PROGRAMMING LANGUAGES FOR SCALABLE SOFTWARE EXTENSION AND COMPOSITION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Nathaniel J. Nystrom

January 2007

PROGRAMMING LANGUAGES FOR SCALABLE SOFTWARE EXTENSION AND COMPOSITION

Nathaniel J. Nystrom, Ph.D.

Cornell University 2007

Large software systems are often constructed by reusing existing code. This dissertation describes several approaches that address the limitations of existing code reuse mechanisms such as class inheritance. The *Polyglot* design pattern enables software systems to be extended in a *scalable* way: the code required to extend the system is proportional to the amount of new functionality provided. This design pattern has been used to implement an extensible compiler framework. *Nested inheritance* is an object-oriented programming language mechanism that supports scalable extensibility in a safer, more natural way than the design pattern approach. Nested inheritance permits modular, type-safe extension of a package (including nested packages and classes), while preserving existing type relationships. *Nested intersection* extends nested intersection to enable composition and extension of two or more packages, combining their types and behavior while resolving conflicts with a relatively small amount of code. Nested intersection is implemented in the language J&. The utility of J& is demonstrated by using it to construct two composable, extensible frameworks: a compiler framework for Java, and a peer-to-peer networking system. Both frameworks support composition of extensions. For example, two compilers adding different, domain-specific features to Java can be composed to obtain a compiler for a language that supports both sets of features.

BIOGRAPHICAL SKETCH

Nathaniel Nystrom was born in Tampa, Florida, in 1973. Following a peripatetic childhood in the southeast United States, England, Germany, and Turkey, he attended Purdue University in West Lafayette, Indiana, earning Bachelor of Science degrees in Computer Science and Mathematics in 1995. After a brief stint at Tektronix in Beaverton, Oregon, he returned to Purdue and received a Master of Science in Computer Science in 1998 under Professor Antony Hosking. He subsequently worked as a software engineer at Hewlett-Packard in Cupertino, California. He then moved to Ithaca, New York, to study at Cornell University from 1999 to 2006. He completed his doctorate in 2006.

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Andrew Myers. Andrew set a high standard and encouraged me to tackle difficult problems. He has been a great teacher and an insightful critic. Andrew has always made himself available to discuss research or just to chat.

During my time at Cornell, I benefited from my interactions with several faculty members, particularly Dexter Kozen, José Martínez, Radu Rugina, Greg Morrisett, Gün Sirer, and Alan Demers. Michael Clarkson Steve Chong, and Xin Qi were great friends and colleagues and contributed much to this thesis. Matthew Fluet, Vicky Weissman, Riccardo Pucella, Jed Liu, Lantian Zheng, Ranveer Chandra, Rama, Hubie Chen, Filip Radliński, Jill Chavez, Tony Faradjian, and David Fang were all wonderful friends and helped keep me sane. Becky Stewart and Stephanie Meik always provided friendly advise with navigating the bureaucracy (and candy too).

At Purdue, Aditya Mathur and Tony Hosking started me along the path towards a career in research. Dave Goldberg and Jesse Cunningham showed me there is beauty in mathematics; Jens Palsberg encouraged my interest in programming languages. John Richardson at Tektronix taught me to program and made me an engineer. Carl Burch and Noubar Partamian at HP and Greg Czajkowski and Laurent Daynès at Sun Labs were great friends and coworkers. I've had many fantastic teachers in my life: thanks to Shannon Connor, Tony Romanello, and John Drake for letting me work ahead; Pamela Stanescu sparked a lifelong passion for history.

Finally, I'd like to thank my parents John and Linda for all of their support, encouragement, and love throughout my education.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Today's software systems consist of millions or tens of millions of lines of code and thousands of interacting functions, modules, and libraries. Developing this code from scratch is infeasible; instead, large software systems are constructed by reusing existing code. Yet, in practice, techniques for code reuse today are often ineffective, messy and unmaintainable, or unsafe.

When implementing new functionality, it is often tempting to copy existing code that performs a similar function and then edit the copy in-place to achieve the desired behavior. Indeed, entire software systems have been implemented this way. This approach is popular because it is both simple and effective; however, "copy and paste" reuse has serious limitations. Over time, both the original code and the code derived from the copy—the *derived code*—may be upgraded with bug fixes and new functionality. Changes to one version of the code are not automatically applied to the other. Developers must carefully patch the derived code with the changes made to the base code, leading to duplication of effort and code maintenance problems. Patching becomes an increasingly arduous task as the two code bases evolve and diverge from each other. Automatic patching tools can help, but are often fragile and can introduce errors. Con-

flicting changes to the base and derived code often must be reconciled by hand, with considerable programmer effort.

Programming languages and design patterns have been developed for extending software without duplicating code, thus avoiding the maintenance problems duplication introduces. In this thesis, we focus on a particular form of reuse: *software extension*. Software extension enhances or refines the behavior of an existing code base without violating its abstractions. Examples of code reuse that violates abstractions include reusing fragments of a procedure body, or reusing a data structure or part of a data structure with different, incompatible types. Abstraction-violating reuse that does not require code duplication is a challenging problem beyond the scope of this work.

Making code extensible requires careful design so that the extension implementer has available the proper *hooks*: interposition points at which new behavior or state can be added. Even with well-structured code, software may not be extensible simply because the right hooks for extension are not available. Design patterns [43] for extensibility structure the code to better expose these hooks to the developer. Programming languages enable more effective reuse by allowing developers to easily create new hooks for extension. For example, in object-oriented languages, methods act as hooks: programmers can extend a class with new functionality by creating a subclass and then overriding methods of the base class. Simply by declaring a method, the programmer of the base system creates a hook that permits the system to be extended with new behavior.

We identify the following requirements for general extension and composition of software systems.

1. **Orthogonal extension.** The base code should be able to be extended with both new data types and new operations on those data types.
2. **Scalability.** Extension of a body of code should require code proportional only to the amount of new functionality provided.

3. **Modularity.** Changes to the extended system should not require recompilation or modification of the base system, and should not change the behavior of existing clients of the base system.
4. **Type safety.** Extensions cannot create run-time type errors.
5. **Ease of use.** Reuse mechanisms should not clutter or obfuscate the code. Ideally, reuse mechanisms should be at least as easy to use as copy-and-paste reuse.
6. **Composition of extensions.** Multiple extensions should be able to be used together, combining their functionality.

This dissertation describes both a design pattern and an object-oriented programming language for scalable, orthogonal extension and composition of large software systems. The design pattern meets the first three requirements listed above: orthogonal extension, scalability, and modularity. The language mechanism, *nested intersection*, meets all of the above requirements.

The design pattern, described in Chapter 2, enables scalable, modular, orthogonal extension of a base system. Using the design pattern, we have implemented an extensible compiler framework called *Polyglot* [84]. The Polyglot framework provides a source-to-source Java compiler that language implementers can extend to compile extended versions of Java. More than 20 compilers for Java language extensions have been implemented with Polyglot; even so, the design pattern has a number of limitations that prevent it from satisfying all of the requirements enumerated above: it is not type-safe, it can be difficult to use, and it does not easily support composition of extensions. These limitations are addressed by the language-based approach.

Nested inheritance is a language mechanism for simultaneously extending a collection of classes with new functionality. Nested inheritance is designed to be applicable to object-oriented languages that, like Java [45] or C++ [107], support nested classes or other containment mechanisms such as packages or namespaces. Using nested in-

heritance, a large system with multiple interacting classes can be extended safely by writing code only for the new functionality. *Nested intersection* builds on nested inheritance to allow the composition of collections of classes to obtain a software system that combines their types and behavior.

With normal class inheritance, methods act as hooks for extension; nested inheritance builds on this idea by enabling nested classes to be used as hooks too. Nested inheritance creates an interaction between containment and inheritance: when a namespace such as a class or package is inherited, all of its components—even nested classes and packages—are inherited too. Inheritance and subtyping relationships among these components are preserved in the derived namespace, where individual methods, classes, and even packages can be refined to add new behavior in a scalable, modular way.

We have designed a new language named J& (pronounced "Jet") that adds nested inheritance and nested intersection to Java [45]. J& demonstrates that nested intersection integrates smoothly into an existing object-oriented language. Nested intersection is a lightweight mechanism that supports type-safe, scalable extensibility and composition, yet it is hardly noticeable to the novice programmer.

We ported the Polyglot framework to J&, stripping out the code for the Polyglot design pattern because it is no longer needed to provide scalable extensibility. Unlike the original Java-based framework, extensions in the new framework are type safe, simpler to write, and can be composed via nested intersection.

1.1 Extensible compilers

To motivate our requirements, we consider the building of an extensible compiler with composable extensions. Compilers are a particularly challenging domain because a compiler has several different interacting dimensions along which it can be extended.

Compilers contain complex data structures representing the source and target code as well as intermediate code and metadata such as types and dataflow analysis values. Analyses, optimizations, and code transformations operate on these data structures. An extension may need to extend both the data types and the operations.

Domain-specific extension or modification of an existing programming language enables more concise, maintainable programs. However, programmers infrequently construct domain-specific language extensions because building and maintaining a compiler is onerous. When developing a compiler for a language extension, it is clearly desirable to build upon an existing compiler for the base language. Furthermore, by composing extensions of the base compiler, one can obtain a compiler tailored for a particular application domain by choosing useful language features from a "menu" of available options.

1.2 Scalable, orthogonal extension

Compiler frameworks must support orthogonal extension, the addition of both new data types (e.g., abstract syntax, types, dataflow analysis values) and operations on those types (e.g., type checking, optimization, translation). However, traditional programming languages permit orthogonal extension by sacrificing scalability: adding new abstract syntax requires changes to all passes, even if the new node types are relevant to only a few passes. Similarly, adding a new pass may require changes to all nodes. This conflict between extending procedures and types creates an incentive to structure a compiler as a few complex passes rather than as a larger number of simple passes, resulting in a less modular compiler that is harder to understand, maintain, and reuse.

John Reynolds [96] observed that there are two complementary ways to organize code: by data representation or by operation. These two approaches often make it difficult to provide orthogonal extension in a scalable, modular way [117]. In the

first case, called *data-directed programming* [1, 30], all operations on a particular representation are grouped together. This form of code organization is exemplified by abstract data types and by objects. In the alternative organization, *operation-directed programming* [30], each operation is implemented as a function with cases for each representation. This form of organization is typical of functional programming.

The two styles can be illustrated by considering a table of data types and operations on those types. A compiler, for example, performs a series of passes over an intermediate representation such as abstract syntax trees (ASTs). The following table shows the implementation of several compiler passes for several AST node types.

	AST Node Types				
Operations	`+`	`==`	`if`	`x`	*e*.`f`
Resolve names	*no-op*	*no-op*	*no-op*	`lookupVar`	`lookupField`
Check exceptions	*no-op*	*no-op*	*no-op*	*no-op*	`throwNull`
Fold constants	`foldAdd`	`foldEq`	*no-op*	*no-op*	*no-op*
Emit code	`emitAdd`	`emitEq`	`emitIf`	`emitVar`	`emitField`

In data-directed programming, code is grouped by type or by type constructor—by column in the table. In operation-directed programming, code is grouped by operation—by row.

Operations are often *sparse* in the sense that they have interesting behavior for only a few data types. Because non-trivial code need be written for only a few types, sparse operations can treat the other types in a default, boilerplate way. In the table above, the "Resolve names", "Check exceptions", and "Fold constants" operations are sparse; they are implemented as no-ops for AST nodes that contain no names, throw no exceptions, or contain no foldable expressions, respectively. Ideally, when a new sparse operation or a new data type is added, code need only be written where non-boilerplate behavior is required. Standard programming methods, however, cannot exploit this sparsity.

The weakness of data-directed programming is that when a new operation is added, it may be necessary to modify several existing data abstractions to support the new

operation. Adding a new operation corresponds to adding a new row to the table. Code needs to be written for each column, adding code for each existing type. In an object-oriented language, new data types are added by writing new classes, and new operations are added by writing new methods. To add new methods for existing data types, it is necessary to modify the classes implementing those types. In a compiler implemented using the data-directed approach in an object-oriented language, each AST node class implements a method for each compiler pass. This technique suffers from the problem that adding a new pass requires adding a method to all existing node classes.

In contrast, with operation-directed programming, the implementation of an operation must be modified each time a new representation is installed. Adding a new data type corresponds to adding a new column and code must be written for each row of the table; that is, existing functions must be modified to support the new data type. Thus, in functional programming languages, it is straightforward to add new functions, but not to add new data types. Data types in functional languages such as ML [80] and Haskell [56] are implemented as tagged variants. Functions perform pattern matching to implement functionality for each variant. If a new variant is added, existing functions that operate on that data type must be rewritten to handle the new variant.

The *Visitor* design pattern [43] is an instance of operation-directed programming in an object-oriented language. This pattern is commonly used to implement compiler passes over abstract syntax trees. There is a hierarchy of classes representing the abstract syntax: an AST node class for each syntactic construct in the source language (e.g., statements, expressions, declarations, types). Each AST node class is a subclass of a base `Node` class. Compiler passes are implemented as *visitors*, objects that encapsulate a traversal over the AST. Each compiler pass is implemented as a subclass of a base `Visitor` class. To allow specialization of visitor behavior for both the AST node type

and the visitor itself, each concrete visitor subclass implements a separate callback, or `visit`, method for every node type.

In a non-extensible compiler, the set of AST nodes is usually fixed. The Visitor pattern works well in this case because it permits scalable addition of new passes, although it sacrifices scalable addition of AST node types. Since each visitor class implements a separate callback method for every node type, visitors written without knowledge of the new node class cannot be used with the new node because they do not implement the callback.

Ordinary class inheritance does not provide scalable extensibility because it operates one class at a time, making it difficult to extend sets of mutually dependent classes that interact through some protocol. New classes can be added at the leaves of the class hierarchy, but in general, more significant changes may be needed to construct the extended software.

For instance, class inheritance does not permit addition of a visitor callback method to the base visitor class; instead, each visitor subclass must be extended individually with the new method. If operations are sparse, most visitor classes will have duplicate implementations of the same boilerplate functionality.

The reuse mechanisms described in this thesis, address this problem, in part, by enabling new functionality to be added into a base class and then automatically inherited by its subclasses. Other language mechanisms such as mixins [14, 40], open classes [28], and virtual superclasses [34] also permit this kind of extension; however, ordinary inheritance does not.

1.3 Type safety

Many systems aim to be extensible, but sacrifice type safety in the process. These systems often load extensions at run time and check for type errors either dynamically or not at all. For example, web browsers such as Firefox [42] can be extended with a wide range of plugins for handling multimedia, managing passwords, enhancing the user interface, and diverse other uses; however, these plugins are not statically type-checked against the base system. The extended system may therefore be fragile and prone to crash because of run-time type errors. Software testing to eliminate these run-time errors can be expensive, and, because they can occur after the software is deployed, fixing the errors can be even more expensive. Static type safety allows a large class of errors to be detected during compilation rather than at run time, thus reducing the cost of software development and maintenance while increasing the reliability of software. Static typing is also desirable because it provides programmers with machine-checkable documentation and enables optimizations.

Because class inheritance does not provide scalable orthogonal extensibility, static type safety is often sacrificed even when using traditional strongly typed object-oriented languages like Java. Design pattern approaches such as the Polyglot design pattern use dynamic type checks to allow base system and extended system code to coexist. Objects created by the extended system can be stored in base system data structures that are unaware of the extended types. For the extended system to use these objects, run-time type checks are performed to coerce the objects from their base system types to the appropriate extended type. These checks may fail at run time.

1.4 Modularity

Modularity is an important requirement for building, maintaining, and deploying large systems. Extension of the base system should not require modification or recompilation of the base system. Separate compilation—compiling code only if a signature on which it depends changes—reduces compile time, accelerating software development and lowering its cost. By enabling the base system to be distributed in binary form, separate compilation also has social and business advantages. The base system can be extended without access to its source code, which may not be available because of intellectual property concerns or because it is too costly.

A related aspect of modularity is that base system code should be available for use within the extended system. *Non-destructive extension* enables existing clients of the base system and the extended system itself to interoperate with code and data of the base system and with other extensions. Non-destructive extension is also important for allowing several extensions of the base system to coexist within the same application.

In an extensible compiler, it should be possible for an extension to compile both extended language source code and base language source code, or to compile source code for several different extensions of the base language. It may also be convenient to implement the extended language by translating it into the base language and using the existing base compiler framework to generate code. In addition, a compiler for an extended language may need the results of existing base-language analyses (e.g., must-return analysis) to generate correct output. The extension compiler may need to run an extended version of the analysis on the source code and the base-language version of the analysis on the target code.

1.5 Ease of use

To make code extensible, it is essential to provide hooks for the extension implementer to add new behavior and state. Design patterns and programming languages enable the implementer of the base system to create these hooks. However, these hooks can often clutter or obfuscate the base code.

One way to provide hooks is through language mechanisms that provide some kind of parametric genericity, such as parameterized types [65], parameterized mixins [14], and functors [80, 67]. Explicit parameterization over types, classes, or modules precisely describes the ways in which extension is permitted. However, it is often an awkward way to achieve extensibility, especially when a number of modules are designed in conjunction with one another and have mutual dependencies. It is often difficult to decide which explicit parameters to introduce for purposes of future extension, and the overhead of declaring and using parameters can be cumbersome. The parameters may be numerous and unintuitive, and it is difficult to instantiate all the generic framework parameters in a consistent way.

Inheritance embodies a different approach to extensibility. By giving names to methods, the programmer creates less obtrusive, *implicit* parameters that can be overridden when the code is reused. Nested inheritance builds on this insight by enabling use of nested classes as hooks too.

Implicit hooks reduce the amount of planning required for later extension. The base system developer can concentrate more on the functionality of the system rather than on how it might be extended. By allowing hooks to be created implicitly, simply by declaring a nested class, nested inheritance allows extensions to refine any nested class with new functionality. Existing code can use the refined nested class without modification.

1.6 Composition

When a system is extended multiple times, it is natural to want to reuse several of these extensions simultaneously within a single combined system. For example, web browsers are often run with several extensions for blocking ads, enhancing the UI, or handling video and audio files.

Composition of extensions is not just a matter of linking. Linking works when the composed software components offer disjoint, complementary functionality. In the general case, two components are not disjoint, but instead may offer similar functionality, because they both extend a common ancestor component. Nested intersection integrates their extensions rather than duplicating the extended components.

While nested inheritance allows extension of entire class hierarchies, nested intersection enables composition of these class hierarchies. When the hierarchies contain common nested packages or classes, these too are composed.

Composed extensions may also provide conflicting functionality. Consequently, it may not be possible to integrate their types and behavior automatically. In this case, the programmer must resolve the conflict; the compiler should report potential conflicts to the programmer and require that they be resolved.

1.7 Outline

The rest of this thesis is organized as follows. Chapter 2 describes the Polyglot design pattern, which supports scalable, modular, orthogonal extension of a base system. This pattern is used in the Polyglot extensible compiler framework; however, it is not statically type safe and, because it is not integrated into the programming language, can also be harder to use than a language-based approach.

To address these concerns, Chapter 3 describes nested inheritance and nested intersection. These mechanisms meet all of the requirements enumerated above. Chapter 4 presents J&, an extension of Java with nested intersection, and also discusses technical challenges, such as the problem of resolving conflicts among composed packages.

Chapter 5 then describes how nested intersection can be used to extend and compose compilers and presents a design pattern for implementing extensible translation passes in J&.

Chapter 6 presents a formal operational semantics and type system for a Java-like calculus extended with nested intersection. This calculus is proved sound in Chapter 7.

Two alternative implementations of J& are described in Chapter 8, and Chapter 9 describes experience using J& to implement and compose extensions in the Polyglot compiler framework [84] and in the Pastry framework for building peer-to-peer systems [99].

Related work is discussed in Chapter 10. Open issues and future directions are presented in Chapter 11. Finally, Chapter 12 concludes.

Chapter 2

The Polyglot Design Pattern

Polyglot is an extensible compiler framework for Java [84]. The framework provides a source-to-source Java base compiler that can be extended to construct compilers for languages with new syntax and new semantics. This chapter describes the design pattern used by the framework to provide scalable, orthogonal extensibility. While we illustrate the pattern for an extensible compiler, its utility is not limited to this application.

2.1 Compiling in Polyglot

The compilation process offers several opportunities for the language extension implementer to customize the behavior of the base compiler framework. This process, including the eventual compilation to Java bytecode [64], is shown in Figure 2.1. The Polyglot framework permits both syntactic and semantic extensions of the base compiler.

The first step in compilation is parsing input source code to produce an abstract syntax tree (AST). The extended AST may contain new kinds of nodes either to represent syntax added to the base language or to record new information in the AST. The core of the compilation process is a series of compilation passes applied to the AST. Both semantic analysis and translation to Java may comprise several such passes. Programmers

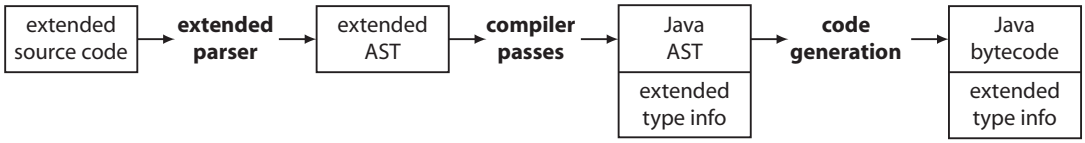

Figure 2.1: Extensible compiler Architecture

can extend existing AST node classes to implement new abstract syntax, and can extend compiler passes to implement new semantics, analyses, and optimizations, or to translate source-language ASTs into target-language (i.e., Java) ASTs. Compilation passes may transform the AST and may modify the symbol table and other data structures that define characteristics of the source and target languages. After all compilation passes complete, a Java AST is produced, from which Java code can be generated. A Java compiler such as `javac` is invoked to compile the Java code to bytecode. To enable separate compilation, source-language type information may be embedded into the bytecode.

The Polyglot compiler framework enables the scalable, orthogonal extension of a base Java compiler. The programmer effort required to add or extend a pass is proportional to the number of AST nodes non-trivially affected by that pass; the effort required to add or extend a node is proportional to the number of passes the node must implement in an interesting way.

2.2 The Polyglot design pattern

There are two common alternatives for implementing compiler passes in an object-oriented language: the data-directed approach where each AST node class implements a method for each compiler pass, and the operation-directed approach using the Visitor pattern. Neither of these approaches is scalable.

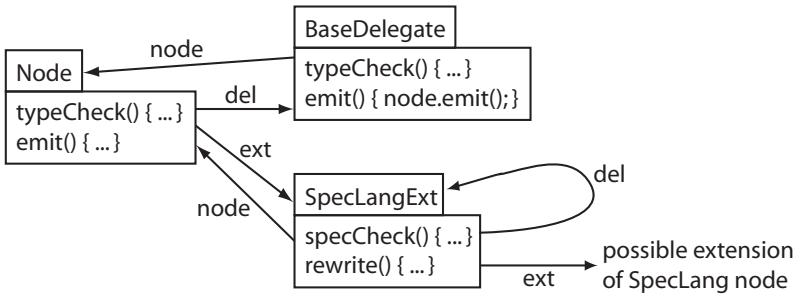

Figure 2.2: Delegates and extensions

Polyglot achieves scalable extensibility by modifying the data-directed approach to allow subclasses to inherit any changes made to their base classes. Compiler passes are implemented by traversing the AST and invoking a method at each node in the tree. Polyglot introduces a delegation mechanism, illustrated in Figure 2.2, that enables orthogonal extension and method override of AST nodes. The figure shows a base compiler `Node` class extended with functionality to implement a new language, called SpecLang, which adds formal specification annotations to the base language.

Because ordinary class inheritance does not permit new methods and new fields to be added into a base class and then automatically inherited by its subclasses, subclassing of node classes does not permit scalable extension of methods in classes with multiple subclasses. Polyglot addresses this problem by adding to each node object a field, labeled `ext` in Figure 2.2, that points to a (possibly `null`) *node extension object*. This field acts as a hook for extending the AST node class. The extension object (`SpecLangExt` in the figure) provides implementations of new methods and fields, thus extending the node interface without subclassing. In effect, a node and its extension object together act as a single AST node. These additional members are accessed by following the `ext` pointer and casting to the extension object type. In the example, `SpecLangExt` extends `Node` with `specCheck()` and `rewrite()` methods. Each AST node class to be extended with a given implementation of these members uses the same

extension object class. Thus, several node classes can be orthogonally extended with a single implementation, avoiding code duplication. Since an extension of the base compiler should be open to further extension, each extension object has an `ext` field similar to the one located in the node object.

Extension objects alone, however, are not sufficient to handle all extensions of an AST node. An extension of the base system may need to override the implementation of an existing pass. As with adding a new method, overriding an existing method can be done by implementing an extension object class and associating it with several AST node classes. The problem is that the node itself or any one of a node's extension objects can implement the overridden method; a mechanism is needed to invoke the correct implementation. Polyglot's solution to this problem is to introduce a level of indirection. For each method in the `Node` interface, a field in the node points to a *delegate* implementing that method, possibly back to the node itself. Because maintaining one object per method is cumbersome, delegate objects are combined by the programmer when possible. In Figure 2.2, the node object has a single `del` field pointing to a delegate (`BaseDelegate`) for both of its methods. Rather than calling a `Node` method directly, calls are made through its delegate object. Language extensions can override a method simply by replacing the delegate with an object containing the new implementation or code to dispatch to a new implementation in an extension object; non-overridden methods can be implemented by dispatching back to the node itself. In Figure 2.2, the `BaseDelegate` overrides the `typeCheck` method with new behavior; the `emit` method dispatches back to the node. Extension objects also have delegates used to override methods declared in the extension object interface.

