1 \leftarrow\!\circ\, m = e_2\rangle\\
&\tau nf(\{\!|r <:_w R :: \kappa = R',\ e|\!\}) = \{\!|r <:_w \tau nf(R) :: \kappa\ = \tau nf(R'),\ e|\!\}\\
&\tau nf(\mathit{Abstype}\ r <:_w R :: \kappa\ \mathit{with}\ x : \tau\ \mathit{is}\ e_1\ \mathit{in}\ e_2) =\\
&\qquad \mathit{Abstype}\ r <:_w \tau nf(R) :: \kappa\ \mathit{with}\ x : \tau nf(\tau)\ \mathit{is}\ e_1\ \mathit{in}\ e_2
\end{aligned}$$

This notion is well-defined since the row and type portion of our calculus is strongly normalizing and confluent. We prove this fact by giving a translation function tr from type expressions τ and row expressions R into $\lambda^{\rightarrow}(\Sigma)$, where $\lambda^{\rightarrow}(\Sigma)$ denotes the typed lambda calculus with function types over signature Σ, defined as follows.

Definition 7.3.3 ($\lambda^{\rightarrow}(\Sigma)$)

$$\begin{array}{lcl}
\mathit{Type\ Constants} & : & typ, row\\
\mathit{Term\ Constants} & : & \mathtt{ar}\colon typ \rightarrow typ \rightarrow typ\\
 & & \mathtt{pr}\colon (typ \rightarrow row) \rightarrow typ\\
 & & \mathtt{ob}\colon (typ \rightarrow row) \rightarrow typ\\
 & & \mathtt{er}\colon row\\
 & & \mathtt{br_m}\colon row \rightarrow typ \rightarrow row\\
 & & \ \mathit{for\ each\ method\ name}\ m.
\end{array}$$

Let the variables of $\lambda^{\rightarrow}(\Sigma)$ contain all of the row and type variables of our calculus. Then define the translation tr as follows:

Definition 7.3.4 (Translation Function)

$$\begin{aligned}
&tr(t) = t\\
&tr(\tau_1 \rightarrow \tau_2) = \mathtt{ar}\ tr(\tau_1)\ tr(\tau_2)\\
&tr(\mathbf{pro}\, t.R) = \mathtt{pr}\,(\lambda t\colon typ.tr(R))\\
&tr(\mathbf{obj}\, t.R) = \mathtt{ob}\,(\lambda t\colon typ.tr(R))\\
&tr(r) = r\\
&tr(\langle\!\langle\rangle\!\rangle) = \mathtt{er}\\
&tr(\langle\!\langle R \,|\, m : \tau\rangle\!\rangle) = \mathtt{br_m}\ tr(R)\ tr(\tau)\\
&tr(\lambda t.R) = \lambda t\colon typ.tr(R)\\
&tr(R\tau) = tr(R)\ tr(\tau)
\end{aligned}$$

We extend tr to the kinds and contexts of our system:

Definition 7.3.5 (Extended Translation Function)

$$\begin{aligned}
&tr(V) = typ\\
&tr((M;\, V)) = row\\
&tr(S \to (M;\, V)) = typ \to row\\
\\
&tr(\epsilon) = \emptyset\\
&tr(,\ , x : \tau) = tr(,\ )\\
&tr(,\ , t : \{t^+\}) = tr(,\ ) \cup \{tr(t) : tr(\{t^+\})\}\\
&tr(,\ , r <:_w R :: \kappa) = tr(,\ ) \cup \{tr(r) :: tr(\kappa)\}
\end{aligned}$$

Note that tr preserves both bound and free variables of expressions, i.e., $BV(U) = BV(tr(U))$ and $FV(U) = FV(tr(U))$ for all row and type expressions U. Furthermore, if $U_1 =_\alpha U_2$, then $tr(U_1) =_\alpha tr(U_2)$ under the same renaming of bound variables.

To show that strong normalization for our system follows from strong normalization for $\lambda^{\to}(\Sigma)$, we need to establish two properties of tr. First, we need to show that the translations of any two terms related via $\to_\beta$ in our system are related via $\to_\beta$ in $\lambda^{\to}(\Sigma)$ and second, that the translation of every well-kinded term in our system is a well-typed $\lambda^{\to}(\Sigma)$ term. Lemma 7.3.6 proves the first of these properties.

Lemma 7.3.6 (Translation Preserves β-Reduction) *If $U_1 \to_\beta U_2$, then $tr(U_1) \to_\beta tr(U_2)$, where U_1 and U_2 are either both row R or both type τ expression in our system.*

Proof The proof of Lemma 7.3.6 is by induction on the structure of U_1. Each inductive case is a case analysis of the possible forms of U_2. The only case which does not follow routinely is when $U_1 = (\lambda t.R)\tau$ and $U_2 = [\tau/t]R$. This case follows from the subsidiary lemma that $[tr(\tau)/t]tr(U) = tr([\tau/t]U)$ for all type and row expressions U, a fact which is proved by induction on the structure of U. ■

Lemma 7.3.7 establishes that tr produces typeable $\lambda^{\to}(\Sigma)$ terms.

Lemma 7.3.7 (Well-Typed Terms Translate to Well-Typed Terms) *If we may derive the judgment $,\ \vdash_W U : -\gamma$ in our system, where $U : -\gamma$ is either $\tau : V$ or $R :: \kappa$, then $tr(,\ ) \rhd tr(U) : tr(\gamma)$ is derivable in $\lambda^{\to}(\Sigma)$. We use $\rhd$ to distinguish $\lambda^{\to}(\Sigma)$-derivations from derivations in our system.*

Proof The proof of Lemma 7.3.7 is by induction on the derivation of $,\ \vdash_W U : -\gamma$. The cases for $(row\ \beta)$ and $(type\ \beta)$ require Lemma 7.3.6 and Subject Reduction for $\lambda^{\to}(\Sigma)$. ■

Lemma 7.3.8 (Strong Normalization for Row and Type Portion) *If $,\ \vdash_W U : -\gamma$ is derivable, then there is no infinite sequence of $\to_\beta$ reductions from U.*

Proof This lemma follows from Lemmas 7.3.6 and 7.3.7 and the fact that $\tau nf(\exists(r <:_w R :: \kappa)\tau) = \exists(r <:_w \tau nf(R) :: \kappa)\tau nf(\tau)$. ■

The following lemma is crucial to showing that confluence for $\lambda^{\rightarrow}(\Sigma)$ implies confluence for the row and type portion of our calculus.

Lemma 7.3.9 (Inverse Translation) *If $tr(U) \rightarrow_\beta W$, then there is a unique expression U' such that $U \rightarrow_\beta U'$ and $tr(U') = W$, where U is either a row R or type τ and W is an expression of $\lambda^{\rightarrow}(\Sigma)$.*

Proof The proof of Lemma 7.3.9 is by induction on the structure of U. It is similar in outline to the proof of Lemma 7.3.6. ■

The confluence of the row and type portion of our system now follows from Lemma 7.3.9 and the fact that all redexes in $\exists(r <:_w R :: \kappa)\tau$ occur in R or in τ.

Lemma 7.3.10 (Confluence for Row and Type Portion) *If $, \vdash U_1 : -\gamma$ is derivable and $U_1 \twoheadrightarrow_\beta U_2$ and $U_1 \twoheadrightarrow_\beta U_3$, then there exists a U_4 such that $U_2 \twoheadrightarrow_\beta U_4$ and $U_3 \twoheadrightarrow_\beta U_4$.*

Since each row and type expression has a unique normal form, the τnf of row and type expressions is a well-defined notion. Because any term expression that has a type has a type in normal form, we may restrict our attention to types and rows in τnf. To this end, we need to extend the definition of τnf to contexts. The τnf of a context , is the context listing the τnf's of the elements of , .

To prove that any judgment derivable in the full system has a related, τnf judgment derivable in $\vdash_N$, we need to establish a number of subsidiary lemmas, which appear in the following few sections. The lemma in question then appears in Section 7.3.

7.4 Remaining Context Lemmas

The following two contexts lemmas are placed here because the second depends on the form of $\vdash_N$ derivations. In particular, the (*row eq*) and (*type eq*) cases cause problems because β-redexes may contain more free row variables than their reducts.

Lemma 7.4.1 (Extraneous Expression Variables) *If the judgment $, , x : \tau, ,' \vdash_N B$ is derivable, where $B \equiv * \,|\, \tau : V \,|\, R :: \kappa$, then $, , ,' \vdash_N B$ is derivable as well.*

Proof The proof is by induction on the derivation of $, , x : \tau, ,' \vdash_N B$. The (*exp var*) case depends on Lemma 7.2.1. All other cases follow immediately from the inductive hypothesis. ■

Lemma 7.4.2 (Extraneous Row Variables) *If the judgment $, , r <:_w R :: \kappa, ,' \vdash_N B$ is derivable, where $B \equiv * \,|\, \tau : V \,|\, R :: \kappa$ and $r \notin FV(,') \cup FV(B)$ then $, , ,' \vdash_N B$ is also derivable.*

Proof The proof is by induction on the derivation of $, , r <:_w R :: \kappa, ,' \vdash_N B$. All cases follow immediately from the inductive hypothesis. ■

7.5 Type Substitution Lemmas

In this section, we present various type substitution lemmas, including lemmas that describe when substituting subtypes for a type variable will produce a subtype. Such lemmas are crucial for proving type soundness because of the rich subtyping supported by the system.

In the following sections, we use meta-variable U to represent either a row R or a type τ. We use the meta-judgment $, \vdash_N U : -\gamma$ to represent judgments of the form $, \vdash_N \tau : V$ and $, \vdash_N R :: \kappa$. Similarly, we use the meta-judgment $, \vdash_N U_1 <:_{(B)} U_2$ represents the judgments $, \vdash_N \tau_1 <: \tau_2$ and $, \vdash_N R_1 <:_B R_2$. In the following lemma, the notation $[\tau : V/t], '$ denotes substituting type τ for type variable t in type and row expressions in $, '$, and variance set V for t in variance sets in $, '$.

Lemma 7.5.1 (Type Substitution) *If the judgments* $, , t : \{t^+\}, , ' \vdash_N A$ *and* $, \vdash \tau : V$ *are both derivable, then*

- *if* $A \equiv *$, *then the judgment* $, , [\tau : V/t], ' \vdash_N *$ *is derivable as well.*
- *if* $A \equiv R :: \kappa$, *then the judgment* $, , [\tau : V/t], ' \vdash_N [\tau/t]R :: [V/t]\kappa$ *is derivable as well.*
- *if* $A \equiv \tau_1 : V_1$, *then the judgment* $, , [\tau : V/t], ' \vdash_N [\tau/t]\tau_1 : [V/t]V_1$ *is derivable as well.*
- *if* $A \equiv U_1 <:_{(B)} U_2$, *then the judgment* $, , [\tau : V/t], ' \vdash_N [\tau/t]U_1 <:_B [\tau/t]U_2$ *is derivable as well.*

Proof The proof is by induction on the derivation of $, , t : \{t^+\}, , ' \vdash_N A$. The type projection case (*type proj*) requires a case analysis on whether or not t is the projected variable. The subtype equality rule ($<: \beta$ *right*) requires the substitution property that if $t_2 \notin FV(\tau) \cup \{t\}$, then

$$[\,[\tau\,/\,t]\tau_2\,/\,t_2]\,([\tau/\,t]R_2) = [\tau\,/t]([\,\tau_2\,/\,t_2]R_2)$$

The remaining cases either follow routinely from the inductive hypothesis or are similar to the cases for (*type arrow*) and (*cov object*), presented below in full detail.

(*type arrow*)

In this case, we start with two assumptions, labeled A1 and A2:

A1 $, , t : \{t^+\}, , ' \vdash_N \tau_1 \to \tau_2 : Merge(\overline{V_1}, V_2)$

A2 $, \vdash_N \tau : V$

By our case analysis, we know that A1 was derived via (*type arrow*). Hence we must have previously derived:

RH1 $, , t : \{t^+\}, , ' \vdash_N \tau_1 : V_1$

RH2 $, , t : \{t^+\}, , ' \vdash_N \tau_2 : V_2$

Applying the inductive hypothesis to RH1 and RH2, respectively, produces:

C1 $, , [\tau : V/t], ' \vdash_N [\tau/t]\tau_1 : [V/t]V_1$

C2 $, , [\tau : V/t], ' \vdash_N [\tau/t]\tau_2 : [V/t]V_2$

From C1 and C2, we may derive via (*type arrow*)

C3 $, , [\tau : V/t], ' \vdash_N [\tau/t]\tau_1 \rightarrow [\tau/t]\tau_2 : Merge(\overline{[V/t]V_1}, [V/t]V_2)$

Applying Lemmas 7.1.8 and 7.1.10 to C3 and simplifying produces:

C4 $, , [\tau : V/t], ' \vdash_N [\tau/t](\tau_1 \rightarrow \tau_2) : [V/t]Merge(\overline{V_1}, V_2)$

which is what we needed to show for (*type arrow*). The case for (*row fn app inv*) is similar but requires Lemma 7.1.9 instead of 7.1.8.

(*cov object*)
In the this case, we start with the assumptions:

A1 $, , t : \{t^+\}, , ' \vdash_N \mathbf{obj}\, t_1 . R_1 : V_1 \backslash t$

A2 $, \vdash_N \tau : V$

Without loss of generality, $t_1 \notin \{t\} \cup FV(\tau) \cup D(V)$. By the case analysis, we know that A1 was derived via (*cov object*). Hence we must have previously derived:

RH1 $, , t : \{t^+\}, , ', t_1 : \{t_1^+\} \vdash_N R_1 :: (M_1; V_1)$

RH2 $Var(t_1, V_1) \in \{+, ?\}$

By applying the inductive hypothesis to RH1 and A2, we get:

C1 $, , [\tau : V/t](, ', t_1 : \{t_1^+\}) \vdash_N [\tau/t]R_1 :: [V/t](M_1; V_1)$

Because $t_1 \neq t$, this derivation is just:

C2 $, , [\tau : V/t], ', t_1 : \{t_1^+\} \vdash_N [\tau/t]R_1 :: (M_1; [V/t]V_1)$

The fact that $t_1 \notin D(V)$ and RH2 implies

C3 $Var(t_1, [V/t]V_1) \in \{+, ?\}$

Applying (*cov object*) to C2 and C3 produces:

C4 $, , [\tau : V/t], ' \vdash_N \mathbf{obj}\, t_1 . [\tau/t]R_1 : ([V/t]V_1) \backslash t$

By Lemma 7.1.15 and the fact that $t_1 \notin FV(\tau)$, C4 is the same as:

C5 $\quad , , [\tau : V/t], ' \vdash_N [\tau/t]\, \mathbf{obj}\, t_1 \mathbf{.} R_1 : [V/t](V_1 \backslash t)$

which is the judgment we needed to derive. The cases for (*non cov object*) and (*pro*) are similar except they use Lemma 7.1.16 instead of Lemma 7.1.15. ■

In the following sections, we use the meta-judgment $, \vdash_N U_1 \cong_{(B)} U_2$ as shorthand for the two judgments $, \vdash_N \tau_1 <: \tau_2$ and $, \vdash_N \tau_2 <: \tau_1$. We also use $,_T$ to denote a context fragment containing only type variables.

Lemma 7.5.2 (Generalized Type Substitution) *If the judgments* $, , t : \{t^+\}, ,_T \vdash_N U : -\gamma$ *and* $, \vdash_N \tau_1 \cong \tau_2$ *are all derivable, then so is the judgment* $, , ,_T \vdash_N [\tau_1/t]U \cong_{(w)} [\tau_2/t]U$.

Proof The proof is by induction on the derivation of $, , t : \{t^+\}, ,_T \vdash_N U : -\gamma$. All cases follow immediately from the inductive hypothesis and Lemma 7.5.1. The case for (*pro*) relies on the ($<:$ *convert*) rule. Similarly, the (*row fn app*) cases rely on ($<:$ *app cong*), and (*row ext*) relies on ($<:$ *cong*). ■

Lemma 7.5.3 (Subtype Substitution I) *If the judgments* $, , t : \{t^+\}, ,_T \vdash_N U : -\gamma$ *and* $, \vdash_N \tau_i : V_i$ *for* $i \in \{1, 2\}$ *are all derivable, then*

- *if* $Var(t, \gamma) = ?$ *then the judgment*
  $, , ,_T \vdash_N [\tau_i/t]U <:_{(B)} [\tau_j/t]U$ *is derivable as well.*
- *if* $Var(t, \gamma) = +$ *and* $, \vdash_N \tau_1 <: \tau_2$ *then the judgment*
  $, , ,_T \vdash_N [\tau_1/t]U <:_{(w,d)} [\tau_2/t]U$ *is derivable as well.*
- *if* $Var(t, \gamma) = -$ *and* $, \vdash_N \tau_1 <: \tau_2$ *then the judgment*
  $, , ,_T \vdash_N [\tau_2/t]U <:_{(w,d)} [\tau_1/t]U$ *is derivable as well.*

where $,_T$ *is a context fragment listing only type variables.*

Proof The proof is by induction on the derivation of $, , t : \{t^+\}, ,_T \vdash_N U : -\gamma$. The case for each typing rule proceeds by a case analysis on the value of $Var(t, \gamma)$. Lemma 7.5.1 is used frequently. The "?" cases are sensitive to the form of the ($<:$ *cong*) and ($<:$ *app cong*) rules. In particular, they require these rules be written with the hypothesis $, \vdash \tau_1 \cong \tau_2$ instead of more simply, using a single type variable τ because ? variance for a type variable t indicates t does not appear in the τnf of the type τ in question; but says nothing about the variance of t in τ itself. The (*pro*) case requires the existence of the ($<:$ *convert*) rule to allow for cases of the form $U \equiv \mathbf{pro}\, t_1 \mathbf{.} r\, t$ where $, , t : \{t^+\}, ,_T \vdash_N r :: T^? \rightarrow \nu$. We present the (*row ext*) case as it is representative of the other cases.

(*row ext*)

We start with two assumptions:

A1 $, , t\colon \{t^+\}, , _T \vdash_N \langle\!\langle R \,|\, m : \tau \rangle\!\rangle :: (M;\, Merge(V_R, V_\tau))$

A2 $, \vdash_N \tau_i\colon V_i$

Because A1 was derived via (*row ext*), we must have previously derived:

RH1 $, , t\colon \{t^+\}, , _T \vdash_N R :: (M, m;\, V_R)$

RH2 $, , t\colon \{t^+\}, , _T \vdash_N \tau : V_\tau$

Now there are three cases to consider depending on the variance of t in $Merge(V_R, V_\tau)$.

Case 1:$Var(t, Merge(V_R, V_\tau)) =?$

This variance for $Merge(V_R, V_\tau)$ implies

C1 $Var(t, V_R) =?$ and

C2 $Var(t, V_\tau) =?$

Applying the inductive hypothesis once to RH1 and twice to RH2, respectively, produces:

C3 $, , , _T \vdash_N [\tau_i/t]R <:_w [\tau_j/t]R$

C4 $, , , _T \vdash_N [\tau_i/t]\tau \cong [\tau_j/t]\tau$

Applying Lemma 7.5.1 twice to A1 gives:

C5 $, , , _T \vdash_N [\tau_j/t]\langle\!\langle R \,|\, m : \tau \rangle\!\rangle :: (M;\, [V_j/t]Merge(V_R, V_\tau))$

C6 $, , , _T \vdash_N [\tau_i/t]\langle\!\langle R \,|\, m : \tau \rangle\!\rangle :: (M;\, [V_i/t]Merge(V_R, V_\tau))$

We may now apply $(<: cong)$ to C3, C4, C5, and C6 to get

C7 $, , , _T \vdash_N [\tau_i/t]\langle\!\langle R \,|\, m : \tau \rangle\!\rangle <:_w [\tau_j/t]\langle\!\langle R \,|\, m : \tau \rangle\!\rangle$

which is what we needed to establish for Case 1.

Case 2:$Var(t, Merge(V_R, V_\tau)) = +$

This variance for $Merge(V_R, V_\tau)$ implies

C1 $Var(t, V_R) \in \{+, ?\}$ and

C2 $Var(t, V_\tau) \in \{+, ?\}$

In this case, we may make the additional assumption

A3 $, \vdash_N \tau_1 <: \tau_2$

Applying the inductive hypothesis to RH1 and RH2 respectively produces:

C3 $, \, , \, , \, _T \vdash_N [\tau_1/t]R <:_{w,d} [\tau_2/t]R$

C4 $, \, , \, , \, _T \vdash_N [\tau_1/t]\tau <: [\tau_2/t]\tau$

Using Lemma 7.5.1 on A1 twice, we may derive:

C5 $, \, , \, , \, _T \vdash_N [\tau_j/t]\langle\!\langle R \,|\, m : \tau \rangle\!\rangle :: (M;\, [V_j/t]Merge(V_R, V_\tau))$

C6 $, \, , \, , \, _T \vdash_N [\tau_i/t]\langle\!\langle R \,|\, m : \tau \rangle\!\rangle :: (M;\, [V_i/t]Merge(V_R, V_\tau))$

We may then apply $(<: d)$ to C3, C4, C5, and C6 to get:

C7 $, \, , \, , \, _T \vdash_N [\tau_1/t]\langle\!\langle R \,|\, m : \tau \rangle\!\rangle <:_{w,d} [\tau_2/t]\langle\!\langle R \,|\, m : \tau \rangle\!\rangle$

which is what we needed to show for Case 2.

Case 3:$Var(t, Merge(V_R, V_\tau)) = -$

This case is dual to Case 2. ■

Lemma 7.5.4 (Subtype Substitution II) *If the judgments* $, \, , t : \{t^+\}, \, , _T \vdash_N U_1 <:_{(B)} U_2$, $, \vdash_N U_2 : -\gamma_2$, *and* $, \vdash_N \tau_i : V_i$ *for* $i \in \{1, 2\}$ *are all derivable, then*

- *if* $Var(t, \gamma_2) = ?$ *then the judgment*
  $, \, , \, , \, _T \vdash_N [\tau_i/t]U_1 <:_{(B)} [\tau_j/t]U_2$ *is derivable as well.*
- *if* $Var(t, \gamma_2) = +$ *and* $, \vdash_N \tau_1 <: \tau_2$ *then*
  $, \, , \, , \, _T \vdash_N [\tau_1/t]U_1 <:_{(w,d)} [\tau_2/t]U_2$ *is derivable as well.*
- *if* $Var(t, \gamma_2) = -$ *and* $, \vdash_N \tau_1 <: \tau_2$ *then*
  $, \, , \, , \, _T \vdash_N [\tau_2/t]U_1 <:_{(w,d)} [\tau_1/t]U_2$ *is derivable as well.*

where $, _T$ *is a context listing only type variables.*

Proof This proof is a case analysis on the variance of t in γ_2. Each case follows from Lemmas 7.5.1 and 7.5.3. We present the $Var(t, \gamma_2) = +$ case, as it is representative of the others.

Case 1: $Var(t, \gamma_2) = +$

By assumption, we have derived

A1 $, \, , t : \{t^+\}, \, , _T \vdash_N U_1 <:_{(B)} U_2$

A2 $, \, , t : \{t^+\}, \, , _T \vdash_N U_2 : -\gamma_2$

A3 $, \, , t : \{t^+\}, \, , _T \vdash_N \tau_i : V_i$ for $i \in \{1, 2\}$

A4 $, \, , t : \{t^+\}, \, , _T \vdash_N \tau_1 <: \tau_2$

We may apply Lemma 7.5.1 to A1 and A3 to produce:

C1 $, , , T \vdash_N [\tau_1/t]U_1 <:_{(B)} [\tau_1/t]U_2$

Applying Lemma 7.5.3 to A2 and A4 gives us:

C2 $, , , T \vdash_N [\tau_1/t]U_2 <:_{(w,d)} [\tau_2/t]U_2$

We may then use $(<\!:\, \mathit{trans})$ on C1 and C2 to derive:

C3 $, , , T \vdash_N [\tau_1/t]U_1 <:_{(w,d)} [\tau_2/t]U_2$

which is the judgment we needed to derive. ∎

Note the use of transitivity in the above lemma. The more straightforward approach of proving the lemma via induction on the derivation of $, \vdash_N U_1 <:_B U_2$ fails because of transitivity. In particular, with depth subtyping, the variances of the sub- and super-types are not related. Hence the inductive hypothesis is not strong enough to connect the variance of γ_2 and the variance of the intermediate row arising in the hypotheses of transitivity.

7.6 Derived Equality Lemmas

To eliminate the equality typing rules in the manner proscribed by $\vdash_N$-derivations, we must show that we can derive all the necessary equality rules in the weakened system. This section collects such lemmas.

Lemma 7.6.1 (Derived Rule Row Equality) *If the judgment* $, \vdash_N (\lambda t.\, R)\tau :: (M;\, V)$ *is derivable, then so is the judgment* $, \vdash_N [\tau/t]R :: (M;\, V)$

Proof The proof is by induction on the derivation of $, \vdash_N (\lambda t.\, R)\tau :: (M;\, V)$, to account for the (*row label*) case, which then follows immediately from the inductive hypothesis. Otherwise, in a $\vdash_N$-derivation, the judgment $, \vdash_N (\lambda t.\, R)\tau :: (M;\, V)$ must have been derived via a (*row fn app*) rule. With any of these four rules, we must have previously derived:

RH1 $, \vdash_N \lambda t.\, R :: T^a \to (M;\, V_1)$

RH2 $, \vdash_N \tau : V_2$

Furthermore, RH1 must have been derived via (*row fn abs*), so we must also have derived:

RH3 $, , t : \{t^+\} \vdash_N R :: (M';\, V_1')$

where $M \subseteq M'$, $a = Var(t, V_1)$, and $V_1 = V_1' \backslash t$. Note that a indicates which of the four (*row fn app*) rules was used to derive $, \vdash_N (\lambda t.\, R)\tau :: (M;\, V)$. Hence, we know that

$$V = \begin{cases} Merge(V_1, V_2) & \text{if } a = + \\ Merge(V_1, \overline{V_2}) & \text{if } a = - \\ Merge(V_1, Invar(V_2)) & \text{if } a = o \\ V_1 & \text{if } a = ? \end{cases}$$

Notice that the right-hand side of this equation is just $[V_2/t]V_1'$. By applying Lemma 7.5.1 to RH2 and RH3, we get

C1 $\;, \vdash_N [\tau/t]R :: (M';\, [V_2/t]V_1')$

which is just

C2 $\;, \vdash_N [\tau/t]R :: (M';\, V)$

An application of (*row label*) then produces the judgment we needed to derive. ∎

The next lemma characterizes a portion of the subtype relation; hence it conceptually belongs in Section 7.9, the subtype characterization section. However, it is needed to establish Lemmas 7.6.5 and 7.6.7 which appear in this section. Hence, it is presented earlier to preserve the property that all lemmas are proved before they are used.