Java code illustrating the pattern is shown in Figure 2.3. In the pattern, calls to all node methods are made through the delegate pointer, thus ensuring that the correct implementation of the method is invoked if the delegate object is replaced by a language

```
interface Base {
    void typeCheck();
    void emit();
}

class Node implements Base {
    Base del; Object ext; ...
    Node typeCheck() { ... }
    void emit() { ... }
}

class Rewriter {
    Node visit(Node n) { return n; } ...
}

class TypeChecker extends Rewriter {
    Node visit(Node n) { return n.del.typeCheck(); } ...
}

class BaseDelegate implements Base {
    Node node; ...
    Node typeCheck() { ... }
    void emit() { node.emit(); }
}

interface Spec {
    Node specCheck();
    Node rewrite();
}

class SpecLangExt implements Spec {
    Node node; Spec del; Object ext; ...
    Node specCheck() { ... }
    Node rewrite() { ... }
}

class SpecChecker {
    Node visit(Node n) { return ((Spec) n.ext).del.specCheck(); } ...
}
```

Figure 2.3: Polyglot design pattern in Java

extension. In our example, the `TypeChecker` class invokes the node `n`'s `typeCheck` method via `n.del.typeCheck()`; similarly, `SpecChecker` invokes the `specCheck` method by following the node's `ext` pointer and invoking through the extension object's delegate: `((Spec) n.ext).del.specCheck()`. An extension of SpecLang could replace the extension object's delegate to override methods declared in the extension, or it could replace the node's delegate to override methods of the node. To access SpecLang's type-checking functionality, this new node delegate may be a subclass of SpecLang's node delegate class or may contain a pointer to the old delegate object.

An important aspect of the design pattern is the use of *factory methods* [43] to create objects, extensions, and delegates. In the Polyglot compiler, each language extension has a *node factory* object that constructs AST nodes for the extension, installing extension objects and delegates as appropriate. The factory contains methods for constructing instances of each AST node class. Factories are important for extensibility since they permit code in the base system to create instances of classes defined by the extension. Hard-coding the names of classes into the base system limits the scalability of the framework by requiring the extension to override all methods whose code contains the name of the class to be refined. By using factories, only a short factory method needs to be overridden when a class is refined.

Node factories for a base compiler and a SpecLang compiler are shown in Figure 2.4 using classes and interfaces declared in Figure 2.3. The factory method `createCall` in `NodeFactory` creates a `Call` AST node. The invocation of the `Call` constructor is wrapped in a call to `extCall`, which decorates the new object with extension and delegate objects. The base implementation of `extCall` invokes `extExp` and `extNode`. The base implementation of `extNode` initializes the `ext` field of the node to `null` and the `del` field to the node itself. The `SpecNodeFactory` subclass of `NodeFactory` that overrides `extCall` to initialize the `del` field to a `CallDel` object, which overrides the

base `Call`'s `typeCheck` method, and delegates back to the base `emit` method. The `SpecNodeFactory` class also overrides `extNode` to decorate all AST node classes created by the factory with a `SpecLangExt` object that adds new functionality to these classes.

Most passes in the Polyglot compiler are structured as non-imperative AST rewriting passes. These passes take an AST as input and produce a new AST as output, leaving the input AST unchanged. Factoring out AST traversal code eliminates the need to duplicate this code when implementing new passes. Each pass is implemented as an *AST rewriter* object that traverses the AST and at each AST node invokes a pass-specific method of the node. This design in shown in Figure 2.5, which shows the code for a `Call` node a type-checker for the SpecLang language. Figure 2.2. At each node, the rewriter invokes a `visitChildren` method to recursively rewrite the node's children using the rewriter and to reconstruct the node if any of the children are modified. A key implementation detail is that when a node is reconstructed, rather than allocating a new node using the node factory, the old node is *cloned* and updated with the new children. Since the base compiler is unaware of any new children added by extensions, cloning ensures these new children are correctly copied into the new node. The node's delegate and extension objects are cloned with the node.

To summarize, running a rewriting pass on an AST proceeds as follows. First, a `Rewriter` for the pass is created and its `visitNode` method is called on the root node of the AST, for example: `new TypeChecker().visitNode(ast)`. This call will eventually return the new AST. The rewriter's `visitNode` method calls the node's `visitChildren` method to rewrite the node's children and reconstruct the node, as described above, and calls the rewriter's `visit` method on the result. The `visit` method is pass-specific: `TypeChecker`'s `visit` method takes a node `n` and invokes `typeCheck` on `n`'s delegate. For base compiler nodes, `n`'s delegate is `n` itself. Extensions may

```
class NodeFactory {
    Call createCall(Exp receiver, String name, List args) {
        return extCall( new Call(receiver, name, args) );
    }
    Call extCall(Call n) {
        return (Call) extExp(n);
    }
    Exp extExp(Exp n) {
        return (Exp) extNode(n);
    }
    Node extNode(Node n) {
        n.ext = null;
        n.del = n;
        return n;
    }
    ...
}

class SpecNodeFactory extends NodeFactory {
    Node extCall(Call n) {
        n = super.extCall(n);
        n.del = new CallDel(n);
        return n;
    }
    Node extNode(Node n) {
        n.ext = new SpecLangExt(n);
        return n;
    }
    ...
}

class CallDel implements Base {
    Call node; ...
    Node typeCheck() { ... node.typeCheck(); ... }
    void emit() { node.emit(); }
    ...
}
```

Figure 2.4: Node factories

```
class Node {
    Object clone() {
        Node n = (Node) super.clone();
        ... // clone ext and del
        return n;
    } ...
}

class Exp extends Node { Type type; ... }

class Call extends Exp {
    Exp receiver;
    String name;
    List args;
    ...
    Node visitChildren(Rewriter v) {
        Exp receiver = (Exp) v.visitNode(this.receiver);
        List args = v.visitList(this.args);
        if (receiver != this.receiver || args != this.args) {
            Call c = (Call) this.clone(); // copy only if changed
            c.receiver = receiver;
            c.args = args;
            return c;
        }
        return this;
    }
    Node typeCheck() {
        // set the type from the method's declared return type
        Call c = (Call) this.clone();
        c.type = ...;
        return c;
    }
}

class Rewriter {
    Node visitNode(Node n) {
        return visit( n.visitChildren(this) );
    }
    Node visit(Node n) { return n; } ...
}

class TypeChecker extends Rewriter {
    Node visit(Node n) { return n.del.typeCheck(); } ...
}
```

Figure 2.5: Fragment of Polyglot type-checking code

create a delegate object to override the behavior of the given pass for a set of node classes. For instance, the SpecLang compiler installs a `CallDel` delegate for all `Call` nodes. `CallDel` overrides the `typeCheck` method and dispatches back to the node's `typeCheck` method as part of its implementation. Passes may also be implemented in an extension object: the `SpecChecker` rewriter invokes the `specCheck` method in the extension object of the node it is passed.

2.3 Scalable extensibility in Polyglot

A language extension may extend the interface of an AST node class through an extension object interface. To add a new pass, an extension object interface is created; the node factory installs instances of the interface into each AST node when the node is created. The compiler pass traverses the AST and invokes the method at each node in the tree. For most nodes, a single extension object class is implemented to define the default, boilerplate, behavior of the pass, typically just an identity transformation on the AST node. This class is overridden for individual node classes where non-trivial work is performed for the pass.

To change the behavior of an existing pass at a given node, the programmer creates a new delegate class implementing the new behavior and associates the delegate with the node at construction time. Like extension classes, the same delegate class may be used for several different AST node classes, allowing functionality to be added to node classes at arbitrary points in the class hierarchy without code duplication.

New kinds of nodes are defined by new node classes; existing node types are extended by adding an extension object to instances of the class. A factory method for the new node type is added to the node factory to construct the node and, if necessary, its delegate and extension objects. The new node inherits default implementations of

all compiler passes from its base class and from the extension's base class. The new node may provide new implementations using method override, possibly via delegation. Methods need be overridden only for those passes that need to perform non-trivial work for that node type.

2.4 Discussion

Because of the limitations of inheritance, design patterns in traditional object-oriented languages provide scalable, orthogonal extensibility by sacrificing type safety. In the Polyglot design pattern, extension objects and delegates must be cast to the appropriate type before use. For instance, in Figure 2.3, the method `SpecChecker.visit` casts the node's extension object to the `Spec` interface to access its `specCheck` method. Because the Java type system is not expressive enough to ensure that the pattern is used correctly, these casts can fail with a run-time type error. Moreover, the casts clutter and obfuscate the code.

Design patterns also require careful planning when designing the base system to ensure the hooks for extension provided by the pattern are available for use. The Polyglot base compiler was intended to be extended with new AST nodes and new compiler passes; the pattern was applied to these classes. However, the pattern was not applied to types, dataflow analysis values, and other data structures used in the base compiler. These data structures can be extended via normal class inheritance, but the extensibility may not scale and often requires duplication of code. For example, if a language extension needs to add a field to both method type objects and constructor type objects to implement procedure pre- and post-conditions, then the two classes implementing method and constructor types each need to be extended with identical code. The Polyglot base compiler does not provide a hook for identical changes to both

classes to be made in one place. Because the design pattern was not applied to type objects, there is no convenient way to add code to a class representing type information so that it is inherited by all subclasses. Despite this limitation, the Polyglot compiler framework has been successfully used to extend the Java type system; this success is due in part to the flatness of the class hierarchy representing Java types: most extensions can be applied to the leaves of the class hierarchy using normal class inheritance.

Because they are not part of the programming language, design patterns can also be cumbersome to use. In Polyglot, factories, extension objects, and delegates complicate both the base compiler and extensions, making them more difficult to use and maintain. If the pattern is not used correctly—particularly, if node factories are not set up correctly—an extension compiler can exhibit unexpected behavior at run time and be difficult to debug. Furthermore, in many cases, class inheritance alone is sufficient to implement a given extension, but the programmer is burdened with implementing the pattern anyway in order to ensure future extensibility. The extension implementer must provide factory methods for new AST node classes, and must remember to call methods through delegates.

To address these limitations, Chapter 3 introduces nested inheritance and nested intersection and the programming language J&. Like the Polyglot design pattern, J& supports scalable, modular, orthogonal extension of a base system. Moreover, the language does so without sacrificing type safety or ease of use. Nested intersection also allows extensions to be composed. Chapter 5 describes how J& can be used to implement an extensible compiler framework, and Chapter 9 describes a port of the Polyglot framework to J&.

Chapter 3

Nested Inheritance and Nested Intersection

Nested inheritance and nested intersection support scalable extension of a base system and scalable composition of those extensions. Nested inheritance [83] builds on the ideas behind virtual classes [68, 69, 52, 38] to enable more code reuse; it is implemented in the language Jx, an extension of the Java programming language. Nested intersection [85] extends nested inheritance with the ability to compose extensions; it is implemented in the language J&, an extension of Jx.

3.1 Nested inheritance

Nested inheritance is inheritance of *namespaces*: packages and classes. In J&, packages are treated like classes with no fields, methods, or constructors. A namespace may contain other namespaces. A namespace may also extend another namespace, inheriting all its members, including nested namespaces. As with ordinary inheritance, the meaning of code inherited from the base namespace is as if it were copied down from the base. A

```
class A {
    class B { int x; }
    class B2 extends B { B next; }
    int m(B b) {
        return b.x;
    }
    B2 n() {
        return new B2();
    }
}

class A2 extends A {
    class B { int y; }
    int m(B b) {
        return b.x + b.y;
    }
}
```

Figure 3.1: Nested inheritance example

derived namespace may *override* any of the members it inherits, including nested classes and packages.

Figure 3.1 shows a simple example of nested inheritance. Class `A` contains nested classes `B` and `B2`. `A`'s subclass `A2` inherits `B` and `B2`. Class `A2` explicitly declares a nested class `B`, overriding `A.B`. Class `A.B2` is inherited into `A2` as the *implicit class* `A2.B2`: although there is no class declaration for `A2.B2`, the programmer can refer to the class.

As with virtual classes [68, 69, 38], overriding of a nested class does not replace the original class, but instead refines, or *further binds* [68], it. The nested class `A2.B` further binds `A.B`. `A2.B` is a subclass of `A.B`, and declarations within `A2.B` (e.g., the field `y`) extend `A.B` as if `A2.B` were an explicitly declared subclass of `A.B`. In general, if a namespace T' extends T, which contains a nested namespace $T.C$, then $T'.C$ inherits members from $T.C$ as well as from $T'.C$'s explicitly named base namespaces (if any). Further binding thus provides a limited form of multiple inheritance: *explicit inheritance*

from the named base of $T'.C$ and *induced inheritance* from the original namespace $T.C$. Unlike with virtual classes, $T'.C$ is a subtype as well as a subclass of $T.C$.

The key feature of nested inheritance that enables scalable extensibility is late binding of type names. When the name of a class or package is inherited into a new namespace, the name is interpreted in the context of the namespace into which it was inherited, rather than where it was originally defined. When the name occurs in a method body, the type it represents may depend on the run-time value of `this`.

In Figure 3.1, the unqualified type names `B` and `B2` are late bound. Thus, the constructor call `new B2()` in the body of method `A.n` allocates an instance of `A.B2` when `n` is invoked on an object of class `A`, and an instance of `A2.B2` when `n` is invoked on an object of class `A2`. Late binding of type names ensures subclass and subtype relationships are preserved by inheritance into a new containing namespace. The class `A2.B2` is a subclass of `A2.B` because `A.B2` is a declared to be a subclass of `B`.

The argument of the method `m` in the class `A` has type `B`, which is also late bound. When called on an instance of `A`, `m` expects an `A.B`; when called on an instance of `A2`, `m` expects an `A2.B`. The name `B` is reinterpreted in the inheriting context. With this change, `A2` might not seem to conform to `A` because a formal parameter type has changed covariantly. However, subtyping between `A2` and `A` is still sound because the type system ensures the `m` method can only be called when its argument is known to be from the same implementation of `A` as the method receiver.

Figures 3.2 and 3.3 show a more elaborate example, fragments of J& code for a simple compiler for the lambda calculus extended with pair expressions. This compiler translates the lambda calculus with pairs into the lambda calculus without pairs. In the example, the `pair` package extends the `base` package, further binding the `Visitor`, `TypeChecker`, and `Compiler` classes, as illustrated by the `base` and `pair` boxes in the inheritance hierarchy of Figure 3.4. The class `pair.TypeChecker` is a subclass

```
package base;

abstract class Exp {
    Type type;
    abstract Exp accept(Visitor v);
}
class Abs extends Exp {
    String x; Exp e; // λx.e
    Exp accept(Visitor v) {
        e = e.accept(v);
        return v.visitAbs(this);
    }
}
class Visitor {
    Exp visitAbs(Abs a) {
        return a;
    }
}
class TypeChecker extends Visitor {
    Exp visitAbs(Abs a) { ... }
}
class Emitter extends Visitor {
    Exp visitAbs(Abs a) {
        print(...);
        return a;
    }
}
class Compiler {
    void main() { ... }
    Exp parse() { ... }
}
```

Figure 3.2: Lambda calculus compiler

```
package pair extends base;

class Pair extends Exp {
    Exp fst, snd;
    Exp accept(Visitor v) {
        fst.accept(v);
        snd.accept(v);
        return v.visitPair(this);
    }
}
class Visitor {
    Exp visitPair(Pair p) { return p; }
}
class TypeChecker extends Visitor {
    Exp visitPair(Pair p) { ... }
}
class TranslatePairs extends Visitor {
    Exp visitPair(Pair p) {
        return ...;
          // (λx.λy.λf. f x y) ⟦p.fst⟧ ⟦p.snd⟧
    }
}
class Compiler {
    void main() {
        Exp e = parse();
        e.accept(new TypeChecker());
        e = e.accept(new TranslatePairs());
        e.accept(new Emitter());
    }
    Exp parse() { ... }
}
```

Figure 3.3: Lambda calculus + pairs compiler

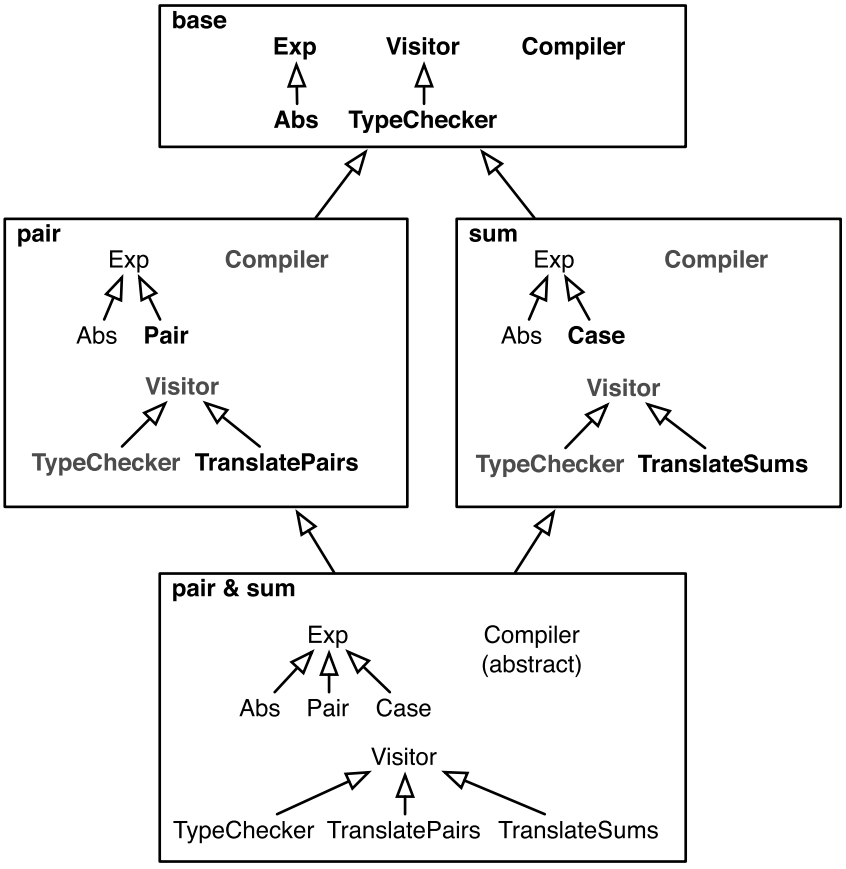

Figure 3.4: Inheritance hierarchy for compiler composition

of both `base.TypeChecker` and `pair.Visitor` and contains both the `visitAbs` and `visitPair` methods.

The unqualified name `Visitor` is late bound. In the context of the `base` package, `Visitor` refers to `base.Visitor`. When a reference to `Visitor` is inherited into `pair`, `Visitor` refers to `pair.Visitor`. Thus, when the method `accept` is called on an instance of `pair.Pair`, it must be called with a `pair.Visitor`, *not* with a `base.Visitor`. This allows `Pair`'s `accept` to invoke the `visitPair` method of the parameter *v*.

Late binding applies to supertype declarations as well. Thus, `pair.Emitter` extends `pair.Visitor` and inherits its `visitPair` method. Late binding of supertype declarations thus provides a form of *virtual superclasses* [69, 34], permitting inheritance relationships among the nested namespaces to be preserved when inherited into a new enclosing namespace. The class hierarchy in the original namespace is replicated in the derived namespace, and in that derived namespace, when a class is further bound, new members added into it are automatically inherited by subclasses in the new hierarchy.

Sets of mutually dependent classes may be extended at once by grouping them into a namespace. For example, the classes `Exp` and `Visitor` in the `base` package are mutually dependent. With ordinary class inheritance, because the extended classes need to know about each other, the `pair` compiler could define `Pair` as a new subclass of `Exp`, but references within `Exp` to class `Visitor` would refer to the old `base` version of `Visitor`, not the appropriate one that understands how to visit pairs. With nested inheritance of the containing namespace, late binding of type names ensures that relationships between classes in the original namespace are preserved when these classes are inherited into a new namespace.

In general, the programmer may want some references to other types to be late bound, while others should refer to a particular fixed class. Late binding is achieved by interpreting unqualified type names like `Visitor` as sugar for types nested within *dependent classes* and *prefix types*. The semantics of these types are described in more detail in Section 4.1. Usually, the programmer need not write down these desugared types; most J& code looks and behaves like Java code.

```
package sum extends base;

class Case extends Exp {
    Exp test, ifLeft, ifRight; ...
}
class Visitor {
    Exp visitCase(Case c) {
        return c;
    }
}
class TypeChecker extends Visitor { ... }
class TranslateSums extends Visitor { ... }
class Compiler {
    void main() { ... }
    Exp parse() { ... }
}
```

Figure 3.5: Lambda calculus + sums compiler

3.2 Nested intersection

To support composition of extensions, J& provides nested intersection: new classes and packages may be constructed by inheriting from multiple packages or classes; the class hierarchies nested within the base namespaces are composed to achieve a composition of their functionalities.

For two namespaces S and T, $S\&T$ is the *intersection* of these two namespaces. Nested intersection is a form of multiple inheritance implemented using *intersection types* [97, 29]: $S\&T$ inherits from and is a subtype of both S and T.

Nested intersection is most useful when composing related packages or classes. When two namespaces that both extend a common base namespace are intersected, their common nested namespaces are themselves intersected: if S and T contain nested namespaces $S.C$ and $T.C$, the intersection $S\&T$ contains $(S\&T).C$, which is equal to $S.C\&T.C$.

```
package pair_and_sum extends pair & sum;

// Resolve conflicting versions of main
class Compiler {
    void main() {
        Exp e = parse();
        e.accept(new TypeChecker());
        e = e.accept(new TranslatePairs());
        e = e.accept(new TranslateSums());
        e.accept(new Emitter());
    }
    Exp parse() { ... }
}
```

Figure 3.6: Compiler composition and conflict resolution

Consider again the lambda calculus compiler from Figure 3.3. Suppose that we had also extended the `base` package to a `sum` package implementing a compiler for the lambda calculus extended with sum types. This compiler is shown in Figures 3.5.

The intersection package `pair & sum`, shown in Figure 3.4, composes the two compilers, producing a compiler for the lambda calculus extended with both product and sum types. Since both `pair` and `sum` contain a class `Compiler`, the new class (`pair & sum`).`Compiler` extends both `pair.Compiler` and `sum.Compiler`. Because both `pair.Compiler` and `sum.Compiler` define a method `main`, the class (`pair & sum`).`Compiler` contains conflicting versions of `main`. The conflict is resolved in Figure 3.6 by creating a new derived package `pair_and_sum` that overrides `main`, defining the order of compiler passes for the composed compiler. A similar conflict occurs with the `parse` method.

3.3 Extensibility requirements

Nested intersection in J& meets all of the requirements listed in Chapter 1, making it a useful language for implementing highly extensible systems.

3.3.1 Orthogonal extension

As explained in Chapter 1, it is well known that there is a tension between extending types and extending the procedures that manipulate them [96]. Nested inheritance solves this problem because late binding of type names causes inherited methods to operate automatically on data types further bound in the inheriting context.

3.3.2 Type safety

Nested inheritance is also type-safe [83]. Dependent classes ensure that extension code cannot use objects of the base system or of other extensions as if they belonged to the extension, which could cause run-time errors. A formal proof of soundness is presented in Chapter 7.

3.3.3 Modularity and scalability

Extensions are subclasses (or subpackages), and hence they are modular. The base code does not need to be modified to extend the system.

Extension is scalable for several reasons; one important reason is that the name of every method, field, and class provides a potential hook that can be used to extend behavior and data representations.

Nested inheritance does not affect the base code, so it is a non-destructive extension mechanism, unlike open classes [28] and aspects [60]. Therefore, base code and

extended code can be used together in the same system, which is important for maintaining backward compatibility with existing clients of the base system.

3.3.4 Ease of use

A strength of nested inheritance as an extension mechanism is that it requires less advance planning to reuse code. Since every class and method provides an implicit hook for further extension, little programmer overhead is needed to identify the possible ways in which the code can be extended. The base system programmer can concentrate on implementing the system's functionality rather than on parameterizing the system for further extension. Nested inheritance allows extensions to refine any nested class with new functionality. Existing code can use the refined nested class without modification.

Nested inheritance largely eliminates the need for factory methods [43] and other design patterns such as the Polyglot pattern that address the problem of scalable extensibility.

J& is largely backward compatible with Java, making it easy for novice programmers to pick up the language and to port existing Java code to the language. In most cases, advanced type system features such as dependent classes and prefix types are hidden from the programmer and need not be written down explicitly.

3.3.5 Composition

Nested intersection enables a namespace to be constructed by inheriting from two or more base namespaces. The class hierarchies nested within the base namespaces are composed to achieve a composition of their functionalities. As described in Section 4.6, the J& compiler detects conflicts between composed class hierarchies and requires the programmer to resolve them.

Chapter 4

The J& Language

This chapter gives an overview of the static and dynamic semantics of J&. A formal semantics is presented in Chapter 6 and proved sound in Chapter 7.

4.1 Dependent classes and prefix types

In most cases, J& code looks and behaves like Java code. However, unqualified type names are really syntactic sugar for nested classes of dependent classes and prefix types, which we introduced in Jx [83]. Figure 4.1 shows a desugared version of the code in Figure 3.1.

In Figure 4.1, the type `this.class` is an example of a *dependent class*. The dependent class p.`class` represents the run-time class of the object referred to by the *final access path* p. Thus, `this.class` is a the run-time class of `this`: if `this` points to an `A`, expressions with type `this.class` are members of the class `A`, and if `this` points to an `A2`, these expressions are members of the class `A2`. A final access path p is either a final local variable, including `this` and final formal parameters, a field access p'.`f`, where p' is itself a final access path and `f` is a final field of p', or a final static field access T.`f`. The class represented by p.`class` is, in general, statically unknown,

```
class A {
    class B { int x; }
    class B2 extends thisclass.B { A[this.class].B next; }
    int m(this.class.B b) {
        return b.x;
    }
    this.class.B2 n() {
        return new this.class.B2();
    }
}

class A2 extends A {
    class B { int y; }
    int m(this.class.B b) {
        return b.x + b.y;
    }
}
```

Figure 4.1: Desugared nested inheritance example

but is *fixed*: for a particular p, all instances of p.`class` have the same run-time class, and *not* a proper subclass, as the object referred to by p. Therefore, if `this` points to an `A`, instances of `this.class` all have class `A` and *not* class `A2`.

The type `A[this.class]` in the body of `A.B2` is an example of a *prefix type*. The prefix type P`[`T`]` represents the enclosing namespace of the class or interface T that is a subtype of the namespace P. Thus, if `this` points to an `A.B` or `A.B2`, expressions with type `A[this.class]` have class `A`, and if `this` points to an `A2.B` or `A2.B2`, expressions with type `A[this.class]` have class `A2`. Similarly, if `this` points to an `A2.B`, then the field `next` with type `A[this.class].B` must point to an `A2.B` (or `A2.B2`) and *not* an `A.B`

It is required that P be a non-dependent type: either a top-level namespace C or a namespace of the form $P'.C$. In typical use T is a dependent class. P may be either a package or a class. Prefix types provide an unambiguous way to name enclosing classes and packages of a class without the overhead of storing references to enclosing instances in each object, as is done in virtual classes. Indeed, if the enclosing namespace

is a package, there are no run-time instances of the package that could be used for this purpose.

Late binding of types is provided by interpreting unqualified names as members of the dependent class `this.class` or of a prefix type of `this.class`. For example, the types in Figure 3.1 are interpreted as the desugared types in Figure 4.1. The compiler resolves the name *C* to the type `this.class`.*C* if the immediately enclosing class contains or inherits a nested namespace named *C*. Similarly, if an enclosing namespace *P* other than the immediately enclosing class contains or inherits *C*, the name *C* resolves to *P*`[this.class]`.*C*. Derived namespaces of the enclosing namespace may further bind and refine *C*. The version of *C* selected is determined by the run-time class of `this`.

Figure 4.2 shows a portion of the lambda calculus compiler from Figures 3.2 and 3.3. The name `Visitor` in this code is sugar for `base[this.class].Visitor`. The dependent class `this.class` represents the run-time class of the object referred to by `this`. The prefix package `base[this.class]` is the enclosing package of `this.class` that is a derived package of `base`. Thus, if `this` is an instance of a class in the package `pair`, `base[this.class]` represents the package `pair`.

Both dependent classes and prefixes of dependent classes are *exact types* [18]: all instances of these types have the same run-time class, but that class is statically unknown in general. Simple types like `base.Visitor` are not exact since variables of this type may contain instances of any subtype of `Visitor`.

4.2 Static contexts

The variable `this` is not in scope in static contexts such as the body of static method or a superclass declaration. Consequently, `this.class` cannot be used in these con-

```
package base;

abstract class Exp {
    Type type;
    abstract Exp accept(Visitor v);
}
class Abs extends Exp {
    String x; Exp e; // λx.e
    Exp accept(Visitor v) {
        e = e.accept(v);
        return v.visitAbs(this);
    }
}
class Visitor {
    Exp visitAbs(Abs a) {
        return a;
    }
}
class TypeChecker extends Visitor {
    ...
}
```

```
package pair extends base;

class Pair extends Exp {
    Exp fst, snd;
    Exp accept(Visitor v) {
        fst.accept(v);
        snd.accept(v);
        return v.visitPair(this);
    }
}
class Visitor {
    Exp visitPair(Pair p) { return p; }
}
```

Figure 4.2: Lambda calculus + pairs compiler

texts. However, support for virtual superclasses requires late binding of types in static contexts.

J& provides the types `thisclass` and `thispackage` for use in static contexts. These types represent the enclosing class or package into which the type reference is inherited. In Figure 4.1, the class `A.B2` extends `thisclass.B`; therefore, `A.B2` is a subclass of `A.B` and `A2.B2` is a subclass of `A2.B`. In non-static contexts in the body of some class, the type `thisclass` is equivalent to `this.class`; the type `thispackage` is equivalent to `P[this.class]`, where *P* is the name of the enclosing package of the current class. Like `this.class`, `thisclass` and `thispackage` and their prefixes are exact types.

4.3 Virtual classes and family polymorphism

Nested classes in J& are similar to *virtual classes* [68, 69, 38]. A virtual class is a nested class that can be further bound in a subclass. A key difference between J& nested classes and virtual classes is that a virtual class is nested within an object, the *enclosing instance*: given an expression *e* of an object type, *e.C* is a virtual class nested within *e*, and the implementation of *e.C* is determined at run time from the value of *e*. In contrast, nested classes in J& are nested within their enclosing class or package, and late binding of types is achieved by using dependent classes. This allows further binding of classes and packages without requiring the programmer to keep track of enclosing instances, which can clutter the code as they are passed between methods.

Like recent type-safe variants of virtual classes [34, 38], J& provides a form of *family polymorphism* [36]. All types indexed by a given dependent class—the dependent class itself, its prefix types, and its nested classes—are members of a *family* of interacting classes and packages. By initializing a variable with instances of dif-

ferent classes, the same code can refer to classes in different families with different behaviors. In the context of a given class, other classes and packages named using `this.class` are in the same family as the actual run-time class of `this`. In Figure 3.3, `pair.Pair.accept`'s formal parameter `v` has type `base[this.class].Visitor`. If `this` is a `pair.Pair`, `base[this.class].Visitor` must be a `pair.Visitor`, ensuring the call to `visitPair` will not cause a run-time type error.

The type system ensures that types in different families (and hence indexed by different access paths) cannot be confused with each other accidentally: a `base` object cannot be used where a `pair` object is expected, for example. However, casts with run-time type checks allow an escape hatch that can enable wider code reuse. When an object is cast to a dependent class p`.class`, a run-time check is done to ensure the object has the same run-time class as p. This feature allows objects indexed by different access paths to be explicitly coerced into another family of types, which is not possible with virtual class mechanisms.

Nested inheritance can operate at every level of the containment hierarchy. A J& class nested within one namespace can be subclassed by a class in a different namespace; virtual classes, in contrast, only support subclassing of other classes nested within the same containing object. For example, suppose a collections library `util` is implemented in J& as a set of mutually dependent interoperating classes. A user can extend the class `util.LinkedList` to a class `MyList` not nested within `util`. A consequence of this feature is that a prefix type P`[`T`]` may be defined even if T is not directly nested within P or within a subtype of P. When the current object `this` is a `MyList`, the prefix type `util[this.class]` is well-formed and refers to the `util` package, even though `MyList` is not a member class of `util`.

4.4 Non-final access paths

To ensure soundness, the type p.`class` is well-formed only if p is final. However, to improve expressiveness and to ease porting of Java programs to J&, a non-final local variable x may be *implicitly coerced* to the type x.`class` under certain conditions. When x is used as an actual argument of a method call, a constructor call, or a `new` expression, or as the source of a field assignment, and if x is not assigned in the expression, then it can be implicitly coerced to type x.`class`. Consider the following code fragment using the classes of Figure 4.2:

```
base.Exp e = new pair.Pair();
e.accept(new base[e.class].TypeChecker());
```

In the call to `accept`, `e` is never assigned and hence its run-time class does not change between the time `e` is first evaluated and the time control is transferred to the method body. If `e` had been assigned, say to a `base.Exp`, the `new` expression would have allocated a `base.TypeChecker` and passed it to `pair.Pair.accept`, leading to a run-time type error. Implicit coercion is not performed for field paths, since it would require reasoning about aliasing and is in general unsafe for multithreaded programs.

4.5 Intersection types

Nested intersection of classes and packages in J& is provided in the form of *intersection types* [97, 29]. An intersection type $S\&T$ inherits all members of its base namespaces S and T. With nested intersection, the nested namespaces of S and T are themselves intersected.

To support composition of classes and packages inherited more than once, J& provides *shared* multiple inheritance: when a subclass (or subpackage) inherits from multiple base classes, the new subclass may inherit the same superclass from more

```
class A {
    class B  { void n() { } }
    class B2 { void n() { } }
    void m() { }
}
```

```
class A1 extends A {
    class B { }
    class C { }
    void m() { }
    void p() { }
}
```

```
class A2 extends A {
    class B { void n() { } }
    class C { }
    void m() { }
    void p() { }
}
```

```
abstract class D extends A1 & A2 { }
```

Figure 4.3: Multiple inheritance with name conflicts

than one immediate superclass; however, instances of the subclass will not contain multiple subobjects for the common superclass. For instance, `pair_and_sum.Visitor` in Figure 3.6 inherits from `base.Visitor` only once, not twice through both `pair` and `sum`. Similarly, the package `pair_and_sum` contains only one `Visitor` class, the composition of `pair.Visitor` and `sum.Visitor`.

Since an intersection class type does not have a class body in the program text, its inherited members cannot be overridden by the intersection itself; however, subclasses of the intersection may override members.

4.6 Name conflicts

When two namespaces declare members with the same name, a *name conflict* may occur in their intersection. How the conflict is resolved depends on where the name was introduced and whether the name refers to a nested class or to a method.

J& distinguishes between two kinds of name conflicts, identified by Borning and Ingalls [10], depending on where the name was introduced into the class hierarchy. If the name was introduced in a common ancestor of the intersected namespaces, members with that name are assumed to be semantically related. Otherwise, the name is assumed to refer to distinct members that coincidentally have the same name, but different semantics. For a given member name M, if $T_1.M$ and $T_2.M$ are semantically related, but refer to different implementations, then $T_1 \,\&\, T_2$ has an *implementation conflict* for M. On the other hand, if $T_1.M$ and $T_2.M$ are semantically distinct, then $T_1 \,\&\, T_2$ has an *unintentional conflict* for M.

4.6.1 Conflicts between nested namespaces

Implementation conflicts for nested namespaces are resolved by intersecting the nested namespaces; that is, when two namespaces are intersected, their corresponding nested namespaces are also intersected. In Figure 4.3, both `A1` and `A2` contain a nested class `B` inherited from `A`. Since a common ancestor introduces `B`, `A1.B` and `A2.B` are semantically related. The intersection type `A1 & A2` contains a nested class `(A1 & A2).B`, which is equivalent to `A1.B & A2.B`. The subclass `D` has an implicit nested class `D.B`, a subclass of `(A1 & A2).B`.

On the other hand, `A1` and `A2` both declare independent nested classes `C`. Even though these classes have the same name, they are assumed to be unrelated; thus, `A1 & A2` has an unintentional conflict for `C`. The class `(A1 & A2).C` is *ambiguous*. In fact, `A1 & A2` contains two nested classes named `C`, one that is a subclass of `A1.C` and one a subclass of `A2.C`. Class `D` and its subclasses can resolve the ambiguity by exploiting prefix type notation: `A1[D].C` refers to the `C` from `A1` and `A2[D].C` refers to the `C` from `A2`. In `A1`, references to the unqualified name `C` are interpreted as `A1[this.class].C`. If `this` is an instance

of D, these references refer to the A1.C. Similarly, references to C in A2 are interpreted as A2[this.class].C, and when this is a D, these references refer to A2.C.

4.6.2 Conflicts between methods

A similar situation occurs with the methods A1.p and A2.p, which have an unintentional conflict in A1&A2 and therefore in D also. As with nested classes D inherits both versions of p. Callers of D.p must resolve the ambiguity by up-casting the receiver to specify which one of the methods to invoke. This solution is also used for "super" calls. If the superclass is an intersection type, the call may be ambiguous. The ambiguity is resolved by up-casting the special receiver super to the desired superclass.

Finally, two or more intersected classes may declare methods that override a method declared in a common base class, causing an implementation conflict for that method. In Java, method calls are dispatched to the method body in the most specific class of the receiver that implements the method. When there is an implementation conflict for a method, there is not a most specific implementation of the method. J& distinguishes between explicit inheritance from a declared superclass and induced inheritance via further binding.

In the explicit case, illustrated by the method m in Figure 4.3, the method in the intersection type A1 & A2 is considered *abstract*. Because it has no class body, the intersection type cannot override the abstract method, and so is an abstract class cannot be instantiated. Subclasses of the intersection type—D in the example— must override m to resolve the conflict, or else also be declared abstract.

Another name conflict occurs with the method n in the implicit class A2.B2. Since both A2.B and A.B2 override A.B's implementation of n, A2.B2 has an implementation conflict for n. However, treating method dispatch conflicts between explicit superclasses (e.g., A2.B) and induced superclasses (A.B2) as a compiler-time error would effectively

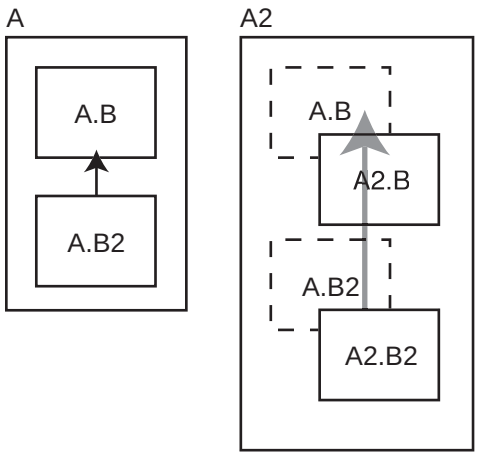

Figure 4.4: Dispatch order for `A.B2` and `A2.B2`

prevent a class from overriding any methods of a class it further binds; its implicit subclasses would inherit both implementations, resulting in an ambiguity the programmer must resolve.

Instead, we exploit the structure of the nested inheritance mechanism and prioritize explicit inheritance over induced inheritance. If a class explicitly inherits only one implementation of a method, the method is not considered ambiguous. In the example of Figure 4.3, the method `A.B2.n` is given priority over `A2.B.n` because the `B2` classes are specializations of the `B` classes. All `B2` classes are regarded as being more specific than any `B` class. The dispatch order for `n` for an `A2.B2` object is thus: `A2.B2`, `A.B2`, `A2.B`, `A.B`. This dispatch order is depicted in Figure 4.4.

4.7 Anonymous intersections

An instance of an intersection class type `A&B` may be created by explicitly invoking constructors of both `A` and `B`:

```
new A() & B();
```

```
class C { void n() { ... } }

class A1 {
    class B1 extends C { }
    class B2 extends C { }
    // B1 & B2 do not conflict
    void m() {
        new A1[this.class].B1() & A1[this.class].B2();
    }
}

class A2 extends A1 {
    class B1 extends C { void n() { ... } }
    class B2 extends C { void n() { ... } }
    // B1 & B2 conflict
}
```

Figure 4.5: Conflicts introduced by late binding

This intersection type is *anonymous*. As in Java, a class body may also be specified in the `new` expression, introducing a new anonymous subclass of `A&B`:

```
new A() & B() { ... };
```

If `A` and `B` have a name conflict that causes their intersection to be an abstract class, a class body must be provided to resolve the conflict.

Further binding may also introduce name conflicts. For example, in Figure 4.5, `A1.B1` and `A1.B2` do not conflict, but `A2.B1` and `A2.B2` do conflict. Since the anonymous intersection in `A1.m` may create an intersection of these two conflicting types, it should not be allowed. Because the type being instantiated is statically unknown, it is a compile-time error to instantiate an anonymous intersection of two or more dependent types (either dependent classes or prefixes of dependent classes); only anonymous intersections of non-dependent, non-conflicting classes are allowed. This restriction ensures the anonymous intersection is a non-empty type.

4.8 Prefix types and intersections

Unlike with virtual classes [38], it is possible in J& to extend classes nested within other namespaces. Multiple nested classes or a mix of top-level and nested classes may be extended, resulting in an intersection of several types with different containers. This flexibility is needed for effective code reuse but complicates the definition of prefix types. Consider this example:

```
class A { class B { B m(); ... } }
class A1 extends A { class B { B x = m(); } }
class A2 extends A { class B { } }
class C extends A1.B & A2.B { }
```

As explained in Section 4.1, the unqualified name `B` in the body of class `A.B` is sugar for the type `A[this.class].B`. The same name `B` in `A1.B` is sugar for `A1[this.class].B`. Since the method `m` and other code in `A.B` may be executed when `this` refers to an instance of `A1.B`, these two references to `B` should resolve to the same type; that is, it must be that `A[this.class]` is equivalent to `A1[this.class]`. This equivalence permits the assignment of the result of `m()` to `x` in `A1.B`. Similarly, the three types `A[C]`, `A1[C]`, and `A2[C]` should all be equivalent.

By defining prefix types as follows, we ensure the desired type equivalence. Two types P_1 and P_2 are *related by further binding* if they both contain nested types C introduced in a common ancestor P_0; that is, $P_1.C$ and $P_2.C$ are induced subclasses of $P_0.C$. For example, `A`, `A1`, `A2` are related by further binding since they all contain a class `B` that further binds `A.B`. We write $P_1 \sim P_2$ for the symmetric, transitive closure of this relation. In general, if $P_1 \sim P_2$, then $P_1[T]$ and $P_2[T]$ should be equivalent; otherwise, P_1 or P_2 might contain an unqualified type name that is late bound differently in the different classes, which can be confusing for the programmer. For example, if `A[C]` and `A1[C]` are not equivalent, then the assignment of the result of `m()` to `x` in `A1.B` should not be allowed, even though both `x` and `m()` have type `B`.

The prefix type P`[`T`]` is defined as the intersection of all types P_2, where $P \sim P_2$ where T has a supertype nested in P and a supertype nested in P_2. Using this definition `A`, `A1` and `A2` are all transitively related by further binding. Thus, `A[C]`, `A1[C]`, and `A2[C]` are all equivalent to `A1&A2`.

Prefix types impose some restrictions on which types may be intersected. If two classes T_1 and T_2 contain conflicting methods, then their intersection is abstract, preventing the intersection from being instantiated. If T_1 or T_2 contain member classes, a prefix type of a dependent class bounded by one of these member classes could resolve to the intersection $T_1 \,\&\, T_2$. To prevent these prefix types from being instantiated, all member classes of an abstract intersection are also abstract.

4.9 Constructors

Like Java, J& initializes objects using constructors. Since J& permits allocation of instances of dependent types, the class being allocated may not be statically known. Constructors in J& are inherited and may be overridden like methods, allowing the programmer to invoke a constructor of a statically known superclass of the class being allocated.

When a class declares a `final` field, it must ensure the field is initialized. Since constructors are inherited from base classes that are unaware of the new field, J& requires that if the field declaration does not have an explicit initializer, all inherited constructors must be overridden to initialize the field.

To ensure fields can be initialized to meaningful values, constructors are inherited only via induced inheritance, not via explicit inheritance. That is, the class $T'.C$ inherits constructors from $T.C$ when T is a supertype of T', but not from other superclasses of $T'.C$. If a constructor were inherited from both explicit and induced superclasses,

then every class that adds a `final` field would have to override the default `Object()` constructor to initialize the field. Since no values are passed into this constructor, the field may not be able to be initialized meaningfully.

Since a dependent class p.`class` may represent any subclass of p's statically known type, a consequence of this restriction is that p.`class` can only be explicitly instantiated if p's statically known class is `final`; in this case, since p.`class` is guaranteed to be equal to that `final` class, a constructor with the appropriate signature exists. The restriction does not prevent nested classes of dependent classes from being instantiated.

A constructor for a given class must explicitly invoke a constructor of its declared superclass. If the superclass is an intersection type, it must invoke a constructor of each class in the intersection. Because of multiple inheritance, superclass constructors are invoked by explicitly naming them rather than by using the `super` keyword as in Java. In Figure 4.6, `B.C` invokes the constructor of its superclass `A` by name.

Because J& implements shared multiple inheritance, an intersection class may inherit more than one subclass of a shared superclass. Invoking a shared superclass constructor more than once may lead to inconsistent initialization of `final` fields, possibly causing a run-time type error if the fields are used in dependent classes. There are two cases, depending on whether the intersection inherits one invocation or more than one invocation of a shared constructor.

In the first case, when the intersection inherits only one invocation of the shared constructor, then all calls to the shared superclass's constructor originate from the same call site. Thus, every inherited call to the shared constructor will pass the same arguments. In this case, the programmer need do nothing; the operational semantics of J& will ensure that the shared constructor is invoked exactly once.

For example, in Figure 4.6, the implicit class `D.C` is a subclass of `B1.C&B2.C` and shares the superclass `A`. Since `B1.C` and `B2.C` both inherit their `C(int)` constructor from

```
class A { A(int x); }

class B {
  class C extends A { C(int x) { A(x+1); } }
}

class B1 extends B {
  class C extends A { void m(); }
}

class B2 extends B { }
  class C extends A { void p(); }
}

class D extends B1 & B2 { }
```

Figure 4.6: Constructors of a shared superclass

`B.C`, both inherited constructors invoke the `A` constructor with the same arguments. There is no conflict and the compiler need only ensure that the constructor of `A` is invoked exactly once, before the body of `D.C`'s constructor is executed. Similarly, if the programmer invokes:

```
new (B1 & B2).C(1);
```

there is only one call to the `A(int)` constructor and no conflict.

If, on the other hand, the intersection contains more than one call site that invokes a constructor of the shared superclass, or of the intersection itself is instantiated so that more than one constructor is invoked, then the programmer must resolve the conflict by specifying the arguments to pass to the constructor of the shared superclass. The call sites inherited into the intersection will *not* be invoked. It is up to the programmer to ensure that the shared superclass is initialized in a way that is consistent with how its subclasses expect the object to be initialized.

In Figure 4.6, if one or both of `B1` and `B2` were to override the `C(int)` constructor, then `B1.C` and `B2.C` would have different constructors with the same signature. One

of them might change how the C constructor invokes A(int). To resolve the conflict, D must further bind C to specify how C(int) should invoke the constructor of A. This behavior is similar to that of constructors of shared virtual base classes in C++.

There would also be a conflict if the programmer were to invoke:

```
new B1.C(1) & B2.C(2);
```

The A(int) constructor would be invoked twice with different arguments. Thus, this invocation is illegal; however, since B1.C&B2.C is equivalent to (B1&B2).C, the intersection can be instantiated using the latter type, as shown above.

4.10 Type substitution

Because types may depend on final access paths, type-checking method calls requires substitution of the actual arguments for the formal parameters. A method may have a formal parameter whose type depends upon another parameter, including this. The actual arguments must reflect this dependency. For example, the class base.Abs in Figure 4.2 contains the following call:

```
v.visitAbs(this_A);
```

to a method of base.Visitor with the signature:

```
void visitAbs(base[this_V.class].Abs a);
```

For clarity, each occurrence of this has been labeled with an abbreviation of its declared type. Since the formal type base[this$_V$.class].Abs depends on the receiver this$_V$, the type of the actual argument this$_A$ must depend on the receiver *v*.

The type checker substitutes the actual argument types for dependent classes occurring in the formal parameter types. In this example, the receiver v has the

type base[this$_A$.class].Visitor, which when substituted for this$_V$.class in base[this$_V$.class].Abs yields base[base[this$_A$.class].Visitor].Abs. This type is equivalent to base[this$_A$.class].Abs.

The type substitution semantics of J& generalize the original Jx substitution rules [83] to increase expressive power. However, to ensure soundness, some care must be taken. If the type of v were base.Visitor, then v might refer at run time to a pair.Visitor while at the same time this$_A$ refers to a base.Abs. Substitution of base.Visitor for this$_V$.class in the formal parameter type would yield base[base.Visitor].Abs, which is equivalent to base.Abs. Since the corresponding actual argument has type base[this$_A$.class].Abs, which is a subtype of base.Abs, the call would incorrectly be permitted, leading to a potential run-time type error. The problem is that there is no guarantee that the run-time classes of this$_A$ and v both have the same enclosing base package.

To remedy this problem, type substitution must *preserve exact types*; that is, when substituting into an exact type—a dependent class or a prefix of a dependent class—the resulting type must also be exact. This ensures that the run-time class or package represented by the type remains fixed. Substituting the type base[this$_A$.class].Visitor. for this$_V$.class is permitted since both base[this$_V$.class] and base[this$_A$.class] are exact. However, substituting base.Visitor for this$_V$.class is illegal since base is not exact; therefore, a call to visitAbs where v is declared to be a base.Visitor is not permitted.

Implicit coercion of a non-final local variable x to dependent class x.class, described in Section 4.4, enhances the expressiveness of J& when checking calls by enabling x.class to be substituted for a formal parameter or this. Since this substitution preserves exactness, the substitution is permitted. If x's declared type were substituted for the formal instead, exactness might not have been preserved.

```
package pair;

class TargetExp = base.Exp;
class Rewriter {
    TargetExp rewrite(Exp e) { ... }
}
```

```
package pair_and_sum extends pair;

class TargetExp = pair.Exp;
class Rewriter {
    TargetExp rewrite(Exp e) { ... }
}
```

Figure 4.7: Static virtual types

4.11 Static virtual types

Dependent classes and prefix types enable classes nested within a given containment hierarchy of packages to refer to each other without statically binding to a particular fixed package. This allows derived packages to further bind a class while preserving its relationship to other classes in the package. It is often useful to refer to other classes *outside* the class's containment hierarchy without statically binding to a particular fixed package. J& provides *static virtual types* to support this feature. Unlike virtual types in BETA [68], a static virtual type is an attribute of an enclosing package or class rather than of an enclosing object.

In Figure 4.7, the package `pair` declares a static virtual type `TargetExp` representing an expression of the target language of a rewriting pass, in this case an expression from the `base` compiler. The `rewrite` method takes an expression with type `pair[this.class].Exp` and returns a `base.Exp`. The `pair_and_sum` package extends the `pair` package and further binds `TargetExp` to `pair.Exp`. A static virtual type can be further bound to any subtype of the original bound. Because `pair_and_sum.TargetExp`

```
class A { }
class A1 extends A { }
class A2 extends A { }

class B { class T = A; }
class B1 extends B { class T = A1; }
class B2 extends B { class T = A2; }

// (B1 & B2).T is A1 & A2
```

Figure 4.8: Static virtual types and intersections

is bound to `pair.Exp`, the method `pair_and_sum.Rewriter.rewrite` must return a `pair.Exp`, rather than a `base.Exp` as in `pair.Rewriter.rewrite`.

With intersections, a static virtual type may be inherited from more than one superclass. Consider the declarations in Figure 4.8. Class `B1&B2` inherits `T` from both `B1` and `B2`. The type `(B1&B2).T` must be a subtype of both `A1` and `A2`; thus, `(B1&B2).T` is bound to `A1&A2`.