Lemma 7.6.2 (Row Function Subtype Characterization) *If* $, \vdash_N \lambda t.\, R_1 <:_B R$ *is derivable, then* R *is of the form* $\lambda t.\, R_2$.

Proof The proof is by induction on the derivation of $, \vdash_N \lambda t.\, R_1 <:_B R$. Most cases are either vacuous or immediate from the inductive hypothesis. The ($<: \beta$ *right*) rule cannot occur because in a $\vdash_N$-derivation, ($<: \beta$ *right*) must occur immediately after a ($<:$ *app*) rule, none of which can produce a subtype judgment whose left-hand side has the form $\lambda t.\, R_1$. The ($<: \beta left$) case follows vacuously, as this rule cannot occur in $\vdash_N$-derivations. ∎

The following two lemmas belong in Section 7.11, the subtyping properties section. However, they are needed to establish Lemmas 7.6.5, 7.6.6, and 7.6.7. As a result, they appear in this section.

Lemma 7.6.3 (Subtyping Implies Well Formed) *If the judgment* $, \vdash_N U_1 <:_{(B)} U_2$ *is derivable, then so are the judgments* $, \vdash_N U_i : -\gamma_i$ *for some* γ_i*'s,* $i \in \{1, 2\}$. *Furthermore,* γ_1 *and* γ_2 *are kind expressions of the same form.*

Proof The proof is by induction on the derivation of $, \vdash_N U_1 <:_{(B)} U_2$. Most of the cases follow immediately from the inductive hypothesis. The (*row proj bound*) case relies on Lemma 7.2.1 and the ($<: \beta$ *right*) case uses Lemma 7.6.1. ∎

Lemma 7.6.4 (Kind of Rows Related Via Subtyping Via ($<: \beta$ *right*)) *If the judgment* $, \vdash_N R_1 <:_B R_2$ *is derived via* ($<: \beta$ *right*), *then the judgments* $, \vdash_N R_i :: \nu_i$, $i \in \{1, 2\}$ *are derivable as well, for some flat row kinds* ν_1 *and* ν_2.

Proof Because we derived $, \vdash_N R_1 <:_B R_2$ via ($<:$ *β right*), R_2 must be of the form $([\tau_2/t]R_2')$, and we must have previously derived

RH1 $, \vdash_N R_1 <:_B (\lambda t.\, R_2')\tau_2$

By Lemma 7.6.3, we may derive both

C1 $, \vdash_N R_1 :: \kappa_1$

C2 $, \vdash_N (\lambda t.\, R_2')\tau_2 :: \kappa_2$

Furthermore, κ_1 and κ_2 must be of the same form. The structure of C2 indicates that it must have been derived from a (*row fn app*) rule. Hence κ_2 must be of the form ν_2. (and κ_1 as well). Applying Lemma 7.6.1 to C2 produces the judgment

C3 $, \vdash_N ([\tau_2/t]R_2') :: \nu_2$

which is the judgment we needed to derive. ∎

Lemma 7.6.5 (Derived Rule Application Equality) *If the judgments* $, \vdash_N \lambda t.\, R_1 <:_B \lambda t.\, R_2$ *and* $, \vdash_N \tau : V$ *are both derivable, then so is the judgment* $, \vdash_N [\tau/t]R_1 <:_B [\tau/t]R_2$.

Proof The proof is by induction on the derivation of $, \vdash_N \lambda t.\, R_1 <:_B \lambda t.\, R_2$. Most cases are vacuously true. The ($<:$ *row refl*) case follows from Lemma 7.6.1, ($<:$ λ) from Lemma 7.5.1, and ($<:$ *row trans*) from Lemma 7.6.2. The fact that the ($<:$ *β right*) case is vacuously true follows from Lemma 7.6.4. ∎

Note the use of transitivity in the following lemma.

Lemma 7.6.6 (Generalized Derived Rule Application Equality) *If we may derive each of the judgments* $, \vdash_N \lambda t.\, R_1 <:_B \lambda t.\, R_2$ *and* $, \vdash_N \tau_1 \cong \tau_2$, *then we may also derive the judgment* $, \vdash_N [\tau_1/t]R_1 <:_B [\tau_2/t]R_2$.

Proof Lemma 7.6.3 implies that we must have previously derived

C1 $, , t : \{t^+\} \vdash_N R_2 :: \nu_2$

for some kind ν_2. Via Lemma 7.5.2, we may derive

C2 $, \vdash_N [\tau_1/t]R_2 <:_w [\tau_2/t]R_2$

Applying Lemma 7.6.5 to $, \vdash_N \lambda t.\, R_1 <:_B \lambda t.\, R_2$ gives us:

C3 $, \vdash_N [\tau_1/t]R_1 <:_B [\tau_1/t]R_2$

Then we may use ($<:$ *row trans*) on C2 and C3 to produce:

C4 $, \vdash_N [\tau_1/t]R_1 <:_B [\tau_2/t]R_2$

which is the judgment we needed to derive. ■

Lemma 7.6.7 (Derived Rule Subtype Application Equality) *If* $, \vdash_N \lambda t.\, R_1 <:_B \lambda t.\, R_2$, $, \vdash_N \lambda t.\, R_2 :: T^a \rightarrow (M; V)$, *and* $, \vdash_N \tau_i : V_i$ *for* $i \in \{1, 2\}$ *are all derivable judgments, then*

- *if* $a = ?$ *then the judgments*
  $, \vdash_N [\tau_i/t]R_1 <:_B [\tau_j/t]R_2$ *are derivable as well, for* $j \in \{1, 2\}$.
- *if* $a = +$ *and* $, \vdash_N \tau_1 <: \tau_2$ *then the judgment*
  $, \vdash_N [\tau_1/t]R_1 <:_{w,d} [\tau_2/t]R_2$ *is derivable as well.*
- *if* $a = -$ *and* $, \vdash_N \tau_1 <: \tau_2$ *then the judgment*
  $, \vdash_N [\tau_2/t]R_1 <:_{w,d} [\tau_1/t]R_2$ *is derivable as well.*

Proof The proof is by induction on the derivation of $, \vdash_N \lambda t.\, R_1 <:_B \lambda t.\, R_2$. The $(<:\ row\ refl)$ case follows from Lemma 7.5.3, and $(<:\ \lambda)$ from Lemma 7.5.4. The trickiest case is the one for $(<:\ row\ trans)$, which follows from Lemmas 7.6.2, and 7.6.5, and the inductive hypothesis. The structure of the proof is similar to that of Lemma 7.5.4. ■

7.7 Normal Form Lemma

This section contains the proof that shows that any judgment derivable in our full system has a related judgment derivable via a $\vdash_N$-derivation.

Lemma 7.7.1 (Normal Form Lemma) *If the judgment* $, \vdash A$ *is derivable, where* $, \vdash A$ *denotes any judgment in the system, then the judgment* $\tau nf(,) \vdash_N \tau nf(A)$ *is derivable as well.*

Proof The proof is by induction on the derivation of $, \vdash A$. Most of the cases follow immediately from the inductive hypothesis and the definition of τnf. The $(row\ fn\ app)$ cases depend on Lemma 7.6.1. The type equality rules follow from the fact that two row or type expressions related via β-reduction have the same τnf. The $(\exists <:\ intro)$ case requires the use of *(type eq)* immediately before the use of $(\exists <:\ intro)$, which is why this possibility is allowed in $\vdash_N$-derivations. The $(<:\ app)$ rules are the most subtle. We present the $(\exists <:\ intro)$ and $(<:\ app\ cong)$ case in detail. The other $(<:\ app)$ share the same structure as $(<:\ app\ cong)$.

$(\exists <:\ intro)$

In this case, we start with the assumption that

A1 $\quad , \vdash_N \{\!|r <:_w R :: \kappa = R_1,\ e|\!\} : \exists(r <:_w R :: \kappa)\tau$

To reach this conclusion, we must have previously derived

RH1 $\quad , \vdash_N R_1 :: \kappa$

RH2 $\Gamma \vdash_N R_1 <:_w R$

RH3 $\Gamma \vdash_N e\colon [R_1/r]\tau$

Applying the inductive hypothesis to these three judgments in turn produces the judgments:

C1 $\tau nf(\Gamma) \vdash_N \tau nf(R_1) :: \kappa$

C2 $\tau nf(\Gamma) \vdash_N \tau nf(R_1) <:_w \tau nf(R)$

C3 $\tau nf(\Gamma) \vdash_N e\colon \tau nf([R_1/r]\tau)$

Notice that since e has a simple type, we are guaranteed that its τnf is just e. It follows from the definition of τnf that

C4 $\tau nf([R_1/r]\tau) \leftrightarrow_\beta [\tau nf(R_1)/r]\tau nf(\tau)$

We may thus apply (*type eq*) to C3 and C4 to produce the judgment

C5 $\tau nf(\Gamma) \vdash_N e\colon [\tau nf(R_1)/r]\tau nf(\tau)$

We may now invoke ($\exists <:$ *intro*) on C1, C2, and C5 to obtain

C6 $\tau nf(\Gamma) \vdash_N \{\!|\tau nf(r <:_w R :: \kappa) = \tau nf(R_1),\ e|\!\} : \exists(\tau nf(r <:_w R :: \kappa))\tau nf(\tau)$

which is just

C7 $\tau nf(\Gamma) \vdash_N \tau nf(\{\!|r <:_w R :: \kappa = R_1,\ e|\!\} : \exists(r <:_w R :: \kappa)\tau)$

the judgment we needed to derive.

(<: *app cong*)

In this case, we start with the assumption that we have derived the following judgment:

A1 $\Gamma \vdash_N R_1\tau_1 <:_B R_2\tau_2$

From the form of the (<: *app cong*) rule, we must have previously derived:

RH1 $\Gamma \vdash_N R_1 <:_B R_2$

RH2 $\Gamma \vdash_N R_2 :: T^a \to \nu$

RH3 $\Gamma \vdash_N \tau_1 \cong \tau_2$

Applying the inductive hypothesis to each of these judgments in turn produces the judgments:

C1 $\tau nf(\Gamma) \vdash_N \tau nf(R_1) <:_B \tau nf(R_2)$

C2 $\tau nf(\Gamma) \vdash_N \tau nf(R_2) :: T^a \to \nu$

C3 $\tau nf(\Gamma) \vdash_N \tau nf(\tau_1) \cong \tau nf(\tau_2)$

From these judgments, we may derive via (<: *app cong*) the judgment:

C4 $\tau nf(\Gamma) \vdash_N \tau nf(R_1)\tau nf(\tau_1) <:_B \tau nf(R_2)\tau nf(\tau_2)$

At this point, there are four cases to consider, depending on whether or not $\tau nf(R_1)$ and $\tau nf(R_2)$ are lambda abstractions or row function variables.

Case 1: $\tau nf(R_1)$ and $\tau nf(R_2)$ are both row variables.

Then $\tau nf(R_i)\tau nf(\tau_i)$ has the form $\tau nf(R_i\tau_i)$ for $i \in \{1, 2\}$ and C5 is the judgment we needed to show derivable.

Case 2: $\tau nf(R_1)$ and $\tau nf(R_2)$ are both lambda abstractions.

In other words, $\tau nf(R_1)$ has the form $\lambda t.\, \tau nf(R_1')$ and $\tau nf(R_2)$ has the form $\lambda t.\, \tau nf(R_2')$ for some rows R_1' and R_2'. Then C1 is just

C5 $\tau nf(\Gamma) \vdash_N \lambda t.\, \tau nf(R_1') <:_B \lambda t.\, \tau nf(R_2')$

Applying Lemma 7.6.6 to C5 and C3 produces the judgment:

C6 $\tau nf(\Gamma) \vdash_N [\tau nf(\tau_1)/t]\tau nf(R_1') <:_B [\tau nf(\tau_2)/t]\tau nf(R_2')$

Since substituting $\tau nf(\tau_2)$ for t cannot introduce any new redexes, this judgment can be written

C7 $\tau nf(\Gamma) \vdash_N \tau nf(R_1\tau_1) <:_B \tau nf(R_2\tau_2)$

the judgment we needed to derive.

Case 3: $\tau nf(R_1)$ is a row variable and $\tau nf(R_2)$ is a lambda abstraction.

In this case, $\tau nf(R_1)\tau nf(\tau_1)$ has the form $\tau nf(R_1\tau_1)$ and $\tau nf(R_2)$ has the form $\lambda t.\, R_2'$ for some row R_2', so C4 may be written

C5 $\tau nf(\Gamma) \vdash_N \tau nf(R_1\tau_1) <:_B (\lambda t.\, \tau nf(R_2'))\tau nf(\tau_2)$

Applying ($<:\ \beta$ *right*) to this judgment produces:

C6 $\tau nf(\Gamma) \vdash_N \tau nf(R_1\tau_1) <:_B [\tau nf(\tau_2)/t]\tau nf(R_2')$

which is just

C7 $\tau nf(\Gamma) \vdash_N \tau nf(R_1\tau_1) <:_B \tau nf(R_2\tau_2)$

which is the judgment we needed to derive.

Case 4: $\tau nf(R_1)$ is a lambda abstraction and $\tau nf(R_2)$ is a row function variable.

Now, $\tau nf(R_1)$ has the form $\lambda t.\, R_1'$ for some row R_1'. By applying Lemma 7.6.2, we can see that this case is impossible. ■

7.8 Properties of Kinding

The lemmas collected in the following section demonstrate properties of the kinding system. The first lemma shows that a given row (or type) has a unique variance in a given context. Intuitively, that variance reflects how type variables appear in the normal form of the row (or type).

Lemma 7.8.1 (Unique Variance) *If the judgments* $, \vdash_N U\colon -\gamma_1$ *and* $, \vdash_N U\colon -\gamma_2$ *are both derivable, where U is either a row expression R or a type expression τ, then* $Var(\gamma_1) = Var(\gamma_2)$.

Proof The proof is by induction on the structure of U. The only tricky case is when U has the form $R\,\tau$, as there are five typing rules that could have been the last in a derivation of the form $, \vdash_N R\,\tau :: (M;\, V)$, the four (*row fn app*) rules and (*row label*). A simple induction on derivations that end with $, \vdash_N R\,\tau :: (M;\, V)$ shows that such a derivation must have included an intermediate judgment of the form $, \vdash_N R\,\tau :: (M';\, V)$, where the rule used to derive this judgment was a (*row fn app*) rule. Hence, we must have derived

A1′ $, \vdash_N R\,\tau :: (M_1';\, V_1)$

A2′ $, \vdash_N R\,\tau :: (M_2';\, V_2)$

each via a (*row fn app*) rule. Regardless of which such rule we used, we must have previously derived:

RH1 $, \vdash_N R :: T^a \to (M_1';\, V_1')$

RH2 $, \vdash_N \tau\colon V_1''$

RH3 $, \vdash_N R :: T^{a'} \to (M_2';\, V_2')$

RH4 $, \vdash_N \tau\colon V_2''$

where $V_1 = f_a(V_1', V_1'')$ and $V_2 = f_{a'}(V_2', V_2'')$. Applying the induction hypothesis first to RH1 and RH3 and then to RH2 and RH4 reveals that $V_1' = V_2'$, $a = a'$, and $V_1'' = V_2''$. These equalities imply that $V_1 = V_2$, which gives us that $Var(M_1;\, V_1) = Var(M_2;\, V_2)$. ■

Lemma 7.8.2 (Grow Method Set) *If the judgments* $, \vdash_N R :: \kappa(M_1)$ *and* $, \vdash_N R :: \kappa(M_2)$ *are both derivable, where*

$$\kappa(M) \ ::= \ T^a \to (M;\, V) \ \mid \ (M;\, V)$$

for some annotation a and some variance set V, then the judgment $, \vdash_N R :: \kappa(M_1 \cup M_2)$ *is derivable as well.*

Proof The proof is by induction on the structure of R. The empty row ($\langle\!\langle\rangle\!\rangle$) and row variable ($r$) cases follow routinely from Lemma 7.2.1. The row extension ($\langle\!\langle R \,|\, m : \tau \rangle\!\rangle$), row function ($\lambda t.\, R$), and row function application ($R\,\tau$) cases follow routinely from the inductive hypothesis. ■

To prove the next lemma about the soundness of the method absence annotations, we need to be able to collect the names of the methods that actually *appear* in a given row. Hence the following definition:

Definition 7.8.3 *Given a context* $\ $, *and a row expression* R *such that the judgment* $\ , \vdash_N R :: \kappa$ *is derivable for some kind* κ, *we define the method name function* $MethNames(\ ,\ ; R)$ *as follows, by recursion on the form of* R:

$$
\begin{array}{lcl}
MethNames(\ ,\ ; \langle\!\langle\rangle\!\rangle) & = & \emptyset \\[1ex]
MethNames(\ ,\ ; \langle\!\langle R \,|\, m : \tau \rangle\!\rangle) & = & MethNames(\ ,\ ; R) \cup \{m\} \\[1ex]
MethNames(\ ,\ ; r) & = & MethNames(\ ,' ; R') \\
 & & \quad \textit{where}\ \ \equiv\ ,',\ r <:_w R' :: \kappa,\ ,'' \\[1ex]
MethNames(\ ,\ ; \lambda t.\, R) & = & MethNames(\ ,\ , t : \{t^+\}\ ; R) \\[1ex]
MethNames(\ ,\ ; R\,\tau) & = & MethNames(\ ,\ ; R)
\end{array}
$$

Lemma 7.8.4 (Soundness of Method Absence Annotations) *If we may derive the judgment* $\ , \vdash_N R :: S \to (M; V)$, *then* $MethNames(\ ,\ ; R) \cap M = \emptyset$.

Proof The proof is by induction on the derivation of $\ , \vdash_N R :: \kappa$. All cases follow routinely. ■

Lemma 7.8.5 (Type Extraction) *If* $\ , \vdash_N [\tau/t]U : -\gamma$ *and* $\ , \vdash_N \tau : V$ *are both derivable and* $t \notin$ *dom*$(\ ,\ )$, *then* $\ ,\ , t : \{t^+\} \vdash_N U : -\gamma_1$ *where* $\gamma = [V/t]\gamma_1$.

Proof The proof of this lemma is by induction on the derivation of $\ , \vdash [\tau/t]U : -\gamma$. Note that U and τ do not range over existential types, so the (*exist*) case is vacuous. Note also that U is assumed to be in normal form (as is τ) and that substituting a type in normal form for a type variable cannot introduce a new redex. Hence if $[\tau/t]U$ has a certain syntactic form and U is not t, then U must have the same syntactic form. Thus for each type rule that can produce a judgment of the form $\ , \vdash_N [\tau/t]U : V$, there are two cases to consider: either U is equal to t or it is not.

U **is** t

By Lemma 7.2.1, we can derive $\ , \vdash_N *$. Since $t \notin$ *dom*$(\ ,\ )$ by assumption, we can derive via (*type var*) the judgment $\ ,\ , t : \{t^+\} \vdash_N *$. By (*type proj*) we get $\ ,\ , t : \{t^+\} \vdash_N t : \{t^+\}$, which is the judgment we need since $[V/t]\{t^+\} = V$.

U is not t

Here we proceed by a case analysis on the last rule in the derivation of $, \vdash [\tau/t]U\colon -\gamma$. We will present the analysis for (*cov obj*) in detail, as it is representative of the difficulties encountered in considering the various typing rules.

In the (*cov obj*) case, we start with the assumptions:

A1 $, \vdash_N \mathbf{obj}\, t_1 \,\mathbf{.}\, R_1 \colon V_1 \backslash t_1$

A2 $, \vdash_N \tau \colon V$

A3 $t \notin dom(,)$

where $\mathbf{obj}\, t_1 \,\mathbf{.}\, R_1$ is of the form $[\tau/t]U$. Because t_1 is a bound variable, we may assume without loss of generality that $t_1 \notin \{t\} \cup dom(,)$. Note that U must be of the form $\mathbf{obj}\, t_1 \,\mathbf{.}\, R_1'$. Hence A1 may be written in the form:

A1′ $, \vdash_N \mathbf{obj}\, t_1 \,\mathbf{.}[\tau/t]R_1' \colon V_1 \backslash t_1$

By assumption, A1' was derived via (*cov obj*). Hence we must have previously derived

RH1 $, , t_1 : \{t_1^+\} \vdash_N [\tau/t]R_1' :: (M_1;\, V_1)$

RH2 $Var(t_1, V_1) \in \{+, ?\}$

By applying Lemma 7.2.3 to A2 and A3, we may produce the judgment

C1 $, , t_1 : \{t_1^+\} \vdash_N \tau : V$

Also, since $t \neq t_1$, A3 implies that

C2 $t \notin dom(, , t_1 : \{t_1^+\})$

We may now apply the induction hypothesis to RH1, C1, and C2 to derive

C3 $, , t_1 : \{t_1^+\}, t\colon \{t^+\} \vdash_N R_1' :: (M_1;\, V_1')$

where

C4 $V_1 = [V/t]V_1'$

By a simple induction, we may show that whenever $, , t\colon \{t^+\}, t_1 : \{t_1^+\}, ,' \vdash_N A$ is derivable, then so is $, , t_1 : \{t_1^+\}, t\colon \{t^+\}, ,' \vdash_N A$. Hence we may derive

C5 $, , t\colon \{t^+\}, t_1 : \{t_1^+\} \vdash_N R_1' :: (M_1;\, V_1')$

Since $t_1 \notin D(V)$, it follows from the definition of variance substitution that $Var(t_1, [V/t]V_1') = Var(t_1, V_1')$. Hence,

C6 $Var(t_1, V_1') = Var(t_1, V_1) \in \{+, ?\}$

We may now apply (*cov obj*) to C5 and C6 to produce

C7 $\;, t\colon \{t^+\} \vdash_N \mathbf{obj}\, t_1 \mathbf{.} R_1' \colon V_1' \backslash t_1$

To see that C7 is the judgment we needed to derive, consider the variance set we get by substituting V for t in $V_1'\backslash t_1$. Because $t_1 \notin D(V) \cup \{t\}$, Variance Lemma 7.1.16 implies that

C8 $[V/t](V_1'\backslash t_1) = ([V/t]V_1')\backslash t_1$

concluding the analysis necessary for the (*cov obj*) case.

The four (*row fn app*) rules may be handled together by appealing to Lemma 7.1.12. In general, the cases for the other typing rules proceed via a similar analysis, differing only in which variance lemmas are needed to show that the variance sets are in the necessary relationship. ∎

7.9 Subtyping Characterization Lemmas

Lemma 7.9.1 (Supertype Characterization for Type Variables) *If the judgment* $\;\vdash_N t <: \tau$ *is derivable, then* τ *is* t.

Proof The proof is by induction on the derivation of $\;\vdash_N t <: \tau$. The ($<:$ *type refl*) case follows immediately from its hypothesis. The ($<:$ *type trans*) case follows from two applications of the inductive hypothesis. ∎

Lemma 7.9.2 (Supertype Characterization for Arrow Types) *If we may derive the judgment* $\;\vdash_N \tau_1 \rightarrow \tau_2 <: \tau'$*, then* τ' *has the form* $\tau_1' \rightarrow \tau_2'$.

Proof The proof is by induction on the derivation of $\;\vdash_N \tau_1 \rightarrow \tau_2 <: \tau'$. The ($<:$ *type refl*) and ($<: \rightarrow$) cases follow immediately from their rule hypotheses. The ($<:$ *type trans*) case follows from two applications of the inductive hypothesis. ∎

Lemma 7.9.3 (Subtype Characterization for Arrow Types) *If we may derive the judgment* $\;\vdash_N \tau' <: \tau_1 \rightarrow \tau_2$*, then* τ' *has the form* $\tau_1' \rightarrow \tau_2'$.

Proof The proof is by induction on the derivation of $\;\vdash_N \tau' <: \tau_1 \rightarrow \tau_2$ The ($<:$ *type refl*) and ($<: \rightarrow$) cases follow immediately from their rule hypotheses. The ($<:$ *type trans*) case follows from two applications of the inductive hypothesis. ∎

Lemma 7.9.4 (Subtype Characterization for Pro Types) *If the judgment* $\;\vdash_N \tau <: \mathbf{pro}\, t \mathbf{.} R_1$ *is derivable, then* τ *must have the form* $\mathbf{pro}\, t \mathbf{.} R_2$

Proof The proof is by induction on the derivation of $, \vdash_N \tau <: \mathbf{pro}\, t\mathbf{.}R_1$. The $(<: type\ refl)$ and $(<: convert)$ cases follow immediately from their rule hypotheses. The $(<: type\ trans)$ case follows from two applications of the inductive hypothesis. ■

Lemma 7.9.5 (Supertype Characterization for Obj Types) *If we may derive the judgment* $, \vdash_N \mathbf{obj}\, t\mathbf{.}R_1 <: \tau$*, then* τ *must have the form* $\mathbf{obj}\, t\mathbf{.}R_2$

Proof The proof is by induction on the derivation of $, \vdash_N \mathbf{obj}\, t\mathbf{.}R_1 <: \tau$. The $(<: type\ refl)$ and $(<: obj)$ cases follow immediately from their rule hypotheses. The $(<: type\ trans)$ case follows from two applications of the inductive hypothesis. ■

Lemma 7.9.6 (Empty Row is Top) *If the judgment* $, \vdash_N R :: \kappa$ *is derivable, then so is the judgment* $, \vdash_N R <:_B ER$*, where* ER *is* $\langle\!\langle\rangle\!\rangle$ *if* κ *is of the form* ν*. Otherwise,* ER *has the form* $\lambda t.\langle\!\langle\rangle\!\rangle$*.*

Proof The proof is by induction on the derivation of $, \vdash_N R :: \kappa$. Lemma 7.6.1 is needed for the $(row\ fn\ app)$ cases. ■

Lemma 7.9.7 (Empty Row Supertype Characterization) *If the judgment* $, \vdash_N \langle\!\langle\rangle\!\rangle <:_B R$ *is derivable,* R *must have the form* $\langle\!\langle\rangle\!\rangle$*.*

Proof The proof is by induction on the derivation of $, \vdash_N \langle\!\langle\rangle\!\rangle <:_B R$. The $(<: row\ refl)$ case is immediate, and the $(<: row\ trans)$ case follows immediately from two applications of the inductive hypothesis. The only other rule that could possibly produce a judgment of the form $, \vdash_N \langle\!\langle\rangle\!\rangle <:_B R$ is $(<: \beta\ right)$. However, the definition of $\vdash_N$-derivations rules this case out, as it limits the use of $(<: \beta\ right)$ to immediately after applications. Since the $\langle\!\langle\rangle\!\rangle$ is not an application, we could not have used $(<: \beta\ right)$ to derive $, \vdash_N \langle\!\langle\rangle\!\rangle <:_B R$. ■

Lemma 7.9.8 (Subtype Characterization For Small Rows) *If we may derive the judgment* $, \vdash_N R_1 = \langle\!\langle \langle\!\langle\rangle\!\rangle \,|\, m : \tau \rangle\!\rangle <:_B R_2$*, then* R_2 *has the form* $\langle\!\langle\rangle\!\rangle$ *or* $\langle\!\langle \langle\!\langle\rangle\!\rangle \,|\, m : \tau' \rangle\!\rangle$*, for some type* τ' *such that the judgment* $, \vdash_N \tau <: \tau'$ *is derivable.*

Proof The proof is by induction on the derivation of $, \vdash_N R_1 = \langle\!\langle \langle\!\langle\rangle\!\rangle \,|\, m : \tau \rangle\!\rangle <:_B R_2$ The $(<: row\ trans)$ case requires Lemma 7.9.7. The $(<: \beta\ right)$ case is vacuously true, because the only place the $(<: \beta\ right)$ rule may appear in a $\vdash_N$ derivation is immediately after a $(<: app)$ rule, in which case R_1 must be of the form $R\,\tau$ for some row R and type τ. ■

The following is a technical lemma is used solely to prove the $(<: \beta\ right)$ case of the next lemma, Lemma 7.9.10.