To enforce exactness preservation by type substitution, static virtual types can be declared `exact`. For a given container namespace T, all members of the `exact` virtual type $T.C$ are of the same fixed run-time class or package. Exact virtual types can be further bound in a subtype of their container. For example, consider these declarations:

```
class B { exact class T = A; }
class B2 extends B { exact class T = A2; }
```

The exact virtual type `B.T` is equivalent to the dependent class `(new A).class`; that is, `B.T` contains only instances with run-time class `A` and not any subtype of `A`. Similarly, `B2.T` is equivalent to `(new A2).class`. If a variable `b` has declared type `B`, then an instance of `b.class.T` may be either a `A` or a `A2`, depending on the run-time class of `b`.

4.12 Genericity

Like virtual types in BETA [68], static virtual types in J& can be used as a genericity mechanism. For example, the following code fragment implements a generic `List` class and a `List` of `Integer`s, `IntList`:

```
class List {
    static abstract class T = Object;
    void add(this.class.T x) { ... }
}
class IntList extends List {
    static class T = Integer;
}
```

By declaring `IntList.T` to be an alias for `Integer`, the `add` method may be called with an argument of type `Integer`. An alternative implementation, using only nested classes might declare `IntList.T` as

```
class IntList extends List {
    class T extends Integer { }
}
```

However, because `IntList.T` and `Integer` are not equal in this case, only instances of `IntList.T` can be added to an `IntList`, not instances of the `Integer` class itself.

Nested inheritance is intended to be a mechanism for extensibility and not for genericity. J& is an extension of Java, which as of version 1.5, Java already has a genericity mechanism, namely parameterized types. Using parameterized types, a list of `Integer` can be implemented more succinctly as the parameterized type `List<Integer>`.

4.13 Supertype declarations

To simplify the semantics and the implementation of J&, a class or package may not extend any of its containing namespaces. This restriction prevents the class or package

from containing or further binding itself. For the same reason, the supertype declaration cannot be an exact type or a type dependent on a field path, including static fields. Therefore, a namespace may not extend `thisclass` or `thispackage` or their prefixes; however, to enable virtual superclasses, a namespace may extend a nested namespace of `thisclass` or `thispackage` (e.g., `thisclass.B` in Figure 4.1).

When further binding a class in a containing namespace, the programmer can change the superclass. This feature allows new functionality to be mixed in to several classes in the new containing class without code duplication. The superclass can only be changed covariantly; that is, it is required that the new superclass be a subtype of the old. This ensures that calls to the superclass using `super` do not cause a type error when inherited into the further bound class.

4.14 Conformance

In J&, a class conforms to its superclass under the same rules as in Java 1.4: a method's parameter types and return type must be identical in both classes. In principle, this rule could be relaxed to permit covariant refinement of method return types and contravariant refinement of method parameter types, but we have not explored this relaxation.

4.15 Final binding

As in Java, classes in J& may be declared `final` to prevent the class from being subclassed. This naturally extends to nested inheritance be requiring that a `final` nested class can be neither subclassed explicitly with an `extends` declaration nor overridden in a subclass of its enclosing class. This *final binding* of nested classes is useful for enabling optimizations and for modeling purposes. In addition, virtual classes in BETA

may be subclassed only if they are final bound. Since J& does not permit inheritance from dependent classes, this restriction is not needed in J&.

Final classes also enable backward compatibility with Java; if all nested classes are `final`, a J& program is a legal Java program.

4.16 Run-time type checking

As in Java, J& code can test the run-time type of an expression using `instanceof` and cast expressions. An expression may also be checked to see if it is a member of a dependent type. To check if an expression *e* is a member of *p*.`class`, the *e*'s run-time class is compared for equality with *p*'s run-time class. Run-time types checks for cast expressions are handled similarly.

4.17 Exceptions

Exceptions in J& are treated similarly to Java: Any subclass of `java.lang.Throwable` can be thrown. Methods must declare the set of exceptions they throw. In J&, the throws set may include dependent types. For instance, in the following code the method `m` throws exceptions dependent on `this` and on the formal parameter `d`.

```
class A {
    class E extends Exception { }
    class B { }
    void m(final A.B b) throws this.class.E, A[b.class].E {
        ...
    }
}
```

A `catch` statement may also catch dependent exceptions. As with Java, a run-time type check is performed on the exception object. The semantics of the check are similar to the `instanceof` expression described in Section 4.16.

```
class A1 {
    static class B1 {
        void m() throws Exception { throw new Exception(); }
    }
    static class B2 extends B1 {
        void m() throws Exception { throw new Exception(); }
    }
}

class A2 extends A1 {
    static class B1 {
        void m() { } // throws nothing: not allowed in J&
    }
}
```

Figure 4.9: Throws sets

Subclasses may refine the set of exceptions a method throws by removing an exception from the set or by adding a subclass of a declared exception. To ensure modularity, a further bound class may not refine the set of exceptions thrown; that is, if $T_2.C$ further binds $T_1.C$, the throws set of $T_2.C.m$ must equal the throws set of $T_1.C.m$. Without this restriction, in the code in Figure 4.9, the implicit class `A2.B2` would inherit `A1.B2.m`, which throws an exception and `A2.B1.m`, which does not. As described in Section 4.6.2, the J& dispatch order for `A2.B2` invokes `A1.B2.m` before `A2.B1.m`. Requiring `A2.B1.m` to have the same throws set as `A1.B1.m` allows `A2` to by type-checked without checking each of its implicit classes.

4.18 Packages

J& supports inheritance of packages, including multiple inheritance. In fact, the most convenient way to use nested inheritance is usually at the package level, because large software is usually contained inside packages, not classes. The semantics of prefix packages and intersection packages are similar to those of prefix and intersection class types,

described above. Since packages do not have run-time instances, the only exact packages are prefixes of a dependent class nested within the package, e.g., `pkg[x.class]`, where `x` is an instance of class `pkg.C`.

To specify package inheritance relationships, the programmer creates a file named `thispackage.jx` in the package directory. This file typically contains a one-line package declaration of the form:

```
package p extends T;
```

The superpackage T is interpreted in the context of the containing namespace of p. Thus, T may mention `thispackage`, enabling T to be a virtual superpackage. Packages declarations may also contain static virtual types and static virtual package declarations, as shown in the following example:

```
package p2 extends thispackage.p1 {
    exact package q = thispackage;
}
```

In this case, `p2.q` is bound to `p2` and can be used as an alias for `p2`.

Package declarations may also be nested within classes. Packages in classes may not themselves contain nested class declarations, but may declare static virtual types and nested packages (which are similarly restricted).

Chapter 5

An Extensible Compiler in J&

Using the language features described in Chapter 4 we can construct composable, extensible systems. In this section, we sketch the design of a composable, extensible compiler. Most of the design described here was used in our port to J& of the Polyglot compiler framework [84] except where necessary to maintain backward compatibility with the Java version of Polyglot.

The `base` package and packages nested within it contain all compiler code for the base language: Java, in the Polyglot framework. The nested packages `base.ast`, `base.types`, and `base.visit` contain classes for AST nodes, types, and visitors that implement compiler passes, respectively. All AST nodes are implemented as subclasses of `base.ast.Node`; compiler passes are implemented as subclasses of `base.visit.Visitor`.

5.1 Orthogonal extension

Scalable, orthogonal extension of the base compiler with new data types and new operations is achieved through nested inheritance. To extend the compiler with new syntax, the `base` package is extended and new subclasses of `Node` can be added to the

`ast` package. New passes can be added to the compiler by creating new subclasses of `visit.Visitor` subclasses.

Because the Visitor design pattern [43] is used to implement compiler passes, when a new AST node class is added to an extension's `ast` package, a `visit` callback method for the class must be added to the extension's `Visitor` class. Because the classes implementing the compiler passes extend `base[this.class].visit.Visitor`, this `visit` method is inherited by all `Visitor` subclasses in the extension. Visitor classes in the framework can transform the AST by returning new AST nodes. The `Visitor` class implements default behavior for the `visit` method by simply returning the node passed to it, thus implementing an identity transformation. Visitors for passes affected by the new syntax can be overridden to support it.

5.2 Composition

Independent compiler extensions can be composed using nested intersection with minimal effort. If the two compiler extensions are orthogonal, as for example with the product and sum type compilers of Section 3.2, then composing the extensions is trivial: the `main` method needs to be overridden in the composing extension to specify the order in which passes inherited from the composed extensions should run.

If the language extensions have conflicting semantics, this will often manifest as a name conflict when intersecting the classes within the two compilers. These name conflicts must be resolved to be able to instantiate the composed compiler, forcing the compiler developer to reconcile the conflicting language semantics.

However, even without name conflicts, there may be semantic conflicts in the composed compiler. The composed compiler will run, but might not implement the

language the programmer expects. In general, it is up to the programmer to detect and resolve semantics conflicts between the composed compilers.

5.3 Extensible rewriters

One challenge for building extensible software systems is to provide extensible data processing, particularly when the input and output data have complex structure. Extensions to the software need to be able to scalably and modularly extend both the transformations performed on the data and the data being transformed. Compilers exhibit this difficulty, because compiler passes perform complex transformations on complex data structures representing program code. For scalable extensibility, it should not be necessary to change data transformers (e.g., compiler passes) if the extensions to the data representation do not interact with the transformation in question.

One challenge for building an extensible compiler is to implement transformations between different program representations. For example, the `pair` compiler from Chapter 3 (Figure 3.3) transforms expressions with pairs into lambda calculus expressions. For a given transformation between two representations, compiler extensions need to be able to scalably and modularly extend both the source and target representations and the transformation itself. However, if the extensions to the source and target representations do not interact with a transformation, it should not be necessary to change the transformation.

A partial solution to this problem is the Visitor design pattern [43], which supports scalable extension of data processing. It allows boilerplate traversal of the input data structure to be factored out and shared. With minor extensions, visitors also support the generation of structured output data.

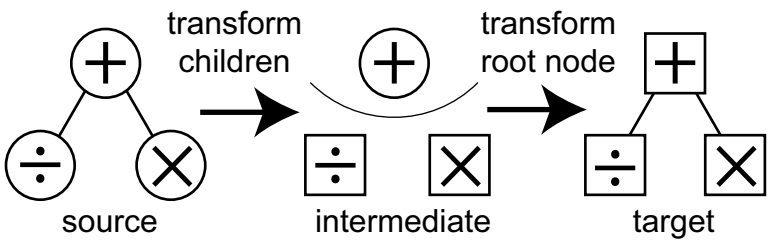

Figure 5.1: AST transformation

Consider an abstract syntax tree (AST) node representing a binary operation. As illustrated in Figure 5.1, most compiler transformations for this kind of node would recursively transform the two child nodes representing the operands, then invoke pass-specific code to transform the binary operation node itself, in general constructing a new node using the new children. The generic code for invoking the pass recursively on the children and constructing a new node can be shared across most compiler passes, avoiding duplication of code for every pass.

However, code for a given base compiler transformation might not be aware of the particular extended AST form used by a given compiler extension. The extension may have added new children to the node in the source representation of which the transformation is unaware. It is therefore hard to write a reusable compiler pass; the pass may fail to transform all the node's children or attributes.

In the `pair` compiler of Figure 3.3, the `TranslatePairs` pass transforms `pair` AST nodes into `base` AST nodes. If this compiler pass is reused in a compiler in which expressions have, say, additional type annotations, the source and target languages node will have children for these additional annotations, but the pass will not be aware of them and will fail to transform them.

Static virtual types (Section 4.11) are used to make a pass aware of any new children added by extensions of the source language, while preserving modularity. The solution is for the compiler to explicitly represent nodes in the intermediate form as trees with

```
package base.ast_struct;

exact package child = ast_struct;

abstract class Exp { }

class Abs extends Exp {
    String x; Exp e; // λx.e
}
```

```
package base.ast extends ast_struct;

exact package child = base.ast[this.class];

abstract class Exp {
    abstract v.class.target.Exp accept(Visitor v);
    void childrenExp(Visitor v, v.class.tmp.Exp t) { }
}

class Abs extends Exp {
    v.class.target.Exp accept(Visitor v) {
        v.class.tmp.Abs t = new v.class.tmp.Abs();
        childrenAbs(v, t);
        return v.visitAbs(this, t);
    }
    void childrenAbs(Visitor v, v.class.tmp.Abs t) {
        childrenExp(v, t);
        t.x = this.x;
        t.e = e.accept(v);
    }
}
```

```
package base.visit;

class Visitor {
    // source language = base[this.class].ast
    // target language <= base.ast;
    exact package target = base.ast;
    package tmp extends ast_struct {
        exact package child = target;
    }
    ...
}
```

Figure 5.2: Extensible rewriting example: base compiler

a root in the source language but children in the target language, corresponding to the middle tree of Figure 5.1. A fragment of a base lambda calculus compiler using this pattern is shown in Figure 5.2. An extension of the base compiler with pairs is shown in Figure 5.3. In this example, a node in intermediate form is an instance of `Pair` (or another class in the `pair` package) with children in the package `base`.

The packages `base.ast_struct` and `pair.ast_struct` define just the structure of each AST node. The `ast_struct` packages are then extended to create `ast` packages for the actual AST nodes. In the `ast_struct` package, children of each AST node reside in a `child` virtual package. The `ast` package extends the `ast_struct` package and further binds `child` to the `ast` package itself; the node classes in `ast` have children in the same package as their parent. By further binding `child` differently, `ast_struct` can be extended to create node classes in which the children are in a different language.

In the pattern, a visitor implementing a transformation pass rewrites a source language AST into target language AST. For example, `TranslatePairs` in Figure 5.3 transforms `pair.ast` nodes into `base.ast` nodes. The key to the design is to create a package `tmp` *inside each visitor class* for the intermediate form nodes of that visitor's specific source and target language. The `Visitor.tmp` package extends the `ast_struct` package, but further binds `child` to the `target` package, which represents the target language of the visitor transformation. Thus, AST node classes in the `tmp` package have children in the `target` package, but parent nodes are in the `tmp` package. Since `tmp` is a subpackage of `ast_struct` in the same enclosing package, nodes in this package have the same structure as nodes in the visitor's `ast_struct` package. Thus, if the `ast_struct` package is overridden to add new children to an AST node class, the intermediate nodes in the `tmp` package will also contain those children.

Each AST node class contains an `accept` method that takes a `Visitor` and invokes a callback in the visitor to transform the node. code to traverse the children of the node. In

```
package pair extends base;
```

```
package pair.ast_struct; // extends base.ast_struct

class Pair extends Exp {
    child.Exp fst, snd;
}
```

```
package pair.ast extends ast_struct; // and extends base.ast

class Pair extends Exp {
    v.class.target.Exp accept(Visitor v) {
        v.class.tmp.Pair t = new v.class.tmp.Pair();
        childrenPair(v, t);
        return v.visitPair(this, t);
    }
    void childrenPair(Visitor v, v.class.tmp.Pair t) {
        childrenExp(v, t);
        t.fst = fst.accept(v);
        t.snd = snd.accept(v);
    }
}
```

```
package pair.visit;

class TranslatePairs extends Visitor {
    exact package target = base.ast;
    target.Exp visitPair(ast.Pair old, tmp.Pair t) {
        return new target.App(... t.fst ... t.snd ...);
            // ((λx.λy.λf. f x y) t.fst) t.snd
    }
}
```

Figure 5.3: Extensible rewriting example: pair compiler

`pair.ast.Pair`, the `accept` method constructs a new `tmp.Pair` specific to the visitor, applies the pass to the pair's children nodes to initialize the intermediate pair object, then invokes the `visitPair` callback with both the original pair and the intermediate pair. In `pair.visit.TranslatePairs`, the callback method uses the `tmp.Pair` to access the rewritten children and creates a new node in the `target` (that is, `base.ast`) package.

Both the `child` and `target` virtual packages are declared to be `exact`. This ensures that the children of a `tmp` node are in the `target` package itself (in this case `base.ast`) and not a derived package of the target (e.g., `pair.ast`).

Chapter 6

Formal Semantics

This chapter presents a formal semantics for the core J& type system and sketches a soundness proof for the semantics. Several language features are not modeled formally, including packages, constructors, and static virtual types. The treatment here is based on the semantics of Jx [83] and uses some ideas from the formal semantics of Tribe [27] and Ernst et al.'s *vc* calculus [38].

6.1 Preliminaries

A grammar for the calculus is shown in Figure 6.1. Throughout the semantics, we use the notation $\overline{a}$ for the list $a_1, \ldots, a_n$ for $n \geq 0$. The length of $\overline{a}$ is written $|\overline{a}|$, and the empty list is written nil. We write $\{\overline{a}\}$ for the set containing the members of the list $\overline{a}$. A term with a list subterm should be interpreted as a list of terms; for example, $\overline{f} = \overline{e}$ should be read $f_1 = e_1, \ldots, f_n = e_n$. We also write $i..j$ for the set $\{i, i+1, \ldots, j\}$.

Programs Pr consist of a list of class declarations $\overline{L}$ and a "`main`" expression e. To avoid cluttering the semantics, we assume a fixed program Pr; all inference rules are implicitly parameterized on Pr. A class declaration L contains a class name C, a superclass declaration T, member classes $\overline{L}$, fields $\overline{F}$, and methods $\overline{M}$. A field declaration

programs	$Pr ::= \langle \overline{L}, e \rangle$
class declarations	$L ::= \texttt{class}\ C\ \texttt{extends}\ T\ \{\overline{L}\ \overline{F}\ \overline{M}\}$
field declarations	$F ::= [\texttt{final}]\ T\ f = e$
method declarations	$M ::= T\ m(\overline{T}\ \overline{x})\ \{e\}$
types	$T ::= \circ \mid T.C \mid p.\texttt{class} \mid P\texttt{[}T\texttt{]} \mid \&\overline{T}$
non-dependent types	$S ::= \circ \mid S.C \mid P\texttt{[}S\texttt{]} \mid \&\overline{S}$
classes	$P ::= \circ \mid P.C$
values	$v ::= \texttt{null} \mid \ell$
access paths	$p ::= v \mid x \mid p.f$
expressions	$e ::= v \mid x \mid e.f \mid e_0.f = e_1$
	$\mid e_0.m(\overline{e}) \mid \texttt{new}\ T(\overline{f} = \overline{e}) \mid e_1;\ e_2$
typing contexts	$\Gamma ::= \emptyset \mid \Gamma, x{:}T \mid \Gamma, \ell{:}S \mid \Gamma, p_1 = p_2$
heaps	$H ::= \emptyset \mid H, \ell \mapsto o$
objects	$o ::= S\ \{\overline{f} = \overline{v}\}$

Figure 6.1: Grammar

F may be final or non-final and consists of a type, field name, and default initializer expression. Methods M have a return type, formal parameters, and a method body; all formal parameters are final.

Following the semantics of Tribe [27], all classes are nested within a single top-level class $\circ$. Types T are either the top-level class $\circ$, nested classes $T.C$, dependent classes $p.\texttt{class}$, prefix types $P\texttt{[}T\texttt{]}$, or intersection types $\&\overline{T}$. The intersection type $\&\overline{T}$ can be read $T_1 \,\&\, \cdots \,\&\, T_n$. A nested class $\circ.C$ of the top-level class is abbreviated as C. Non-dependent types are written S and class names are written P. In the calculus, the prefix type $P\texttt{[}T\texttt{]}$ is well-formed only if some supertype of T is immediately enclosed by a subclass of P. More general prefix types can be constructed by desugaring to this form: for example, if `c` has type `A.B.C`, then `A[c.class]` desugars to `A[A.B[c.class]]`.

A value is either `null` or a location ℓ, which maps to an object on the heap of type S. A final access path p is either a value, a parameter x, or a final field access $p.f$. Expressions are values, parameters x, field accesses, field assignments, calls, allocation expressions, or sequences. Constructors are not modeled in the semantics; instead, a `new` expression may explicitly initialize fields of the new object. Fields not explicitly

$\boxed{CT(P)}$

$$\frac{Pr = \langle \overline{L}, e \rangle}{CT(\circ) = \texttt{class}\ \circ\ \texttt{extends}\ \&\mathsf{nil}\ \{\overline{L}\}}$$

$$\frac{\begin{array}{c} CT(P) = \texttt{class}\ C'\ \texttt{extends}\ T'\ \{\overline{L'}\ \overline{F'}\ \overline{M'}\} \\ \texttt{class}\ C\ \texttt{extends}\ T\ \{\ldots\} \in \overline{L'} \end{array}}{CT(P.C) = \texttt{class}\ C\ \texttt{extends}\ T\ \{\ldots\}}$$

Figure 6.2: Class table

initialized by the `new` expression are initialized by the default initializer in the field declaration.

Type checking is performed in a typing context Γ, which is a list of variable bindings $x{:}T$, location bindings $\ell{:}S$, and path equivalence constraints $p_1 = p_2$. Location bindings are used to type-check the heap during evaluation. Path equivalence constraints are used to assert equivalence of dependent types during evaluation. They are similar to the aliasing equations in the Tribe type system [27].

A heap H maps locations ℓ to objects o. An object is simply a record labeled with a non-dependent type S.

6.2 Non-dependent types

We begin by presenting definitions for classes P and non-dependent types S. All dependent types are bounded by a non-dependent type, which is used for looking up nested classes, fields, and methods.

$\boxed{\vdash P : \mathsf{class}}$

$$\frac{CT(P) \neq \bot}{\vdash P : \mathsf{class}} \qquad \text{(DEF-CT)}$$

$$\frac{\vdash P_1 : \mathsf{class} \qquad \vdash P_1 \sqsubset P_2 \qquad \vdash P_2.C : \mathsf{class}}{\vdash P_1.C : \mathsf{class}} \qquad \text{(DEF-INH)}$$

Figure 6.3: Well-formed classes

$\boxed{\vdash S : \mathsf{nondep}}$

$$\frac{\mathsf{mem}(S) \neq \emptyset}{\vdash S : \mathsf{nondep}}$$

Figure 6.4: Well-formed non-dependent types

6.2.1 Class lookup

The class table, CT, defined in Figure 6.2, maps class names P to class declarations. The class declaration for the top-level class $\circ$ simply contains the program's class declarations. We write $CT(P) = \bot$ if P has no definition.

The judgment $\vdash P : \mathsf{class}$, shown in Figure 6.3, states that P is a well-formed class; the judgment holds either when P is a class in the class table or when P further binds a defined class. The rule DEF-CT says a class is well-formed if it is in the class table CT. The rule DEF-INH states that $P_1.C$ is well-formed if $P_2.C$ is well-formed and if P_1 is a subclass of P_2, which is written $\vdash P_1 \sqsubset P_2$ and defined in Figure 6.6 in Section 6.2.2.

The judgment $\vdash S : \mathsf{nondep}$, defined in Figure 6.4, states that S is a well-formed non-dependent type. The definition uses the mem function, defined in Figure 6.5, which returns the set of classes P comprising a non-dependent type S. The subtyping rules, described in Section 6.3.10, ensure type S is equivalent to the intersection of all classes in $\mathsf{mem}(S)$. The mem function for $P[S]$ uses the prefix function, defined in Section 6.2.3.

$\boxed{\mathsf{mem}(S)}$

$$\frac{\vdash P : \mathsf{class}}{\mathsf{mem}(P) = \{P\}}$$

$$\frac{D = \{P_i \in \mathsf{mem}(S) \mid \vdash P_i.C : \mathsf{class}\}}{\mathsf{mem}(S.C) = \bigcup_{P_i \in D} P_i.C}$$

$$\mathsf{mem}(P\texttt{[}S\texttt{]}) = \mathsf{prefix}(P, S)$$

$$\mathsf{mem}(\&\overline{S}) = \bigcup_{S_i \in \overline{S}} \mathsf{mem}(S_i)$$

Figure 6.5: Class membership

6.2.2 Subclassing and further binding

Inheritance among classes is defined in Figure 6.6. The rules are similar to those defined for the language Tribe [27]. The judgment $\vdash P_1 \sqsubset_{\mathsf{sc}} P_2$ states that P_1 is a declared subclass of P_2. The rule SC simply looks up the superclass using the class table CT, substituting the container for occurrences of `this.class` in the superclass. Type substitution is defined in Figure 6.17. By the program well-formedness rules, described in Section 6.3.12, the only access path allowed in a superclass declaration is the `this` path, ensuring that the result of substituting for `this` is a non-dependent type.

The judgment $\vdash P_1.C \sqsubset_{\mathsf{fb}} P_2.C$ in rule FB states that $P_1.C$ further binds $P_2.C$ when P_1 inherits from P_2 and $P_2.C$ is well-formed. The $\sqsubset$ relation is derived from the explicit subclassing and further binding relations: $\vdash P_1 \sqsubset P_2$ if P_1 either explicitly subclasses or further binds P_2. The reflexive, transitive closure of $\sqsubset$ is $\sqsubset^*$.

The function $\mathsf{supers}(S)$ returns the set of all superclasses of S.

The relation $\sim$ in Figure 6.8 is an equivalence relation between classes that contain a common nested class C. This relation is used to define membership in a non-dependent prefix type $P\texttt{[}S\texttt{]}$.