Lemma 7.9.9 (Substitution Preserves Method Names) *If the judgments* $, \vdash_N \tau : V$ *and* $, , t : \{t^+\}, ,_T \vdash R :: \kappa$ *are derivable, where* $,_T$ *is a context fragment listing only type variables, then* $MethNames(, , t : \{t^+\}, ,_T ; R) = MethNames(, , ,_T ; [\tau/t]R)$*.*

Proof The proof is by induction on the derivation of $, , t : \{t^+\}, ,_T \vdash R :: \kappa$. All cases follow routinely from the inductive hypothesis. ∎

Lemma 7.9.10 (Subtyping Implies Method Name Superset) *If judgment* $, \vdash_N R_1 <:_B R_2$ *is derivable, then* $MethNames(, ; R_2) \subseteq MethNames(, ; R_1)$.

Proof The proof is by induction on the derivation of $, \vdash_N R_1 <:_B R_2$. All cases follow routinely from the inductive hypothesis except for $(<: \beta\ right)$, which requires Lemmas 7.6.3 and 7.9.9. ∎

Lemma 7.9.11 (Flat Row Supertype Characterization) *If we may derive the row-subtyping judgment* $, \vdash_N \langle\!\langle R_1' \,|\, m : \tau' \rangle\!\rangle <:_B R_2$ *and* $MethNames(, ; \langle\!\langle R_1' \,|\, m : \tau' \rangle\!\rangle) = MethNames(, ; R_2)$, *then* R_2 *has the form* $\langle\!\langle R_2' \,|\, m : \tau_2' \rangle\!\rangle$.

Proof The proof is by induction on the derivation of $, \vdash_N \langle\!\langle R_1' \,|\, m : \tau' \rangle\!\rangle <:_B R_2$. The $(<: row\ refl)$, $(<: cong)$, and $(<: d)$ cases follow immediately from the form of their conclusion judgments. The $(<: w)$ rule cannot have been the last rule in the derivation of $, \vdash_N \langle\!\langle R_1' \,|\, m : \tau' \rangle\!\rangle <:_B R_2$ because this rule forces R_2 to have a strictly smaller method name set, violating the second assumption above. The $(<: row\ trans)$ case follows from two applications of Lemma 7.9.10 and the inductive hypothesis. Lemma 7.9.10 is necessary to show that the method name set for the intermediate row expression is equal to $MethNames(, ; \langle\!\langle R_1' \,|\, m : \tau' \rangle\!\rangle)$ and $MethNames(, ; R_2)$. ∎

To characterize what kinds of rows can be subtypes of row variables, we need the following definition, which essentially calculates the position of row variable r in context , , counting from the left.

Definition 7.9.12 *Given a context , and a row variable* r *such that the judgment* $, \vdash_N r :: \kappa$ *is derivable for some kind* κ, *we define the Rank of* r *with respect to , as follows:*

$$Rank(r ; , , a) = \begin{cases} RowLen(,) + 1 & \textit{if } r = var(a) \\ Rank(r ; ,) & \textit{otherwise} \end{cases}$$

where $RowLen(,)$ *is the number of row variables listed in , and*

$$var(a) = \begin{cases} x & \textit{if } a = x : \tau \\ t & \textit{if } a = t : \{t^+\} \\ r & \textit{if } a = r <:_w R :: \kappa \end{cases}$$

Lemma 7.9.13 (Row Variable Subtype Characterization) *If the judgment* $, \vdash_N R <:_B r$ *is derivable, then* R *must be a row variable* q *such that* $Rank(q; ,) \geq Rank(r; ,)$.

Proof The proof is by induction on the derivation of $, \vdash_N R <:_B r$. The $(<: row\ refl)$ case follows immediately, $(row\ proj\ bound)$ follows routinely from Lemma 7.2.5, and $(<: row\ trans)$ follows immediately from the inductive hypothesis. The $(<: w)$ rule cannot produce a judgment of this form, because all row variables have functional kind, whereas the row expressions related via $(<: w)$ must have flat kind. Similarly, by Lemma 7.6.4, the $(<: \beta\ right)$ rule cannot have produced the judgment $, \vdash_N R <:_B r$ as r has functional kind. ■

To prove some further properties of subtyping, which are useful in proving Lemma 7.11.1, we need the following definition and two lemmas.

Definition 7.9.14 *Given a context* $,$ *and a row expression* R *such that the judgment* $, \vdash_N R :: \kappa$ *is derivable for some kind* κ*, we define the main row variable rank function* $MainRank(, ; R)$ *as follows, by recursion on the form of* R*:*

$$\begin{array}{lcl}
MainRank(, ; \langle\!\langle\rangle\!\rangle) & = & 0 \\
MainRank(, ; r) & = & Rank(r; ,) \\
MainRank(, ; \langle\!\langle R \,|\, m : \tau \rangle\!\rangle) & = & MainRank(, ; R) \\
MainRank(, ; \lambda t.\, R) & = & MainRank(, , t : \{t^+\} ; R) \\
MainRank(, ; R\tau) & = & MainRank(, ; R)
\end{array}$$

The follow technical lemma is needed to prove Lemma 7.9.16.

Lemma 7.9.15 *If the judgments* $, , t : \{t^+\}, ,_T \vdash_N R :: \kappa$ *and* $, , ,_T \vdash_N [\tau/t]R :: \kappa'$ *are derivable, where* $,_T$ *is a context fragment listing only type variables, then* $MainRank(, , ,_T ; [\tau/t]R) = MainRank(, , t : \{t^+\}, ,_T ; R)$.

Proof The proof is by induction on the derivation of $, , t : \{t^+\}, ,_T \vdash_N R :: \kappa$. The *(row proj)* case follows from the fact that for any row variable r, $Rank(, , ,_T ; r) = Rank(, , t : \{t^+\}, ,_T ; r)$ and $[\tau/t]r = r$. The rank of r is equal in the two contexts because the Rank function only counts row variables. The *(empty row)* case follows from the fact that

$$Rank(, , ,_T ; \langle\!\langle\rangle\!\rangle) = Rank(, , t : \{t^+\}, ,_T ; \langle\!\langle\rangle\!\rangle) = 0.$$

The remaining cases follow immediately from the inductive hypothesis. ■

Lemma 7.9.16 (Subtyping Implies Greater Main Rank) *If the judgment* $, \vdash_N R_1 <:_B R_2$ *is derivable, then* $MainRank(, ; R_1) \geq MainRank(, ; R_2)$.

Proof The proof is by induction on the derivation of $, \vdash_N R_1 <:_B R_2$. The *(row proj bound)* case follows from the fact that if $, \vdash_N r <:_B R$ is derived via *(row proj bound)*, then $,$ must have the form $,', r <:_w R :: \kappa, ,''$, and we must have derived the $,' \vdash_N R :: \kappa'$ for some kind κ. This judgment implies via Lemma 7.2.5 that $MainRank(, ; R) \leq RowLen(,')$, while $MainRank(, ; r) = RowLen(,') + 1$. Hence we have that $MainRank(, ; r) \geq MainRank(, ; R)$.

The $(<: row\ refl)$ case follows from the fact that $MainRank(,\ ; r) = MainRank(,\ ; r)$, while the $(<: \beta\ right)$ case follows from the inductive hypothesis and Lemma 7.9.15. We present this case in detail below. All other cases follow routinely from the inductive hypothesis and the definition of $MainRank$.

$(<: \beta\ right)$

In this case, our initial assumption has the form

A1 $\ ,\ \vdash_N R_1 <:_B [\tau_2/t]R_2$

which must have been derived from

RH1 $\ ,\ \vdash_N R_1 <:_B (\lambda t.\, R_2)\tau_2$

Lemmas 7.6.3 and 7.6.4 indicate that we must have previously derived

RH2 $\ ,\ \vdash_N (\lambda t.\, R_2)\tau_2 :: \nu$

for some kind ν. An inspection of the typing rules then reveals that to derive RH2, we must have previously derived the judgments

RH3 $\ ,\ ,t : \{t^+\} \vdash_N R_2 :: \nu'$

RH4 $\ ,\ \vdash_N \tau_2 : V$

for some kind ν' and variance set V. Lemma 7.5.1 applied to RH3 and RH4 allows us to produce the judgment

C1 $\ ,\ \vdash_N [\tau_2/t]R_2 :: \nu''$.

We may now apply Lemma 7.9.15 to RH3 and C1 to conclude that

C2 $MainRank(,\ ; [\tau_2/t]R_2) = MainRank(,\ , t : \{t^+\}\, ; R_2)$

Applying the induction hypothesis to RH1 gives us the inequality

C3 $MainRank(,\ ; R_1) \geq MainRank(,\ ; [\tau_2/t]R_2)$

Hence

C4 $MainRank(,\ ; R_1) \geq MainRank(,\ , t : \{t^+\}\, ; R_2)$

By definition, the right-hand side of this inequality is just $MainRank(,\ ; (\lambda t.\, R_2)\tau_2)$. Hence we get the desired inequality

C5 $MainRank(,\ ; R_1) \geq MainRank(,\ ; [\tau_2/t]R_2)$ ■

Lemma 7.9.17 (Supertype Characterization for Row Variables) *If the judgment*

$$, \; , r <:_w R :: \kappa, \; ,' \underset{N}{\vdash} r <:_B R'$$

is derivable, then either R' is r or the free row variables of R' are contained in the domain of , .

Proof The proof is by induction on the derivation of $, \; , r <:_w R :: \kappa, \; ,' \vdash_N r <:_B R'$. It is not difficult to show that the $(<:$ *row refl*$)$ case satisfies the first possibility, while the (*row proj bound*) case satisfies the second. The $(<:$ *row trans*$)$ case follows routinely from the inductive hypothesis and Lemma 7.9.16. The $(<: \beta$ *right*$)$ case cannot occur because it is only applicable to rows that have flat kind (Lemma 7.6.4), whereas r has functional kind. ■

Lemma 7.9.18 (Subtype Characterization for Row Application) *If the judgment*

$$, \; \underset{N}{\vdash} R_1 <:_B R_2' \tau_2' \qquad\qquad (*)$$

is derivable, and if $MethNames(, \; ; R_1) = MethNames(, \; ; R_2' \, \tau_2')$ and $MainRank(, \; ; R_1) = MainRank(, \; ; R_2' \, \tau_2')$, then R_1 must be of the form $R_1' \, \tau_1'$. Furthermore, the last rule in the derivation of $(*)$ cannot have been $(<: \beta$ right$)$.

Proof The proof is by induction on the derivation of $(*)$. The $(<:$ *row refl*$)$ case follows routinely from its hypothesis. The (*row proj bound*) case is vacuously true since the rows it relates must have functional kind, whereas $R_2' \, \tau_2'$ has flat kind. The $(<:$ *app*$)$ rules follow immediately from their rule hypotheses. The $(<: w)$ rule cannot have been the last rule in the derivation of $(*)$ because this rule forces $R_2' \, \tau_2'$ to have a method name set that is strictly smaller than the one for R_1, violating the second assumption above. The $(<:$ *row trans*$)$ case follows from two applications of the inductive hypothesis.

The only other possibility for the final rule in the derivation of $(*)$ is $(<: \beta$ *right*$)$. We will show that this case is not possible. For $(*)$ to have been derived via $(<: \beta$ *right*$)$ in a $\vdash_N$-derivation, R_1 must have the form of a row function application. Furthermore, the analysis in Lemma 7.7.1 reveals that both R_1 and $R_2' \, \tau_2'$ must be in normal form. Hence $(*)$ must have the form:

A1′ $, \; \vdash_N r_1 \, \tau_1 <:_B r_2 \, \tau_2'$

for some row variables r_1 and r_2 and some type τ_1. The definition of $MainRank$ reveals that

C1 $MainRank(, \; ; r_1 \, \tau_1) = MainRank(, \; ; r_1) = Rank(r_1 \, ; \; , \;)$

C2 $MainRank(, \; ; r_2 \, \tau_2') = MainRank(, \; ; r_2) = Rank(r_2 \, ; \; , \;)$

By the third assumption above, these two ranks must be equal. Hence $r_1 = r_2 = r$ and A1' has the form

A1″ $, \; \vdash_N r \, \tau_1 <:_B r \, \tau_2'$

Furthermore, the form of the $(<:\ \beta\ right)$ rule reveals that $r\,\tau_2'$ must be of the form $[\tau/t](r\,\tau_2'')$ for some types τ and τ_2'', and we must have previously derived

RH1 $\quad , \vdash_N r\,\tau_1 <:_B (\lambda t.\, r\,\tau_2'')\tau$

The definition of $\vdash_N$-derivations further reveals that RH1 must have been derived via a $(<:\ app)$ rule. In any of the four possible rules, we must have previously derived the judgment

RH2 $\quad , \vdash_N r <:_B \lambda t.\, r\,\tau_2''$

However, Lemma 7.9.17 reveals that any supertype of a row function variable r must be either r itself, or an expression whose free variables all appear strictly prior to r in context , . Since r does not appear strictly prior to itself in , , judgment RH2 is not derivable. Hence we cannot have derived $(*)$ via $(<:\ \beta\ right)$. ■

7.10 Subtyping Implies Component Subtyping Lemmas

The lemmas in this section show that when two rows or types are related via subtyping, then their sub-type and sub-row expressions are also in the subtyping relation. These lemmas are primarily used to in the proof of Lemma 7.11.1.

Lemma 7.10.1 (Arrow Subtyping Implies Component Subtyping) *If we may derive judgment* $, \vdash_N \tau_1 \to \tau_2 <: \tau_1' \to \tau_2'$, *then the judgments* $, \vdash_N \tau_1' <: \tau_1$ *and* $, \vdash_N \tau_2 <: \tau_2'$ *are derivable as well.*

Proof The proof is by induction on the derivation of $, \vdash_N \tau_1 \to \tau_2 <: \tau_1' \to \tau_2'$. The $(<:\ type\ refl)$ and $(<:\ \to)$ cases follow routinely from their rule hypotheses. The $(<:\ type\ trans)$ case follows from Lemma 7.9.2 and the inductive hypothesis. Lemma 7.9.2 is used to show that the intermediate type in the transitivity must be a function type. ■

Lemma 7.10.2 (Probj Subtyping Implies Row Subtyping) *If we may derive the judgment* $, \vdash_N \mathbf{probj_1}\, t.R_1 <: \mathbf{probj_2}\, t.R_2$, *then the judgment* $, , t\colon \{t^+\} \vdash_N R_1 <:_B R_2$ *is derivable as well. Furthermore, if* $\mathbf{probj_1}$ *and* $\mathbf{probj_2}$ *are both* **pro**, *then* B *must be* w *and we may derive the judgment* $, , t\colon \{t^+\} \vdash_N R_2 <:_B R_1$ *as well.*

Proof The proof is by induction on the derivation of $, \vdash_N \mathbf{probj_1}\, t.R_1 <: \mathbf{probj_2}\, t.R_2$. The $(<:\ type\ trans)$ case requires Lemma 7.9.4 and a subsidiary lemma, easily proved by induction, that all supertypes of **probj** types are themselves **probj** types. The other cases follow routinely from the inductive hypothesis. ■

Lemma 7.10.3 (Row Function Subtyping Implies Body Subtyping) *If we may derive the judgment* $, \vdash_N \lambda t.\, R_1 <:_B \lambda t.\, R_2$, *then the judgment* $, , t\colon \{t^+\} \vdash_N R_1 <:_B R_2$ *is derivable as well.*

Proof The proof is by induction on the derivation of , $\vdash_N \lambda t.\, R_1 <:_B \lambda t.\, R_2$. The ($<:$ *row refl*) and ($<:$ λ) cases follow routinely from their rule hypotheses. The ($<:$ *row trans*) case follows from Lemma 7.6.2, which requires the intermediate row expression to have the form $\lambda t.\, R_3$, and the inductive hypothesis. Finally, the ($<:$ β *right*) rule cannot have been the last rule in the derivation of , $\vdash_N \lambda t.\, R_1 <:_B \lambda t.\, R_2$ because $\lambda t.\, R_1$ has functional kind and Lemma 7.6.4 reveals that the rows related via ($<:$ β *right*) have flat kind. ∎

Lemma 7.12.4 in Section 7.12 is a more general version of this lemma. However, this weaker version is needed to prove Lemma 7.11.1, which in turn is used to prove 7.12.4.

Lemma 7.10.4 (Flat Row Subtyping Implies Component Subtyping) *If we may derive the judgment*

$$, \underset{N}{\vdash} \langle\!\langle R_1 \,|\, m : \tau_1 \rangle\!\rangle <:_B \langle\!\langle R_2 \,|\, m : \tau_2 \rangle\!\rangle \qquad (*)$$

and establish the equality $MethNames(,\ ; \langle\!\langle R_1 \,|\, m : \tau_1 \rangle\!\rangle) = MethNames(,\ ; \langle\!\langle R_2 \,|\, m : \tau_2 \rangle\!\rangle)$*, then the judgments* , $\vdash_N R_1 <:_B R_2$ *and* , $\vdash_N \tau_1 <: \tau_2$ *are derivable.*

Proof The proof is by induction on the derivation of $(*)$. The ($<:$ *row refl*), ($<:$ *cong*), and ($<:$ d) cases follow follow routinely from their rule hypotheses. The ($<:$ w) rule cannot derive the judgment $(*)$. (Note that Lemmas 7.8.4 and 7.9.10 reveal that m could not be in the method set of the $\langle\!\langle R_2 \,|\, m : \tau_2 \rangle\!\rangle$.) The ($<:$ *row trans*) case follows from Lemma 7.9.11 and the inductive hypothesis. Finally, the ($<:$ β *right*) rule cannot have produced $(*)$ because this rule can only appear immediately after a (*row fn app*) rule, in which case $\langle\!\langle R_1 \,|\, m : \tau_1 \rangle\!\rangle$ would have to have the form of an application. ∎

Lemma 7.10.5 (Row Application Subtyping Implies Row Function Subtyping) *If we may derive the judgment*

$$, \underset{N}{\vdash} R_1'\, \tau_1' <:_B R_2'\, \tau_2' \qquad (*)$$

and if the equalities $MethNames(,\ ; R_1'\, \tau_1') = MethNames(,\ ; R_2'\, \tau_2')$ *and* $MainRank(,\ ; R_1'\, \tau_1') = MainRank(,\ ; R_2'\, \tau_2')$ *both hold, then the judgment* , $\vdash_N R_1' <:_B R_2'$ *is derivable.*

Proof The proof is by induction on the derivation of $(*)$. The ($<:$ *row refl*) case follows routinely from its hypothesis. The ($<:$ *app*) rules follow immediately from their rule hypotheses. The ($<:$ *row trans*) case follows Lemma 7.9.18 from two applications of the inductive hypothesis. Finally, it follows from Lemma 7.9.18 that the ($<:$ β *right*) rule cannot have been the last rule in the derivation of $(*)$, and hence that case is vacuously true. ∎

Note that the above proof says nothing about the subtyping relationship between τ_1' and τ_2'. The reason for this silence is that their relationship depends on the variance of R_2', and in general, variance is not preserved by subtyping. In particular, if $(*)$ is derived via ($<:$ *row trans*), we cannot show a connection between τ_1' and τ_2'. We are able to show that the intermediate row in the transitivity must have the form R'_3, τ'_3, but cannot connect the variance of R'_3 to that of R'_2. As a result, we cannot connect τ'_3 to τ'_1, and hence cannot connect τ'_1 or τ'_2. This point is addressed in the next lemma, and is the source of its complexity.

7.11 Properties of Subtyping

This section, which conceptually includes Lemmas 7.6.3 and 7.6.4, contains lemmas connecting the subtyping relation to other parts of the type system. The first lemma listed in this section, Lemma 7.11.1, is the most difficult lemma in the entire proof of type soundness. In the following lemma, we use U to range over rows R and types τ. We use the judgment form $, \vdash_N U :- \gamma$ to denote either the judgment $, \vdash_N R :: \kappa$ or $, \vdash_N \tau : V$. Note that Lemma 7.6.3 implies that if the judgment $, \vdash_N U_1 <:_B U_2$ is derivable, then U_1 and U_2 are either both rows or both types.

Lemma 7.11.1 (Mutual Subtyping Implies Same Variance) *If the judgments* $, \vdash_N U_1 <: U_2$ *and* $, \vdash_N U_2 <: U_1$ *are both derivable, then for some kind* γ *we may derive judgments* $, \vdash_N U_1 :- \gamma$ *and* $, \vdash_N U_2 :- \gamma$*, as well. Furthermore, if* U_1 *and* U_2 *are rows, then we may also derive the judgments* $, \vdash_N U_1 <:_w U_2$ *and* $, \vdash_N U_2 <:_w U_1$*.*

Proof The proof is by induction on the structure of U_1.

Case 1: $U_1 \equiv \langle\!\langle\rangle\!\rangle$

This case follows routinely from Lemmas 7.9.7 and 7.2.1.

Case 2: $U_1 \equiv r$

In this case, our initial assumptions have the form:

A1 $, \vdash_N r <:_B R_2$

A2 $, \vdash_N R_2 <:_B r$

Lemma 7.9.13 applied to A2 reveals that R_2 must be a row variable q such that $Rank(q\,;\,,\,) \geq Rank(r\,;\,,\,)$. Applying Lemma 7.9.13 to A1 further reveals that $Rank(r\,;\,,\,) \geq Rank(q\,;\,,\,)$. Hence we may conclude that r is q. It follows routinely from A1 and Lemma 7.6.3 that we may derive the judgments $, \vdash_N r :: \kappa$ and $, \vdash_N r <:_w r$, just the judgments we needed.

Case 3: $U_1 \equiv \langle\!\langle R'_1 \,|\, m : \tau'_1 \rangle\!\rangle$

In this case, our initial assumptions have the form:

A1 $, \vdash_N \langle\!\langle R'_1 \,|\, m : \tau'_1 \rangle\!\rangle <:_B R_2$

A2 $, \vdash_N R_2 <:_B \langle\!\langle R'_1 \,|\, m : \tau'_1 \rangle\!\rangle$

Lemma 7.9.10, applied once to A1 and once to A2, produces the equation

C1 $MethNames(,\ ; \langle\!\langle R'_1 \,|\, m : \tau'_1 \rangle\!\rangle) = MethNames(,\ ; R_2)$

We may now invoke Lemma 7.9.11 on A1 and C1 to conclude that R_2 must have the form $\langle\!\langle R'_2 \,|\, m : \tau'_2 \rangle\!\rangle$. Applying Lemma 7.10.4 once to A1 and once to A2 produces the judgments

C2 $, \vdash_N R'_1 <:_B R'_2$

C3 $, \vdash_N \tau'_1 <: \tau'_2$

C4 $, \vdash_N R'_2 <:_B R'_1$

C5 $, \vdash_N \tau'_2 <: \tau'_1$

Applying the inductive hypothesis to C2 and C4 allows us to conclude that the judgments

C6 $, \vdash_N R'_1 <:_w R'_2$

C7 $, \vdash_N R'_2 <:_w R'_1$

C8 $, \vdash_N R'_1 :: (M;\, V')$

C9 $, \vdash_N R'_2 :: (M;\, V')$

are derivable, for some method set M and variance set V'. Lemma 7.6.3 applied to A1 reveals that we may derive the judgments

C10 $, \vdash_N R'_1 :: (M_1, m;\, V')$

C11 $, \vdash_N R'_2 :: (M_2, m;\, V')$

Lemma 7.8.1 reveals that the two variance sets must equal V'. It then follows from Lemma 7.8.2 and (*row label*) that we may derive the judgments

C12 $, \vdash_N R'_1 :: (M \cup \{m\};\, V')$

C13 $, \vdash_N R'_2 :: (M \cup \{m\};\, V')$

Now, by applying the inductive hypothesis to C3 and C5 we may derive the judgments

C14 $, \vdash_N \tau'_1 : V$

C15 $, \vdash_N \tau'_2 : V$

for some variance set V. We may now use (*row ext*) to produce the judgments

C16 $, \vdash_N \langle\!\langle R'_1 \,|\, m : \tau'_1 \rangle\!\rangle :: (M;\, Merge(V', V))$

C17 $, \vdash_N \langle\!\langle R'_2 \,|\, m : \tau'_2 \rangle\!\rangle :: (M;\, Merge(V', V))$

We may now conclude via ($<:$ *cong*) that the judgments

C18 $, \vdash_N \langle\!\langle R'_1 \,|\, m : \tau'_1 \rangle\!\rangle <:_w \langle\!\langle R'_2 \,|\, m : \tau'_2 \rangle\!\rangle$

C19 $, \vdash_N \langle\!\langle R'_2 \,|\, m : \tau'_2 \rangle\!\rangle <:_w \langle\!\langle R'_1 \,|\, m : \tau'_1 \rangle\!\rangle$

are both derivable. Judgments C16, C17, C18, and C19 are just the judgments that we needed to derive.

Case 4: $U_1 \equiv \lambda t.\, R_1'$

This case follows routinely from Lemmas 7.6.2 and 7.10.3 and the inductive hypothesis.

Case 5: $U_1 \equiv R_1'\, \tau_1'$

In this case, our initial assumptions have the form:

A1 $\;,\ \vdash_N R_1'\, \tau_1' <:_B R_2$

A2 $\;,\ \vdash_N R_2 <:_B R_1'\, \tau_1'$

Lemma 7.9.10, applied once to A1 and once to A2, produces the equation

C1 $MethNames(,\ ;\, R_1'\, \tau_1') = MethNames(,\ ;\, R_2)$

Similarly, two applications of Lemma 7.9.16 to A1 an A2 reveal

C2 $MainRank(,\ ;\, R_1'\, \tau_1') = MainRank(,\ ;\, R_2)$

We may now apply Lemma 7.9.18 to A1, C1, and C2 to conclude that R_2 must have the form $R_2'\, \tau_2'$. Lemma 7.10.5 applied once to A1 and once to A2 produces the derivations

C3 $\;,\ \vdash_N R_1' <:_B R_2'$

C4 $\;,\ \vdash_N R_2' <:_B R_1'$

Invoking the inductive hypothesis on R_1' reveals that we may derive the following four judgments:

C5 $\;,\ \vdash_N R_1' <:_w R_2'$

C6 $\;,\ \vdash_N R_2' <:_w R_1'$

C7 $\;,\ \vdash_N R_1' :: T^a \rightarrow (M;\, V)$

C8 $\;,\ \vdash_N R_2' :: T^a \rightarrow (M;\, V)$

for some annotation a, method set M, and variance set V.

Now we encounter the problem that makes this case by far the trickiest in the proof: it is difficult to connect via subtyping the types τ_1' and τ_2'. Notice that Lemma 7.10.5 does *not* relate these types. The complication arises when A1 is derived via $(<:\ \textit{row trans})$. Lemma 7.9.18 reveals that the intermediate row must have the form $R_3'\, \tau_3'$, but it says nothing about the variance of R_3'. Indeed, unless we are in the context of this induction, there is nothing to force R_3' to have the same variance as R_1' and R_2'. Hence we must prove via a nested induction that the types τ_1' and τ_2' are appropriately related.

Nested Lemma If the judgments

$$\begin{array}{ll} ,\ \vdash_N R'_1\,\tau'_1 <:_B R'_2\,\tau'_2 & \qquad (*) \\ ,\ \vdash_N R'_2\,\tau'_2 <:_B R'_1\,\tau'_1 & \\ ,\ \vdash_N R'_1 :: T^a \to (M; V) & \\ ,\ \vdash_N R'_2 :: T^a \to (M; V) & \\ ,\ \vdash_N R'_1 <:_w R'_2 & \\ ,\ \vdash_N R'_2 <:_w R'_1 & \end{array}$$

are all derivable, and

$$\begin{array}{lcl} MethNames(,\ ; R'_1\,\tau'_1) & = & MethNames(,\ ; R'_2\,\tau'_2) \\ MainRank(,\ ; R'_1\,\tau'_1) & = & MainRank(,\ ; R'_2\,\tau'_2) \end{array}$$

then

$$\begin{array}{lll} \text{if } a = +, & \text{the judgment} & ,\ \vdash_N \tau'_1 <: \tau'_2 \text{ is derivable.} \\ \text{if } a = -, & \text{the judgment} & ,\ \vdash_N \tau'_2 <: \tau'_1 \text{ is derivable.} \\ \text{if } a = o, & \text{the judgments} & ,\ \vdash_N \tau'_1 <: \tau'_2 \text{ and} \\ & & ,\ \vdash_N \tau'_2 <: \tau'_1 \text{ are derivable.} \end{array}$$

Nested Proof The proof is by induction on the derivation of $(*)$. The induction is necessary for the $(<:$ *trans*$)$ case. The $(<:$ *row refl*$)$, $(<:$ *app cong*$)$, $(<:$ *app cov*$)$, and $(<:$ *app contra*$)$ cases all follow routinely from their rule hypotheses. The $(<:$ *app vac*$)$ is vacuously true, as in this case a will be ?. Lemma 7.9.18 reveals that $(<:\ \beta$ *right*$)$ cannot have produced judgment $(*)$. All that remains is the $(<:$ *row trans*$)$ case.

$(<:$ *row trans*$)$

In this case, $(*)$ must have been derived from judgments of the form

IRH1 $,\ \vdash_N R'_1\,\tau'_1 <:_B R_3$

IRH2 $,\ \vdash_N R_3 <:_B R'_2\,\tau'_2$

By two applications of Lemma 7.9.16, we know that

IC1 $MainRank(,\ ; R'_2\,\tau'_2) \leq MainRank(,\ ; R_3) \leq MainRank(,\ ; R'_1\,\tau'_1)$

By assumption, however, $MainRank(,\ ; R'_2\,\tau'_2) = MainRank(,\ ; R'_1\,\tau'_1)$. Hence the inequalities in IC1 are actually equalities:

IC2 $MainRank(,\ ; R'_2\,\tau'_2) = MainRank(,\ ; R_3) = MainRank(,\ ; R'_1\,\tau'_1)$

Similarly, we may conclude via Lemma 7.9.10 and our assumption about $MethNames$ that

IC3 $MethNames(,\ ; R'_2\,\tau'_2) = MethNames(,\ ; R_3) = MethNames(,\ ; R'_1\,\tau'_1)$

We may now invoke Lemma 7.9.18 to reveal that R_3 must have the form $R'_3\,\tau'_3$. Furthermore, Lemma 7.10.5 reveals that we may derive the judgments

IC4 $\quad , \vdash_N R'_3 <:_B R'_2$

IC5 $\quad , \vdash_N R'_1 <:_B R'_3$

An applications of ($<:$ *row trans*) give us the judgment

IC6 $\quad , \vdash_N R'_3 <:_B R'_1$

We may now apply the outer inductive hypothesis to IC5 and IC6 to produce the judgments

IC7 $\quad , \vdash_N R'_3 <:_w R'_1$

IC8 $\quad , \vdash_N R'_1 <:_w R'_3$

IC9 $\quad , \vdash_N R'_1 :: T^{a'} \rightarrow (M'; V')$

IC10 $\quad , \vdash_N R'_3 :: T^{a'} \rightarrow (M'; V')$

for some annotation a', method set M', and variance set V'. By Lemma 7.8.1, $a' = a$ and $V' = V$. By Lemma 7.8.2 then, the derivations $i \in \{1, 2, 3\}$ are all derivable:

IC10 $\quad , \vdash_N R'_i :: T^a \rightarrow (M''; V)$

where $M'' = M \cup M'$. Two applications of ($<:$ *row trans*) now reveal that the judgments

IC11 $\quad , \vdash_N R'_3 <:_w R'_2$

IC12 $\quad , \vdash_N R'_2 <:_w R'_3$

are derivable. Two more applications of ($<:$ *row trans*) allow us to derive the judgments

IC13 $\quad , \vdash_N R'_3\,\tau'_3 <:_B R'_1\,\tau'_1$

IC14 $\quad , \vdash_N R'_2\,\tau'_2 <:_B R'_3\,\tau'_3$

We may now apply the inner inductive hypothesis to IHR1, IC13, IC10, IC7, IC8, IC2, and IC3 to conclude:

if $a = +$,	the judgment	$, \vdash_N \tau'_1 <: \tau'_3$ is derivable.
if $a = -$,	the judgment	$, \vdash_N \tau'_3 <: \tau'_1$ is derivable.
if $a = o$,	the judgments	$, \vdash_N \tau'_1 <: \tau'_3$ and
		$, \vdash_N \tau'_3 <: \tau'_1$ are derivable.

Similarly, we may apply the inner inductive hypothesis to IHR2, IC14, IC10, IC11, IC12, IC2, and IC3 to conclude:

if $a = +$,	the judgment	$, \vdash_N \tau'_3 <: \tau'_2$ is derivable.
if $a = -$,	the judgment	$, \vdash_N \tau'_2 <: \tau'_3$ is derivable.
if $a = o$,	the judgments	$, \vdash_N \tau'_3 <: \tau'_2$ and
		$, \vdash_N \tau'_2 <: \tau'_3$ are derivable.

Putting these subtyping relations together via $(<:$ *type trans*$)$, we get the relationships:

if $a = +$,	the judgment	$, \vdash_N \tau'_1 <: \tau'_2$ is derivable.
if $a = -$,	the judgment	$, \vdash_N \tau'_2 <: \tau'_1$ is derivable.
if $a = o$,	the judgments	$, \vdash_N \tau'_1 <: \tau'_2$ and
		$, \vdash_N \tau'_2 <: \tau'_2$ are derivable.

which are just the connections we needed to show. □

Returning to the outer induction, we now have two cases to consider: either $a =?$ or it does not. In the first case, we may apply $(<:$ *app vac*$)$ twice to conclude

C9 $, \vdash_N R'_1\, \tau'_1 <:_w R'_2\tau'_2$

C10 $, \vdash_N R'_2\, \tau'_2 <:_w R'_1\tau'_1$

and (*row fn app vac*) twice to produce:

C11 $, \vdash_N R'_1\, \tau'_1 :: (M;\, V)$

C12 $, \vdash_N R'_2\, \tau'_2 :: (M;\, V)$

These four judgment, C9, C10, C11, and C12, are just the judgments we need to derive in the $a =?$ case.

For the $a \neq ?$, we may invoke the inner lemma twice to show that we may derive the judgments

C13 $, \vdash_N \tau'_1 <: \tau'_2$

C14 $, \vdash_N \tau'_2 <: \tau'_1$

By the inductive hypothesis applied to C13 and C14, we get the judgments

C15 $, \vdash_N \tau'_1 : V''$

C16 $, \vdash_N \tau'_2 : V''$

for some variance set V''. We may now apply $(<:$ *app cong*$)$ twice to conclude that the judgments

C17 $, \vdash_N R'_1\, \tau'_1 <:_w R'_2\tau'_2$

C18 $, \vdash_N R'_2\, \tau'_2 <:_w R'_1\tau'_1$

are derivable. Rule (*row fn app cong*) gives us:

C19 $, \vdash_N R'_1\, \tau'_1 :: (M;\, Merge(V, V''))$

C20 $, \vdash_N R'_2\, \tau'_2 :: (M;\, Merge(V, V''))$

Judgments C17, C18, C19, and C20 are the judgments we needed to show derivable in the $a \neq ?$ case.

Case 6: $U_1 \equiv t$

The desired result follows routinely from Lemmas 7.9.1 and 7.6.3 and the inductive hypothesis.

Case 7: $U_1 \equiv \tau_1 \rightarrow \tau_2$

Here, Lemmas 7.9.2 and 7.10.1 and the inductive hypothesis produce the desired judgments.

Case 8: $U_1 \equiv \textbf{pro}\, t \centerdot R_1'$

This case follows routinely from Lemmas 7.9.4 and 7.10.2.

Case 9: $U_1 \equiv \textbf{obj}\, t \centerdot R_1'$

Finally, this case follows routinely from Lemmas 7.9.5, 7.10.2, and 7.8.1. ■

Lemma 7.11.2 (Width Subtyping Implies Lower Kind) *If* $, \vdash_N R_1 <:_w R_2$ *is derivable, then* $\kappa_1 \le \kappa_2$ *where* κ_i *is the unique variance of* R_i *in* , *for* $i \in \{1, 2\}$.

Proof The proof is by induction on the derivation of $, \vdash_N R_1 <:_w R_2$. All of the cases rely on Lemmas 7.2.3 and 7.6.3. The ($<:$ *app cong*) and ($<:$ *cong*) cases rely on Lemma 7.11.1 to show that congruent types have the same variance. The ($<:$ *app*) cases require Variance Lemma 7.1.25, while the ($<:$ *cong*) and ($<:$ *w*) case use Variance Lemmas 7.1.22 and 7.1.23, respectively. All other cases, except ($<:$ β *right*), which is presented below, follow routinely from the inductive hypothesis.

($<:$ β *right*)

In this case, the judgment we are assumed to have derived has the form:

A1 $\quad , \vdash_N R_1 <:_w [\tau_2/t]R_2$

To reach this conclusion, we must have previously derived

RH1 $\quad , \vdash_N R_1 <:_w (\lambda t.\, R_2)\tau_2$

Lemma 7.6.3 reveals that we may derive

C1 $\quad , \vdash_N R_1 :: (M_1;\, V_1)$

C2 $\quad , \vdash_N (\lambda t.\, R_2)\tau_2 :: (M_2;\, V_2)$

for some method sets M_1, M_2 and variance sets V_1, V_2. Invoking the inductive hypothesis on RH1 reveals that

C3 $\quad V_1 \le V_2$

We now need to establish the relationship between V_2 and the unique variance of $[\tau_2/t]R_2$ in , . An analysis of the rules that must have been used to conclude C2 shows that we may derive the judgments

C4 $\quad , , t\colon \{t^+\} \vdash_N R_2 :: (M_2;\, V_2')$

C5 $\ , \vdash_N \tau_2 : V$

Furthermore, $V_2 = f_a(V_2' \backslash t, V)$, where $a = Var(t, V_2')$. Applying Lemma 7.5.1 to C4 and C5 yields the judgment

C6 $\ , \vdash_N [\tau_2/t]R_2 :: (M_2;\ [V/t]V_2')$

Variance lemma 7.1.26 reveals that $f_a(V_2' \backslash t, V) = [V/t]V_2'$; hence we have that $V_1 \leq [V/t]V_2'$, which is the needed relationship between the variance of R_1 and $[\tau/t]R_2$. ■

7.12 Method Extraction Lemmas

The lemmas in this section are needed to prove Lemma 7.14.2. Essentially, they show us that width supertyping only forgets about the existence of methods, not any information about the present methods. To prove this property, we need to introduce some definitions.

Definition 7.12.1 *Given a context , and a row expression R such that the judgment $\ , \vdash_N R :: \kappa$ is derivable for some kind κ, we define the method extraction function $Meth_m(\ ;\ R)$ as follows, by recursion on the form of R:*

$$
\begin{array}{lcl}
Meth_m(\ ;\ \langle\!\langle\rangle\!\rangle) & = & \langle\!\langle\rangle\!\rangle \\[1em]
Meth_m(\ ;\ \langle\!\langle R \,|\, m' : \tau \rangle\!\rangle) & = & \left\{ \begin{array}{ll} \langle\!\langle m : \tau \rangle\!\rangle & \textit{if } m = m' \\ Meth_m(\ ;\ R) & \textit{otherwise} \end{array} \right. \\[1em]
Meth_m(\ ;\ r) & = & Meth_m(\ ,' ;\ R') \\
 & & \quad \textit{where } \ \equiv\ ,',\ r <:_w R' :: \kappa,\ ,'' \\[1em]
Meth_m(\ ;\ \lambda t.\, R) & = & \lambda t.\, Meth_m(\ ,\ t : \{t^+\} ;\ R) \\[1em]
Meth_m(\ ;\ R\tau) & = & [\tau/t]R' \\
 & & \quad \textit{if } Meth_m(\ ;\ R) = \lambda t.\, R'
\end{array}
$$

Lemma 7.12.2 (Characterization of Extracted Methods) *If $\ , \vdash_N R :: \kappa$ is derivable, then so is $\ , \vdash_N Meth_m(\ ;\ R) :: \kappa$ where $\kappa \leq \kappa'$. Furthermore,*

- *if $\kappa = \nu$, then*
$$Meth_m(\ ;\ R) = \left\{ \begin{array}{ll} \langle\!\langle\rangle\!\rangle & \textit{if } m \notin MethNames(\ ;\ R) \\ \langle\!\langle m : \tau \rangle\!\rangle & \textit{otherwise} \end{array} \right.$$
- *if $\kappa = S \rightarrow \nu$, then*
$$Meth_m(\ ;\ R) = \left\{ \begin{array}{ll} \lambda t.\langle\!\langle\rangle\!\rangle & \textit{if } m \notin MethNames(\ ;\ R) \\ \lambda t.\langle\!\langle m : \tau \rangle\!\rangle & \textit{otherwise} \end{array} \right.$$

Proof The proof is by induction on the derivation of $, \vdash_N R :: \kappa$. The (*row proj*) case requires Lemma 7.2.3. The (*row ext*) case depends on Variance Lemma 7.1.23. We present in detail the (*row fn app*) cases, as they are the most involved. The other cases follow routinely from the inductive hypothesis.

(*row fn app*)

In each of the (*row fn app*) cases, our assumption is of the form:

A1 $, \vdash_N R\tau :: (M;\, f_a(V_1, V_2))$

where a is $+$ for the covariant application rule, $-$ for the contravariant, o for the invariant, and $?$ for the vacuous. To reach this judgment, we must have previously derived

RH1 $, \vdash_N R :: T^a \to (M;\, V_1)$

RH2 $, \vdash_N \tau : V_2$

Applying the inductive hypothesis to RH1 produces

C1 $, \vdash_N Meth_m(,\ ; R) :: T^{a'} \to (M';\, V_1')$

C2 $a' \leq a$ and $V_1 \leq V_1'$

C3

$$Meth_m(,\ ; R) = \begin{cases} \lambda t.\langle\!\langle\rangle\!\rangle & \text{if } m \notin MethNames(,\ ; R) \\ \lambda t.\langle\!\langle m : \tau' \rangle\!\rangle & \text{otherwise} \end{cases}$$

Having derived C1 and RH2, we may invoke the (*row fn app*) rule corresponding to a' to get

C4 $, \vdash_N Meth_m(,\ ; R)\tau :: (M;\, f_{a'}(V_1', V_2))$

There are now two cases to consider: either $m \in Meth_m(,\ ; R)$ or it is not.

Case 1: $m \in Meth_m(,\ ; R)$

In this case, we know by C3 that $Meth_m(,\ ; R) = \lambda t.\langle\!\langle m : \tau' \rangle\!\rangle$, so C4 has the form

C5 $, \vdash_N (\lambda t.\langle\!\langle m : \tau' \rangle\!\rangle)\tau :: (M;\, f_{a'}(V_1', V_2))$

Lemma 7.6.1 allows us to derive the judgment

C6 $, \vdash_N [\tau/t]\langle\!\langle m : \tau' \rangle\!\rangle :: (M;\, f_{a'}(V_1', V_2))$

By the definition of $Meth_m$, this judgment may be written

C7 $, \vdash_N Meth_m(,\ ; R\,\tau) :: (M;\, f_{a'}(V_1', V_2))$

Since Lemma 7.1.25 applied to C2 shows that

C8 $f_a(V_1, V_2) \leq f_{a'}(V_1', V_2)$.

C7 is just the judgment we needed to derive.

Case 2: $m \in Meth_m(, \; ; R)$ This case is analogous to Case 1. ∎

The following is a technical lemma needed to establish the $(<\!: \beta\ right)$ case of Lemma 7.12.4.

Lemma 7.12.3 (Substitution Characterization for Method Search) *If both of the judgments* $, \; , t\colon \{t^+\}, \; ,_T \vdash_N R :: \kappa$ *and* $, \vdash_N \tau : V$ *are derivable, where* $,_T$ *is a context fragment containing only type variables, then for all method names* m,

$$[\tau/t]Meth_m(, \; , t\colon \{t^+\}, \; ,_T \; ; R) = Meth_m(, \; , \; ,_T \; ; [\tau/t]R).$$

Proof The proof is by induction on the derivation of $, \; , t\colon \{t^+\}, \; ,_T \vdash_N R :: \kappa$. The (*row proj*) case uses Lemma 7.12.2. The (*row fn app*) cases require Lemmas 7.5.1 and 7.12.2 and standard properties of substitution. The case for the (*row ext*) rule proceeds by a case analysis on whether or not m is the method added to the row. All other cases follow routinely from the inductive hypothesis. ∎

The following lemma is the key step in showing the soundness of (*probj* $\Leftarrow$).

Lemma 7.12.4 (Row Subtyping Implies Method Subtyping) *If judgment* $, \vdash_N R_1 <:_B R_2$ *is derivable, then for all method names* m*, the judgment* $, \vdash_N Meth_m(, \; ; R_1) <:_B Meth_m(, \; ; R_2)$ *is derivable as well. Furthermore, if* $B = w$ *and* $m \in MethNames(, \; ; R_2)$*, then the judgment* $, \vdash_N Meth_m(, \; ; R_2) <:_w Meth_m(, \; ; R_1)$ *is also derivable.*

Proof The proof is by induction on the derivation of $, \vdash_N R_1 <:_B R_2$.

The $(<\!: row\ refl)$ case requires Lemma 7.12.2. The case for (*row proj bound*) needs Lemmas 7.12.2 and 7.2.3. The $(<\!:$ *app cong*$)$ case follows routinely from Lemmas 7.12.2 and 7.6.6. The $(<\!:$ *app cov*$)$ and $(<\!:$ *app contra*$)$ cases follow routinely from Lemmas 7.6.3, 7.12.2 and 7.6.5. The cases for $(<\!:$ *cong*$)$ and $(<\!: d)$ follow easily from the inductive hypothesis. The case for $(<\!: w)$ relies on Lemmas 7.9.10, 7.6.3, 7.8.4, and 7.9.6. The $(<\!:$ *row trans*$)$ case follows easily from Lemma 7.9.10.

Somewhat surprisingly, the case for $(<\!:$ *app vac*$)$ is quite tricky; it is presented below in full detail. Less surprisingly, the $(<\!: \beta\ right)$ case also requires some work. It is described below in detail as well.

$(<\!:$ *app vac*$)$

In this case, our initial assumption has the form

A1 $, \vdash_N R_1\tau_1 <:_B R_2\tau_2$

To reach this conclusion, we must have previously derived the judgments

RH1 $, \vdash_N R_1 <:_B R_2$

RH2 $, \vdash_N R_2 :: T^? \to \nu_2$

RH3 $, \vdash_N \tau_i : V_i$

Invoking the inductive hypothesis on RH1 produces the judgment

C1 $, \vdash_N Meth_m(, \ ; R_1) <:_B Meth_m(, \ ; R_2)$

We must now transform this subtyping judgment on the row functions to the corresponding judgments on the row function applications. To that end, we need to know the general form of $Meth_m(, \ ; R_1)$ and $Meth_m(, \ ; R_2)$. By invoking Lemma 7.12.2 for R_1 and Lemmas 7.6.3 and 7.12.2 for R_2, we obtain the judgments

C2 $, \vdash_N Meth_m(, \ ; R_1) :: T^a \to \nu'_1$, where $Meth_m(, \ ; R_1) = \lambda t.\, R'_1$

and

C3 $, \vdash_N Meth_m(, \ ; R_2) :: T^? \to \nu'_2$, where $Meth_m(, \ ; R_2) = \lambda t.\, R'_2$

Note that since R_2's argument variance is ? (in RH2), the argument variance of $Meth_m(, \ ; R_2)$ must be ? as well. (This conclusion follows from Lemma 7.12.2 and the definition of variance ordering.)

We may now invoke *(row fn app vac)* on C1, C3, and RH3 to produce the judgment

C4 $, \vdash_N (\lambda t.\, R'_1)\tau_1 <:_B (\lambda t.\, R'_2)\tau_2$

Applying Lemma 7.6.5 to C4 yields

C5 $, \vdash_N [\tau_1/t]R'_1 <:_B [\tau_2/t]R'_2$

which is just

C6 $, \vdash_N Meth_m(, \ ; R_1\tau_1) <:_B Meth_m(, \ ; R_2\tau_2)$

If we also know that $B = w$ and $m \in MethNames(, \ ; R_2\tau_2)$, then the induction hypothesis also reveals that we may derive the judgment

C7 $, \vdash_N Meth_m(, \ ; R_2) <:_w Meth_m(, \ ; R_1)$

It also follows from Lemma 7.11.2 that the a in judgment C2 must be ?. Then by the same analysis as above we may derive the judgment

C8 $\;, \vdash_N Meth_m(,\ ; R_2\tau_2) <:_w Meth_m(,\ ; R_1\tau_1)$

($<: \beta$ *right*)

In this case, our assumption has the form

A1 $\;, \vdash_N R_1 <:_B [\tau_2/t]R_2$

To conclude A1, we must have previously derived

RH1 $\;, \vdash_N R_1 <:_B (\lambda t.\, R_2)\tau_2$

From the definition of $\vdash_N$-derivations, RH1 must have been derived via a (*row fn app*) rule. Hence R_1 must be of the form $R_1'\,\tau_1$, for some row R_1' and type τ_1. The fact that the desired judgment

D1 $\;, \vdash_N Meth_m(,\ ; R_1) <:_B Meth_m(,\ ; [\tau_2/t]R_2)$

is derivable then follows from a case analysis on which of the (*row fn app*) rules was used to derive RH1, the inductive hypothesis, and Lemmas 7.12.2 and 7.6.5.