$\boxed{\vdash P_1 \sqsubset_{\mathsf{sc}} P_2}$

$$\frac{\begin{array}{c}\vdash P_1 \sqsubset^* P \\ CT(P.C) = \texttt{class}\ C\ \texttt{extends}\ T\ \{\ldots\} \\ T\{\!\{\emptyset;\ P_1/\texttt{this}\}\!\} = S \\ P_2 \in \mathsf{mem}(S)\end{array}}{\vdash P_1.C \sqsubset_{\mathsf{sc}} P_2} \qquad \text{(SC)}$$

$\boxed{\vdash P_1 \sqsubset_{\mathsf{fb}} P_2}$

$$\frac{\vdash P_1 \sqsubset P_2 \qquad \vdash P_2.C : \mathsf{class}}{\vdash P_1.C \sqsubset_{\mathsf{fb}} P_2.C} \qquad \text{(FB)}$$

$\boxed{\vdash P_1 \sqsubset P_2}$

$$\frac{\vdash P_1 \sqsubset_{\mathsf{sc}} P_2}{\vdash P_1 \sqsubset P_2} \qquad \text{(INH-SC)}$$

$$\frac{\vdash P_1 \sqsubset_{\mathsf{fb}} P_2}{\vdash P_1 \sqsubset P_2} \qquad \text{(INH-FB)}$$

Figure 6.6: Subclassing and further binding

$$\mathsf{supers}(S) = \bigcup_{P \in \mathsf{mem}(S)} \{P' \mid \vdash P \sqsubset^* P'\}$$

Figure 6.7: Superclasses

$\boxed{\vdash P_1 \sim P_2}$

$$\frac{\vdash P_1.C \sqsubset_{\mathsf{fb}} P.C \qquad \vdash P_2.C \sqsubset_{\mathsf{fb}} P.C}{\vdash P_1 \sim P_2} \quad \text{(REL-FB)}$$

$$\vdash P \sim P \quad \text{(REL-REFL)}$$

$$\frac{\vdash P_1 \sim P_2}{\vdash P_2 \sim P_1} \quad \text{(REL-SYM)}$$

$$\frac{\vdash P_1 \sim P_2 \qquad \vdash P_2 \sim P_3}{\vdash P_1 \sim P_3} \quad \text{(REL-TRANS)}$$

Figure 6.8: Related by further binding

$$\begin{aligned}\mathsf{prefix}(P,S) = \{P' \mid \exists C, C'.\ & \\ & \vdash P \sim P' \\ & \wedge P.C \in \mathsf{supers}(S) \\ & \wedge P'.C' \in \mathsf{supers}(S)\}\end{aligned}$$

Figure 6.9: Auxiliary functions

6.2.3 Prefix types

The meaning of a non-dependent prefix type $P[S]$ is defined by the prefix function in Figure 6.9. The P-prefix of a non-dependent type S is the intersection of all classes P' where P and P' transitively share a nested class—that is, P and P' are equivalent under the $\sim$ relation—and S extends nested classes of both P and P'. The intuition behind the definition is that S extends some class that is contained in the intersection of P and P'. This definition ensures that if P is a subtype of P', then $P[S]$ is equal to $P'[S]$, as desired in Section 4.8.

$\boxed{\Gamma \vdash p : T \ \mathsf{final}}$

$$\frac{\Gamma \vdash S : \mathsf{type}}{\Gamma \vdash \mathtt{null} : S \ \mathsf{final}} \quad \text{(F-NULL)}$$

$$\frac{\ell : S \in \Gamma}{\Gamma \vdash \ell : S \ \mathsf{final}} \quad \text{(F-LOC)}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T \ \mathsf{final}} \quad \text{(F-VAR)}$$

$$\frac{\Gamma \vdash p : T \ \mathsf{final} \qquad \mathsf{ftype}(\Gamma, T, f) = \mathtt{final}\ T_f}{\Gamma \vdash p.f : T_f \ \mathsf{final}} \quad \text{(F-GET)}$$

Figure 6.10: Final access paths

6.3 Static semantics

6.3.1 Final access paths

The judgment $\Gamma \vdash p : T$ final in Figure 6.10 states that the access path p is a well-typed final access path in context Γ. The `null` path can take on any non-dependent type. A location path ℓ has the type declared in the typing context. A variable path x has the type declared in the context. Finally a field path $p.f$ is final if p is final with type T, and the type of the field path is determined by looking up the field type.

6.3.2 Aliasing

To type-check field accesses, the type system keeps track of aliases. The judgment $\Gamma \vdash p_1 = p_2$, defined Figure 6.11, states that two final access paths are aliases.

$\boxed{\Gamma \vdash p_1 = p_2}$

$$\frac{\ell.f = v \in \Gamma}{\Gamma \vdash \ell.f = v} \quad \text{(A-ENV)}$$

$$\frac{\Gamma \vdash p_1 = p_2 \qquad \Gamma \vdash p_1.f : T_f \text{ final} \qquad \Gamma \vdash p_2.f : T_f \text{ final}}{\Gamma \vdash p_1.f = p_2.f} \quad \text{(A-FIELD)}$$

$$\frac{\Gamma \vdash p : T \text{ final}}{\Gamma \vdash p = p} \quad \text{(A-REFL)}$$

$$\frac{\Gamma \vdash p_2 = p_1}{\Gamma \vdash p_1 = p_2} \quad \text{(A-SYM)}$$

$$\frac{\Gamma \vdash p_1 = p_2 \qquad \Gamma \vdash p_2 = p_3}{\Gamma \vdash p_1 = p_3} \quad \text{(A-TRANS)}$$

Figure 6.11: Aliasing

6.3.3 Non-dependent bounding types

The judgment $\Gamma \vdash T \lhd S$ in Figure 6.12 states that T has a non-dependent bounding type S. The only interesting rule is for dependent classes, BD-FIN. The rule uses aliasing to ensure that the type p_1.`class` has the same bound as p_2.`class` if p_1 and p_2 are aliases. Aliasing must be considered since if p_1 and p_2 are aliases, we want p_1.`class` and p_2.`class` to have equivalent bounds. Because of BD-FIN, the bounding type is not necessarily unique.

6.3.4 Member lookup

Method and field lookup functions are shown in Figure 6.13. For a class P, $\mathsf{ownFields}(P)$ and $\mathsf{ownMethods}(P)$ is the set of fields and methods declared in the class. Using these definitions, the set of fields and methods declared or inherited by a non-dependent type S is defined by the $\mathsf{fields}(S)$ and $\mathsf{methods}(S)$ functions. The function fnames returns the

$\boxed{\Gamma \vdash T \lhd S}$

$$\Gamma \vdash P \lhd P \qquad \text{(BD-SIMP)}$$

$$\frac{\Gamma \vdash T \lhd S}{\Gamma \vdash T.C \lhd S.C} \qquad \text{(BD-NEST)}$$

$$\frac{\begin{array}{c}\Gamma \vdash p_1 = p_2 \\ \Gamma \vdash p_1 : T_1 \text{ final} \qquad \Gamma \vdash T_1 \lhd S_1 \\ \Gamma \vdash p_2 : T_2 \text{ final} \qquad \Gamma \vdash T_2 \lhd S_2\end{array}}{\Gamma \vdash p_1.\texttt{class} \lhd S_1 \,\&\, S_2} \qquad \text{(BD-FIN)}$$

$$\frac{\Gamma \vdash T \lhd S}{\Gamma \vdash P[T] \lhd P[S]} \qquad \text{(BD-PRE)}$$

$$\frac{\forall i.\ \Gamma \vdash T_i \lhd S_i}{\Gamma \vdash \&\overline{T} \lhd \&\overline{S}} \qquad \text{(BD-MEET)}$$

Figure 6.12: Type bounds

set of field names for a list of fields $\overline{F}$. The ftype function returns the declared type of a field f of an arbitrary type T in typing context Γ. The mtype function provides similar functionality for methods.

The method body for a method m in type S is returned by mbody. For simplicity, the formal semantics presented here do not specify what method body to dispatch to when one method overrides another; precise specification of method dispatch is not necessary to prove soundness of the type system.

6.3.5 Access paths

The function $\mathsf{paths}(T)$ in Figure 6.14 returns the set of access paths in the structure of type T. The paths function is used in the well-formedness rule for intersection types, in Section 6.3.7.

$$\frac{CT(P) = \texttt{class}\ C\ \texttt{ext}\ T\ \{\overline{L}\ \overline{F}\ \overline{M}\}}{\begin{array}{c}\mathsf{ownFields}(P) = \overline{F} \\ \mathsf{ownMethods}(P) = \overline{M}\end{array}}$$

$$\frac{CT(P) = \bot}{\begin{array}{c}\mathsf{ownFields}(P) = \emptyset \\ \mathsf{ownMethods}(P) = \emptyset\end{array}}$$

$$\mathsf{fields}(S) = \bigcup_{P_i \in \mathsf{supers}(S)} \mathsf{ownFields}(P_i)$$

$$\mathsf{methods}(S) = \bigcup_{P_i \in \mathsf{supers}(S)} \mathsf{ownMethods}(P_i)$$

$$\frac{\overline{F} = [\texttt{final}]\ \overline{T}\ \overline{f} = \overline{e}}{\mathsf{fnames}(\overline{F}) = \{\overline{f}\}}$$

$$\frac{\begin{array}{c}\Gamma \vdash T \lhd S \\ \mathsf{fields}(S) = \overline{F} \\ F_i = [\texttt{final}]\ T_f\ f = e\end{array}}{\mathsf{ftype}(\Gamma, T, f) = [\texttt{final}]\ T_f}$$

$$\frac{\begin{array}{c}\Gamma \vdash T \lhd S \\ \mathsf{fields}(S) = \overline{F} \\ F_i = [\texttt{final}] T\ f = e\end{array}}{\mathsf{finit}(S, f) = e}$$

$$\frac{\begin{array}{c}\Gamma \vdash T \lhd S \\ \mathsf{methods}(S) = \overline{M} \\ M_i = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\}\end{array}}{\mathsf{mtype}(\Gamma, T, m) = (\overline{x} : \overline{T}) \to T_{n+1}}$$

$$\frac{\begin{array}{c}\Gamma \vdash T \lhd S \\ \mathsf{methods}(S) = \overline{M} \\ M_i = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\}\end{array}}{\mathsf{mbody}(S, m) = M_i}$$

Figure 6.13: Member lookup

$$
\begin{aligned}
\mathsf{paths}(\circ) &= \emptyset \\
\mathsf{paths}(T.C) &= \mathsf{paths}(T) \\
\mathsf{paths}(p.\texttt{class}) &= \{p\} \\
\mathsf{paths}(P[T]) &= \mathsf{paths}(T) \\
\mathsf{paths}(\&\overline{T}) &= \bigcup_{T_i \in \overline{T}} \mathsf{paths}(T_i)
\end{aligned}
$$

Figure 6.14: Access paths

$$
\begin{aligned}
\mathsf{prefixExact}(\circ, k) &= \mathsf{false} \\
\mathsf{prefixExact}(T.C, k) &= \begin{cases} \mathsf{false} & \text{if } k = 0 \\ \mathsf{prefixExact}(T, k-1) & \text{otherwise} \end{cases} \\
\mathsf{prefixExact}(p.\texttt{class}, k) &= \mathsf{true} \\
\mathsf{prefixExact}(P[T], k) &= \mathsf{prefixExact}(T, k+1) \\
\mathsf{prefixExact}(\&\overline{T}, k) &= \bigvee_{T_i \in \overline{T}} \mathsf{prefixExact}(T_i, k) \\
\mathsf{exact}(T) &= \mathsf{prefixExact}(T, 0)
\end{aligned}
$$

Figure 6.15: Prefix exactness

6.3.6 Exactness

When type-checking calls, type substitution must preserve exact types; that is, if an exact type is substituted into, the result must be exact also. Since a type may have an embedded exact type, we define exactness using $\mathsf{prefixExact}(T,k)$, defined in Figure 6.15. The function $\mathsf{prefixExact}(T,k)$ is true if the kth prefix of T is an exact type for $k \geq 0$. A type T is exact if $\mathsf{prefixExact}(T,0)$ holds. Since, if T is exact, all of its prefixes (if they exist) are also exact, if $\mathsf{prefixExact}(T,k)$, then $\mathsf{prefixExact}(T,k+1)$; thus, $\mathsf{prefixExact}(p.\texttt{class},k)$ for any k.

$\boxed{\Gamma \vdash T : \mathsf{type}}$

$$\frac{CT(P) \neq \bot}{\Gamma \vdash P : \mathsf{type}} \quad \text{(WF-SIMP)}$$

$$\frac{\Gamma \vdash T : \mathsf{type} \qquad \Gamma \vdash T \lhd S \qquad \vdash S.C : \mathsf{nondep}}{\Gamma \vdash T.C : \mathsf{type}} \quad \text{(WF-NEST)}$$

$$\frac{\Gamma \vdash p : T\ \mathsf{final}}{\Gamma \vdash p.\mathtt{class} : \mathsf{type}} \quad \text{(WF-FIN)}$$

$$\frac{\Gamma \vdash P : \mathsf{type} \qquad \Gamma \vdash T : \mathsf{type} \qquad \Gamma \vdash T \lhd S \qquad \mathsf{prefix}(P,S) \neq \emptyset}{\Gamma \vdash P[T] : \mathsf{type}} \quad \text{(WF-PRE)}$$

$$\frac{\begin{array}{c}\forall i.\ \Gamma \vdash T_i : \mathsf{type} \\ \forall p_i, p_j \in \mathsf{paths}(\&\overline{T}).\ \Gamma \vdash p_i = p_j \\ \forall T_i, T_j \in \overline{T}.\ \mathsf{prefixExact}(T_i,k) \Leftrightarrow \mathsf{prefixExact}(T_j,k)\end{array}}{\Gamma \vdash \&\overline{T} : \mathsf{type}} \quad \text{(WF-MEET)}$$

Figure 6.16: Type well-formedness

6.3.7 Type well-formedness

Type well-formedness is defined in Figure 6.16. The judgment $\Gamma \vdash T : \mathsf{type}$ states that type T is well-formed in a context Γ. A class P is well-formed if it is in the class table CT. A nested type $T.C$ is well-formed if T is well-formed and has bound S and if $S.C$ is a well-formed non-dependent type. From the two rules WF-SIMP and WF-NEST, it is easy to see that if $\vdash P : \mathsf{class}$ then $\Gamma \vdash P : \mathsf{type}$ for any typing context Γ. A dependent class $p.\mathtt{class}$ is well-formed if p is a final access path. A prefix type $P[T]$ is well-formed if P and T are both well-formed and if T has simple bound S and $\mathsf{prefix}(P,S)$ is not empty; in other words, there is some superclass of T whose enclosing class is related to P by further binding. Finally, an intersection type $\&\overline{T}$ is well-formed if all three of the following conditions hold:

1. All constituent types T_i are well-formed.
2. All access paths free in $\&\overline{T}$ are aliases, ensuring all paths refer to the same run-time class. If this condition does not hold, the intersection could be the empty type.
3. All constituent types T_i have the same level of exactness; that is if $\mathsf{prefixExact}(T_i, k)$ for any i, then $\mathsf{prefixExact}(T_j, k)$ for all T_j. Thus, for example, $P[p.\texttt{class}]\ \&\ p.\texttt{class}$ is not well-formed, which is desirable since the intersection is empty. The conditional also helps to ensure that after substituting of an intersection type into a path preserves exactness.

6.3.8 Type substitution

The rules for type substitution are shown in Figure 6.17. The function $T\{\!\{\Gamma;\ T_x/x\}\!\}$ substitutes T_x for x in T. The typing context Γ is used to look up field types when substituting a non-dependent class into a field-path dependent class. T_x should be well-formed in Γ and a subtype of x's declared type.

6.3.9 Typing

For arbitrary expressions, the judgment $\Gamma \vdash e : T$, defined in Figure 6.18, states that e has type T in context Γ.

Any final access path p has type $p.\texttt{class}$ by T-FIN. The subtyping rule S-FIN and the subsumption rule T-SUB give the standard typing rules for values and parameters x:

$$\frac{\ell : S \in \Gamma}{\Gamma \vdash \ell : S} \qquad\qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

By T-GET, the type of a field access $e.f$ is obtained by looking up the field in T, the static type of e. The rule T-SET checks if the source expression in an assignment has the same type as the target expression.

$\boxed{T\{\!\{\Gamma;\ T_x/x\}\!\}}$

$$\circ\{\!\{\Gamma;\ T_x/x\}\!\} = \circ$$

$$T.C\{\!\{\Gamma;\ T_x/x\}\!\} = T\{\!\{\Gamma;\ T_x/x\}\!\}.C$$

$$v.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = v.\texttt{class}$$

$$\frac{x \neq y}{y.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = y.\texttt{class}}$$

$$x.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = T_x$$

$$\frac{p.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = p'.\texttt{class}}{p.f.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = p'.f.\texttt{class}}$$

$$\frac{\begin{array}{c}p.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = T_p \\ T_p \neq p'.\texttt{class} \\ \mathsf{ftype}(\Gamma, T_p, f) = [\texttt{final}]\ T_f\end{array}}{p.f.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = T_f}$$

$$\frac{T\{\!\{\Gamma;\ T_x/x\}\!\} = T'}{P[\![T]\!]\{\!\{\Gamma;\ T_x/x\}\!\} = P[\![T']\!]}$$

$$\frac{\forall i.\ T_i\{\!\{\Gamma;\ T_x/x\}\!\} = T_i'}{\&\overline{T}\{\!\{\Gamma;\ T_x/x\}\!\} = \&\overline{T'}}$$

Figure 6.17: Type substitution

$\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma \vdash p : T \ \mathsf{final}}{\Gamma \vdash p : p.\mathtt{class}} \quad \text{(T-FIN)}$$

$$\frac{\Gamma \vdash e : T \qquad \mathsf{ftype}(\Gamma, T, f) = [\mathtt{final}]\ T_f}{\Gamma \vdash e.f : T_f} \quad \text{(T-GET)}$$

$$\frac{\Gamma \vdash e_0 : T_0 \qquad \Gamma \vdash e_1 : T_f \qquad \mathsf{ftype}(\Gamma, T_0, f) = T_f}{\Gamma \vdash e_0.f = e_1 : T_f} \quad \text{(T-SET)}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_0 : T_0^0 \qquad \forall i = 1..n.\ \Gamma \vdash e_i : T_i^i \\ n = |\overline{e}| = |\overline{x}| \qquad x_0 = \mathtt{this} \\ \mathsf{mtype}(\Gamma, T_0^0, m) = (\overline{x} : \overline{T^0}) \to T_{n+1}^0 \\ \forall i \in 1..n+1, j \in 1..i.\ T_i^{j-1}\{\!\{\Gamma;\ T_{j-1}^{j-1}/x_{j-1}\}\!\} = T_i^j \\ \forall i \in 1..n, j \in 1..i.\ \mathsf{prefixExact}(T_i^{j-1}, k) \Rightarrow \mathsf{prefixExact}(T_i^j, k) \\ \forall i \in 1..n+1, j \in 1..i.\ p.f \in \mathsf{paths}(T_i^{j-1}) \Rightarrow p' \in \mathsf{paths}(T_i^j) \wedge \Gamma \vdash p' = p\{e_{j-1}/x_{j-1}\}.f \end{array}}{\Gamma \vdash e_0.m(\overline{e}) : T_{n+1}^{n+1}} \quad \text{(T-CALL)}$$

$$\frac{\Gamma \vdash T : \mathsf{type} \qquad \Gamma \vdash \overline{e} : \overline{T} \qquad \forall f_i \in \overline{f}.\ \mathsf{ftype}(\Gamma, T, f_i) = [\mathtt{final}]\ T_i}{\Gamma \vdash \mathtt{new}\ T(\overline{f} = \overline{e}) : T} \quad \text{(T-NEW)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1;\ e_2 : T_2} \quad \text{(T-SEQ)}$$

$$\frac{\Gamma \vdash e : T_1 \qquad \Gamma \vdash T_1 \leq T_2}{\Gamma \vdash e : T_2} \quad \text{(T-SUB)}$$

Figure 6.18: Typing

The most complex rule is the call rule, T-CALL. Calls are checked by looking up the method type, then substituting in the receiver type and the actual argument types for `this` and the formal parameters. Types rather than values are substituted because the actual arguments may not be final access paths. Type substitution is defined in Figure 6.17. Formal parameter i has type T_i^0. The type T_i^j is the result of substituting the actuals 0 (the receiver) through $j-1$ into T_i^0. For the call to type-check, the actuals e_i must have the same type as the fully substituted formal types T_i^i. The call itself has type T_{n+1}^{n+1}, where T_{n+1}^0 is the method's declared return type.

To ensure subtyping is preserved by the substitution, substitution must preserve prefix-exactness, defined in Figure 6.15. This ensures that if the type of formal i is dependent on another formal j (or `this`), the ith actual value has a type dependent on actual j (or the actual receiver). Substitution of the return type need not preserve exactness since it is in a covariant position and the result of the substitution can be less precise.

Two different objects used as actuals may have the same dependent type, but may contain final fields that point to objects of different classes. To ensure that a substituted field path is dependent on the actual target, not on another object of the same class that may have initialized the field differently, substitution must also preserve field paths.

A `new` expression is well-typed via T-NEW if it initializes only declared fields of a well-formed type. By T-SEQ, a sequence expression takes the type of the second expression in the sequence. Finally, T-SUB is the standard subsumption rule.

6.3.10 Subtyping and type equivalence

Subtyping rules are defined in Figure 6.19. The judgment $\Gamma \vdash T_1 \leq T_2$ states that T_1 is a subtype of T_2 in context Γ. The rules ensure that syntactically different types

$\boxed{\Gamma \vdash T_1 \leq T_2}$

$$\frac{}{\Gamma \vdash T \leq T} \text{(S-REFL)} \qquad \frac{\Gamma \vdash T_1 \leq T_2 \qquad \Gamma \vdash T_2 \leq T_3}{\Gamma \vdash T_1 \leq T_3} \text{(S-TRANS)}$$

$$\frac{\begin{array}{c}\Gamma \vdash T \leq P \\ CT(P.C) = \texttt{class}\ C\ \texttt{extends}\ T'\ \{\ldots\} \\ T'\{\!\{\Gamma;\ T/\texttt{this}\}\!\} = T''\end{array}}{\Gamma \vdash T.C \leq T''} \text{(S-SUP)}$$

$$\frac{\Gamma \vdash T : \mathsf{type} \qquad \Gamma \vdash T \lhd S}{\Gamma \vdash T \leq S} \text{(S-BOUND)}$$

$$\frac{\Gamma \vdash T_1 \leq T_2 \qquad \Gamma \vdash T_2.C : \mathsf{type}}{\Gamma \vdash T_1.C \leq T_2.C} \text{(S-NEST)}$$

$$\frac{\Gamma \vdash p : T\ \mathsf{final}}{\Gamma \vdash p.\texttt{class} \leq T} \text{(S-FIN)}$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 \leq T_2 \\ \Gamma \vdash P[T_2] : \mathsf{type}\end{array}}{\Gamma \vdash P[T_1] \leq P[T_2]} \text{(S-PRE-1)}$$

$$\frac{\vdash P_1 \sim P_2 \vee \vdash P_1 \sqsubset P_2 \qquad \begin{array}{c}\Gamma \vdash P_1[T] : \mathsf{type} \\ \Gamma \vdash P_2[T] : \mathsf{type}\end{array}}{\Gamma \vdash P_1[T] \approx P_2[T]} \text{(S-PRE-2)}$$

$$\frac{\Gamma \vdash T \leq P.C}{\Gamma \vdash T \leq P[T].C} \text{(S-PRE-OUT)}$$

$$\frac{\Gamma \vdash P[T.C] : \mathsf{type}}{\Gamma \vdash T \approx P[T.C]} \text{(S-PRE-IN)}$$

$$\Gamma \vdash \&\overline{T} \leq T_i \text{(S-MEET-LB)}$$

$$\frac{\forall i.\ \Gamma \vdash T \leq T_i}{\Gamma \vdash T \leq \&\overline{T}} \text{(S-MEET-G)}$$

$$\frac{\Gamma \vdash p_1 = p_2}{\Gamma \vdash p_1.\texttt{class} \approx p_2.\texttt{class}} \text{(S-ALIAS)}$$

$$\frac{\begin{array}{cc}\Gamma \vdash U_1 \lhd S_1 & \Gamma \vdash U_2 \lhd S_2 \\ \Gamma \vdash U_1 : \mathsf{type} & \Gamma \vdash U_2 : \mathsf{type} \\ \mathsf{exact}(U_1) & \emptyset \vdash S_1 \approx S_2\end{array}}{\Gamma \vdash U_1 \leq U_2} \text{(S-EVAL)}$$

Figure 6.19: Subtyping

representing the same sets of values are considered equal. The judgment $\Gamma \vdash T_1 \approx T_2$ is sugar for the pair of judgments $\Gamma \vdash T_1 \leq T_2$ and $\Gamma \vdash T_2 \leq T_1$.

Subtyping is reflexive and transitive. The rule S-SUP states that a type is a subclass of its declared superclass; the enclosing class of the subtype T is substituted in for `this` in the superclass.

S-BOUND states that a type is a subtype of its non-dependent bounding type. The rule S-NEST states that a nested class C is covariant with its containing class; that is, further binding implies subtyping. S-FIN states that a dependent class is a subtype of its declared bound; with F-NULL, this rule also implies that `null.class` is a subtype of any well-formed simple type.

Subtyping of prefix types is covariant by the rules S-PRE-1 and S-PRE-2. S-PRE-OUT and S-PRE-IN, and relate prefix types to non-prefix types.

S-MEET-LB and S-MEET-G are from Compagnoni and Pierce [29] and define subtyping for intersection types. Together these two rules imply that intersection types are associative and commutative and that the singleton intersection type $\&T$ is equivalent to its element type T. With the other rules above, these rules also imply the intuitive judgments $\Gamma \vdash P[\&\overline{T}] \leq P[T_i]$ and $\Gamma \vdash (\&\overline{T}).C \leq T_i.C$.

The rule S-EVAL states that a fully evaluated type (i.e., a type containing only value paths) is a supertype of any fully evaluated exact type with the same bounding type. This rule ensures, for example, that ℓ_1`.class` $\approx \ell_2$`.class` if ℓ_1 and ℓ_2 both point to objects of the same type.