If we also know that $B = w$ and $m \in MethNames(,\ ; R_2\tau_2)$, then we also need to derive the judgment

D2 $\;, \vdash_N Meth_m(,\ ; [\tau_2/t]R_2) <:_w Meth_m(,\ ; R_1)$

To this end, we apply Lemmas 7.6.3 and 7.9.9 to RH1 to conclude that

C2 $\;MethNames(,\ ; [\tau_2/t]R_2) = MethNames(,\ ; (\lambda t.\, R_2)\tau_2)$

Hence we know that

C3 $\;m \in MethNames(,\ ; (\lambda t.\, R_2)\tau_2)$

We may now invoke the inductive hypothesis on RH1 and C3 to produce the judgment

C4 $\;, \vdash_N Meth_m(,\ ; (\lambda t.\, R_2)\tau_2) <:_w Meth_m(,\ ; R_1)$

Since $Meth_m(,\ ; \lambda t.\, R_2) = \lambda t.\, Meth_m(,\ , t\colon \{t^+\}\, ; R_2)$, it follows from the definition of $Meth_m$ that

C5 $\;Meth_m(,\ ; (\lambda t.\, R_2)\tau_2) = [\tau_2/t]MethNames(,\ , t\colon \{t^+\}\, ; R_2)$

Applying Lemmas 7.6.3 and 7.12.3 to RH1 reveals that

C6 $\;[\tau_2/t]MethNames(,\ , t\colon \{t^+\}\, ; R_2) = Meth_m(,\ ; [\tau_2/t]R_2)$

Hence C4 is just

C4′ $\;, \vdash_N Meth_m(,\ ; [\tau_2/t]R_2) <:_w Meth_m(,\ ; R_1)$

which is D2, the judgment we needed to show derivable. ■

Lemma 7.12.5 (Extracted Method is Mutual Subtype of Width Supertype) *If judgment* $, \vdash_N R <:_w \langle\!\langle m : \tau \rangle\!\rangle$ *is derivable, then the judgment* $, \vdash_N Meth_m(, \; ; R) \cong_w \langle\!\langle m : \tau \rangle\!\rangle$ *is derivable as well. Furthermore,* $Meth_m(, \; ; R) = \langle\!\langle m : \tau' \rangle\!\rangle$ *for some type* τ'

Proof It follows from Lemma 7.12.4 that we may derive the judgments

C1 $, \vdash_N Meth_m(, \; ; R) \cong_w \langle\!\langle m : \tau \rangle\!\rangle$

From Lemma 7.9.10, we may conclude that $m \in MethNames(, \; ; R)$, and Lemma 7.6.3 reveals that judgment

C2 $, \vdash_N R :: \kappa$

is derivable for some kind κ. We may now conclude via Lemma 7.12.2 that for some type τ', $Meth_m(, \; ; R) = \langle\!\langle m : \tau' \rangle\!\rangle$. ■

Lemma 7.12.6 (Width Supertypes are Mutual Subtypes) *If we may derive the judgments* $, \vdash_N R <:_w \langle\!\langle m : \tau_1 \rangle\!\rangle$ *and* $, \vdash_N R <:_w \langle\!\langle m : \tau_2 \rangle\!\rangle$*, then we may derive the judgments* $, \vdash_N \tau_1 \cong \tau_2$ *as well.*

Proof It follows from Lemma 7.12.5 that $Meth_m(, \; ; R) = \langle\!\langle m : \tau \rangle\!\rangle$ for some type τ, and that the judgments

C1 $, \vdash_N \langle\!\langle m : \tau \rangle\!\rangle \cong_w \langle\!\langle m : \tau_1 \rangle\!\rangle$

C2 $, \vdash_N \langle\!\langle m : \tau \rangle\!\rangle \cong_w \langle\!\langle m : \tau_2 \rangle\!\rangle$

are derivable. By two applications of $(<: row\ trans)$, we may derive the judgments

C3 $, \vdash_N \langle\!\langle m : \tau_1 \rangle\!\rangle \cong_w \langle\!\langle m : \tau_2 \rangle\!\rangle$

We may now invoke Lemma 7.9.8 twice to obtain the judgments

C4 $, \vdash_N \tau_1 \cong \tau_2$ ■

7.13 Row Substitution

This section contains a fairly standard row-substitution lemma, used to show that polymorphic method bodies may be instantiated as necessary. The other lemma in this section is a technical lemma that allows us to strengthen assumptions about row variables in judgments. In this section and in the following, we will use the shorthand $\kappa_{min(\Gamma)}$ to stand for the kind $T^0 \to (\emptyset; Invar(TVar(,)))$. Intuitively, this is the "least" well-formed kind possible in context , . It is this kind we use to type check method bodies, since it makes the least assumptions about the subtyping possible in future extensions of the host object. In the following lemma, we must consider the possibility of substituting a row with lesser variance for r because of the $(pro\ ext)$ and $(pro\ ov)$ typing rules.

Lemma 7.13.1 (Row Substitution)

If $\;, \;, r <:_w R' :: \kappa_r, \;, '\vdash_N A, \quad , \vdash_N R :: \kappa, \quad , \vdash_N R <:_B R', \quad$ *and* $\kappa_r \leq \kappa$

then $\;, \;, [R/r], '\vdash_N A',$

where

- *if* $A \equiv *,$ *then* $A' \equiv *$
- *if* $A \equiv U{:}-\gamma,$ *then* $A' \equiv [R/r]U{:}-\gamma'$ *where* $\gamma \leq \gamma'$ *(and* $\gamma = \gamma'$ *if* $\kappa = \kappa_r$*)*
- *if* $A \equiv \sigma{:}V,$ *then* $A' \equiv [R/r]\sigma : V'$ *where* $V \leq V'$ *(and* $V = V'$ *if* $\kappa = \kappa_r$*)*
- *if* $A \equiv U_1 <:_{(B')} U_2,$ *then* $A' \equiv [R/r]U_1 <:_{(B+B')} [R/r]U_2$
- *if* $A \equiv e : \tau,$ *then* $A' \equiv e : [R/r]\tau$
- *if* $A \equiv e : \sigma,$ *then* $A' \equiv [R/r]e : [R/r]\sigma$

Proof

The proof is by induction on the derivation of $, \;, r <:_w R' :: \kappa, \;, '\vdash_N A$. The row projection cases (*row proj*) and (*row proj bound*) require a case analysis on whether or not r is the projected variable. The subtype equality rule ($<:$ *β right*) and rules (*probj* $\Leftarrow$) and ($\exists <:$ *intro*) require the substitution property that if $u \notin FV(R)$, then

$$[[R/r]U'/u]([R/r]U) = [R/r][U'/u]U$$

where U is either a row or type expression and u either a row or type variable. The remaining cases either follow routinely from the inductive hypothesis or are similar to the case for (*pro ext*), presented below in full detail.

(*pro ext*)

We start with three assumptions, labeled A1, A2, and A3:

A1 $\;, \;, r <:_w R' :: \kappa_r, \;, '\vdash_N \langle e_1 \longleftrightarrow\!\!+\; m = e_2\rangle : \mathbf{pro}\, t.\langle\!\langle R_1 \mid m : \tau\rangle\!\rangle$

A2 $\;, \vdash_N R <: R'$

A3 $\;, \vdash_N R :: \kappa$

Note that we may assume without loss of generality that $t \notin dom(,)$. Since we know that A1 was derived via (*pro ext*), we must have previously derived the rule hypotheses RH1, RH2, RH3, and RH4:

RH1 $\;, \;, r <:_w R' :: \kappa_r, \;, '\vdash_N e_1 : \mathbf{pro}\, t.R_1$

RH2 $, \, , r <:_w R' :: \kappa_r, \, ,', t \colon \{t^+\} \vdash_N R_1 :: (\{m\}; V)$

RH3 $, \, , r <:_w R' :: \kappa_r, \, ,', p <:_w P :: \kappa_p \vdash_N e_2 : [\mathbf{pro}\, t\boldsymbol{.} p\, t\,/t](t \to \tau)$
where $P = \lambda t.\langle\!\langle R_1 \,|\, m : \tau \rangle\!\rangle$ and $\kappa_p = \kappa_{min(\Gamma,\, r <:_w R' :: \kappa_r,\, \Gamma')}$

RH4 $p \notin FV(\tau)$

Applying the inductive hypothesis to RH1, A2, and A3, we get:

C1 $, \, , [R/r], ' \vdash_N e_1 : [R/r]\,\mathbf{pro}\, t\boldsymbol{.} R_1$

Because $t \notin FV(R)$, C1 is just:

C2 $, \, , [R/r], ' \vdash_N e_1 : \mathbf{pro}\, t\boldsymbol{.}[R/r]R_1$

This judgment is the first hypothesis of the (*pro ext*) typing rule we will use to build the desired judgment. To get the second and third such hypothesis judgments, we need to apply the inductive hypothesis first to RH2, A2, and A3 and then to RH3, A2, and A3:

C3 $, \, , [R/r], ', t \colon \{t^+\} \vdash_N [R/r]R_1 :: (\{m\}; V)$

C4 $, \, , [R/r](, ', p <:_w P :: \kappa_p) \vdash_N e_2 : [R/r]([\mathbf{pro}\, t\boldsymbol{.} p\, t\,/t](t \to \tau))$

Because $t \notin FV(R)$ and $TVar(, \, , r <:_w R' :: \kappa_r, \, , ') = TVar(, \, , [R/r], ')$, C4 is just

C5 $, \, , [R/r], ', p <:_w P' :: \kappa_p \vdash_N e_2 : [\mathbf{pro}\, t\boldsymbol{.} p\, t\,/t](t \to [R/r]\tau)$
where $P' = \lambda t.\langle\!\langle [R/r]R_1 \,|\, m : [R/r]\tau \rangle\!\rangle$ and $\kappa_p = \kappa_{min(\Gamma,\, [R/r]\Gamma')}$

The final hypothesis that we need to establish is that $p \notin FV([R/r]\tau)$. From RH4, we know that $p \notin FV(\tau)$. By applying Lemma 7.2.5 first to A3 and then to RH3, we learn that $FV(R) \subseteq dom(, \,)$ and that $p \notin dom(, \,)$. Hence $p \notin FV(R)$. Thus we get

C6 $p \notin FV([R/r]\tau)$

Applying (*pro ext*) to C2, C3, C5, and C6 produces

C7 $, \, , [R/r], ' \vdash_N \langle e_1 \leftarrow\!\!+ m = e_2 \rangle : \mathbf{pro}\, t\boldsymbol{.}\langle\!\langle [R/r]R_1 \,|\, m : [R/r]\tau \rangle\!\rangle$

Because $t \notin FV(R)$, C7 is just

C8 $, \, , [R/r], ' \vdash_N \langle e_1 \leftarrow\!\!+ m = e_2 \rangle : [R/r]\mathbf{pro}\, t\boldsymbol{.}\langle\!\langle R_1 \,|\, m : \tau \rangle\!\rangle$

which is what we needed to show. ■

Lemma 7.13.2 is crucial to the proofs of Lemma 7.14.2 and Lemma 7.14.9.

Lemma 7.13.2 (Bound Transformation) *If the judgments*

$$, , p <:_w P :: T^o \to (M_p; Invar(TVar(,))), , ' \underset{N}{\vdash} B$$

$$, , r <:_w R :: T^o \to (M_r; Invar(TVar(,))) \vdash *$$

and

$$, \underset{N}{\vdash} R <:_w P$$

are all derivable, $r \notin dom(, ')$ and $M_p \subseteq M_r$, then the judgment

$$, , r <:_w R :: T^o \to (M_r; Invar(TVar(,))), [r/p], ' \vdash [r/p]B$$

is also derivable, where B is any judgment not involving strictly σ-types.

Proof The proof is by induction on the derivation of

$$, , p <:_w P :: T^o \to (M_p; Invar(TVar(,))), , ' \underset{N}{\vdash} B$$

All cases except those that require a row variable in their contexts follow immediately from the inductive hypothesis. The (*row var*), (*pro ext*), and (*pro ov*) cases follow routinely from the inductive hypothesis. The (*row proj*) and (*row proj bound*) cases follow routinely from the inductive hypothesis and a case analysis on whether or not p is the projected variable.

The proof of this lemma is very sensitive to the form of the (*pro ext*) proof rule. ■

7.14 Expression Lemmas

Finally, we reach proofs specific to the expression portion of our language. This section includes a standard expression substitution lemma (Lemma 7.14.10) as well as lemmas that show that the method bodies inside of well-typed prototypes and objects are themselves well-typed and appropriately polymorphic (Lemmas 7.14.2,7.14.3, and 7.14.4). These lemmas are then used to show that each of the operational semantics reduction axioms exhibit the subject reduction property (Lemmas 7.14.5, 7.14.6, 7.14.7, 7.14.8, 7.14.9, 7.14.12, and 7.14.13]). Lemma 7.14.16 then reveals that a derivation from a judgment $e \vdash_N \sigma$ can only depend on the form of σ, not on the form of e. Combining this result with the subject reduction results for the reduction axioms gives us Theorem 7.14.17 (Subject Reduction) for the full operational semantics.

The section starts with a lemma showing that any type we can give to an expression must be a well-formed type.

Lemma 7.14.1 (Expression Types are Well-Formed) *If $, \vdash_N e : \tau$ is derivable, then $, \vdash_N \tau : V$ is as well, for some variance set V.*

Proof The proof is by induction on the derivation of $, \vdash_N e : \tau$. The (*exp proj*) case follows from Lemma 7.2.3. The (*subsumption*) case uses Lemma 7.6.3. The case for (*exp abs*) relies on Lemmas 7.4.1 and 7.2.1. The (*probj* $\Leftarrow$) case depends on Lemmas 7.6.3 and 7.5.1. The case for (*pro ext*) is presented in detail below. All other cases follow immediately from the inductive hypothesis.

(*pro ext*)

The judgment we are assumed to have derived has the form

A1 $, \vdash_N \langle e_1 \longleftrightarrow m = e_2 \rangle : \mathbf{pro}\, t\,\mathbf{.}\langle\!\langle R \,|\, m : \tau \rangle\!\rangle$

To reach this conclusion, we must have previously concluded (among other things)

RH1 $, , t : \{t^+\} \vdash_N R :: (\{m\}; V)$
RH2 $, , r <:_w \lambda t.\langle\!\langle R \,|\, m : \tau \rangle\!\rangle :: \kappa_{min(\Gamma)} \vdash_N e_2 : [\mathbf{pro}\, t\,\mathbf{.} r\, t/t](t \rightarrow \tau)$
RH3 $r \notin FV(\tau)$

Applying the inductive hypothesis to RH2 allows us to derive

C1 $, , r <:_w \lambda t.\langle\!\langle R \,|\, m : \tau \rangle\!\rangle :: \kappa_{min(\Gamma)} \vdash_N [\mathbf{pro}\, t\,\mathbf{.} r\, t/t](t \rightarrow \tau) : V_1$

for some variance set V_1. As this judgment must have been derived via (*type arrow*), we must have previously derived

C2 $, , r <:_w \lambda t.\langle\!\langle R \,|\, m : \tau \rangle\!\rangle :: \kappa_{min(\Gamma)} \vdash_N [\mathbf{pro}\, t\,\mathbf{.} r\, t/t]\tau : V_2$

Note that without loss of generality, we may assume that $t \notin dom(,) \cup \{r\}$ and that we may derive the judgment

C3 $, , r <:_w \lambda t.\langle\!\langle R \,|\, m : \tau \rangle\!\rangle :: \kappa_{min(\Gamma)} \vdash_N \mathbf{pro}\, t\,\mathbf{.} r\, t : V_3$.

Hence we may apply Lemma 7.8.5 to C2 to produce the judgment

C4 $, , r <:_w \lambda t.\langle\!\langle R \,|\, m : \tau \rangle\!\rangle :: \kappa_{min(\Gamma)}, t : \{t^+\} \vdash_N \tau : V_4$

The fact that variance sets do not contain row variables and RH3 allow us to apply Lemma 7.4.2 to conclude

C5 $, , t : \{t^+\} \vdash_N \tau : V_4$

To this judgment and RH1, we may first apply (*row ext*) and then (*pro*) to produce the necessary judgment:

C6 $, \vdash_N \mathbf{pro}\, t\,\mathbf{.}\langle\!\langle R \,|\, m : \tau \rangle\!\rangle : V_5$

for some variance set V_5. ■

The following two lemmas are used to show that the method bodies that appear within expressions of **pro** type are type correct.

Lemma 7.14.2 (Method Bodies Type Correct) *If the judgment* $, \vdash_N \langle e_1 \mathrel{\leftarrow\!\!\circ} m{=}e_2\rangle : \mathbf{pro}\, t.R$ *is the conclusion of a* $\vdash_N$ *derivation in which the last rule was not* (*type eq*)*, then there exists a type* τ *and a row variable* r *such that the judgments*

$$, , t\colon \{t^+\} \underset{N}{\vdash} R <:_w \langle\!\langle m : \tau \rangle\!\rangle$$

and

$$, , r <:_w \lambda t.\, R :: \kappa_{min(\Gamma)} \underset{N}{\vdash} e_2 : [\mathbf{pro}\, t.r\, t/t](t \to \tau)$$

are derivable. Furthermore, if $\leftarrow\!\!\circ$ *is* $\leftarrow$*, then the judgment* $, \vdash_N e_1 : \mathbf{pro}\, t.R$ *is derivable as well.*

Proof

The proof is by induction on the derivation of $, \vdash_N \langle e_1 \mathrel{\leftarrow\!\!\circ} m{=}e_2\rangle : \mathbf{pro}\, t.R$. The induction is necessary for the (*subsumption*) case, which I will present in detail. The (*pro over*) and (*pro ext*) cases follow routinely from their hypotheses.

(*subsumption*)

In this case, our initial hypothesis has the form

A1 $\quad , \vdash_N \langle e_1 \mathrel{\leftarrow\!\!\circ} m{=}e_2\rangle : \mathbf{pro}\, t.R$

To reach this conclusion via (*subsumption*), we must have previously derived

RH1 $\quad , \vdash_N \langle e_1 \mathrel{\leftarrow\!\!\circ} m{=}e_2\rangle : \tau$

RH2 $\quad , \vdash_N \tau <: \mathbf{pro}\, t.R$

Lemma 7.9.4 reveals that τ must be of the form $\mathbf{pro}\, t.R'$. Hence, RH1 has the form:

C1 $\quad , \vdash_N \langle e_1 \mathrel{\leftarrow\!\!\circ} m{=}e_2\rangle : \mathbf{pro}\, t.R'$

Now, recall that in a $\vdash_N$-form derivation, the only occurrences of (*type eq*) must immediately precede an occurrence of ($\exists <:$ *intro*). Since the derivation of RH1 preceded an instance of subsumption, it could not have been derived via (*type eq*). Hence we may invoke the inductive hypothesis to conclude that there exists a type τ' such that the judgments

C2 $\quad , , t\colon \{t^+\} \vdash_N R' <:_w \langle\!\langle m : \tau' \rangle\!\rangle$

C3 $\quad , , r <:_w \lambda t.\, R' :: \kappa_{min(\Gamma)} \vdash_N e_2 : [\mathbf{pro}\, t.r\, t/t](t \to \tau')$

are derivable, and if $\leftarrow\!\!\circ$ is $\leftarrow$, then the judgment

C4 $\quad , \vdash_N e_1 : \mathbf{pro}\, t.R'$

is derivable as well. Type τ' will be the type we are interested in. Applying Lemma 7.10.2 to RH2, we may reach the judgments

C5 $\;, \;, t : \{t^+\} \vdash_N R' \cong_w R$

By invoking $(<:\ \mathit{trans})$, we may conclude

C6 $\;, \;, t : \{t^+\} \vdash_N R <:_w \langle\!\langle m : \tau' \rangle\!\rangle$

We may now invoke Lemma 7.2.1 on A1, Lemma 7.6.3 on C5, and the fact that $\kappa_{min(\Gamma)}$ is the least variance $\lambda t.\, R$ can have in , to derive via $(\mathit{row\ var})$ the judgment

C7 $\;, \;, p <:_w \lambda t.\, R :: \kappa_{min(\Gamma)} \vdash_N *$

where $p \notin \mathit{dom}(\;, \;) \cup \{r\}$. We may now apply Lemma 7.13.2 to C3, C7, and C5 to derive

C8 $\;, \;, p <:_w \lambda t.\, R :: \kappa_{min(\Gamma)} \vdash_N e_2 : [\mathbf{pro}\, t\mathbf{.} p\, t/t](t \to \tau')$

(Note that Lemma 7.2.5 implies that $p \notin FV(\tau')$.) Finally, we may derive from C5 and Lemma 7.6.3 via $(<:\ \mathit{convert})$ the judgment

C9 $\;, \vdash_N \mathbf{pro}\, t\mathbf{.}R' <: \mathbf{pro}\, t\mathbf{.}R$

We may now conclude via $(\mathit{subsumption})$ applied to C4 and C9 that if $\leftarrow\!\!\!\circ$ is $\leftarrow$, then we may derive the judgment

C10 $\;, \vdash_N e_1 : \mathbf{pro}\, t\mathbf{.}R$

Conclusions C6, C8, C10 are just the judgments we needed to derive for type τ'. ■

Corollary 7.14.3 *If the judgment* $\;, \vdash_N \langle e_1 \leftarrow\!\!\!\circ\, m{=}e_2 \rangle : \mathbf{pro}\, t\mathbf{.}R$ *is the conclusion of a* $\vdash_N$ *derivation in which the last rule was not* $(\mathit{type\ eq})$*, then there exists a type* τ *such that the judgments* $\;, \;, t : \{t^+\} \vdash_N R <:_w \langle\!\langle m : \tau \rangle\!\rangle$ *and* $\;, \vdash_N e_2 : [\mathbf{pro}\, t\mathbf{.}R/t](t \to \tau)$ *are derivable.*

Proof If we have derived $\;, \vdash_N \langle e_1 \leftarrow\!\!\!\circ\, m{=}e_2 \rangle : \mathbf{pro}\, t\mathbf{.}R$, then by Lemma 7.14.2, there exist type τ and row variable r such that we may derive the judgments

C1 $\;, \;, t : \{t^+\} \vdash_N R <:_w \langle\!\langle m : \tau \rangle\!\rangle$

C2 $\;, \;, r <:_w \lambda t.\, R :: \kappa_{min(\Gamma)} \vdash_N e_2 : [\mathbf{pro}\, t\mathbf{.}r\, t/t](t \to \tau)$

It follows from Lemma 7.6.3, $(<:\ \mathit{row\ refl})$, and the fact that $\kappa_{min(\Gamma)}$ is the least variance $\lambda t.\, R$ can have in , that we may apply Lemma 7.13.1 to C2 to produce the judgment

C3 $\;, \vdash_N e_2 : [\lambda t.\, R/r]([\mathbf{pro}\, t\mathbf{.}r\, t/t](t \to \tau))$

Since Lemma 7.2.5 implies that $r \notin FV(\tau)$, C3 may be rewritten as

C4 $\;, \vdash_N e_2 : [\mathbf{pro}\, t . (\lambda t.\, R)\, t/t](t \to \tau)$

Lemma 7.7.1 applied to C4 reveals that we may derive the judgment

C5 $\;, \vdash_N e_2 : [\mathbf{pro}\, t . R/t](t \to \tau)$

Conclusion C5 is the judgment we needed to show derivable. ■

The following lemma is used in conjunction with Lemma 7.14.2 to show that method bodies that appear in expressions with **obj** type may be given the appropriate types.

Lemma 7.14.4 (Object Types Come From Pro Types) *If* $, \vdash_N \langle e_1 \leftarrow\!\!\!\circ\; m{=}e_2\rangle : \mathbf{obj}\, t . R$ *is the conclusion of a* $\vdash_N$ *derivation in which the last rule was not* (*type eq*)*, then there exists a type* $\mathbf{pro}\, t . R'$ *such that the judgments* $, \vdash_N \langle e_1 \leftarrow\!\!\!\circ\; m{=}e_2\rangle : \mathbf{pro}\, t . R'$ *and* $, \vdash_N \mathbf{pro}\, t . R' <: \mathbf{obj}\, t . R$ *are both derivable. Furthermore, the last rule in the derivation of* $, \vdash_N \langle e_1 \leftarrow\!\!\!\circ\; m{=}e_2\rangle : \mathbf{pro}\, t . R'$ *is not* (*type eq*)*.*

Proof The proof is by induction on the derivation of $, \vdash_N \langle e_1 \leftarrow\!\!\!\circ\; m{=}e_2\rangle : \mathbf{obj}\, t . R$. The form of this judgment reveals that the last rule in its derivation must have been (*subsumption*). Hence we must have previously derived the judgments

RH1 $\;, \vdash_N \mathbf{probj}\, t . R'' <: \mathbf{obj}\, t . R$

RH2 $\;, \vdash_N \langle e_1 \leftarrow\!\!\!\circ\; m{=}e_2\rangle : \mathbf{probj}\, t . R''$.

There are now two cases to consider: either **probj** is **pro** or it is **obj**. In the first case, RH1 and RH2 are exactly the judgments we needed to derive. The second case requires more work. Because RH2 is not immediately used in as the hypothesis to ($\exists <:$ *intro*), we cannot have derived RH2 via (*type eq*). Hence we may apply the inductive hypothesis to obtain the judgments

C1 $\;, \vdash_N \mathbf{pro}\, t . R''' <: \mathbf{probj}\, t . R''$

C2 $\;, \vdash_N \langle e_1 \leftarrow\!\!\!\circ\; m{=}e_2\rangle : \mathbf{pro}\, t . R'''$.

The inductive hypothesis also reveals that the last rule in the derivation of C2 was not (*type eq*). A simple application of ($<:$ *type trans*) to C1 and RH1 now produces the desired result. ■

Lemma 7.14.5 (Subject Reduction for Message Sending) *If the judgment* $, \vdash_N \langle e_1 \leftarrow\!\!\!\circ\; m{=}e_2\rangle \Leftarrow m : \tau$ *is derivable, then so is the judgment* $, \vdash_N e_2 \langle e_1 \leftarrow\!\!\!\circ\; m{=}e_2\rangle : \tau$.