6.3.11 Example

As an example, consider the code in Figure 6.20. For clarity, each occurrence of `this` has been labeled with an abbreviation of its declared type. The call to `visit` in `Exp.accept` has the type `Compiler[this`$_{\texttt{E}}$`.class].Exp` as follows. To type-check the

```
class Compiler {
    class Exp {
        Compiler[this_E.class].Exp
            accept(Compiler[this_E.class].Visitor v) {
            v.visit(this_E)
        }
    }
    class Visitor {
        Compiler[this_V.class].Exp
            visit(Compiler[this_V.class].Exp e) { e }
    }
}
```

Figure 6.20: Example code

call to `visit`, let $\Gamma = \emptyset$, $\texttt{this}_\texttt{E}$:`Compiler.Exp`, `v`:`Compiler[this`$_\texttt{E}$`.class].Visitor`. It is easy to see that:

$$\Gamma \vdash \texttt{v}:\texttt{Compiler[this}_\texttt{E}\texttt{.class].Visitor}$$
$$\Gamma \vdash \texttt{this}_\texttt{E}:\texttt{Compiler.Exp}$$

The mtype function returns the declared type of `visit`:

$$\mathsf{mtype}(\Gamma, \texttt{Compiler[this}_\texttt{E}\texttt{.class].Visitor}, \texttt{visit}) = (e:\texttt{Compiler[this}_\texttt{V}\texttt{.class].Exp}) \to \texttt{Compiler[this}_\texttt{V}\texttt{.class].Exp}$$

By T-CALL, the declared type of the actual receiver v is substituted for $\texttt{this}_\texttt{V}$ in the formal parameter type and the return type:

$$\texttt{Compiler[this}_\texttt{V}\texttt{.class].Exp}\{\!\{\Gamma;\ \texttt{Compiler[this}_\texttt{E}\texttt{.class].Visitor}/\texttt{this}_\texttt{V}\}\!\} = \texttt{Compiler[Compiler[this}_\texttt{E}\texttt{.class].Visitor].Exp}$$

Therefore, the call can be typed by T-CALL as follows:

$$\Gamma \vdash \texttt{v.visit(this}_\texttt{E}\texttt{)}:\texttt{Compiler[Compiler[this}_\texttt{E}\texttt{.class].Visitor].Exp}$$

Using subsumption, the result type can be written as an equivalent simpler type. First, by S-PRE-IN, the following type equivalence can be derived:

$$\Gamma \vdash \texttt{Compiler[Compiler[this}_{\texttt{E}}\texttt{.class].Visitor]} \approx \texttt{Compiler[this}_{\texttt{E}}\texttt{.class]}$$

and then by S-NEST, the following equivalence can be derived:

$$\Gamma \vdash \texttt{Compiler[Compiler[this}_{\texttt{E}}\texttt{.class].Visitor].Exp} \approx \texttt{Compiler[this}_{\texttt{E}}\texttt{.class].Exp}$$

Finally, by T-SUB, the call can be typed as:

$$\Gamma \vdash \texttt{v.visit(this}_{\texttt{E}}\texttt{)}:\texttt{Compiler[this}_{\texttt{E}}\texttt{.class].Exp}$$

Now consider what happens if v were declared to have type Compiler.Visitor. In this case, the call will not type-check. The substitution of this type for $\texttt{this}_{\texttt{V}}$ would have the result type Compiler[Compiler.Visitor].Exp:

$$\texttt{Compiler[this}_{\texttt{V}}\texttt{.class].Exp}\{\!\{\Gamma;\ \texttt{Compiler.Visitor}/\texttt{this}_{\texttt{V}}\}\!\} = \texttt{Compiler[Compiler.Visitor].Exp}$$

By S-PRE-IN and S-NEST, this type is equivalent to Compiler.Exp. But, the rule T-CALL requires that since $\mathsf{prefixExact}(\texttt{Compiler[this}_{\texttt{V}}\texttt{.class].Exp}, 1)$ it must be that $\mathsf{prefixExact}(\texttt{Compiler[Compiler.Visitor.class].Exp}, 1)$ also. Since this is not the case, T-CALL cannot be applied, and the call to visit will not type-check.

6.3.12 Program typing

Program typing rules are presented in Figure 6.21. The P-OK says the program *Pr* is well-formed if all class declarations are well-formed, if the "main" expression is well-typed, and if the transitive closure of the inheritance relation $\sqsubset$ is acyclic. The last requirement is needed to ensure that the type system is well-founded.

$$\frac{\circ \vdash \overline{L}\ \mathsf{ok} \qquad \emptyset \vdash e : T \qquad \emptyset \vdash T : \mathsf{type} \qquad \sqsubset^{+}\ \text{acyclic}}{\vdash \langle \overline{L}, e \rangle\ \mathsf{ok}} \tag{P-OK}$$

$$\frac{\begin{array}{c} P.C \vdash \overline{L}\ \mathsf{ok} \qquad P.C \vdash \overline{F}\ \mathsf{ok} \qquad P.C \vdash \overline{M}\ \mathsf{ok} \\ P \vdash T\ \mathsf{super\ ok} \\ \forall P_i \in \mathsf{supers}(P.C) \backslash \{P.C\}.\ \vdash P.C \text{ conforms to } P_i \end{array}}{P \vdash \mathtt{class}\ C\ \mathtt{extends}\ T\ \{\overline{L}\ \overline{F}\ \overline{M}\}\ \mathsf{ok}} \tag{L-OK}$$

$$\frac{\begin{array}{c} T \neq \circ \\ \mathtt{this} : P \vdash T : \mathsf{type} \\ \mathsf{paths}(T) \subseteq \{\mathtt{this}\} \\ \neg \mathsf{exact}(T) \end{array}}{P \vdash T\ \mathsf{super\ ok}}$$

$$\frac{\begin{array}{c} CT(P) = \mathtt{class}\ C\ \mathtt{extends}\ T\ \{\overline{L}\ \overline{F}\ \overline{M}\} \\ CT(P') = \mathtt{class}\ C'\ \mathtt{extends}\ T'\ \{\overline{L'}\ \overline{F'}\ \overline{M'}\} \\ \forall i, j.\ \begin{pmatrix} L_i = \mathtt{class}\ D\ \mathtt{extends}\ T_i\ \{\ldots\} \wedge \\ L'_j = \mathtt{class}\ D\ \mathtt{extends}\ T'_j\ \{\ldots\} \end{pmatrix} \Rightarrow \mathtt{this} : P \vdash T_i \leq T'_j \\ \mathsf{fnames}(\overline{F}) \cap \mathsf{fnames}(\overline{F'}) = \emptyset \\ \forall i, j.\ \begin{pmatrix} M_i = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\} \wedge \\ M'_j = T'_{n+1}\ m(\overline{T'}\ \overline{x'})\ \{e'\} \end{pmatrix} \Rightarrow P \vdash M_i \text{ overrides } M'_j \end{array}}{\vdash P \text{ conforms to } P'}$$

$$\frac{\begin{array}{c} M = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\} \\ M' = T'_{n+1}\ m(\overline{T'}\ \overline{x'})\ \{e'\} \\ |\overline{x}| = |\overline{x'}| = |\overline{y}| \qquad \overline{y} \cap (\overline{x} \cup \overline{x'}) = \emptyset \\ \overline{T}\{\overline{y}/\overline{x}\} = \overline{T'}\{\overline{y}/\overline{x'}\} \\ T_{n+1}\{\overline{y}/\overline{x}\} = T'_{n+1}\{\overline{y}/\overline{x'}\} \end{array}}{P \vdash M \text{ overrides method } M'}$$

$$\frac{\emptyset \vdash T : \mathsf{type} \qquad \emptyset \vdash e : T}{P \vdash [\mathtt{final}]\ T\ f = e\ \mathsf{ok}} \tag{F-OK}$$

$$\frac{\Gamma = \mathtt{this} : P, \overline{x} : \overline{T} \qquad \Gamma \vdash \mathsf{env} \qquad \Gamma \vdash T : \mathsf{type} \qquad \Gamma \vdash e : T}{P \vdash T\ m(\overline{T}\ \overline{x})\ \{e\}\ \mathsf{ok}} \tag{M-OK}$$

Figure 6.21: Program typing

By L-OK, a class declaration is well-formed if all its members are well-formed and its superclass is well-formed in a context containing only `this` bound to the class's container. Additionally, the only access path embedded in the superclass declaration can be `this`. The class must also conform to all of its superclasses.

A class P conforms to P' if all of the following hold:

- If both P and P' have a member class D, then $P.D$'s declared superclass is a subtype of $P'.D$'s.
- The field names of P and P' are disjoint. This requirement simplifies the semantics by ensuring field names are unique.
- If both P and P' define a method m, then the method in P correctly overrides the method P'.

Method M in P correctly overrides M' if the number of formal parameters are equal, the parameter types of M are supertypes of the parameter types of M', and the return type of M is a subtype of M'. Subtyping checks are done with fresh names substituted in for the parameter names occurring in the types. Using the judgment $\Gamma \vdash \mathsf{env}$, it is required that the type of formal parameter i depends only on `this` and formal parameters 1 through $i-1$.

Finally, field and method declarations are well-formed by rules F-OK and M-OK, respectively, if the types occurring in the signatures well-formed and if the initializer is method body is well-typed.

6.3.13 Typing contexts

Typing contexts are deemed well-formed by the judgment $\Gamma \vdash \mathsf{env}$, defined in Figure 6.22. The rules disallow rebinding of variables and locations and require that types

$\boxed{\Gamma \vdash \mathsf{env}}$

$$\emptyset \vdash \mathsf{env}$$

$$\frac{\Gamma \vdash \mathsf{env} \qquad x \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash T : \mathsf{type}}{\Gamma, x:T \vdash \mathsf{env}}$$

$$\frac{\Gamma \vdash \mathsf{env} \qquad \ell \notin \mathrm{dom}(\Gamma) \qquad \emptyset \vdash S : \mathsf{type}}{\Gamma, \ell:S \vdash \mathsf{env}}$$

$$\frac{\Gamma \vdash \mathsf{env} \qquad \Gamma \vdash \ell.f : T \ \mathsf{final} \qquad \Gamma \vdash v : T}{\Gamma, \ell.f = v \vdash \mathsf{env}}$$

Figure 6.22: Well-formed typing contexts

$\boxed{\Gamma\{v/x\}}$

$$\begin{aligned}
\emptyset\{v/x\} &= \emptyset \\
(\Gamma, x:T)\{v/x\} &= \Gamma \\
(\Gamma, y:T)\{v/x\} &= \Gamma\{v/x\}, y:T\{v/x\} \\
(\Gamma, \ell:S)\{v/x\} &= \Gamma\{v/x\}, \ell:S \\
(\Gamma, p_1 = p_2)\{v/x\} &= \Gamma\{v/x\}, p_1\{v/x\} = p_2\{v/x\}
\end{aligned}$$

Figure 6.23: Substitution on typing contexts

be well-formed. Substitution on typing contexts is straightforward and is defined in Figure 6.23.

6.3.14 Heaps

A heap H is a function from memory locations to objects o; we write $H(\ell) = o$ if $o \mapsto \ell$ is in H. We write $H[\ell := o]$ for H with $H(\ell)$ remapped to o, that is:

$$\begin{aligned} \emptyset[\ell := o] &= \ell \mapsto o \\ (H, \ell \mapsto o')[\ell := o] &= H, \ell \mapsto o \\ (H, \ell' \mapsto o')[\ell := o] &= H[\ell := o], \ell' \mapsto o' \qquad (\ell \neq \ell') \end{aligned}$$

In the operational semantics and in the soundness proof, run-time values are typed using a typing context constructed from the heap. A typing context $\lfloor H \rfloor$ is constructed from H by inserting location types and aliasing information for fields into the context. To type the heap properly, the typing context include path aliasing constraints of the form $\ell.f = v$.

$$\begin{aligned} \lfloor \emptyset \rfloor &= \emptyset \\ \lfloor H, \ell \mapsto S\ \{\overline{f} = \overline{v}\} \rfloor &= \lfloor H \rfloor, \ell : S, \ell.\overline{f'} = \overline{v'} \\ &\quad \text{where } \overline{f'} = \{f_i \in \overline{f} \mid \mathsf{ftype}(\emptyset, S, f_i) = \texttt{final}\ T_i\} \end{aligned}$$

The equivalence constraints and S-ALIAS ensure that if $\ell_1.f$ steps to ℓ_2, then $\ell_2.\texttt{class}$ is a subtype of $\ell_1.f.\texttt{class}$, which is essential for proving type preservation.

Figure 6.24 shows the heap typing rules. The judgment $H \vdash \ell : \mathsf{loc}$ states that a location ℓ is well-formed for a heap H if it maps to an object of type S containing all declared fields of S and each value stored in those fields has the correct type and, if a location, is also well-formed in H. Rule H-NULL states that the `null` value is always well-formed.

$$\frac{\begin{array}{c} H(\ell) = S\,\{\overline{f} = \overline{v}\} \\ \mathsf{fnames}(\mathit{fields}(S)) = \{\overline{f}\} \\ \mathsf{ftype}(\emptyset, S, \overline{f}) = \overline{T} \\ \lfloor H \rfloor \vdash \overline{v} : \overline{T} \\ \overline{v} \subseteq \mathrm{dom}(H) \cup \{\mathtt{null}\} \end{array}}{H \vdash \ell : \mathsf{loc}} \qquad \text{(H-LOC)}$$

$$\frac{\forall \ell \in \mathrm{dom}(H).\ H \vdash \ell : \mathsf{loc}}{\vdash H} \qquad \text{(HEAP)}$$

$$\frac{\vdash H \qquad \mathsf{locs}(e) \subseteq \mathrm{dom}(H)}{\vdash e, H} \qquad \text{(CONFIG)}$$

Figure 6.24: Well-formed heaps

A heap H is well-formed, written $\vdash H$, if all locations in its domain are well-formed. Finally, a configuration is well-formed, written $\vdash e, H$ if H is well-formed and all free locations of e, $\mathsf{locs}(e)$, are in H.

6.4 Operational semantics

This section presents a small-step operational semantics. The semantics are defined with a reduction relation $\longrightarrow$, which maps a configuration of an expression e and a heap H to a new configuration. Result configurations consist of a new heap and a result r, which is either an expression or NullError. The notation $e, H \longrightarrow r, H'$ means that expression e and heap H step to result r and heap H'. The initial configuration for program $\langle \overline{L}, e \rangle$ is $e, \emptyset$. Final configurations are of the form v, H or $\mathsf{NullError}, H$.

Figure 6.25 defines additional syntax used in the operational semantics.

results	r	$::= e \mid \mathsf{NullError}$
evaluated types	U	$::= \circ \mid U.C \mid \ell.\texttt{class} \mid P\texttt{[}U\texttt{]} \mid \&\overline{U}$
evaluation contexts	E	$::= [\cdot]$
		$\mid E.f$
		$\mid \texttt{new}\ TE(\overline{f} = \overline{e})$
		$\mid \texttt{new}\ U(\overline{f} = \overline{v}, f = E, \overline{f'} = \overline{e})$
		$\mid E.f = e$
		$\mid \ell.f = E$
		$\mid E.m(\overline{e})$
		$\mid \ell.m(\overline{v}, E, \overline{e})$
		$\mid E;\ e$
type evaluation contexts	TE	$::= TE.C$
		$\mid E.\texttt{class}$
		$\mid P\texttt{[}TE\texttt{]}$
		$\mid \&(\overline{U}, TE, \overline{T})$
null errors	NE	$::= \texttt{null}.f$
		$\mid \texttt{null}.f = e$
		$\mid \texttt{null}.m(\overline{e})$
		$\mid \texttt{new}\ TE[\texttt{null}](\overline{f} = \overline{e})$
		$\mid \mathsf{NullError}$

Figure 6.25: Additional syntax

6.4.1 Evaluation contexts

Order of evaluation is captured by an evaluation context E, an expression with a hole $[\cdot]$. Since types are dependent, `new` expressions need to evaluate the class of the object being allocated. A type evaluation context *TE* specifies how dependent types are evaluated. Fully evaluated types U contain only location paths ℓ.`class` and not field paths.

6.4.2 Null dereferences

The nonterminal *NE* in Figure 6.25 specifies expressions with dereferences of the `null` value. These expressions all evaluate to the result NullError. Attempting to allocate an object whose type embedded has an embedded `null` will also evaluate to NullError.

6.4.3 Reduction rules

The reduction rules are shown in Figure 6.26. Order of evaluation is captured by an evaluation context E and the congruence rule R-CONG. Since types are dependent, expressions used in types must be evaluated as well. We write U for a type containing no redex.

The rule R-NULL propagates a dereference of a `null` pointer out through the evaluation contexts to produce a NullError, simulating a Java `NullPointerException`.

The rules R-GET and R-SET get and set a field in a heap object, respectively. R-CALL uses the mbody function defined in Figure 6.13 to locate the most specific implementation of method m. The actual values for the receiver and arguments are then substituted into the method body.

There are two rules for evaluating `new` expressions. R-NEW looks up all fields of the type being allocated and steps to a configuration containing initializers for those fields. R-ALLOC is applied when all initializers have been evaluated. A new location is

$\boxed{e, H \longrightarrow r, H}$

$$\frac{e, H \longrightarrow e', H'}{E[e], H \longrightarrow E[e'], H'} \quad \text{(R-CONG)}$$

$$E[NE], H \longrightarrow \mathsf{NullError}, H \quad \text{(R-NULL)}$$

$$\frac{H(\ell) = S\ \{\overline{f} = \overline{v}\}}{\ell.f_i, H \longrightarrow v_i, H} \quad \text{(R-GET)}$$

$$\frac{\begin{array}{c} H(\ell) = S\ \{\overline{f} = \overline{v}\} \\ H' = H[\ell := S\ \{f_1 = v_1, \ldots, f_i = v, \ldots, f_n = v_n\}] \end{array}}{\ell.f_i = v, H \longrightarrow v, H'} \quad \text{(R-SET)}$$

$$\frac{\ell : S \in \lfloor H \rfloor \qquad \mathsf{mbody}(S, m) = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\} \qquad n = |\overline{v}| = |\overline{x}|}{\ell.m(\overline{v}), H \longrightarrow e\{\ell, \overline{v}/\mathtt{this}, \overline{x}\}, H} \quad \text{(R-CALL)}$$

$$\frac{\begin{array}{c} \lfloor H \rfloor \vdash U \lhd S \qquad \mathsf{fnames}(\mathsf{fields}(S)) = \{\overline{f}, \overline{f'}\} \\ |\overline{f'}| \neq 0 \qquad \mathsf{finit}(S, \overline{f'}) = \overline{e'} \end{array}}{\mathtt{new}\ U(\overline{f} = \overline{v}), H \longrightarrow \mathtt{new}\ U(\overline{f} = \overline{v}, \overline{f'} = \overline{e'}), H} \quad \text{(R-NEW)}$$

$$\frac{\begin{array}{c} \lfloor H \rfloor \vdash U \lhd S \qquad \mathsf{fnames}(\mathsf{fields}(S)) = \{\overline{f}\} \\ \ell \notin \mathrm{dom}(H) \qquad H' = H, \ell \mapsto S\ \{\overline{f} = \overline{v}\} \end{array}}{\mathtt{new}\ U(\overline{f} = \overline{v}), H \longrightarrow \ell, H'} \quad \text{(R-ALLOC)}$$

$$v;\ e, H \longrightarrow e, H \quad \text{(R-SEQ)}$$

Figure 6.26: Operational semantics

allocated and the object is installed in the heap. To ensure that all fields are accounted for, field names are looked up in a typing context containing only location bindings. This ensures the bounding non-dependent type of the new instance's type U is unique and is as tight a bound as possible.

Chapter 7

Soundness

This chapter presents a soundness proof for the semantics presented in Chapter 6. The soundness theorem states that the result of evaluating a well-typed program is either a value or a `null` dereference error.

Theorem 7.1 *(Soundness)* If $\vdash \langle \overline{L}, e \rangle$ ok, and $\emptyset \vdash e : T$, and $e, \emptyset \longrightarrow^* r, H$ where r is in normal form, then either $r = v$ and $\lfloor H \rfloor \vdash v : T$ or $r = \mathsf{NullError}$.

To prove soundness we use the standard technique of proving subject reduction and progress lemmas [121].

Lemma 7.2 *(Subject reduction)* If $\vdash e, H$, $\lfloor H \rfloor \vdash e : T$, and $e, H \longrightarrow r, H'$, then either

- $r = e'$, $\vdash e', H'$, and $\lfloor H' \rfloor \vdash e' : T$, or
- $r = \mathsf{NullError}$.

Lemma 7.3 *(Progress)* If $\vdash e, H$ and $\lfloor H \rfloor \vdash e : T$, then either $e = v$, or there is an r and an H' such that $e, H \longrightarrow r, H'$.

The subject reduction proof is the more complicated of the two. We first prove several preliminary lemmas about typing contexts, non-dependent bounding types, and substitution. The subject reduction proof follows.

After the subject reduction proof, a few more lemmas needed to prove progress are presented. Finally, the progress lemma itself is proved and the soundness theorem follows.

7.1 Typing contexts

We first prove some lemmas about typing contexts. We say a context Γ_2 extends Γ_1 if there is a Γ such that $\Gamma_2 = \Gamma_1, \Gamma$. The following lemma states that if a judgment holds in a particular context Γ, it holds in an extended context Γ, Γ'.

Lemma 7.4 *(Weakening)* If Γ' extends Γ and $\Gamma' \vdash \mathsf{env}$, then all of the following hold:

1. If $\Gamma \vdash p : T$ final, then $\Gamma' \vdash p : T$ final.
2. If $\Gamma \vdash T : \mathsf{type}$ then $\Gamma' \vdash T : \mathsf{type}$.
3. If $\Gamma \vdash T \lhd S$ then $\Gamma' \vdash T \lhd S$.
4. If $\mathsf{ftype}(\Gamma, T, f) = T_f$ then $\mathsf{ftype}(\Gamma', T, f) = T_f$.
5. If $\mathsf{mtype}(\Gamma, T, m) = (\overline{x} : \overline{T}) \to T_{n+1}$, then $\mathsf{mtype}(\Gamma', T, m) = (\overline{x} : \overline{T}) \to T_{n+1}$.
6. If $T\{\!\{\Gamma;\ T_v/x\}\!\} = T'$ then $T\{\!\{\Gamma';\ T_v/x\}\!\} = T'$.
7. If $\Gamma \vdash T_1 \leq T_2$ then $\Gamma' \vdash T_1 \leq T_2$.
8. If $\Gamma \vdash e : T$ then $\Gamma' \vdash e : T$.

Proof. The proof is by induction on the derivation of the appropriate judgment. □

7.2 Heap contexts

During evaluation, the heap is updated as objects are allocated and non-final fields are assigned. The following definitions and lemmas state that the typing context constructed from the updated heap extends the context constructed from the original heap.

Definition 7.5 We say a heap H_2 *remaps* H_1 if

- $H_1 = H_2 = \emptyset$, or
- $H_1 = H_1', \ell \mapsto S\ \{\overline{f} = \overline{v}\}$, and $H_2 = H_2', \ell \mapsto S\ \{\overline{f} = \overline{v'}\}$, and H_2' remaps H_1', and for all f_i, if $\mathsf{ftype}(\emptyset, S, f_i) = \texttt{final}\ T$, then $v_i = v_i'$.

Definition 7.6 H_2 *extends* H_1 if H_2 remaps H_1, or there is an H such that H extends H_1 and $H_2 = H, \ell \mapsto o$ and $\ell \notin \mathrm{dom}(H)$.

The remapped and extended heaps extend the typing context derived from the original heap, allowing the extension lemma (Lemma 7.4) above to be used.

Lemma 7.7 If H_2 remaps H_1, then $\lfloor H_2 \rfloor$ extends $\lfloor H_1 \rfloor$.

Proof. The proof is by structural induction on H_2.

Case $H_2 = \emptyset$:

Then $H_1 = \emptyset$. Trivial.

Case $H_2 = H_2', \ell \mapsto S\ \{\overline{f} = \overline{v}\}$:

Then $H_2 = H_1', \ell \mapsto S\ \{\overline{f} = \overline{v'}\}$. By the induction hypothesis, $\lfloor H_2' \rfloor = \lfloor H_1' \rfloor$. For all final fields f of S, $H_1(\ell)[f] = H_2(\ell)[f]$. Therefore, $\lfloor H_2 \rfloor = \lfloor H_1 \rfloor$ by construction. □

Lemma 7.8 If H_2 extends H_1, then $\lfloor H_2 \rfloor$ extends $\lfloor H_1 \rfloor$.

Proof. The proof is by structural induction on H.

- If H_2 remaps H_1, then $\lfloor H_2 \rfloor = \lfloor H_1 \rfloor$ by Lemma 7.7.
- Otherwise, there is an H such that H extends H_1 and $H_2 = H, \ell \mapsto S\ \{\overline{f} = \overline{v}\}$ and $\ell \notin \mathrm{dom}(H)$. By the induction hypothesis, $\lfloor H \rfloor$ extends $\lfloor H_1 \rfloor$ and by construction $\lfloor H_2 \rfloor = \lfloor H \rfloor, \ell : S, \ell.\overline{f'} = \overline{v'}$ where $\overline{f'}$ are the final fields of S. □

7.3 Non-dependent bounding types

Next, we want to show that if T_1 is a subtype of T_2, then the non-dependent bound of T_1 extends a bound of T_2. We first show that type substitution and bounding types commute. The lemma will only be applied to declared supertypes, which can contain only the `this` access path; hence, to simplify the proof, the lemma is restricted to types with access paths containing a single variable x.

Lemma 7.9 If $\Gamma \vdash T_1 \lhd S_1$, $\mathsf{paths}(T) \subseteq \{x\}$, $T\{\!\{\Gamma;\ T_1/x\}\!\} = T_2$, and $T\{\!\{\Gamma;\ S_1/x\}\!\} = S_2$, then $\Gamma \vdash T_2 \lhd S_2$.

Proof. The proof is by induction on the structure of T.

Case $T = \circ$:

Then $T_2 = \circ = S_2$.

Case $T = T'.C$:

Let $T'\{\!\{\Gamma;\ T_1/x\}\!\} = T_2'$ and $T'\{\!\{\Gamma;\ S_1/x\}\!\} = S_2'$. Then $T_2 = T_2'.C$ and $S_2 = S_2'.C$. By the induction hypothesis, $\Gamma \vdash T_2' \lhd S_2'$. By BD-NEST, $\Gamma \vdash T_2'.C \lhd S_2'.C$.

Case $T = x.\texttt{class}$:

Then $T_2 = T_1$ and $S_2 = S_1$. Trivial.

Case $T = P[T']$:

Let $T'\{\!\{\Gamma;\ T_1/x\}\!\} = T_2'$ and $T'\{\!\{\Gamma;\ S_1/x\}\!\} = S_2'$. Then $T_2 = P[T_2']$ and $S_2 = P[S_2']$. By the induction hypothesis, $\Gamma \vdash T_2' \lhd S_2'$. By BD-PRE, $\Gamma \vdash P[T_2'] \lhd P[S_2']$.

Case $T = \&\overline{T}$:

Follows from the induction hypothesis and BD-MEET. $\square$

Then, we show that subtyping tightens the non-dependent bound.

Lemma 7.10 If $\Gamma \vdash T_1 \leq T_2$ and $\Gamma \vdash T_1 \lhd S_1$, then there is an S_2 such that $\Gamma \vdash T_2 \lhd S_2$ and $\vdash S_1 \sqsubset^* S_2$.

Proof. The proof is by induction on the subtyping derivation.

Case S-REFL:

Trivial.

Case S-TRANS:

Trivial via the induction hypothesis.

Case S-SUP:

Then $T_1 = T.C$, and there is a P where $CT(P.C) = \texttt{class}\ C\ \texttt{extends}\ T'\ \{\ldots\}$, $\Gamma \vdash T \leq P$, and $T'\{\!\{\Gamma;\ T/\texttt{this}\}\!\} = T_2$. By BD-NEST, $S_1 = S.C$ where $\Gamma \vdash T \lhd S$. By the induction hypothesis, $\vdash S \sqsubset^* P$, and by FB, $\vdash S.C \sqsubset^* P.C$. Let $T'\{\!\{\Gamma;\ S/\texttt{this}\}\!\} = S'$. Then by SC, $\vdash P.C \sqsubset^* S'$. By Lemma 7.9, $S' = S_2$. Therefore, by transitivity of $\sqsubset^*$, we have $\vdash S_1 \sqsubset^* S_2$.

Case S-BOUND:

Trivial since $T_2 = S_1 = S_2$.

Case S-NEST:

Follows from the induction hypothesis.

Case S-FIN:

Follows from BD-FIN.

Case S-PRE-1:

Follows from definition of prefix and BD-PRE.

Case S-PRE-2:

Follows from definition of prefix and BD-PRE.

Case S-PRE-OUT:

Follows from the induction hypothesis.

Case S-PRE-IN:

Follows from definition of prefix and BD-PRE.

Case S-MEET-LB:

Follows from BD-MEET.

Case S-MEET-G:

Follows from the induction hypothesis.

Case S-ALIAS:

Then $T_1 = p_1.\texttt{class}$ and $T_2 = p_2.\texttt{class}$ and $\Gamma \vdash p_1 = p_2$. Also, $\Gamma \vdash p_1 : T_1'$ final and $\Gamma \vdash p_2 : T_2'$ final, where $\Gamma \vdash T_1' \lhd S_1'$ and $\Gamma \vdash T_2' \lhd S_2'$. By BD-FIN, $\Gamma \vdash T_1 \lhd S_1' \,\&\, S_2'$ and $\Gamma \vdash T_2 \lhd S_2' \,\&\, S_1'$. The case holds since $\mathsf{supers}(S_1' \,\&\, S_2') = \mathsf{supers}(S_2' \,\&\, S_1')$.

Case S-EVAL:

Trivial since $S_1 = S_2$. □

A corollary of Lemma 7.10 is that a field lookup on a subtype returns the same type.

Lemma 7.11 and $\Gamma \vdash T_1 \leq T_2$, and $\mathsf{ftype}(\Gamma, T_1, f) = [\texttt{final}]\ T_f$, then $\mathsf{ftype}(\Gamma, T_2, f) = [\texttt{final}]\ T_f$.

Proof. Follows immediately from Lemma 7.10 and the definition of ftype. □

7.4 Final access paths

This lemma states that if a final access path p has a given type T, that type must be a supertype of $p.\texttt{class}$.

Lemma 7.12 If $\Gamma \vdash p : T_p$ final and $\Gamma \vdash p : T$, then $\Gamma \vdash p.\texttt{class} \leq T$.

Proof. The proof is by induction on the height of the subtyping derivation. There are only two ways to derive $\Gamma \vdash p : T$:

Case T-FIN:

Then $T = p.\texttt{class}$ and the case holds by S-REFL.

Case T-SUB:

Then $\Gamma \vdash p : T'$ and $\Gamma \vdash T' \leq T$. By the induction hypothesis, $\Gamma \vdash p.\texttt{class} \leq T'$. Therefore by S-TRANS, $\Gamma \vdash p.\texttt{class} \leq T$. □

A related lemma for location paths states that the declared type S of a location is a subtype of any other typing with a non-dependent type S'.

Lemma 7.13 If $\Gamma \vdash \ell : S'$ and $\Gamma \vdash \ell : S$ final, then $\Gamma \vdash S \leq S'$.

Proof. Since $\Gamma \vdash \ell : S$ final, by F-LOC we must have $\ell : S \in \Gamma$. The only way to derive $\Gamma \vdash \ell : S'$ is to use T-FIN to derive $\Gamma \vdash \ell : \ell.\mathtt{class}$, and then to derive $\Gamma \vdash \ell.\mathtt{class} \leq S'$, and then S-TRANS to derive $\Gamma \vdash \ell : S'$. The only way to derive $\Gamma \vdash \ell.\mathtt{class} \leq S'$ is to derive $\Gamma \vdash \ell.\mathtt{class} \leq S$ by S-FIN and then to derive $\Gamma \vdash S \leq S'$, which is assumed. This proves the lemma. □

7.5 Type substitution

We next prove some lemmas about type substitution. We want to show that type substitution preserves type well-formedness. First, we show that the result of a type substitution has a tighter non-dependent bound.

Lemma 7.14 If $x : T_x \in \Gamma$, and $\Gamma \vdash T_v \leq T_x$, and $\Gamma \vdash T \lhd S$, and $\Gamma \vdash T_v : \mathsf{type}$, and $T\{\!\{\Gamma;\ T_v/x\}\!\} = T'$, then there is an S' such that $\Gamma \vdash T' \lhd S'$ and $S' \sqsubset^* S$.

Proof. The proof is by induction on type substitution derivation.

Case $T = \circ$:

Trivial since $T' = T$.

Case $T = T_0.C$:

Then $T' = T'_0.C$ where $T_0\{\!\{\Gamma;\ T_v/x\}\!\} = T'_0$.

Let $\Gamma \vdash T_0 \lhd S_0$. By the induction hypothesis, $\Gamma \vdash T'_0 \lhd S'_0$ and $\vdash S'_0 \sqsubset^* S_0$. By BD-NEST, $\Gamma \vdash T'_0.C \lhd S'_0.C$. The case holds by the definitions of mem and supers.

Case $T = p_1.\mathtt{class}$:

Then by BD-FIN, $\Gamma \vdash p_1 = p_2$, $\Gamma \vdash p_1 : T_1$ final, $\Gamma \vdash p_2 : T_2$ final, $\Gamma \vdash T_1 \lhd S_1$, $\Gamma \vdash T_2 \lhd S_2$, and $S = S_1 \,\&\, S_2$. We consider p_1 by cases.

Case $p_1 = v$:

Then $T' = T$.

Case $p_1 = y$:

Then $T' = T$.

Case $p_1 = x$:

Then $T' = T_v$ and $\Gamma \vdash T_v \lhd S'$. Since $x : T_x \in \Gamma$, $\Gamma \vdash x : T_x$ final by F-VAR. Since $T = x.\texttt{class}$, we have $T_x = T_1$ and $T_x \vdash S_1 \lhd$. Since $\Gamma \vdash T_v \leq T_x$, by Lemma 7.10, $\vdash S' \sqsubset^* S_1$. Since $\Gamma \vdash x = p_2$, by S-ALIAS we have $\Gamma \vdash x.\texttt{class} \approx p_2.\texttt{class}$. By Lemma 7.10, $\vdash S_1 \sqsubset^* S_2$. Therefore $\vdash S' \sqsubset^* S_1 \,\&\, S_2$.

Case $p_1 = p_0.f$:

Let $p_0.\texttt{class}\{\!\{\Gamma;\ T_v/x\}\!\} = T_p$. By F-GET, we have $\Gamma \vdash p_0 : T_p$ final, and $\mathsf{ftype}(\Gamma, T_p, f) = \texttt{final}\ T_f$, and $\Gamma \vdash p_0.f : T_f$ final. By BD-FIN, we have $\Gamma \vdash T_f \lhd S$.

Case $T_p = p_0'.\texttt{class}$:

Then $T' = p_0'.f.\texttt{class}$. By F-GET, we have $\Gamma \vdash p_0' : T_p'$ final, $\mathsf{ftype}(\Gamma, T_p', f) = \texttt{final}\ T_f$, and $\Gamma \vdash p_0'.f : T_f$ final. Since $\Gamma \vdash T_f \lhd S$, by BD-FIN, $\Gamma \vdash T' \lhd S$ and $S = S'$.

Otherwise:

T_p is not a path type. Then by the definition of type substitution, $T' = T_f$ where $\mathsf{ftype}(\Gamma, T_p, f) = T_f$. Since $\Gamma \vdash T_f \lhd S$, we have $\Gamma \vdash T' \lhd S$ and $S = S'$.

Case $T = P[T_0]$:

Let $\Gamma \vdash T_0 \lhd S_0$. By the induction hypothesis, $\Gamma \vdash T_0' \lhd S_0'$ and $\vdash S_0' \sqsubset^* S_0$. Thus, by BD-PRE, $\Gamma \vdash P[T_0'] \lhd S'$ where $S' = \mathsf{prefix}(P, S_0')$. Since $S = \mathsf{prefix}(P, S_0)$, by the definition of prefix, $\vdash S' \sqsubset^* S$.

Case $T = \&\overline{T}$:

Then $T' = \&\overline{T'}$. Let $\Gamma \vdash T_i \lhd S_i$. By the induction hypothesis, for all i $\Gamma \vdash T_i' \lhd S_i'$ and $\vdash S_i' \sqsubset^* S_i$. By BD-MEET, $\Gamma \vdash T_i' \lhd S_i'$. The case holds by the definitions of mem and supers. $\square$

A corollary of the Lemma 7.14 is that field lookups have the same result on a substituted type.

Lemma 7.15 If $x : T_x \in \Gamma$, and $\Gamma \vdash T_v \leq T_x$, and $\mathsf{ftype}(\Gamma, T, f) = [\texttt{final}]\ T_f$, and $\Gamma \vdash T_v :$ type, and $T\{\!\{\Gamma;\ T_v/x\}\!\} = T'$, then $\mathsf{ftype}(\Gamma, T', f) = [\texttt{final}]\ T_f$.

Proof. Follows from Lemma 7.14 and the definition of ftype. $\square$

Next, we prove a few lemmas needed to show type well-formedness is preserved by type substitution.

Lemma 7.16 If $\vdash S_1 \sqsubset^* S_0$, then $\mathsf{prefix}(P, S_1) \supseteq \mathsf{prefix}(P, S_0)$.

Proof. By the definition of $\sqsubset^*$, $\mathsf{supers}(S_1) \supseteq \mathsf{supers}(S_0)$ and from the definition of prefix, it follows that $\mathsf{prefix}(P, S_1) \supseteq \mathsf{prefix}(P, S_0)$. $\square$

Lemma 7.17 If $\Gamma \vdash p_1 = p_2$, then either $p_1 = p_2$ or p_1 and p_2 have no free variables.

Proof. The proof is by induction on the derivation of $\Gamma \vdash p_1 = p_2$. $\square$

Lemma 7.18 Assume $x : T_x \in \Gamma$, and $\Gamma \vdash T_v \leq T_x$, and $\Gamma \vdash T :$ type, and $\Gamma \vdash T_v :$ type, and $T\{\!\{\Gamma;\ T_v/x\}\!\} = T'$. If for all p_1 and p_2 in $\mathsf{paths}(T)$, $\Gamma \vdash p_1 = p_2$, then for all p_1' and p_2' are in $\mathsf{paths}(T')$, $\Gamma \vdash p_1' = p_2'$,

Proof. If $p_1 = p_2$, then necessarily $p_1' = p_2'$. Otherwise, by Lemma 7.17, p_1 and p_2 have no free variables and therefore $p_1 = p_1'$ and $p_2 = p_2'$; the lemma then holds by the induction hypothesis. □

Lemma 7.19 If $x : T_x \in \Gamma$, and $\Gamma \vdash T_v \leq T_x$, and $\Gamma \vdash T_1 \,\&\, T_2 : \mathsf{type}$, and $\Gamma \vdash T_v : \mathsf{type}$, and $T_1 \,\&\, T_2 \{\!\{\Gamma;\ T_v/x\}\!\} = T_1' \,\&\, T_2'$, and for all p_1 and p_2 in $\mathsf{paths}(T_1 \,\&\, T_2)$, $\Gamma \vdash p_1 = p_2$, and $\mathsf{prefixExact}(T_1, k) \Leftrightarrow \mathsf{prefixExact}(T_2, k)$, then $\mathsf{prefixExact}(T_1', h) \Leftrightarrow \mathsf{prefixExact}(T_2', h)$ for some h.

Proof. By Lemma 7.18, for all p_1' and p_2' are in $\mathsf{paths}(T_1' \,\&\, T_2')$, $\Gamma \vdash p_1' = p_2'$. If x is not free in T, then $T_1' = T_1$ and $T_2' = T_2$ and the lemma holds with $h = k$. Otherwise, by Lemma 7.17, $p_1 = p_2$. The lemma holds by induction on the structure of p_1. □

Lemma 7.20 If $x : T_x \in \Gamma$, and $\Gamma \vdash T_v \leq T_x$, and $\Gamma \vdash T : \mathsf{type}$, and $\Gamma \vdash T_v : \mathsf{type}$, and $T \{\!\{\Gamma;\ T_v/x\}\!\} = T'$, then $\Gamma \vdash T' : \mathsf{type}$.

Proof. The proof is by induction on type substitution derivation.

Case $T = \circ$:

Trivial.

Case $T = T_0.C$:

Then $T \{\!\{\Gamma;\ T_v/x\}\!\} = T_0'.C = T_0 \{\!\{\Gamma;\ T_v/x\}\!\}.C$. By the induction hypothesis, T_0' is well-formed. Let $\Gamma \vdash T_0 \lhd S_0$ and $\Gamma \vdash T_0' \lhd S_0'$. By Lemma 7.14, $\vdash S_0' \sqsubset^* S_0$. Therefore $\vdash S_0'.C : \mathsf{nondep}$, and by WF-NEST, $\Gamma \vdash T_0'.C : \mathsf{type}$.

Case $T = p.\mathtt{class}$:

We consider p by cases.

Case $p = v$:

Trivial.

Case $p = y \neq x$:

Trivial.

Case $p = x$:

Then $T\{\!\{\Gamma;\ T_v/x\}\!\} = T_v$ and the case follows from the assumption that T_v is well-formed.

Case $p = p_0.f$:

Let $p_0.\texttt{class}\{\!\{\Gamma;\ T_v/x\}\!\} = T_p$. Then, by Lemma 7.15, $\mathsf{ftype}(\Gamma, T_p, f) = T_f = \mathsf{ftype}(\Gamma, p_0.\texttt{class}, f)$. There are two cases:

Case $T_p = p_0'.\texttt{class}$:

Then $T' = p_0'.f.\texttt{class}$. Since by Lemma 7.15, ftype is unchanged by the substitution, we can derive $\Gamma \vdash p_0'.f : T_f$ final by F-GET. Hence, by WF-FIN, we have $\Gamma \vdash p_0'.f.\texttt{class} : \mathsf{type}$.

Otherwise:

Assume $T_p \neq p_0'.\texttt{class}$. Then $T' = T_f$. By the induction hypothesis, $\Gamma \vdash T_f : \mathsf{type}$.

Case $T = P[T_0]$:

Then $T' = P[T_0']$ where $T_0\{\!\{\Gamma;\ T_v/x\}\!\} = T_0'$. By the induction hypothesis, T_0' is well-formed. Let $\Gamma \vdash T_0 \lhd S_0$ and $\Gamma \vdash T_0' \lhd S_1$. By Lemma 7.14, $\vdash S_1 \sqsubset^* S_0$. Therefore, by Lemma 7.16, $\mathsf{prefix}(P, S_1) \supseteq \mathsf{prefix}(P, S_0)$. Hence, $\mathsf{prefix}(P, S_1) \neq \emptyset$. Finally, by WF-PRE, we can derive $\Gamma \vdash T' : \mathsf{type}$.

Case $T = \&\overline{T}$:

Then $T' = \&\overline{T'}$ where for all i, $T_i\{\!\{\Gamma;\ T_v/x\}\!\} = T_i'$. By WF-MEET, for all i, $\Gamma \vdash T_i$: type. Therefore, by the induction hypothesis, for all i, $\Gamma \vdash T_i' : \mathsf{type}$. If x is not free in T, then $T = T'$ and the case holds trivially. So, assume x is free in T.

By WF-MEET, all p in $\mathsf{paths}(T)$ are aliases. By Lemma 7.18, all p' in $\mathsf{paths}(T')$ are aliases.

By WF-MEET, for all i and j, $\mathsf{prefixExact}(T_i, k) \Rightarrow \mathsf{prefixExact}(T_j, k)$. Thus, by Lemma 7.19, $\mathsf{prefixExact}(T_i', h) \Rightarrow \mathsf{prefixExact}(T_j', h)$ for some h. Thus, we can derive by WF-MEET, $\Gamma \vdash T' : \mathsf{type}$. □

7.6 Value substitution

We now prove several value substitution lemmas. First, we show that after substituting v for x in a final access path, the new path's declared type is a subtype of the original path's declared type.

Lemma 7.21 If $x : T_x \in \Gamma$, and $\Gamma\{v/x\} \vdash v : T_v$ and $\Gamma\{v/x\} \vdash T_v \leq T_x$, and $\Gamma \vdash p : T$ final, then $\Gamma\{v/x\} \vdash p\{v/x\} : T_s$ final. where $\Gamma\{v/x\} \vdash T_s \leq T\{v/x\}$.

Proof. The proof is by induction on the derivation of $\Gamma \vdash p : T$ final. Let $p' = p\{v/x\}$

Case F-NULL:

Then $p = p'$. By F-NULL, $\Gamma\{v/x\} \vdash \mathtt{null} : T\{v/x\}$ final.

Case F-LOC:

Then $p = p'$ and $T = T\{v/x\}$.

Case F-VAR:

Let $p = y \neq x$. Then $p = p'$. Then $y : T \in \Gamma$. If x is not free in T, then $T\{v/x\} = T$. If, on the other hand, x is free in T, then $y : T\{v/x\} \in \Gamma\{v/x\}$ and we can derive $\Gamma\{v/x\} \vdash y : T\{v/x\}$ final by F-VAR. Now, let $p = x$. Then $p' = v$ and $T = T_x$ and $T\{v/x\} = T_s = T_v$. Since we assumed $\Gamma\{v/x\} \vdash T_v \leq T_x$, the case holds trivially.

Case F-GET:

Then $p = p_0.f$, $\Gamma \vdash p_0 : T_0$ final, $\mathsf{ftype}(\Gamma, T_0, f) = T_f$, and $T = T_f$. By the induction hypothesis, $\Gamma\{v/x\} \vdash p_0\{v/x\} : T_0'$ final, where $\Gamma\{v/x\} \vdash T_0' \leq T_0\{v/x\}$. By Lemma 7.26, $\mathsf{ftype}(\Gamma, T_0\{v/x\}, f) = T_f$; therefore, $\mathsf{ftype}(\Gamma, T_0', f) = T_f$. Thus, we can derive by F-GET, $\Gamma \vdash p_0\{v/x\}.f : T_f\{p_0\{v/x\}/\texttt{this}\}$ final, which can be rewritten: $\Gamma \vdash p_0.f\{v/x\} : (T_f\{p_0/\texttt{this}\})\{v/x\}$ final. □

Next, we prove a useful pair of lemmas that allows many of the type substitution (T for x) lemmas proved above in Section 7.5 to be used easily to prove value substitution (v for x) lemmas.

Lemma 7.22 If $x : T_x \in \Gamma$, and $\Gamma\{v/x\} \vdash v : T_v$ and $\Gamma\{v/x\} \vdash T_v \leq T_x$, and $\Gamma \vdash p.\texttt{class}$: type, then $p.\texttt{class}\{\!\{\Gamma;\ v.\texttt{class}/x\}\!\} = p\{v/x\}.\texttt{class}$.

Proof. The proof is by structural induction on p. Let $p' = p\{v/x\}$.

Case $p = x$:

Then $p' = v$. The case follows trivially since $x.\texttt{class}\{\!\{\Gamma;\ v.\texttt{class}/x\}\!\} = v.\texttt{class}$.

Case $p = p_0.f$:

Then $p' = p_0\{v/x\}.f$, and by the induction hypothesis, $p_0.\texttt{class}\{\!\{\Gamma;\ v.\texttt{class}/x\}\!\} = p_0\{v/x\}.\texttt{class}$. Thus, $p_0.f.\texttt{class}\{\!\{\Gamma;\ v.\texttt{class}/x\}\!\} = p_0\{v/x\}.f.\texttt{class}$.

Otherwise:

$p' = p$ and the case holds trivially. □

Lemma 7.23 (Type substitution lifting) If $x : T_x \in \Gamma$, and $\Gamma\{v/x\} \vdash v : T_v$ and $\Gamma\{v/x\} \vdash T_v \leq T_x$, and $\Gamma \vdash T : \mathsf{type}$, then $T\{\!\{\Gamma;\ v.\mathtt{class}/x\}\!\} = T\{v/x\}$.

Proof. The proof is by structural induction on T.

Case $T = \circ$:

Trivial.

Case $T = T_0.C$:

Follows from the induction hypothesis and definition of type substitution.

Case $T = p.\mathtt{class}$:

Then $T\{v/x\} = p'.\mathtt{class}$ where $p' = p\{v/x\}$. The case follows from Lemma 7.22.

Case $T = P[T_0]$:

Then $T\{v/x\} = P[T_0\{v/x\}]$. By the induction hypothesis, $T_0\{\!\{\Gamma;\ v.\mathtt{class}/x\}\!\} = T_0\{v/x\}$. Since $\mathsf{exact}(P[T_0])$, we also have $\mathsf{exact}(P[T_0\{v/x\}])$. Hence, the case holds by the definition of type substitution,

Case $T = \&\overline{T}$:

Follows from the induction hypothesis and definition of type substitution. □

Using the lifting lemma, we can show that value substitution preserves type well-formedness, tightens the non-dependent bound of a type, and preserves the result of field and method lookups.

Lemma 7.24 If $x : T_x \in \Gamma$, and $\Gamma\{v/x\} \vdash v : T_v$ and $\Gamma\{v/x\} \vdash T_v \leq T_x$, and $\Gamma \vdash T : \mathsf{type}$ then $\Gamma\{v/x\} \vdash T\{v/x\} : \mathsf{type}$.

Proof. Follows from Lemma 7.23 and Lemma 7.20. □

Lemma 7.25 If $x : T_x \in \Gamma$, and $\Gamma\{v/x\} \vdash v : T_v$ and $\Gamma\{v/x\} \vdash T_v \leq T_x$, and $\Gamma \vdash T \lhd S$, then $\Gamma\{v/x\} \vdash T\{v/x\} \lhd S'$ where $\vdash S' \sqsubset^* S$.

Proof. Follows from Lemma 7.14 and Lemma 7.23. □

Lemma 7.26 If $x : T_x \in \Gamma$, and $\Gamma\{v/x\} \vdash v : T_v$ and $\Gamma\{v/x\} \vdash T_v \leq T_x$, and $\mathsf{ftype}(\Gamma, T, f) = [\texttt{final}]\ T_f$, then $\mathsf{ftype}(\Gamma\{v/x\}, T\{v/x\}, f) = [\texttt{final}]\ T_f$.

Proof. Follows from Lemma 7.25 and the definition of fields. □

Lemma 7.27 If $x : T_x \in \Gamma$, and $\Gamma\{v/x\} \vdash v : T_v$ and $\Gamma\{v/x\} \vdash T_v \leq T_x$, and $\mathsf{mtype}(\Gamma, T, m) = (\overline{x} : \overline{T}) \to T_{n+1}$, then $\mathsf{mtype}(\Gamma\{v/x\}, T\{v/x\}, m) = (\overline{x} : \overline{T}) \to T_{n+1}$.

Proof. Follows from Lemma 7.25 and the definition of methods. □

Next, we want to show that value substitution on a type preserves the subtyping relation. We first show that exactness is preserved by value substitution.

Lemma 7.28 If $x : T_x \in \Gamma$, and $\Gamma\{v/x\} \vdash v : T_v$ and $\Gamma\{v/x\} \vdash T_v \leq T_x$, and $\mathsf{exact}(T)$, then $\mathsf{exact}(T\{v/x\})$.

Proof. By inspection of definition of exact. □

Next we show that type substitution is preserved by value substitution if variable capture is avoided.

Lemma 7.29 If $x : T_x \in \Gamma$, and $\Gamma\{v/x\} \vdash v : T_v$ and $\Gamma\{v/x\} \vdash T_v \leq T_x$, and $T\{\!\{\Gamma;\ T_y/y\}\!\} = T'$, and x is not free in T, then $T\{\!\{\Gamma\{v/x\};\ T_y\{v/x\}/y\}\!\} = T'\{v/x\}$.

Proof. The proof is by induction on type substitution derivation.

Case $T = \circ$:

Trivial.

Case $T = T_0.C$:

Follows from the induction hypothesis.

Case $T = p.\texttt{class}$:

We prove the case by structural induction on p.

Case $p = v$:

Trivial.

Case $p = z \neq y$:

Trivial.

Case $p = y$:

Then $T' = T_y$ and $T'\{v/x\} = T_y\{v/x\}$. By the definition of type substitution, $y.\texttt{class}\{\!\{\Gamma\{v/x\};\ T_y\{v/x\}/y\}\!\} = T_y\{v/x\}$.

Case $p = p_0.f$:

Let $p_0.\texttt{class}\{\!\{\Gamma;\ T_y/y\}\!\} = T_p$. and $p_0.\texttt{class}\{\!\{\Gamma\{v/x\};\ T_y\{v/x\}/y\}\!\} = T'_p$. By the induction hypothesis, we have $T'_p = T_p\{v/x\}$.