Proof The proof is by induction on the derivation of $, \vdash_N \langle e_1 \leftarrow\!\!\!\circ\; m{=}e_2\rangle \Leftarrow m : \tau$. The induction is necessary for the (*subsumption*) and (*type eq*) cases, which then follow routinely. The (*pro* $\Leftarrow$) case follows from the inductive hypothesis and Lemmas 7.14.3, 7.12.6, 7.14.1, and 7.5.1. The most complex case is the one for (*obj* $\Leftarrow$). In this case, the proof amounts to going back to the **pro** type that we must have derived for the expression $\langle e_1 \leftarrow\!\!\!\circ\; m{=}e_2\rangle$, using the fact that we had this typing to prove e_2 has the necessary type, and then using subsumption to get the needed type for $e_2 \langle e_1 \leftarrow\!\!\!\circ\; m = e_2\rangle$. In more detail:

(*obj* $\Leftarrow$)

In this case, our initial hypothesis has the form:

A1 $\;, \vdash_N \langle e_1 \leftarrow\!\!\!\circ\; m = e_2 \rangle \Leftarrow m : [\mathbf{obj}\, t\boldsymbol{.}R/t]\tau$

Since A1 was derived via (*obj* $\Leftarrow$), we must have previously derived the judgments

RH1 $\;, \vdash_N \langle e_1 \leftarrow\!\!\!\circ\; m = e_2 \rangle : \mathbf{obj}\, t\boldsymbol{.}R$

RH2 $\;, , t : \{t^+\} \vdash_N R <:_w \langle\!\langle m : \tau \rangle\!\rangle$

Lemma 7.14.4 reveals that the judgments

C1 $\;, \vdash_N \langle e_1 \leftarrow\!\!\!\circ\; m = e_2 \rangle : \mathbf{pro}\, t\boldsymbol{.}R'$

C2 $\;, \vdash_N \mathbf{pro}\, t\boldsymbol{.}R' <: \mathbf{obj}\, t\boldsymbol{.}R$

are derivable in such a way that the last rule in deriving C1 was not (*type eq*). Hence by Lemma 7.14.3, the following judgments are also derivable:

C3 $\;, , t : \{t^+\} \vdash_N R' <:_w \langle\!\langle m : \tau_2 \rangle\!\rangle$

C4 $\;, \vdash_N e_2 : [\mathbf{pro}\, t\boldsymbol{.}R'/t](t \rightarrow \tau_2)$

We may now apply (*exp app*) to C4, C1 to produce the judgment

C5 $\;, \vdash_N e_2 \langle e_1 \leftarrow\!\!\!\circ\; m = e_2 \rangle : [\mathbf{pro}\, t\boldsymbol{.}R'/t]\tau_2$

By Lemma 7.10.2 and ($<:$ *row trans*), we may conclude

C6 $\;, , t : \{t^+\} \vdash_N R' <:_B \langle\!\langle m : \tau \rangle\!\rangle$

Applying Lemma 7.12.4 to C3 and C6 respectively produces the judgments

C7 $\;, , t : \{t^+\} \vdash_N Meth_m(, \;; R') \cong_w \langle\!\langle m : \tau_2 \rangle\!\rangle$

C8 $\;, , t : \{t^+\} \vdash_N Meth_m(, \;; R') <:_B \langle\!\langle m : \tau \rangle\!\rangle$

By ($<:$ *row trans*), we may derive

C9 $\;, , t : \{t^+\} \vdash_N \langle\!\langle m : \tau_2 \rangle\!\rangle <:_B \langle\!\langle m : \tau \rangle\!\rangle$

By Lemma 7.9.8, we may conclude

C10 $\;, , t : \{t^+\} \vdash_N \tau_2 <: \tau$

Now we need to know the variance of t in τ to establish the needed subtyping judgment. To that end, we invoke Lemma 7.6.3 on C2 and RH2, respectively, to reveal that the judgments

C11 $\;, \vdash_N \mathbf{obj}\, t\boldsymbol{.}R : V_1$

C12 $, \vdash_N \langle\!\langle m : \tau \rangle\!\rangle :: (M;\, V_2)$

are derivable, for some variance sets V_1 and V_2. The form of judgment C11 indicates it must have been derived via (*cov obj*). Hence we must have previously derived

C13 $, , t \colon \{t^+\} \vdash_N R :: (M;\, V_1')$

C14 $Var(t, V_1') \in \{+, ?\}$

where $V_1 = V_1' \backslash t$.

The form of judgment C12 reveals that it must have been derived via (*row ext*). Hence we must have previously derived the judgment

C15 $, , t \colon \{t^+\} \vdash_N \tau : V_2$

Then Lemma 7.11.2 applied to RH2 implies that

C16 $V_1' \leq V_2$

By the definition of variance inequality, then, we get that

C17 $Var(t, V_2) \in \{+, ?\}$

By Lemma 7.5.4, applied to C10, C15, C14, C2, we may derive

C18 $, \vdash_N [\mathbf{pro}\, t\mathbf{.} R'/t]\tau_2 <: [\mathbf{obj}\, t\mathbf{.} R/t]\tau$

Now we may derive via (*subsumption*) applied to C5 and C18

C19 $, \vdash_N e_2\langle e_1 \leftarrow\!\!\ominus\; m = e_2\rangle : [\mathbf{obj}\, t\mathbf{.} R/t]\tau$

which is the judgment we needed to derive. ∎

Lemma 7.14.6 (Subject Reduction for Cancel Over Over) *If we may derive the judgment* $, \vdash_N \langle\langle e_1 \leftarrow m_3 = e_2\rangle \leftarrow m_3 = e_3\rangle : \tau$, *then we may derive judgment* $, \vdash_N \langle e_1 \leftarrow m_3 = e_3\rangle : \tau$ *as well.*

Proof The proof is by induction on the derivation of $, \vdash_N \langle\langle e_1 \leftarrow m_3 = e_2\rangle \leftarrow m_3 = e_3\rangle : \tau$ because of the (*type eq*) and (*subsumption*) cases, which then follow immediately from the inductive hypothesis. The (*pro over*) case follows routinely from Lemma 7.14.2. ∎

Lemma 7.14.7 (Subject Reduction for Add Over) *If* $, \vdash_N \langle e_1 \leftarrow\!\!\!+\; m_2 = e_2\rangle : \tau$ *is derivable, then judgment* $, \vdash_N \langle\langle e_1 \leftarrow\!\!\!+\; m_2 = e_2\rangle \leftarrow m_2 = e_2\rangle : \tau$ *is derivable as well.*

Proof The proof is by induction on the derivation of $, \vdash_N \langle e_1 \leftarrow\!\!\!+\; m_2 = e_2\rangle : \tau$ because of the (*type eq*) and (*subsumption*) cases. The (*pro ext*) case follows routinely from Lemmas 7.14.1 and 7.9.6. ∎

Lemma 7.14.8 (Subject Reduction for Permute Over Over) *If we may derive the judgment* $, \vdash_N \langle\langle e_1 \leftarrow m_2 = e_2\rangle \leftarrow m_3 = e_3\rangle : \tau$*, then we may derive* $, \vdash_N \langle\langle e_1 \leftarrow m_3 = e_3\rangle \leftarrow m_2 = e_2\rangle : \tau$ *as well.*

Proof The proof is by induction on the derivation of $, \vdash_N \langle\langle e_1 \leftarrow m_2 = e_2\rangle \leftarrow m_3 = e_3\rangle : \tau$ because of the (*type eq*) and (*subsumption*) cases, which then follow immediately from the inductive hypothesis. The (*pro ov*) case follows routinely from Lemma 7.14.2. ∎

Lemma 7.14.9 (Subject Reduction Switch Extend Over) *If we may derive the judgement* $, \vdash_N \langle\langle e_1 \leftarrow m_2 = e_2\rangle \leftarrow\!\!+\, m_3 = e_3\rangle : \tau$*, then* $, \vdash_N \langle\langle e_1 \leftarrow\!\!+\, m_3 = e_3\rangle \leftarrow m_2 = e_2\rangle : \tau$ *is derivable as well.*

Proof The proof is by induction on the derivation of $, \vdash_N \langle\langle e_1 \leftarrow m_2 = e_2\rangle \leftarrow\!\!+\, m_3 = e_3\rangle : \tau$. The induction is necessary for the (*subsumption*) and (*type eq*) cases, which then follow routinely. The only other possibility is (*pro ext*), which we present in detail below.

(*pro ext*)

In this case, we assume we have derived

A1 $, \vdash_N \langle\langle e_1 \leftarrow m_2 = e_2\rangle \leftarrow\!\!+\, m_3 = e_3\rangle : \mathbf{pro}\, t.\langle\!\langle R \mid m_3 : \tau_3\rangle\!\rangle$

via (*pro ext*). Hence we must have previously derived the judgments

RH1 $, \vdash_N \langle e_1 \leftarrow m_2 = e_2\rangle : \mathbf{pro}\, t.R$

RH2 $, , t : \{t^+\} \vdash_N R :: (\{m_3\}; V)$

RH3 $, , r <:_w \lambda t.\langle\!\langle R \mid m_3 : \tau_3\rangle\!\rangle :: \kappa_{min(\Gamma)} \vdash_N e_3 : [\mathbf{pro}\, t.r\, t/t](t \to \tau_3)$

RH4 $r \notin FV(\tau_3)$

By applying Lemma 7.14.2 to RH1, we may derive the judgments

C1 $, \vdash_N e_1 : \mathbf{pro}\, t.R$

C2 $, , t : \{t^+\} \vdash_N R <:_w \langle\!\langle m_2 : \tau_2\rangle\!\rangle$

C3 $, , p <:_w \lambda t. R :: \kappa_{min(\Gamma)} \vdash_N e_2 : [\mathbf{pro}\, t.p\, t/t](t \to \tau_2)$

for some type τ_2 and some row variable $p \notin dom(,)$. We may now apply (*pro ext*) to C1, RH2, RH3, and RH4 to derive the judgment

C4 $, \vdash_N \langle e_1 \leftarrow\!\!+\, m_3 = e_3\rangle : \mathbf{pro}\, t.\langle\!\langle R \mid m_3 : \tau_3\rangle\!\rangle$

To override this object's m_2 method, we need to know some subtyping properties of R. To obtain the first of these properties, we invoke Lemma 7.6.3 on A1 to show that $\langle\!\langle R \mid m_3 : \tau_3\rangle\!\rangle$ is well-formed in context $, , t : \{t^+\}$ and then apply $(<: w)$ to C2 to derive the judgment

C5 $\;, \, , t : \{t^+\} \vdash_N \langle\!\langle R \,|\, m_3 : \tau_3 \rangle\!\rangle <:_w \langle\!\langle m_2 : \tau_2 \rangle\!\rangle$

Similarly, to obtain the second needed subtyping property, we may use $(<: row\ refl)$, $(<: w)$, and $(<: \lambda)$ to derive the judgment

C6 $\;, \vdash_N \lambda t. \langle\!\langle R \,|\, m_3 : \tau_3 \rangle\!\rangle <:_w \lambda t.\, R$

We may obtain the final preliminary judgment that we need by applying Lemma 7.2.1 to RH3 to derive the judgment

C7 $\;, \, , r <:_w \lambda t. \langle\!\langle R \,|\, m_3 : \tau_3 \rangle\!\rangle :: \kappa_{min(\Gamma)} \vdash_N *$

We may now invoke Lemma 7.13.2 on C3, C7, and C6 to derive

C8 $\;, \, , r <:_w \lambda t. \langle\!\langle R \,|\, m_3 : \tau_3 \rangle\!\rangle :: \kappa_{min(\Gamma)} \vdash_N e_2 : [\mathbf{pro}\, t \,\boldsymbol{.}\, r\, t/t](t \rightarrow \tau_2)$

(Note that Lemma 7.2.5 implies that $p \notin FV(\tau_2)$). Applying $(pro\ ov)$ to C4, C5, and C8 allows us to derive

C9 $\;, \vdash_N \langle \langle e_1 \leftarrow\!\!+\, m_3 = e_3 \rangle \leftarrow m_2 = e_2 \rangle : \mathbf{pro}\, t \,\boldsymbol{.} \langle\!\langle R \,|\, m_3 : \tau_3 \rangle\!\rangle$

the judgment we needed to show derivable. ∎

Lemma 7.14.10 (Expression Substitution) *If both of the judgments* $, \, , x : \tau_2, \, ,' \vdash_N e_1 : \sigma_1$ *and* $, \vdash_N e_2 : \tau_2$ *are derivable, then so is the judgment* $, \, , \, ,' \vdash_N [e_2/x]e_1 : \sigma_1$ *.*

Proof The proof is by induction on the derivation of $, \, , x : \tau_2, \, ,' \vdash_N e_1 : \sigma_1$. The $(exp\ proj)$ case follows from a case analysis on whether or not x is the projected variable. It requires Lemmas 7.4.1 and 7.2.3 to appropriately adjust the contexts. The other cases follow routinely from the inductive hypothesis and Lemma 7.4.1. ∎

Lemma 7.14.11 (Function Body Substitution) *If the judgment* $, \vdash_N \lambda x.\, e_2 : \tau_1 \rightarrow \tau_2$ *is the result of a* $\vdash_N$*-derivation in which the last rule was not* $(type\ eq)$ *and the judgment* $, \vdash_N e_1 : \tau_1$ *is derivable, then the judgment* $, \vdash_N [e_1/x]e_2 : \tau_1$ *is derivable as well.*

Proof The proof is by induction on the derivation of $, \vdash_N \lambda x.\, e_2 : \tau_1 \rightarrow \tau_2$. The $(subsumption)$ case follows from Lemmas 7.9.3 and 7.10.1 and two applications of the $(subsumption)$ typing rule. The $(exp\ abs)$ case follows routinely from Lemma 7.14.10. ∎

Lemma 7.14.12 (Function Application Subject Reduction) *If judgment* $, \vdash_N (\lambda x.\, e_1)\, e_2 : \tau$ *is derivable, then the judgment* $, \vdash_N [e_2/x]e_1 : \tau$ *is derivable as well.*

Proof The proof is by induction on the derivation of $, \vdash_N (\lambda x.\, e_1)\, e_2 : \tau$ because of the $(subsumption)$ and $(type\ eq)$ rules, which then follow immediately from the inductive hypothesis. The $(exp\ app)$ case follows immediately from Lemma 7.14.11. ∎

Lemma 7.14.13 (Class Subject Reduction) *If the judgment*

$$, \underset{N}{\vdash} \mathit{Abstype}\ r <:_w R :: \kappa\ \mathit{with}\ x : \tau\ \mathit{is}\ \{\!|r <:_w R :: \kappa = R',\ e_1|\!\}\ \mathit{in}\ e_2 : \sigma$$

is derivable, then so is the judgment $, \vdash_N [R'/r,\ e_1/x]e_2 : \sigma$.

Proof The proof is by induction on the derivation of

$$, \underset{N}{\vdash} \mathit{Abstype}\ r <:_w R :: \kappa\ \mathit{with}\ x : \tau\ \mathit{is}\ \{\!|r <:_w R :: \kappa = R',\ e_1|\!\}\ \mathit{in}\ e_2 : \sigma$$

to handle the (*subsumption*) and (*type eq*) cases, which then follow immediately from the inductive hypothesis. The only other possible case is $(\exists <: \mathit{elim})$, which we describe in detail.

$(\exists <: \mathit{elim})$

In this case, we must have previously derived the judgments

RH1 $, \vdash_N \{\!|r <:_w R :: \kappa = R',\ e_1|\!\} : \exists(r <:_w R :: \kappa)\tau$

RH2 $, , r <:_w R :: \kappa, x : \tau \vdash_N e_2 : \sigma$

RH3 $, \vdash_N \sigma : V$

Because (*subsumption*) and (*type eq*) rules do not apply to proper σ-types, the last rule in the derivation of RH1 must have been $(\exists <: \mathit{intro})$. Hence we must have previously derived the judgments

RH4 $, \vdash_N R' :: \kappa$

RH5 $, \vdash_N R' <:_w R$

RH6 $, \vdash_N e_1 : [R'/r]\tau$

By applying Lemma 7.13.1 to RH2, RH4, and RH5, we may derive the judgment

C1 $, , x : [R'/r]\tau \vdash_N [R'/r]e_2 : [R'/r]\sigma$

We may now apply Lemma 7.14.10 to C1 and RH6 to derive the judgment

C2 $, \vdash_N [e_1/x][R'/r]e_2 : [R'/r]\sigma$

Lemma 7.2.5 applied to RH3 implies that $r \notin FV(\sigma)$; hence $[R'/r]\tau_2$ is just σ. Since $x \notin FV(R')$ and $r \notin FV(e_1)$, C2 is just

C3 $, \vdash_N [R'/r, e_1/x]e_2 : \sigma$

which is just the judgment we needed to show derivable. ■

Lemma 7.14.14 (Axiom Reduction Subject Reduction) *If the judgment* $, \vdash_N e : \tau$ *is derivable, and* $e \xrightarrow{axm} e'$, *then the judgment* $, \vdash_N e' : \tau$ *is also derivable.*

Proof The proof is a case analysis on which of the reduction axioms $e \overset{axm}{\longrightarrow} e'$ is an instance of. Depending on the case, the desired result follows from Lemma 7.14.5, 7.14.6, 7.14.7, 7.14.8, 7.14.9, 7.14.12, or 7.14.13. ■

Lemma 7.14.15 (Subexpressions are Typed) *If we may derive the judgment* , $\vdash_N C[e] : \sigma$, *then we must have previously derived a judgment of the form* , $' \vdash_N e : \sigma'$,

Proof The proof is by induction on the structure of C. All cases follow routinely from the inductive hypothesis. ■

Lemma 7.14.16 (Typing Depends only on Expression Types) *If the judgment* , $\vdash_N C[e] : \sigma$ *is derived from the judgment* , $' \vdash_N e : \sigma'$ *and the judgment* , $' \vdash_N e' : \sigma'$ *is derivable, then the judgment* , $\vdash_N C[e'] : \sigma$ *is derivable as well.*

Proof The proof is by induction on the derivation of , $\vdash_N C[e] : \sigma$. All cases follow immediately from either the assumptions or the inductive hypothesis because the typing rules do not assume anything about the form of the expressions they manipulate; the rules only make assumptions about the *types* of their hypothesis expressions. ■

Theorem 7.14.17 (Subject Reduction) *If the judgment* , $\vdash_N e : \sigma$ *is derivable, and* $e \overset{ceval}{\longrightarrow} e'$, *then the judgment* , $\vdash_N e' : \sigma$ *is derivable as well.*

Proof From the definition of reduction, this theorem can be restated as follows: If the judgment , $\vdash_N C[e_1] : \sigma$ is derivable, and $e_1 \overset{axm}{\longrightarrow} e_2$, then the judgment , $\vdash_N C[e_2] : \sigma$ is derivable as well. To prove this restated theorem, we first invoke Lemma 7.14.15 to show that the judgment , $\vdash_N C[e_1] : \sigma$ must have been derived from a judgment of the form , $' \vdash_N e_1 : \sigma'$. Lemma 7.14.14 then allows us to derive the judgment , $' \vdash_N e_2 : \sigma'$. Finally, we may use Lemma 7.14.16 to show that the judgment , $\vdash_N C[e_2] : \sigma$ is derivable. ■

7.15 Type Soundness

Now that we have subject reduction, we need to formalize the notion of message-not-understood errors to show that our type system prevents them. Intuitively, a message-not-understood error occurs when a message m is sent to an expression that does not define an object with an m-method. To formalize this notion, we define mutually recursive functions *eval* and get_m via proof rules in the style of structured operational semantics. The ideas behind this proof system are discussed below. The full system is given in Appendix F.

The *eval* function is the standard lazy evaluator from lambda calculus, extended to our object calculus in the following way. Object expressions other than message sends evaluate to themselves. On expressions of the form $e \Leftarrow m$, *eval* uses the function get_m to extract the m method from

e. This behavior is specified in the $(ev \Leftarrow)$ proof rule, shown below. In the rule, meta-variable z represents either an expression or the special "value" *error* and *ev* stands for either *getm* or *eval*.

$$(ev \Leftarrow) \qquad \frac{\begin{array}{c} get_m(e) = \langle e_1 \leftarrow m = e_2 \rangle \\ ev(e_2 \langle e_1 \leftarrow n = e_2 \rangle) = z \end{array}}{ev(e \Leftarrow m) = z}$$

We may read this rule as follows. Once get_m has extracted the m method from e by returning an expression of the form $\langle e_1 \leftarrow m = e_2 \rangle$, we "send" the message m to this expression by applying e_2 to the object. This resulting expression is then recursively evaluated to z, which is then returned as the value of the original message send.

How does get_m extract an m-method from its expression? There are two different forms of expressions from which get_m may immediately extract an m method: $\langle e_1 \leftarrow m = e_2 \rangle$ and $\langle e_1 \leftarrow\!\!+ m = e_2 \rangle$. The corresponding axioms are:

$$(getm \leftarrow) \qquad get_m(\langle e_1 \leftarrow m = e_2 \rangle) = \langle e_1 \leftarrow m = e_2 \rangle$$

$$(getm \leftarrow\!\!+) \qquad get_m(\langle e_1 \leftarrow\!\!+ m = e_2 \rangle) = \langle \langle e_1 \leftarrow\!\!+ m = e_2 \rangle \leftarrow m = e_2 \rangle$$

The second of these rules converts its object $\langle e_1 \leftarrow\!\!+ m = e_2 \rangle$ to the equivalent object $\langle \langle e_1 \leftarrow\!\!+ m = e_2 \rangle \leftarrow m = e_2 \rangle$ so that get_m returns objects in a standard form.

To extract an m method from more complicated expressions, we recursively use the *eval* and get_m functions. The $(ev \Leftarrow)$ rule given above, when ev has the value get_m, is representative of these cases.

How could get_m fail to find an m method? There are four different ways in which get_m may immediately "realize" that its object does not have the required method m. Its object could be a variable, a lambda abstraction, an abstract data type, or the empty object $\langle\rangle$. These possibilities are described by the following four axioms:

$$(getm\ var) \qquad get_m(x) = error$$

$$(getm\ \lambda) \qquad get_m(\lambda x.\, e) = error$$

$$(getm\ \exists) \qquad get_m(\{\!|r <:_w R :: \kappa = R',\ e_1|\!\}) = error$$

$$(getm\ \langle\rangle) \qquad get_m(\langle\rangle) = error$$

When called on more complicated expressions, get_m fails to find its desired method if one of its recursive calls fails. The $(ev \Leftarrow err)$ rule

$$(ev \ \Leftarrow \ err) \qquad \frac{get_n(e) = error}{ev(e \Leftarrow n) = error}$$

and the $(ev \ \Leftarrow)$ give above (when z is $error$) are representative of these cases. These rules reflect the fact that there are two ways we could fail to find an m-method in an expression of the form $e \Leftarrow n$. The first, described by $(ev \ \Leftarrow \ err)$, occurs when we cannot find an n-method in e. The second, described by the $(ev \ \Leftarrow)$ rule, occurs when we cannot find an m method in the expression obtained by invoking e's n-method.

We need to define these mutually recursive functions instead of using a single *eval* function because the notion of a value changes within the context of a message send. In particular, when we are not looking for a method, any object expression of the form $\langle e \leftarrow\!\!\circ\, m = e' \rangle$ is a value. If we are looking for an m method, then expressions of the form $\langle e \leftarrow m = e' \rangle$ are still values. However, if we are looking for an n method, $\langle e \leftarrow\!\!\circ\, m = e' \rangle$ is not a value and must be evaluated further.

Our specific *eval* and get_m functions are designed so that we may demonstrate that our type system prevents message-not-understood errors in programs. The same technique would allow us to show that terms typed as function expressions in programs either diverge or reduce to lambda abstractions. To do this, we would need to add a third evaluation function, get_λ, that "looks" for lambda abstractions and returns a tagged error value $function\text{-}error$ when called on an expression that cannot reduce to a lambda abstraction. To simplify the presentation, we consider only message-not-understood errors here.

Using the proof rules, we may show formally that typeable programs of our object calculus do not produce message-not-understood errors.

Lemma 7.15.1 *If $ev(e_1) = e_2$, then $e_1 \overset{ceval}{\twoheadrightarrow} e_2$.*

Proof The proof is by induction on the derivation of $ev(e_1) = e_2$. The base case for the $(getm \ \longleftarrow\!\!\!+)$ axiom follows from the $(add\ ov)$ bookkeeping rule. The other base cases are either vacuous, since $error$ is not an expression, or immediate.

The inductive case for the $(ev \ \Leftarrow)$ proof rule is given below since it is representative of the non-vacuous inductive cases. If we have derived $ev(e \Leftarrow n) = e'$ via $(ev \ \Leftarrow)$, then we must previously have derived that

$$ev(e) = \langle e_1 \leftarrow n = e_2 \rangle$$

and

$$ev(e_2 \langle e_1 \leftarrow n = e_2 \rangle) = e'.$$

Note that $ev(e_2 \langle e_1 \leftarrow n = e_2 \rangle)$ cannot equal $error$, since we know that $ev(e \Leftarrow n) = e'$ and e' is an expression, not *error*. Thus

$$\begin{aligned} e \Leftarrow n &\overset{ceval}{\twoheadrightarrow} \langle e_1 \leftarrow n = e_2 \rangle \Leftarrow n \\ &\overset{ceval}{\rightarrow} e_2 \langle e_1 \leftarrow n = e_2 \rangle \\ &\overset{ceval}{\rightarrow} e' \end{aligned}$$

The first step above follows from the inductive hypothesis and fact that the reduction rules are a congruence relation. The second step follows from the $(\Leftarrow)$-reduction rule, and the final one via the inductive hypothesis. ■

It remains to show that if we may derive $\emptyset \vdash e : \tau$, then $eval(e) \neq error$. Because there are two ways in which $eval(e)$ could not equal $error$, either by returning an expression or being undefined (which happens when e diverges under lazy evaluation), it is simpler to prove the contrapositive:

Lemma 7.15.2 *If $ev(e) = error$, then*

- *if ev is get_m, then $\emptyset \not\vdash e : \mathbf{probj}\, t.R$ for any row R and type τ such that the judgment $t : \{t^+\} \vdash_N R <:_w \langle\!\langle m : \tau \rangle\!\rangle$ is derivable.*
- *if ev is $eval$, then $\emptyset \not\vdash e : \sigma$ for any type σ,*

where $\Gamma \not\vdash A$ indicates that the judgment $\Gamma \vdash A$ is not derivable.

Proof The proof is by induction on the derivation of $ev(e) = error$. The base cases are either vacuous or follow by inspection of the typing rules. We give the inductive cases for the $(ev \Leftarrow err)$ and $(ev \Leftarrow)$ rules, since they are representative of the non-vacuous inductive cases.

$(ev \Leftarrow err)$

If we derive $ev(e) = error$ via $(ev \Leftarrow err)$, then e must be of the form $e' \Leftarrow n$, and we must have previously derived that $get_n(e') = error$. Applying the inductive hypothesis, we get that either

$$\emptyset \not\vdash e' : \mathbf{probj}\, t.R$$

for any row R such that the judgment $t : \{t^+\} \vdash_N R <:_w \langle\!\langle m : \tau \rangle\!\rangle$ is derivable, or

$$\emptyset \not\vdash e : \sigma$$

for any type σ, depending on whether ev is actually $getm$ or $eval$. Then an inspection of the typing rule for message send $(\mathbf{probj} \Leftarrow)$ reveals that

$$\emptyset \not\vdash e' \Leftarrow n : \tau''$$

for any type τ''. *A fortiori,*

$$\emptyset \not\vdash e' \Leftarrow n : \mathbf{probj}\, t.R$$

for any row R and type τ such that the judgment $t : \{t^+\} \vdash_N R <:_w \langle\!\langle m : \tau \rangle\!\rangle$ is derivable, which is what we needed to show.

$(ev\ \Leftarrow)$

If we derive $ev(e) = error$ via $(ev\ \Leftarrow)$, then e must be of the form $e' \Leftarrow n$. Also, we must have previously derived that $get_n(e') = \langle e_1 \leftarrow n = e_2 \rangle$ and $ev(e_2\langle e_1 \leftarrow n = e_2 \rangle) = error$. Applying the inductive hypothesis to the second of these equations, we get that either

$$\emptyset \not\vdash e_2\langle e_1 \leftarrow n = e_2 \rangle : \mathbf{probj}\, t.R$$

for any row R such that the judgment $t : \{t^+\} \vdash_N R <:_w \langle\!\langle m : \tau \rangle\!\rangle$ is derivable, or

$$\emptyset \not\vdash e_2\langle e_1 \leftarrow n = e_2 \rangle : \sigma$$

for any type σ, depending on whether ev is actually $getm$ or $eval$. By Lemma 7.15.1, $e' \overset{ceval}{\twoheadrightarrow} \langle e_1 \leftarrow n = e_2 \rangle$, so

$$\begin{aligned} e' \Leftarrow n \quad &\overset{ceval}{\twoheadrightarrow} \quad \langle e_1 \leftarrow n = e_2 \rangle \Leftarrow n \\ &\overset{ceval}{\rightarrow} \quad e_2\langle e_1 \leftarrow n = e_2 \rangle. \end{aligned}$$

Hence we get by Subject Reduction (Theorem 7.14.17) that either

$$\emptyset \not\vdash e' \Leftarrow n : \mathbf{probj}\, t.R$$

for any row R such that the judgment $t : \{t^+\} \vdash_N R <:_w \langle\!\langle m : \tau \rangle\!\rangle$ is derivable, or

$$\emptyset \not\vdash e' \Leftarrow n : \sigma$$

for any type σ, depending on whether ev is actually $getm$ or $eval$, which is what we needed to show. ■

The contrapositive of the above lemma then gives us type soundness:

Theorem 7.15.3 (Type Soundness) *If the judgment $\emptyset \vdash e : \tau$ is derivable, then $eval(e) \neq error$.*

Appendix A

Shape Program: Typecase Version

The following program, written in C, illustrates how a shape-manipulating program might be written in a conventional programming language. In Section 2.3, we compare this organization to that of the object-oriented program in Appendix B.

```
#include <stdio.h>
#include <stdlib.h>



   /*
    * We use the following enumeration type to ``tag'' shapes.
    * The first field of each shape struct stores what particular
    * kind of shape it is.
    */
enum ShapeTag {Circle, Rectangle};



   /*
    * The following struct Pt and functions newPt and copyPt are
    * used in the implementations of the Circle and Rectangle
    * shapes below.
    */
struct Pt {
   float x;
   float y;
};

struct Pt* newPt(float xval, float yval) {
   struct Pt* p = (struct Pt *)malloc(sizeof(struct Pt));
   p->x = xval;
   p->y = yval;
   return p;
};

struct Pt* copyPt(struct Pt* p) {
   struct Pt* q = (struct Pt *)malloc(sizeof(struct Pt));
   q->x = p->x;
   q->y = p->y;
   return q;
};


   /*
    * The Shape struct provides a flag that is used to get some static
    * type checking in the operation functions (center, move, rotate,
    * and print) below.
    */
struct Shape {
   enum ShapeTag tag;
};


   /*
    * The following Circle struct is our representation of a circle.
    * The first field is a type tag to indicate that this struct
    * represents a circle.  The second field stores the circle's
    * center point and the third field holds its radius.
    */
struct Circle {
   enum ShapeTag  tag;
   struct Pt*     center;
   float          radius;
};
```

```
   /*
    * The function newCircle creates a Circle struct from a given
    * center point and radius.  It sets the type tag to ‘‘Circle.’’
    */
struct Circle* newCircle(struct Pt* cp, float r) {
   struct Circle* c = (struct Circle*)malloc(sizeof(struct Circle));
   c->center=copyPt(cp);
   c->radius=r;
   c->tag=Circle;
   return c;
};


   /*
    * The function deleteCircle frees resources used by a Circle.
    */
void deleteCircle(struct Circle* c) {
   free (c->center);
   free (c);
};


   /*
    * The following Rectangle struct is our representation of a rectangle.
    * The first field is a type tag to indicate that this struct
    * represents a rectangle.  The next two fields store the rectangle’s
    * top-left and bottom-right corner points.
    */
struct Rectangle {
   enum ShapeTag  tag;
   struct Pt*     topleft;
   struct Pt*     botright;
};


   /*
    * The function newRectangle creates a rectangle in the location
    * specified by parameters tl and br.  It sets the type tag to
    * ``Rectangle.''
    */
struct Rectangle* newRectangle(struct Pt* tl, struct Pt* br)  {
   struct Rectangle* r = (struct Rectangle*)malloc(sizeof(struct Rectangle));
   r->topleft=copyPt(tl);
   r->botright=copyPt(br);
   r->tag=Rectangle;
   return r;
};


   /*
    * The function deleteRectangle frees resources used by a Rectangle.
    */
void deleteRectangle(struct Rectangle* r) {
   free (r->topleft);
   free (r->botright);
   free (r);
};


   /*
    * The center function returns the center point of whatever shape
    * it is passed.  Because the computation depends on whether the
    * shape is a Circle or a Rectangle, the function consists of a
    * switch statement that branches according to the type tag stored
    * in the shape s.  If the tag is Circle, for instance, we know
    * the parameter is really a circle struct and hence that it has
    * a ``center'' component which we can return.  Note that we need
    * to insert a typecast to instruct the compiler that we have a
    * circle and not just a shape.  Note also that this program
    * organization assumes that the type tags in the struct are
    * set correctly.  If some programmer incorrectly modifies a type tag
    * field, the program will no longer work and the problem cannot
    * be detected at compile time because of the typecasts.
    */
```

```
struct Pt* center (struct Shape* s) {
   switch (s->tag) {
    case Circle: {
       struct Circle* c = (struct Circle*) s;
       return copyPt(c->center);
    };
    case Rectangle: {
       struct Rectangle* r = (struct Rectangle*) s;
       return newPt((r->botright->x - r->topleft->x)/2,
                    (r->botright->x - r->topleft->x)/2);
    };
   };
};


   /*
    * The move function receives a Shape parameter s and moves it
    * dx units in the x-direction and dy units in the y-direction.
    * Because the code to move a Shape depends on the kind of shape,
    * this function inspects the Shape's type tag field within a switch
    * statement.  Within the individual cases, typecasts are used to
    * convert the generic shape parameter to a Circle or Rectangle as
    * appropriate.
    */
void move (struct Shape* s,float dx, float dy) {
   switch (s->tag) {
    case Circle: {
         struct Circle* c = (struct Circle*) s;
         c->center->x    += dx;
         c->center->y    += dy;
      };
      break;
    case Rectangle: {
       struct Rectangle* r = (struct Rectangle*) s;
       r->topleft->x      += dx;
       r->topleft->y      += dy;
       r->botright->x     += dx;
       r->botright->y     += dy;
    };
   };
};


   /*
    * The rotate function rotates the shape s ninety degrees.  Like
    * the center and move functions, this code uses a switch statement
    * that checks the type of shape being manipulated.
    */
void rotate (struct Shape* s) {
   switch (s->tag) {
    case Circle:
         /* Rotating a circle is not a very interesting operation! */
      break;
    case Rectangle: {
         struct Rectangle* r = (struct Rectangle*)s;
         float d = ((r->botright->x - r->topleft->x) -
                    (r->topleft->y - r->botright->y))/2.0;
         r->topleft->x  += d;
         r->topleft->y  += d;
         r->botright->x -= d;
         r->botright->y -= d;
      };
      break;
   };
};


   /*
    * The print function outputs a description of its Shape parameter.
    * This function again selects its processing based on the type tag
    * stored in the Shape struct.
    */
void print (struct Shape* s) {
   switch (s->tag) {
    case Circle: {
         struct Circle* c = (struct Circle*) s;
         printf("circle at  %.1f  %.1f  radius %.1f \n",
                c->center->x, c->center->y, c->radius);
      };
      break;
    case Rectangle: {
         struct Rectangle* r = (struct Rectangle*) s;
         printf("rectangle at  %.1f  %.1f   %.1f  %.1f \n",
                r->topleft->x, r->topleft->y,
                r->botright->x, r->botright->y);
      };
      break;
   };
};


   /*
    * The body of this program just tests some of the above functions.
    */
void main() {
   struct Pt* origin = newPt(0,0);
   struct Pt* p1     = newPt(0,2);
   struct Pt* p2     = newPt(4,6);

   struct Shape* s1  = (struct Shape*)newCircle(origin,2);
   struct Shape* s2  = (struct Shape*)newRectangle(p1,p2);

   print(s1);
   print(s2);

   rotate(s1);
   rotate(s2);

   move(s1,1,1);
   move(s2,1,1);
   print(s1);
   print(s2);

   deleteCircle((struct Circle*)s1);
   deleteRectangle((struct Rectangle*)s2);

   free(origin);
   free(p1);
   free(p2);
};
```

Appendix B

Shape Program: Object-Oriented Version

```
#include <stdio.h>

   // (The following is a running C++ program, but it does not represent
   // an ideal C++ implementation.  The code has been kept simple so
   // that it can be understood by readers who are not well-versed in C++).


   // The following class Pt is used by the shape objects below.  Since
   // Pt is a class in this version of the program, the ‘‘newPt’’ and
   // ‘‘copyPt’’ functions may be implemented as class member functions.
   // For readability, we have in-lined the function definitions and
   // named both of these functions ‘‘Pt’’; these overloaded functions
   // are differentiated by the types of their arguments.
class Pt {
 public:
   Pt(float xval, float yval) {
      x = xval;
      y = yval;
   };

   Pt(Pt* p) {
      x = p->x;
      y = p->y;
   };
   float x;
   float y;
};


   // Class Shape is an example of a ``pure abstract base class,''
   // which means that it exists solely to provide an interface to
   // classes derived from it.  Since it provides no implementations
   // for the methods center, move, rotate, and print, no ``shape''
   // objects can be created.  Instead, we use this class as a base
   // class.  Our circle and rectangle shapes will be derived from
   // it.  This class is useful because it allows us to write
   // functions that expect ``shape'' objects as arguments.  Since
   // our circles and rectangles are subtypes of shape, we may pass
   // them to such functions in a type-safe way.
class Shape {
public:
  virtual Pt* center()=0;
  virtual void move(float dx, float dy)=0;
  virtual void rotate()=0;
  virtual void print()=0;
};


   // Class Circle consolidates the center, move, rotate, and print
   // functions for circles.  It also contains the object constructor
   // ``Circle,'' corresponding to the function ``newCircle'' and the
   // object destructor ``~Circle, corresponding to the function
   // ``deleteCircle'' from the typecase version.  Note that the
   // compiler guarantees that the Circle's methods are only called on
   // objects of type Circle.  The programmer does not need to keep an
   // explicit tag field in the object.
class Circle : public Shape {
 public:
   Circle(Pt* cn, float r) {
      center_ = new Pt(cn);
      radius_ = r;
   };

   virtual ~Circle() {
      delete center_;
   };

   virtual Pt* center() {
      return new Pt(center_);
   };

   void move(float dx, float dy) {
      center_->x += dx;
      center_->y += dy;
   };

   void rotate() {
         /* Rotating a circle is not a very interesting operation! */
   };

   void print() {
      printf("circle at  %.1f  %.1f  radius %.1f \n",
             center_->x, center_->y, radius_);
   };

 private:
   Pt*   center_;
   float radius_;
};


   // Class Rectangle consolidates the center, move, rotate, and print
   // functions for rectangles.  It also contains the object constructor
   // ``Rectangle,'' corresponding to the function ``newRectangle'' and the
   // object destructor ``~Rectangle, corresponding to the function
   // ``deleteRectangle'' from the typecase version.  Note that the
   // compiler guarantees that the Rectangle's methods are only called on
   // objects of type Rectangle.  The programmer does not need to keep an
   // explicit tag field in the object.
class Rectangle : public Shape {
 public:
   Rectangle(Pt* tl, Pt* br) {
      topleft_  = new Pt(tl);
      botright_ = new Pt(br);
   };

   virtual ~Rectangle() {
      delete topleft_;
      delete botright_;
   };

   Pt* center() {
      return new Pt((botright_->x - topleft_->x)/2,
                    (botright_->x - topleft_->x)/2);
   };

   void move(float dx,float dy) {
      topleft_->x  += dx;
      topleft_->y  += dy;
      botright_->x += dx;
      botright_->y += dy;
   };

   void rotate() {
      float d = ((botright_->x - topleft_->x) -
                 (topleft_->y - botright_->y))/2.0;
      topleft_->x  += d;
      topleft_->y  += d;
      botright_->x -= d;
      botright_->y -= d;
   };

   void print () {
      printf("rectangle coordinates  %.1f  %.1f   %.1f  %.1f \n",
          topleft_->x, topleft_->y,
          botright_->x, botright_->y);
  };

 private:
  Pt* topleft_;
  Pt* botright_;
};



  /*
   * The body of this program just tests some of the above functions.
   */
void main() {
  Pt* origin = new Pt(0,0);
  Pt* p1     = new Pt(0,2);
  Pt* p2     = new Pt(4,6);

  Shape* s1 = new Circle(origin, 2 );
  Shape* s2 = new Rectangle(p1, p2);

  s1->print();
  s2->print();

  s1->rotate();
  s2->rotate();

  s1->move(1,1);
  s2->move(1,1);

  s1->print();
  s2->print();

  delete s1;
  delete s2;

  delete origin;
```

```
   delete p1;
   delete p2;
}
```

Appendix C

Full Formal System

In this appendix, we summarize the syntax of the formal system presented in Chapter 6.

Expressions

$$\begin{aligned} e ::= \quad & x \mid c \mid \lambda x.\, e \mid e_1 e_2 \mid \\ & \langle\rangle \mid e \Leftarrow m \mid \langle e_1 \leftarrow m = e_2 \rangle \mid \langle e_1 \longleftarrow\!\!\!+\; m = e_2 \rangle \mid \\ & \{\!|\, r <:_w R :: \kappa = R',\ e \,|\!\} \mid \\ & \mathit{Abstype}\ r <:_w R :: \kappa\ \mathit{with}\ x : \tau\ \mathit{is}\ e_1\ \mathit{in}\ e_2 \end{aligned}$$

Types

$$\begin{aligned} \sigma ::= \quad & \tau \mid \exists (r <:_w R :: \kappa) \tau \\ \tau ::= \quad & t \mid \tau_1 \rightarrow \tau_2 \mid \mathbf{pro}\, t.R \mid \mathbf{obj}\, t.R \end{aligned}$$

Rows

$$R ::= \quad r \mid \langle\!\langle\rangle\!\rangle \mid \langle\!\langle R \mid m{:}\tau \rangle\!\rangle \mid \lambda t.\, R \mid R\tau$$

Variance Annotations

$$\begin{aligned} b ::= \quad & + \mid - \mid o \\ a ::= \quad & b \mid ? \end{aligned}$$

Variance Sets

$$V ::= \quad \{\vec{t^b}\}$$

Method Absence Annotations

$$M ::= \quad \{\vec{m}\}$$

Kinds

$$\begin{aligned} k ::= \quad & V \mid \kappa \\ S ::= \quad & T^a \qquad \text{Symbol } T \text{ is a terminal.} \\ \kappa ::= \quad & S \rightarrow \nu \mid \nu \\ \nu ::= \quad & (M; V) \end{aligned}$$

Contexts

$$, \; ::= \quad \epsilon \,|\, , \, x{:}\tau \,|\, , \, t{:}V \,|\, , \, r <:_w R :: \kappa$$

C.1 Subtyping Annotation

$B ::= \; w \,|\, w, d \,| B_1 + B_2$ $\qquad$ B indicates row subtyping forms.

Here, $w < w, d$ and $+$ denotes the least upper bound of B_1 and B_2 with respect to this ordering.

C.2 Judgment Forms

$, \vdash *$	well-formed context
$, \vdash \sigma{:}V$	well-formed type with variance V
$, \vdash R :: \kappa$	row has kind
$, \vdash \tau_1 <: \tau_2$	type τ_1 subtype of τ_2
$, \vdash R_1 <:_B R_2$	row R_1 is a B-subtype of R_2
$, \vdash e{:}\sigma$	term e has type σ

C.3 Judgment Shorthands

We will use the meta-judgment $, \vdash A$ to range over all of the above judgments. In addition, we use the meta-variable U to range over types τ and rows R. The meta-judgment $, \vdash U{:}-\gamma$ represents judgments of the form $, \vdash \tau{:}V$ and $, \vdash R :: \kappa$. Similarly, meta-judgment $, \vdash U_1 <:_{(B)} U_2$ represents the judgments $, \vdash \tau_1 <: \tau_2$ and $, \vdash R_1 <:_B R_2$. Finally, meta-judgment $, \vdash U_1 \cong_{(B)} U_2$ is short for the two judgments $, \vdash U_1 <:_{(B)} U_2$ and $, \vdash U_2 <:_{(B)} U_1$. We will also use the syntax $\mathbf{probj}\, t\mathbf{.}R$ as shorthand for either $\mathbf{pro}\, t\mathbf{.}R$ or $\mathbf{obj}\, t\mathbf{.}R$.

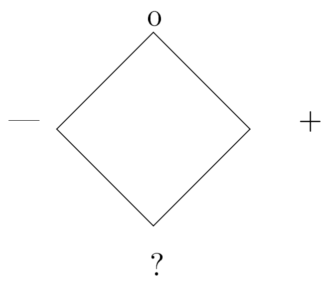
C.4 Context Access Functions

- We use the function $dom(,)$ to denote the set of row, type, and expression variables that are listed in context $,$.
- We use the function $TVar(,)$ to denote the set of type variables t such that the type assumptions $t{:}\,\{t^+\} \in \, ,$.

C.5 Ordering on Variance Annotations

$$+ \leq o,\ - \leq o,\ ? \leq -,\ ? \leq +$$

Pictorially,



We will use the notation $\overline{b}$ to indicate the complement of annotation b with respect to the above ordering. In other words, $\overline{o} = o$, $\overline{+} = -$, $\overline{-} = +$, and $\overline{?} = ?$.

C.6 Operations on Variance Sets

$$\begin{array}{lcl}
Var(t, V) & = & \begin{cases} b & \text{if } t^b \in V \\ ? & \text{if } t^{b'} \notin V \text{ for all } b'. \end{cases} \\[2ex]
D(V_1, \ldots, V_n) & = & \{t \,|\, t^b \in V_i \text{ for some } b{,}i \in 1, \ldots, n\} \\[2ex]
GVar(t, V_1, \ldots, V_n) & = & lub\{Var(t, V_1), \ldots, Var(t, V_n)\} \\[2ex]
Merge(V_1, \ldots, V_n) & = & \{t^{GVar(t, V_1, \ldots, V_n)} \,|\, t \in D(V_1, \ldots, V_n)\} \\[2ex]
V \backslash t & = & \{t_1^b \,|\, t \neq t_1 \text{ and } t_1^b \in V\} \\[2ex]
\overline{V} & = & \{t^{\overline{b}} \,|\, t^b \in V\} \\[2ex]
Invar(V) & = & \{t^o \,|\, t \in D(V)\}
\end{array}$$

C.7 Generalized Variance

$$\begin{array}{lcl}
Var(t, (M;\, V)) & = & Var(t, V) \\
Var(t, T^a \rightarrow \nu) & = & Var(t, \nu) \\
Var(V) & = & V \\
Var(M;\, V) & = & V \\
Var(T^a \rightarrow \nu) & = & T^a \rightarrow Var(\nu)
\end{array}$$

C.8 Variance Shorthand

We will use the notation $\kappa_{min(\Gamma)}$ as a shorthand for the kind $T^o \rightarrow (\emptyset; Invar(TVar(,)))$.

C.9 Variance Substitutions

$$[V_2/t]V_1 = \begin{cases} Merge(V_1', V_2) & \text{if } V_1 = V_1', t^+ \\ Merge(V_1', \overline{V_2}) & \text{if } V_1 = V_1', t^- \\ Merge(V_1', Invar(V_2)) & \text{if } V_1 = V_1', t^o \\ V_1 & \text{if } t \notin D(V_1) \end{cases}$$

C.10 Ordering on Kinds

$$\begin{array}{rcl} V_1 \leq V_2 & \text{iff} & \forall t, Var(t, V_2) \leq Var(t, V_1) \\ (M_1; V_1) \leq (M_2; V_2) & \text{iff} & V_1 \leq V_2 \\ T^{b'} \rightarrow \nu_1 \leq T^b \rightarrow \nu_2 & \text{iff} & b \leq a \text{ and } \nu_1 \leq \nu_2 \end{array}$$

Appendix D

Typing rules

This appendix summarizes the typing rules given in Section 6.2.

D.1 Context Rules

(*start*)

$$\frac{}{\epsilon \vdash *}$$

(*type var*)

$$\frac{\begin{array}{c} , \vdash * \\ t \notin dom(,\ ) \end{array}}{,\ , t\colon \{t^{+}\} \vdash *}$$

(*row var*)

$$\frac{\begin{array}{c} , \vdash R_1 :: S_1 \to (M_1;\, V_1) \\ S_0 \to (M_0;\, V_0) \leq S_1 \to (M_1;\, V_1) \\ D(V_0) \subseteq dom(,\ ) \qquad M_0 \subseteq M_1 \\ r \notin dom(,\ ) \end{array}}{,\ , (r <:_w R_1 :: S_0 \to (M_0;\, V_0)) \vdash *}$$

(*exp var*)

$$\frac{\begin{array}{c} , \vdash \tau : V \\ x \notin dom(,\ ) \end{array}}{,\ , x\colon \tau \vdash *}$$

$$(\textit{weakening}) \qquad \frac{\begin{array}{c} ,_1, ,_2 \vdash A \\ ,_1, a, ,_2 \vdash * \end{array}}{,_1, a, ,_2 \vdash A}$$

$$\text{where } a ::= x{:}\tau \mid t{:}V \mid r <:_w R :: \kappa$$

D.2 Rules for Type Expressions

$$(\textit{type proj}) \qquad \frac{\begin{array}{c} , \vdash * \\ t : \{t^+\} \in , \end{array}}{, \vdash t : \{t^+\}}$$

$$(\textit{type arrow}) \qquad \frac{\begin{array}{c} , \vdash \tau_1 : V_1 \\ , \vdash \tau_2 : V_2 \end{array}}{, \vdash \tau_1 \rightarrow \tau_2 : Merge(\overline{V_1}, V_2)}$$

$$(\textit{pro}) \qquad \frac{, , t{:}\{t^+\} \vdash R :: (M;\, V)}{, \vdash \mathbf{pro}\, t\mathbf{.}R : Invar(V \setminus t)}$$

$$(\textit{cov object}) \qquad \frac{\begin{array}{c} , , t{:}\{t^+\} \vdash R :: (M;\, V) \\ Var(t, V) \in \{+, ?\} \end{array}}{, \vdash \mathbf{obj}\, t\mathbf{.}R : V \setminus t}$$

$$(\textit{exist}) \qquad \frac{, , r <:_w :: S \rightarrow (M;\, V_2) \vdash \tau : V_1}{, \vdash \exists (r <:_w :: S \rightarrow (M;\, V_2))\tau : Merge(V_1, V_2)}$$

D.3 Rules for Row Expressions

$$(\textit{row proj}) \qquad \frac{\begin{array}{c} , \vdash * \\ r <:_w R :: \kappa \in dom(,) \end{array}}{, \vdash r :: \kappa}$$

(*empty row*)

$$\frac{,\ \vdash *}{,\ \vdash \langle\!\langle\rangle\!\rangle :: (M;\ \emptyset)}$$

(*row label*)

$$\frac{\begin{array}{c},\ \vdash R :: S^i \to (M;\ V)\\ N \subseteq M \qquad i \in \{0,1\}\end{array}}{,\ \vdash R :: S^i \to (N;\ V)}$$

(*row fn abs*)

$$\frac{,\ ,t{:}\,\{t^+\} \vdash R :: (M;\ V)}{,\ \vdash \lambda t.\, R :: T^{Var(t,V)} \to (M;\ V\backslash t)}$$

(*row fn app cov*)

$$\frac{\begin{array}{c},\ \vdash R :: T^+ \to (M;\ V_1)\\ ,\ \vdash \tau : V_2\end{array}}{,\ \vdash R\tau :: (M;\ Merge(V_1, V_2))}$$

(*row fn app contra*)

$$\frac{\begin{array}{c},\ \vdash R :: T^- \to (M;\ V_1)\\ ,\ \vdash \tau : V_2\end{array}}{,\ \vdash R\tau :: (M;\ Merge(V_1, \overline{V_2}))}$$

(*row fn app inv*)

$$\frac{\begin{array}{c},\ \vdash R :: T^o \to (M;\ V_1)\\ ,\ \vdash \tau : V_2\end{array}}{,\ \vdash R\tau :: (M;\ Merge(V_1, Invar(V_2)))}$$

(*row fn app vac*)

$$\frac{\begin{array}{c},\ \vdash R :: T^? \to (M;\ V_1)\\ ,\ \vdash \tau : V_2\end{array}}{,\ \vdash R\tau :: (M;\ V_1)}$$

(*row ext*)

$$\frac{\begin{array}{c},\ \vdash R :: (\{\vec{m}, m\}, V_1)\\ ,\ \vdash \tau : V_2\end{array}}{,\ \vdash \langle\!\langle R \mid m{:}\,\tau\rangle\!\rangle :: (\{\vec{m}\};\ Merge(V_1, V_2))}$$

D.4 Subtyping Rules for Types

$(<:\ \mathit{type\ refl})$

$$\frac{,\ \vdash \tau : V}{,\ \vdash \tau <: \tau}$$

$(<:\rightarrow)$

$$\frac{\begin{array}{c},\ \vdash \tau_1' <: \tau_1 \\ ,\ \vdash \tau_2 <: \tau_2'\end{array}}{,\ \vdash \tau_1 \rightarrow \tau_2 <: \tau_1' \rightarrow \tau_2'}$$

$(<:\ \mathit{obj})$

$$\frac{\begin{array}{c},\ ,\ t : \{t^+\} \vdash R_1 <:_B R_2 \\ ,\ \vdash \mathbf{probj}\, t.R_1 : V_1 \\ ,\ ,\ t : \{t^+\} \vdash R_2 :: (M; V_2) \qquad Var(t, V_2) \in \{?, +\}\end{array}}{,\ \vdash \mathbf{probj}\, t.R_1 <: \mathbf{obj}\, t.R_2}$$

$(<:\ \mathit{convert})$

$$\frac{\begin{array}{c},\ ,t : \{t^+\} \vdash R_1 \cong_w R_2 \\ ,\ ,t : \{t^+\} \vdash R_i :: (M_i; V_i) \qquad i \in \{1, 2\}\end{array}}{,\ \vdash \mathbf{pro}\, t.R_1 <: \mathbf{pro}\, t.R_2}$$

$(<:\ \mathit{type\ trans})$

$$\frac{\begin{array}{c},\ \vdash \tau_1 <: \tau_2 \\ ,\ \vdash \tau_2 <: \tau_3\end{array}}{,\ \vdash \tau_1 <: \tau_3}$$

D.5 Subtyping Rules for Rows

$(<:\ \mathit{row\ refl})$

$$\frac{,\ \vdash R :: \kappa}{,\ \vdash R <:_B R}$$

$(\mathit{row\ proj\ bound})$

$$\frac{\begin{array}{c},\ \vdash * \\ r <:_w R :: \kappa \in\ ,\end{array}}{,\ \vdash r <:_w R}$$

$(<: \lambda)$
$$\frac{\begin{array}{c} ,\ ,\ t : \{t^{+}\} \vdash R_1 <:_B R_2 \\ ,\ ,\ t : \{t^{+}\} \vdash R_2 :: \nu \end{array}}{,\ \vdash \lambda t.\, R_1 <:_B \lambda t.\, R_2}$$

$(<: \mathit{app\ cong})$
$$\frac{\begin{array}{c} ,\ \vdash R_1 <:_B R_2 \\ ,\ \vdash R_2 :: T^{a} \to \nu \\ ,\ \vdash \tau_1 \cong \tau_2 \end{array}}{,\ \vdash R_1 \tau_1 <:_B R_2 \tau_2}$$

$(<: \mathit{app\ cov})$
$$\frac{\begin{array}{c} ,\ \vdash R_1 <:_B R_2 \\ ,\ \vdash R_2 :: T^{+} \to \nu \\ ,\ \vdash \tau_1 <: \tau_2 \end{array}}{,\ \vdash R_1 \tau_1 <:_{B+d} R_2 \tau_2}$$

$(<: \mathit{app\ contra})$
$$\frac{\begin{array}{c} ,\ \vdash R_1 <:_B R_2 \\ ,\ \vdash R_2 :: T^{-} \to \nu \\ ,\ \vdash \tau_2 <: \tau_1 \end{array}}{,\ \vdash R_1 \tau_1 <:_{B+d} R_2 \tau_2}$$

$(<: \mathit{app\ vac})$
$$\frac{\begin{array}{c} ,\ \vdash R_1 <:_B R_2 \\ ,\ \vdash R_2 :: T^{?} \to \nu \\ ,\ \vdash \tau_1 : V_1 \qquad ,\ \vdash \tau_2 : V_2 \end{array}}{,\ \vdash R_1 \tau_1 <:_B R_2 \tau_2}$$

$(<: \mathit{cong})$
$$\frac{\begin{array}{c} ,\ \vdash R_1 <:_B R_2 \\ ,\ \vdash \tau_1 \cong \tau_2 \\ ,\ \vdash \langle\!\langle R_i \,|\, m : \tau_i \rangle\!\rangle :: \nu_i \qquad i \in \{1, 2\} \end{array}}{,\ \vdash \langle\!\langle R_1 \,|\, m : \tau_1 \rangle\!\rangle <:_B \langle\!\langle R_2 \,|\, m : \tau_2 \rangle\!\rangle}$$

$(<: d)$
$$\frac{\begin{array}{c} ,\ \vdash R_1 <:_B R_2 \qquad ,\ \vdash \tau_1 <: \tau_2 \\ ,\ \vdash \langle\!\langle R_i \,|\, m : \tau_i \rangle\!\rangle :: \nu_i \qquad i \in \{1, 2\} \end{array}}{,\ \vdash \langle\!\langle R_1 \,|\, m : \tau_1 \rangle\!\rangle <:_{B+d} \langle\!\langle R_2 \,|\, m : \tau_2 \rangle\!\rangle}$$

$(<:w)$

$$\frac{, \vdash R_1 <:_B R_2 \qquad , \vdash \langle\!\langle R_1 \,|\, m\!:\!\tau \rangle\!\rangle :: \nu}{, \vdash \langle\!\langle R_1 \,|\, m\!:\!\tau \rangle\!\rangle <:_{B+w} R_2}$$

$(<:row\ trans)$

$$\frac{, \vdash R_1 <:_B R_2 \qquad , \vdash R_2 <:_{B'} R_3}{, \vdash R_1 <:_{B+B'} R_3}$$

D.6 Type and Row Equality Rules

$(row\ \beta)$

$$\frac{, \vdash R :: \kappa \quad R \to_\beta R'}{, \vdash R' :: \kappa}$$

$(type\ \beta)$

$$\frac{, \vdash \tau : V \quad \tau \to_\beta \tau'}{, \vdash \tau' : V}$$

$(type\ eq)$

$$\frac{, \vdash e : \sigma \quad \sigma \leftrightarrow_\beta \sigma' \quad , \vdash \sigma' : V}{, \vdash e : \sigma'}$$

$(<:\beta\ right)$

$$\frac{, \vdash R_1 <:_B (\lambda t.\, R_2)\tau_2}{, \vdash R_1 <:_B [\tau_2/t]R_2}$$

$(<:\beta\ left)$

$$\frac{, \vdash (\lambda t.\, R_1)\tau_1 <:_B R_2}{, \vdash [\tau/t]R_1 <:_B R_2}$$

D.7 Rules for Assigning Types to Terms

$(exp\ proj)$

$$\frac{, \vdash * \qquad x : \tau \in ,}{, \vdash x : \tau}$$

$(subsumption)$
$$\frac{,\ \vdash e : \tau_1 \quad ,\ \vdash \tau_1 <: \tau_2}{,\ \vdash e : \tau_2}$$

$(exp\ abs)$
$$\frac{,\ , x{:}\tau_1 \vdash e : \tau_2}{,\ \vdash \lambda x.\, e : \tau_1 \rightarrow \tau_2}$$

$(exp\ app)$
$$\frac{,\ \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad ,\ \vdash e_2 : \tau_1}{,\ \vdash e_1\, e_2 : \tau_2}$$

$(empty\ pro)$
$$\frac{,\ \vdash *}{,\ \vdash \langle\rangle : \mathbf{pro}\, t \boldsymbol{.} \langle\!\langle\rangle\!\rangle}$$

$(pro\ ext)$
$$\frac{\begin{array}{l},\ \vdash e_1 : \mathbf{pro}\, t \boldsymbol{.} R \\ ,\ , t{:}\{t^+\} \vdash R :: (\{m\}; V) \\ ,\ , I_r \vdash e_2 : [\mathbf{pro}\, t \boldsymbol{.} rt/t](t \rightarrow \tau) \qquad r \notin V(\tau)\end{array}}{,\ \vdash \langle e_1 \leftarrow\!\!+\, m = e_2\rangle : \mathbf{pro}\, t \boldsymbol{.} \langle\!\langle R \,|\, m{:}\tau \rangle\!\rangle}$$
where $I_r = r <:_w \lambda t. \langle\!\langle R \,|\, m : \tau \rangle\!\rangle :: \kappa_{min(\Gamma)}$.

$(pro\ over)$
$$\frac{\begin{array}{l},\ \vdash e_1 : \mathbf{pro}\, t \boldsymbol{.} R \\ ,\ , t{:}\{t^+\} \vdash R <:_w \langle\!\langle m{:}\tau \rangle\!\rangle \\ ,\ , I_r \vdash e_2 : [\mathbf{pro}\, t \boldsymbol{.} rt/t](t \rightarrow \tau)\end{array}}{,\ \vdash \langle e_1 \leftarrow m{=}e_2\rangle : \mathbf{pro}\, t \boldsymbol{.} R}$$
where $I_r = r <:_w \lambda t.\, R :: \kappa_{min(\Gamma)}$.

$(probj \Leftarrow)$
$$\frac{\begin{array}{c},\ \vdash e : \mathbf{probj}\, t \boldsymbol{.} R \\ ,\ , t{:}\{t^+\} \vdash R <:_w \langle\!\langle m{:}\tau \rangle\!\rangle\end{array}}{,\ \vdash e \Leftarrow m : [\mathbf{probj}\, t \boldsymbol{.} R/t]\tau}$$

$$(\exists <:\ \mathit{intro})\qquad \frac{\begin{array}{c}, \vdash R_1 :: \kappa \\ , \vdash R_1 <:_w R \\ , \vdash e : [R_1/r]\tau\end{array}}{, \vdash \{\!|r <:_w R :: \kappa = R_1,\ e|\!\} : \exists(r <:_w R :: \kappa)\tau}$$

$$(\exists <:\ \mathit{elim})\qquad \frac{\begin{array}{c}, \vdash e_1 : \exists(r <:_w R :: \kappa)\tau \\ , , r <:_w R :: \kappa,\ x{:}\tau \vdash e_2 : \tau_2 \\ , \vdash \tau_2 : V\end{array}}{, \vdash \mathit{Abstype}\ r <:_w R :: \kappa\ \mathit{with}\ x : \tau\ \mathit{is}\ e_1\ \mathit{in}\ e_2 : \tau_2}$$

Appendix E

Operational Semantics

Axioms:

$$\begin{array}{llcl}
(\textit{switch ext ov}) & \langle\langle e_1 \leftarrow m_2{=}e_2\rangle \leftarrow\!\!+\, m_3{=}e_3\rangle & \xrightarrow{book} & \langle\langle e_1 \leftarrow\!\!+\, m_3{=}e_3\rangle \leftarrow m_2{=}e_2\rangle \\
(\textit{perm ov ov}) & \langle\langle e_1 \leftarrow m_2{=}e_2\rangle \leftarrow m_3{=}e_3\rangle & \xrightarrow{book} & \langle\langle e_1 \leftarrow m_3{=}e_3\rangle \leftarrow m_2{=}e_2\rangle \\
(\textit{add ov}) & \langle e_1 \leftarrow\!\!+\, m_2{=}e_2\rangle & \xrightarrow{book} & \langle\langle e_1 \leftarrow\!\!+\, m_2{=}e_2\rangle \leftarrow m_2{=}e_2\rangle \\
(\textit{cancel ov ov}) & \langle\langle e_1 \leftarrow m_3{=}e_2\rangle \leftarrow m_3{=}e_3\rangle & \xrightarrow{book} & \langle e_1 \leftarrow m_3{=}e_3\rangle \\
(\beta) & (\lambda x.\, e_1)e_2 & \xrightarrow{eval} & [e_2/x]e_1 \\
(\Leftarrow) & \langle e_1 \leftarrow\!\!\circ\, m{=}e_2\rangle \Leftarrow m & \xrightarrow{eval} & e_2\langle e_1 \leftarrow\!\!\circ\, m{=}e_2\rangle \\
 & & & \text{where } \leftarrow\!\!\circ \text{ may be either } \leftarrow\!\!+ \text{ or } \leftarrow. \\
(\textit{Abstype}) & \textit{Abstype } r <:_w R :: \kappa \textit{ with } x{:}\tau & \xrightarrow{eval} & [R'/r, e_1/x]e_2 \\
 & \textit{is } \{\!|r <:_w R :: \kappa = R',\ e_1|\!\} \textit{ in } e_2 & &
\end{array}$$

Evaluation Contexts:

$$\begin{array}{lcl}
C[\,] & ::= & [\,] \mid \lambda x.\, C[\,] \mid C[\,]\ e_2 \mid e_1\ C[\,] \mid C[\,] \Leftarrow m \mid \\
 & & \langle C[\,] \leftarrow\!\!\circ\, m{=}e_2\rangle \mid \langle e_1 \leftarrow\!\!\circ\, m{=}C[\,]\rangle \mid \\
 & & \{\!|r <:_w R{::}\kappa = R',\ C[\,]\,|\!\} \\
 & & \textit{Abstype } r <:_w R :: \kappa \textit{ with } x : \tau \textit{ is } C[\,] \textit{ in } e_2 \\
 & & \textit{Abstype } r <:_w R :: \kappa \textit{ with } x : \tau \textit{ is } e_1 \textit{ in } C[\,]
\end{array}$$

Congruence Closure:

In the following rule, we write $\xrightarrow{axm}$ to denote either $\xrightarrow{eval}$ or $\xrightarrow{book}$.

$$\frac{e \xrightarrow{axm} e'}{C[e] \xrightarrow{ceval} C[e']}$$

Appendix F

Definition of Evaluation Strategy

Axioms

$(getm\ var)$	$get_m(x) = error$
$(getm\ \langle\rangle)$	$get_m(\langle\rangle) = error$
$(getm\ \lambda)$	$get_m(\lambda x.\, e) = error$
$(getm\ \exists)$	$get_m(\{\!\vert r <:_w R :: \kappa = R',\ e_1 \vert\!\}) = error$
$(getm\ \leftarrow)$	$get_m(\langle e_1 \leftarrow m = e_2\rangle) = \langle e_1 \leftarrow m = e_2\rangle$
$(getm\ \leftarrow\!\!+)$	$get_m(\langle e_1 \leftarrow\!\!+\ m = e_2\rangle) = \langle\langle e_1 \leftarrow\!\!+\ m = e_2\rangle \leftarrow m = e_2\rangle$
$(eval\ var)$	$eval(x) = x$
$(eval\ \langle\rangle)$	$eval(\langle\rangle) = \langle\rangle$
$(eval\ \lambda)$	$eval(\lambda x.\, e) = \lambda x.\, e$
$(eval\ \leftarrow)$	$eval(\langle e_1 \leftarrow m = e_2\rangle) = \langle e_1 \leftarrow m = e_2\rangle$
$(eval\ \leftarrow\!\!+)$	$eval(\langle e_1 \leftarrow\!\!+\ m = e_2\rangle) = \langle e_1 \leftarrow\!\!+\ m = e_2\rangle$

(*eval* ∃) $$eval(\{\!|r <:_w R :: \kappa = R',\ e_1|\!\}) = \{\!|r <:_w R :: \kappa = R',\ e_1|\!\}$$

Inference Rules In the following, meta-variable z represents either an expression or *error* and *ev* represents either *eval* or *getm*.

(*ev app err*) $$\frac{eval(e_1) = error}{ev(e_1 e_2) = error}$$

(*ev app*) $$\frac{\begin{array}{c} eval(e_1) = \lambda x.\, e_1' \\ ev([e_2/x]e_1') = z \end{array}}{ev(e_1 e_2) = z}$$

(*ev* ⇐ *err*) $$\frac{get_n(e) = error}{ev(e \Leftarrow n) = error}$$

(*ev* ⇐) $$\frac{\begin{array}{c} get_n(e) = \langle e_1 \leftarrow n = e_2 \rangle \\ ev(e_2 \langle e_1 \leftarrow n = e_2 \rangle) = z \end{array}}{ev(e \Leftarrow n) = z}$$

(*ev* ∃ *err*) $$\frac{eval(e_1) = error}{ev(Abstype\ r <:_w R :: \kappa\ with\ x : \tau\ is\ e_1\ in\ e_2) = error}$$

(*ev* ∃) $$\frac{\begin{array}{c} eval(e_1) = \{\!|r <:_w R :: \kappa = R',\ e_3|\!\} \\ ev([R'/r, e_3/x]e_2) = z \end{array}}{ev(Abstype\ r <:_w R :: \kappa\ with\ x : \tau\ is\ e_1\ in\ e_2) = z}$$

(*getm* ← *err*) $$\frac{\begin{array}{c} n \neq m \\ get_m(e_1) = error \end{array}}{get_m(\langle e_1 \leftarrow n = e_2 \rangle) = error}$$

(*getm* ←) $$\frac{\begin{array}{c} n \neq m \\ get_m(e_1) = \langle e_3 \leftarrow m = e_4 \rangle \end{array}}{get_m(\langle e_1 \leftarrow n = e_2 \rangle) = \langle \langle e_3 \leftarrow n = e_2 \rangle \leftarrow m = e_4 \rangle}$$

$(getm\ \leftarrow\!\!+\ err)$

$$\frac{\begin{array}{c} n \neq m \\ get_m(e_1) = error \end{array}}{get_m(\langle e_1 \leftarrow\!\!+ n = e_2\rangle) = error}$$

$(getm\ \leftarrow\!\!+)$

$$\frac{\begin{array}{c} n \neq m \\ get_m(e_1) = \langle e_3 \leftarrow m = e_4\rangle \end{array}}{get_m(\langle e_1 \leftarrow\!\!+ n = e_2\rangle) = \langle\langle e_3 \leftarrow\!\!+ n = e_2\rangle \leftarrow m = e_4\rangle}$$

Bibliography

[Aba94] M. Abadi. Baby Modula-3 and a theory of objects. *J. Functional Programming*, 4(2):249–283, April 1994. Also appeared as SRC Research Report 95.

[AC95] M. Abadi and L. Cardelli. A theory of primitive objects: Second-order systems. *Science of Computer Programming*, 25(2-3):81–116, December 1995. A preliminary version appeared in the 1994 Proc. of European Symposium on Programming.

[AC96a] M. Abadi and L. Cardelli. An imperative object calculus. *Theory and Practice of Object Systems*, 1(3):151–166, 1996. Earlier version appeared in TAPSOFT '95 proceedings.

[AC96b] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer, 1996.

[AC96c] M. Abadi and L. Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 125(2):78–102, 1996. Earlier version appeared in TACS '94 proceedings, LNCS 789.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language.* Addison Wesley, 1996.

[App92] Apple Computer. *Dylan: An Object-Oriented Dynamic Language.* Apple Computer, 1992.

[AR88] N. Adams and J. Rees. Object-oriented programming in Scheme. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 277–288, July 1988.

[BDMN73] G.M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin.* Studentlitteratur, Box 1717, S-222 01 Lund, Sweden; Auerbach, Philadelphia, 1973.

[BL90] F. Belz and D.C. Luckham. A new approach to prototyping Ada-based hardware/software systems. In *Proc. ACM Tri-Ada'90 Conference*, December 1990.

[BL95] V. Bono and L. Liquori. A subtyping for the Fisher-Honsell-Mitchell lambda calculus of objects. In L. Pacholsky and J. Tiuryn, editors, *Proc. of Int'l Conf. of Computer Science Logic*, pages 16–30, Berlin, June 1995. Springer LNCS 933.

[Boo91] G. Booch. *Object-Oriented Design with Applications.* Benjamin Cummings, Redwood City, CA, 1991.

[Bru92] K. Bruce. The equivalence of two semantic definitions of inheritance in object-oriented languages. In *Proc. Mathematical Foundations of Programming Language Semantics*, pages 102–124, Berlin, 1992. Springer LNCS 598.

[Bru93] K. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Proc. 20th ACM Symp. Principles of Programming Languages*, pages 285–298, 1993.

[BSv95] K. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proc. 9th European Conference on Object-Oriented Programming*, pages 26–51, Aarhus, Denmark, 1995. Springer LNCS 952.

[Bv93] K. Bruce and R. van Gent. TOIL: a new type-safe object-oriented imperative language. Manuscript, 1993.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Special issue devoted to *Symp. on Semantics of Data Types,* Sophia-Antipolis (France), 1984.

[Car95] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, January 1995.

[CDJ$^+$89] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The Modula-3 type system. In *Sixteenth ACM Symp. Principles of Programming Languages*, pages 202–212, 1989.

[CGL95] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995. A preliminary version was presented at the *1992 ACM Conf. on Lisp and Functional Programming.*

[Cha95] C. Chambers. The Cecil language: Specification and rationale. Available electronically from http://www.cs.washington.edu/research/projects/cecil/www/Papers/cecil-spec.html, December 1995.

[CHC90] W.R. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 125–135, January 1990.

[CL90] L. Cardelli and X. Leroy. *Abstract types and the dot notation*, pages 479–504. IFIP State of the Art Reports. North Holland, March 1990. Also appeared as SRC Research Report 56.

[CL94] C. Chambers and G.T. Leavens. Typechecking and modules for multi-methods. In *ACM Conf. Object-oriented Programming: Systems, Languages and Applications*, pages 1–15, October 1994.

[CM91] L. Cardelli and J.C. Mitchell. Operations on records. *Math. Structures in Computer Science*, 1(1):3–48, 1991. Summary in *Math. Foundations of Prog. Lang. Semantics,* Springer LNCS 442, 1990, pp 22–52.

[Com94] A.B. Compagnoni. *Higher-Order Subtyping with Intersection Types.* PhD thesis, Katholieke Universiteit Nijmegen, 1994.

[Coo87] W.R. Cook. A *self*-ish model of inheritance. Manuscript, 1987.

[Coo89a] W.R. Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

[Coo89b] W.R. Cook. A proposal for making Eiffel type-safe. In *European Conf. on Object-Oriented Programming*, pages 57–72, 1989.

[Coo92] W.R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *ACM Conf. Object-oriented Programming: Systems, Languages and Applications*, pages 1–15, 1992.

[CU89] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 146–160, 1989.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[DF96] P. DiBlasio and K. Fisher. A concurrent object calculus. In *CONCUR '96 Proc.*, Pisa, 1996. Springer-Verlag. To appear.

[Dij72] E.W. Dijkstra. Notes on structured programming. In O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming.* Academic Press, 1972.

[ES90] M. Ellis and B. Stroustrop. *The Annotated C^{++} Reference Manual.* Addison-Wesley, 1990.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *ACM Conf. Object-Oriented Programming: Systems, Languages and Applications*, pages 169–184, October 1995.

[FHM94] K. Fisher, F. Honsell, and J.C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. Computing (*formerly *BIT)*, 1:3–37, 1994. Preliminary version appeared in *Proc. IEEE Symp. on Logic in Computer Science,* 1993, 26–38.

[FM95a] K. Fisher and J.C. Mitchell. A delegation-based object calculus with subtyping. In *Proc. 10th Int'l Conf. Fundamentals of Computation Theory (FCT'95)*, pages 42–61. Springer LNCS 965, 1995.

[FM95b] K. Fisher and J.C. Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):189–220, 1995. Preliminary version appeared in TACS '94 proceedings.

[GM94] C.A. Gunter and J.C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming.* MIT Press, Cambridge, MA, 1994.

[GR83] A. Goldberg and D. Robson. *Smalltalk–80: The Language and its Implementation.* Addison Wesley, 1983.

[HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21-st ACM Symp. on Principles of Programming Languages*, 1994.

[HP95] M. Hofmann and B. Pierce. Positive subtyping. In *Proc. 22nd ACM Symp. on Principles of Programming Languages*, pages 186–197, San Francisco, January 1995. To appear in *Information and Computation.*

[KLM94] D. Katiyar, D. Luckham, and J.C. Mitchell. A type system for prototyping languages. In *Proc. 21-st ACM Symp. on Principles of Programming Languages*, 1994.

[KLMM94] D. Katiyar, D. Luckham, S. Meldal, and J.C. Mitchell. Polymorphism and subtyping in interfaces. In *ACM Workshop on Interface Definition Languages*, 1994.

[L$^+$81] B. Liskov et al. *CLU Reference Manual.* Springer LNCS 114, Berlin, 1981.

[Liq96] L. Liquori. An extended theory of primitive objects: First and second order systems. Technical Report CS-23-96, Dipartimento di Informatica, Universitá di Torino, 1996.

[LSAS77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Comm. ACM*, 20:564–576, 1977.

[Mey92] B. Meyer. *Eiffel: The Language.* Prentice-Hall, 1992.

[MHF93] J.C. Mitchell, F. Honsell, and K. Fisher. A lambda calculus of objects and method specialization. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 26–38, 1993.

[Mit86] J.C. Mitchell. Representation independence and data abstraction. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 263–276, January 1986.

[Mit90] J.C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 109–124, January 1990.

[Mit91] J.C. Mitchell. On the equivalence of data representations. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 305–330. Academic Press, 1991.

[MM85] J.C. Mitchell and A.R. Meyer. Second-order logical relations. In *Logics of Programs*, pages 225–236, Berlin, June 1985. Springer-Verlag LNCS 193.

[MMM91] J.C. Mitchell, S. Meldal, and N. Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Proc. 18th ACM Symp. on Principles of Programming Languages*, pages 270–278, January 1991.

[MP88] J.C. Mitchell and G.D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proc. 12th ACM Symp. on Principles of Programming Languages,* 1985.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Nel91] G. Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall Series in Innovative Technology. Prentice Hall, 1991.

[Pie93] B.C. Pierce. Mutable objects. Draft report; available electronically, May 1993.

[PT93] B.C. Pierce and D.N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[PT94] B.C. Pierce and D.N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, 1994. Preliminary version appeared in Principles of Programming Languages, 1993, under the title "Object-Oriented Programming Without Recursive Types".

[RA82] J. Rees and N. Adams. T, a dialect of Lisp, or lambda: The ultimate software tool. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 114–122, August 1982.

[Rey83] J.C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, Amsterdam, 1983.

[RR96] J.H. Reppy and J.G. Riecke. Classes in Object ML via modules, 1996. Presented at FOOL3 workshop.

[Sny86] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proc. ACM Symp. on Object-Oriented Programming Systems, Languages, and Applications*, pages 38–46, October 1986.

[Ste84] G.L. Steele. *Common Lisp: The Language.* Digital Press, 1984.

[Str86] B. Stroustrop. *The C++ Programming Language.* Addison-Wesley, 1986.

[US 80] US Dept. of Defense. *Reference Manual for the Ada Programming Language.* GPO 008-000-00354-8, 1980.

[US91] D. Ungar and R.B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation*, 4(3):187–206, 1991. Preliminary version appeared in *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications,* 1987, 227-241.