Case $T_p = p_1.\texttt{class}$:

Then $T'_p = p_1\{v/x\}.\texttt{class}$ and $p_0.f.\texttt{class}\{\!\{\Gamma;\ T_y/y\}\!\} = p_1.f.\texttt{class}$. Therefore, $p_0.f.\texttt{class}\{\!\{\Gamma\{v/x\};\ T_y\{v/x\}/y\}\!\} = p_1\{v/x\}.f.\texttt{class}$, which equals $p_1.f.\texttt{class}\{v/x\}$.

Otherwise:

Then, $T_p \neq p_1.\texttt{class}$. Since $\mathsf{ftype}(\Gamma, T_p, f) = T_f$, by Lemma 7.26, we have $\mathsf{ftype}(\Gamma\{v/x\}, T_p\{v/x\}, f) = T_f$. Since by F-OK, $\emptyset \vdash T_f : \mathsf{type}$, x is not free in T_f, and hence $T_f\{v/x\} = T_f$.

Case $T = P[T_0]$:

Follows from the induction hypothesis.

Case $T = \&\overline{T}$:

Follows from the induction hypothesis. □

Using the above lemmas, we can finally show that value substitution preserves subtyping.

Lemma 7.30 If $x : T_x \in \Gamma$, and $\Gamma\{v/x\} \vdash v : T_v$ and $\Gamma\{v/x\} \vdash T_v \leq T_x$, and $\Gamma \vdash T_1 \leq T_2$, then $\Gamma\{v/x\} \vdash T_1\{v/x\} \leq T_2\{v/x\}$.

Proof. The proof is by induction on the derivation of $\Gamma \vdash T_1 \leq T_2$.

Case S-REFL:

Trivial.

Case S-TRANS:

Trivial via the induction hypothesis.

Case S-SUP:

Follows from the induction hypothesis and Lemma 7.29. and S-TRANS.

Case S-BOUND:

By Lemma 7.24 and Lemma 7.25 and S-TRANS.

Case S-NEST:

Follows from the induction hypothesis and Lemma 7.24.

Case S-FIN:

Lemma 7.21.

Case S-PRE-1:

Follows from the induction hypothesis and Lemma 7.24.

Case S-PRE-2:

By Lemma 7.24.

Case S-PRE-OUT:

Follows from the induction hypothesis.

Case S-PRE-IN:

By Lemma 7.24.

Case S-MEET-LB:

By Lemma 7.24.

Case S-MEET-G:

Follows from the induction hypothesis.

Case S-ALIAS:

Follows from definition of $\Gamma\{v/x\}$.

Case S-EVAL:

Trivial since $U_i\{v/x\} = U_i$. $\square$

This lemma is the main substitution lemma and states that typing is preserved by substitution.

Lemma 7.31 *(Substitution)* If $x : T_x \in \Gamma$, and $\Gamma\{v/x\} \vdash v : T_v$ and $\Gamma\{v/x\} \vdash T_v \leq T_x$, and $\Gamma \vdash e : T$, then $\Gamma\{v/x\} \vdash e\{v/x\} : T\{v/x\}$.

Proof. The proof is by induction on the derivation of $\Gamma \vdash e : T$.

Let $e' = e\{v/x\}$ and $T' = T\{v/x\}$.

Case T-FIN:

Then $e = p$ and $T = p.\texttt{class}$ and $e' = p\{v/x\}$ and $T' = p\{v/x\}.\texttt{class}$. The case follows from Lemma 7.21.

Case T-GET:

Then $e = e_0.f$, $\Gamma \vdash e_0 : T_0$, $\mathsf{ftype}(\Gamma, T_0, f) = [\texttt{final}]\ T_f$, and $T = T_f$, and $e' = e_0\{v/x\}.f = e'_0.f$.

Since $\Gamma \vdash e_0 : T_0$, by the induction hypothesis we have $\Gamma\{v/x\} \vdash e_0\{v/x\} : T_0\{v/x\}$. By Lemma 7.26, and $\mathsf{ftype}(\Gamma\{v/x\}, T_0\{v/x\}, f) = [\texttt{final}]\ T_f$. Since by F-OK, $\emptyset \vdash T_f : \mathsf{type}$, x is not free in T_f, and hence $T' = T_f\{v/x\} = T_f = T$. The case holds by T-GET.

Case T-SET:

The proof of this case is similar to the proof of the previous case for T-GET.

Case T-SEQ:

Follows from the induction hypothesis.

Case T-NEW:

Follows from Lemma 7.24, Lemma 7.26, and the induction hypothesis.

Case T-CALL:

Follows from the induction hypothesis, Lemma 7.27, Lemma 7.29, and Lemma 7.28.

Case T-SUB:

Then $\Gamma \vdash e : T''$ where $\Gamma \vdash T'' \leq T$. By the induction hypothesis, $\Gamma\{v/x\} \vdash e\{v/x\} : T''\{v/x\}$. By Lemma 7.30, $\Gamma\{v/x\} \vdash T''\{v/x\} \leq T\{v/x\}$. Thus, by T-SUB, $\Gamma\{v/x\} \vdash e\{v/x\} : T\{v/x\}$. $\square$

The following lemma relates type and value substitution. It states that a value substitution of v for x in a type T results in a subtype of the type substitution of v's static type T_v for occurrences of x in T.

Lemma 7.32 If $x : T_x \in \Gamma$, and $\Gamma\{v/x\} \vdash v : T_v$ and $\Gamma\{v/x\} \vdash T_v \leq T_x$, and $\Gamma \vdash \mathsf{env}$, and $T\{\!\{\Gamma\{v/x\};\ T_v/x\}\!\} = T'$, then $\Gamma\{v/x\} \vdash T\{v/x\} \leq T'$.

Proof. The proof is by induction on type substitution derivation.

Case $T = \circ$:

Trivial.

Case $T = T_0.C$:

Then $T\{v/x\} = T_0\{v/x\}.C$ and $T' = T_0'.C$ where $T_0\{\!\{\Gamma\{v/x\};\ T_v/x\}\!\} = T_0'$. By the induction hypothesis $\Gamma\{v/x\} \vdash T_0\{v/x\} \leq T_0'$; therefore, by S-NEST, $\Gamma\{v/x\} \vdash T_0\{v/x\}.C \leq T_0'.C$.

Case $T = p.\texttt{class}$:

Then $T\{v/x\} = p\{v/x\}.\texttt{class}$. We consider p by cases.

Case $p = v$:

Trivial.

Case $p = y \neq x$:

Trivial.

Case $p = x$:

Then $T = x.\texttt{class}$ and $T' = T_v$ and $T\{v/x\} = v.\texttt{class}$. Since $\Gamma\{v/x\} \vdash v : T'$, $\Gamma\{v/x\} \vdash v.\texttt{class} \leq T'$ by Lemma 7.12.

Case $p = p_0.f$:

Then $T = p_0.f.\texttt{class}$ and $T\{v/x\} = p_0\{v/x\}.f.\texttt{class}$. Applying the definition of type substitution, let $p_0.\texttt{class}\{\!\{\Gamma\{v/x\};\ T_v/x\}\!\} = T_p$. Then, by the induction hypothesis, $\Gamma\{v/x\} \vdash p_0\{v/x\}.\texttt{class} \leq T_p$. There are two cases for T_p.

Case $T_p \neq p_0'.\texttt{class}$ for any p_0':

Then, $\mathsf{ftype}(\Gamma\{v/x\}, T_p, f) = T_f$. By F-GET, we have $\Gamma\{v/x\} \vdash p_0\{v/x\}.f : T_f$ final. Therefore, by Lemma 7.12, we can derive the subtyping judgment $\Gamma\{v/x\} \vdash p_0\{v/x\}.f.\texttt{class} \leq T_f\{p_0\{v/x\}/\texttt{this}\}$. Since T_f has no free variables, $T_f = T_f\{p_0\{v/x\}/\texttt{this}\}$.

Case $T_p = p_0'.\texttt{class}$:

It must be that $p_0' = p_0\{v/x\}$. The case follows trivially from S-REFL.

Case $T = P[T_0]$:

Then $T\{v/x\} = P[T_0\{v/x\}]$. and $T' = P[T_0']$ where and $T_0\{\!\{\Gamma\{v/x\};\ T_v/x\}\!\} = T_0'$. By the induction hypothesis we have $\Gamma\{v/x\} \vdash T_0\{v/x\} \leq T_0'$; therefore, by S-PRE-1 we have $\Gamma\{v/x\} \vdash T\{v/x\} \leq T'$.

Case $T = \&\overline{T}$:

Then $T\{v/x\} = \&\overline{T\{v/x\}}$ and $T' = \&\overline{T'}$ where for all i, $T_i\{\!\{\Gamma\{v/x\};\ T_v/x\}\!\} = T_i'$. By the induction hypothesis we have $\Gamma\{v/x\} \vdash T_i\{v/x\} \leq T_i'$; therefore, by S-MEET-G we have $\Gamma\{v/x\} \vdash T\{v/x\} \leq T'$. $\square$

7.7 Inheritance and subtyping

Here we prove some lemmas about inheritance and subtyping.

Lemma 7.33 If $\vdash P_1 \sqsubset P_2$, then $\emptyset \vdash P_1 \leq P_2$.

Proof. The proof is by induction on the derivation of $\vdash P_1 \sqsubset P_2$. There are two cases:

Case INH-SC:

If $\vdash P_1 \sqsubset_{\mathsf{sc}} P_2$, then $P_1 = P_1'.C$ and there is a P such that $\vdash P_1' \sqsubset^* P$, and $CT(P.C) =$ `class` C `extends` T $\{\ldots\}$, and $T\{\!\{\emptyset;\ P_1'/\texttt{this}\}\!\} = S$, and $P_2 \in \mathsf{mem}(S)$. By the induction hypothesis and S-TRANS, $\emptyset \vdash P_1' \leq P$. Thus, by S-SUP, we can derive $\emptyset \vdash P_1'.C \leq P_2$.

Case INH-FB:

If $\vdash P_1 \sqsubset_{\mathsf{fb}} P_2$, then $P_1 = P_1'.C$ and $P_2 = P_2'.C$ and $\vdash P_1' \sqsubset P_2'$. By the induction hypothesis, $\emptyset \vdash P_1' \leq P_2'$. By S-NEST, $\emptyset \vdash P_1 \leq P_2$. $\square$

Lemma 7.34 If $P \in \mathsf{supers}(S)$, then $\emptyset \vdash S \leq P$.

Proof. Trivial from Lemma 7.33. $\square$

7.8 Method lookup agreement

This lemma states that a method type lookup and a method body lookup on the same type S agree with each other.

Lemma 7.35 If $\mathsf{mtype}(\emptyset, S, m) = (\overline{x}{:}\overline{T}) \to T_{n+1}$, then $\mathsf{mbody}(S, m) = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\}$.

Proof. Follows immediately from definition of mtype and mbody. □

7.9 Subject reduction

The subject reduction lemma states that a well-formed configuration steps to another well-formed configuration or to a configuration containing NullError. We first show that if a final access path p steps to p', then p' is also a final access path and furthermore it is an alias of p.

Lemma 7.36 If $\vdash p, H$, and $\lfloor H \rfloor \vdash p : T$ final, and $p, H \longrightarrow p', H$, and $\lfloor H \rfloor \vdash p' : T'$ final, then $\lfloor H \rfloor \vdash p = p'$.

Proof. The proof is by induction on the derivation of $\lfloor H \rfloor \vdash p : T$ final.

Since p can make a step, $p = p_0.f$. We consider p_0 by cases.

Case $p_0 = \texttt{null}$:

Then $p = \texttt{null}.f$ and R-NULL is the only rule that can apply.

Case $p_0 = \ell$:

Then $p = \ell.f$ and R-GET is the only rule that can apply, $p' = v_i = H(\ell)[f_i]$ where $H(\ell) = S\ \{\overline{f} = \overline{v}\}$. By the construction of $\lfloor H \rfloor$, $\lfloor H \rfloor$ must include $\ell.f_i = v_i$.

Case $p_0 \neq v$:

Then R-CONG is the only rule that can apply and $p_0, H \longrightarrow p_0', H$. By F-GET, $\lfloor H \rfloor \vdash p_0 : T_0$ final. By the induction hypothesis, $\lfloor H \rfloor \vdash p_0 = p_0'$. Thus, by A-FIELD, $\lfloor H \rfloor \vdash p_0.f = p_0'.f$. $\square$

We also prove that if p steps to p', then the declared type of p' is a subtype of the declared type of p.

Lemma 7.37 If $\vdash p, H$, and $\lfloor H \rfloor \vdash p : T$ final, and $p, H \longrightarrow p', H$, then $\vdash p', H$ and $\lfloor H \rfloor \vdash p' : T'$ final, where $\lfloor H \rfloor \vdash T' \leq T$.

Proof. The proof is by induction on the derivation of $\lfloor H \rfloor \vdash p : T$ final.

Since p can make a step, $p = p_0.f$. We consider p_0 by cases.

Case $p_0 = \texttt{null}$:

Then $p = \texttt{null}.f$ and R-NULL is the only rule that can apply.

Case $p_0 = \ell$:

Then $p = \ell.f$ and R-GET is the only rule that can apply. Then, $p' = v_i = H(\ell)[f_i]$ where $H(\ell) = S\ \{\overline{f} = \overline{v}\}$. By F-GET, $\lfloor H \rfloor \vdash \ell : T_0$ final, and $\mathsf{ftype}(\lfloor H \rfloor, T_0, f) = T$. By F-OK, T must be of the form S_f.

If $v_i = \texttt{null}$, then $\lfloor H \rfloor \vdash v_i : S_i$ final by F-NULL for any S_i. Specifically, let $S_i = T$. Since $\vdash p, H$, by CONFIG and HEAP, we have $H \vdash \ell : \mathsf{loc}$. Thus, by H-LOC, we have $\lfloor H \rfloor \vdash v_i : T$.

If $v_i = \ell_i$, then $\lfloor H \rfloor \vdash v_i : S_i$ final by F-LOC where $\ell_i : S_i \in \lfloor H \rfloor$. By Lemma 7.13, $\lfloor H \rfloor \vdash S_i \leq T$. By H-LOC, we can also derive $v_i \in \mathrm{dom}(H) \cup \{\texttt{null}\}$. If $v_i = \ell_i$, then $v_i \in \mathrm{dom}(H)$. Therefore by CONFIG, $\vdash v_i, H$.

Case $p_0 \neq v$:

Then R-CONG is the only rule that can apply and $p_0, H \longrightarrow p_0', H$. By F-GET, $\lfloor H \rfloor \vdash p_0 : T_0$ final, and $\mathsf{ftype}(\lfloor H \rfloor, T_0, f) = T$. By the induction hypothesis, $\lfloor H \rfloor \vdash p_0' : T_0'$ final and $\lfloor H \rfloor \vdash T_0' \leq T_0$. By Lemma 7.11, $\mathsf{ftype}(\lfloor H \rfloor, T_0', f) = T$. Hence, we can derive by F-GET, $\lfloor H \rfloor \vdash p_0'.f : T$ final.

By the induction hypothesis, $\vdash p_0', H$. Therefore, since $\mathsf{locs}(p_0'.f) = \mathsf{locs}(p_0')$, by CONFIG we can derive $\vdash p_0'.f, H$. $\square$

This lemma states that if a dependent type steps to another dependent type, the bound on the result type is tighter.

Lemma 7.38 If $\lfloor H \rfloor \vdash TE[p] \lhd S$ and $p, H \longrightarrow p', H$, then $\lfloor H \rfloor \vdash TE[p'] \lhd S'$ where $\vdash S' \sqsubset^* S$.

Proof. The proof is by induction on $\lfloor H \rfloor \vdash TE[p] \lhd S$.

Case $TE = TE_0.C$:

Then $TE[p] = TE_0[p].C$. By BD-NEST, $\lfloor H \rfloor \vdash TE_0[p] \lhd S_0$ where $S = S_0.C$. By the induction hypothesis, $\lfloor H \rfloor \vdash TE_0[p'] \lhd S_0'$. Thus, we can derive by BD-NEST. $\lfloor H \rfloor \vdash TE_0[p'] \lhd S_0'.C$. Also, by the induction hypothesis, $\vdash S_0' \sqsubset^* S_0$. We therefore have $\vdash S_0'.C \sqsubset^* S_0.C$ by the definition of INH-FB.

Case $TE = E.\texttt{class}$:

Then $TE[p] = E.\texttt{class}[p] = E[p].\texttt{class}$. By BD-FIN, $\lfloor H \rfloor \vdash E[p] = p_2$, $\lfloor H \rfloor \vdash E[p] : T_1$ final, $\lfloor H \rfloor \vdash p_2 : T_2$ final, $\lfloor H \rfloor \vdash T_1 \lhd S_1$, $\lfloor H \rfloor \vdash T_2 \lhd S_2$, and $S = S_1 \,\&\, S_2$.

By Lemma 7.37, $\lfloor H \rfloor \vdash E.\texttt{class}[p'] : T'$ final where $\lfloor H \rfloor \vdash T' \leq T$.

Let $\lfloor H \rfloor \vdash T' \lhd S'$. By BD-FIN, we can derive $\lfloor H \rfloor \vdash E[p'].\texttt{class} \lhd S' \,\&\, S_2$, By Lemma 7.10, $\vdash S' \sqsubset^* S_1$. Therefore, $\vdash S' \,\&\, S_1 \sqsubset^* S_1 \,\&\, S_2$.

Case $TE = P[TE_0]$:

Then $TE[p] = P[TE_0[p]]$. By BD-PRE, $\lfloor H \rfloor \vdash TE_0[p] \lhd S_0$, and $S = P[S_0]$. By the induction hypothesis, $\lfloor H \rfloor \vdash TE_0[p'] \lhd S_0'$ where $\vdash S_0' \sqsubset^* S_0$. By BD-PRE, we have $S' = P[S_0']$.

By Lemma 7.16, $\mathsf{prefix}(P, S_0') \supseteq \mathsf{prefix}(P, S_0)$. Therefore $S' \sqsubset^* S$. Thus, by BD-PRE, $\lfloor H \rfloor \vdash P[TE_0[p']] \lhd S'$.

Case $TE = \&(\overline{U}, TE_0, \overline{T})$:

Then $TE[p] = \&(\overline{U}, TE_0[p], \overline{T})$. By WF-MEET, $\lfloor H \rfloor \vdash TE_0[p] : \mathsf{type}$. By the induction hypothesis, $\lfloor H \rfloor \vdash TE_0[p'] : \mathsf{type}$. All other components of the intersection do not change and therefore remain well-formed. Thus, we can derive by WF-MEET, $\lfloor H \rfloor \vdash \&(\overline{U}, TE_0[p'], \overline{T}) : \mathsf{type}$. $\square$

This lemma states that if a dependent type steps to another dependent type, the result type is well-formed.

Lemma 7.39 If $\lfloor H \rfloor \vdash TE[p] : \mathsf{type}$ and $p, H \longrightarrow p', H$, then $\lfloor H \rfloor \vdash TE[p'] : \mathsf{type}$.

Proof. The proof is by induction on $\lfloor H \rfloor \vdash TE[p] : \mathsf{type}$.

Case $TE = TE_0.C$:

Then $TE[p] = TE_0[p].C$. By WF-NEST, $\lfloor H \rfloor \vdash TE_0[p] : \mathsf{type}$, $\lfloor H \rfloor \vdash TE_0[p] \lhd S$, and $\vdash S.C : \mathsf{nondep}$. By the induction hypothesis, $\lfloor H \rfloor \vdash TE_0[p'] : \mathsf{type}$. By Lemma 7.38, $\lfloor H \rfloor \vdash TE_0[p'] \lhd S'$ where $\vdash S' \sqsubset^* S$. Since $\vdash S' \sqsubset^* S$ and since $\vdash S.C : \mathsf{nondep}$, we also have $\vdash S'.C : \mathsf{nondep}$. Thus, we can derive by WF-NEST. $\lfloor H \rfloor \vdash TE_0[p'] : \mathsf{type}$.

Case $TE = E.\mathtt{class}$:

Then $TE[p] = E.\texttt{class}[p]$. By WF-FIN, $\lfloor H \rfloor \vdash E.\texttt{class}[p] : T$ final. By Lemma 7.37, $\lfloor H \rfloor \vdash E.\texttt{class}[p'] : T'$ final. Hence, by WF-FIN, we can derive $\lfloor H \rfloor \vdash E.\texttt{class}[p'] :$ type.

Case $TE = P\texttt{[}TE_0\texttt{]}$:

Then $TE[p] = P\texttt{[}TE_0[p]\texttt{]}$. By WF-PRE, $\lfloor H \rfloor \vdash P :$ type, $\lfloor H \rfloor \vdash TE_0[p] :$ type, $\lfloor H \rfloor \vdash P\texttt{[}TE_0[p]\texttt{]} \lhd S$, and $\mathsf{prefix}(P,S) \neq \emptyset$.

By the induction hypothesis, $\lfloor H \rfloor \vdash TE_0[p'] :$ type. By Lemma 7.38 $\lfloor H \rfloor \vdash TE_0[p'] \lhd S'_0$, where $\vdash S'_0 \sqsubset^* S_0$. By Lemma 7.16, $\mathsf{prefix}(P,S'_0) \supseteq \mathsf{prefix}(P,S_0)$. Therefore $\mathsf{prefix}(P,S'_0) \neq \emptyset$. Hence, by WF-PRE, we can derive $\lfloor H \rfloor \vdash P\texttt{[}TE_0[p']\texttt{]} :$ type.

Case $TE = \&(\overline{U}, TE_0, \overline{T})$:

Then $TE[p] = \&(\overline{U}, TE_0[p], \overline{T})$. By WF-MEET, $\lfloor H \rfloor \vdash TE_0[p] :$ type. By the induction hypothesis, $\lfloor H \rfloor \vdash TE_0[p'] :$ type. All other components of the intersection do not change and therefore remain well-formed.

Since the structure of $TE_0[p]$ and $TE_0[p']$ are the same, it is easy to see that $\mathsf{prefixExact}(TE_0[p],k) \Leftrightarrow \mathsf{prefixExact}(TE_0[p'],k)$.

Since all T_i in $\mathsf{exacts}(TE[p])$ are equivalent up to aliasing, and since by Lemma 7.36 $\lfloor H \rfloor \vdash p = p'$, we have all T_i in $\mathsf{exacts}(TE[p'])$ are equivalent up to aliasing,

Thus, we can derive by WF-MEET, $\lfloor H \rfloor \vdash \&(\overline{U}, TE_0[p'], \overline{T}) :$ type. $\square$

The following lemma states that if one well-formed configuration steps to another well-formed configuration without changing the heap, then the result configuration in an evaluation context is also well-formed.

Lemma 7.40 If $\vdash E[e], H$, and $e, H \longrightarrow e', H$, and $\vdash e', H$, then $\vdash E[e'], H$.

Proof. Since $\mathsf{locs}(E[e']) \subseteq \mathsf{locs}(E[e]) \cup \mathsf{locs}(e')$, and $\mathsf{locs}(E[e]) \subseteq \mathrm{dom}(H)$, and $\mathsf{locs}(e') \subseteq \mathrm{dom}(H)$, we have $\mathsf{locs}(E[e']) \subseteq \mathrm{dom}(H)$. Since $\vdash e', H$, we have $\vdash H$. Thus, by CONFIG, $\vdash E[e'], H$. $\square$

Next, we show that if a type T is exact and has a subtype that is a location dependent type $\ell.\texttt{class}$, that T is a subtype of $\ell.\texttt{class}$; thus, the two types are equivalent.

Lemma 7.41 If $\lfloor H \rfloor \vdash \ell.\texttt{class} \leq T$ and $\mathsf{exact}(T)$, then $\lfloor H \rfloor \vdash T \leq \ell.\texttt{class}$.

Proof. The proof is by induction on the derivation of $\lfloor H \rfloor \vdash \ell.\texttt{class} \leq T$.

Case S-REFL:

Trivial.

Case S-TRANS:

Follows from the induction hypothesis.

Case S-SUP:

Vacuous.

Case S-BOUND:

Vacuous.

Case S-NEST:

Vacuous.

Case S-FIN:

Vacuous since S-FIN requires T be an S, which is not exact.

Case S-PRE-1:

Vacuous.

Case S-PRE-2:

Vacuous.

Case S-PRE-OUT:

Vacuous.

Case S-PRE-IN:

Then $T = P[\ell.\texttt{class}].C$. Trivial by S-PRE-IN.

Case S-MEET-LB:

Vacuous.

Case S-MEET-G:

Then $T = \&\overline{T}$ and $\lfloor H \rfloor \vdash \ell.\texttt{class} \leq T_i$ for all i. By WF-MEET, all T_i are exact. By the induction hypothesis, $\lfloor H \rfloor \vdash T_i \leq \ell.\texttt{class}$. Therefore, $\lfloor H \rfloor \vdash T \leq \ell.\texttt{class}$ by S-MEET-LB.

Case S-ALIAS:

Trivial.

Case S-EVAL:

Trivial. □

Finally, we prove the subject reduction lemma.

Lemma 7.2 *(Subject reduction)* If $\vdash e, H$, $\lfloor H \rfloor \vdash e : T$, and $e, H \longrightarrow r, H'$, then either

- $r = e'$, $\vdash e', H'$, and $\lfloor H' \rfloor \vdash e' : T$, or
- $r = \mathsf{NullError}$.

Proof. The proof is by induction on the typing derivation $\lfloor H \rfloor \vdash e : T$. We first consider the case where the derivation of $\lfloor H \rfloor \vdash e : T$ ends with an application of T-SUB. Then $\lfloor H \rfloor \vdash e : T'$ where $\lfloor H \rfloor \vdash T' \leq T$.

If $r = e'$, then by the induction hypothesis, $\lfloor H' \rfloor \vdash e' : T'$. By Lemma 7.4, since $\lfloor H' \rfloor$ extends $\lfloor H \rfloor$. $\lfloor H' \rfloor \vdash T' \leq T$. Thus, by T-SUB we can derive $\lfloor H' \rfloor \vdash e' : T$. Thus, for the remainder of the proof we need only consider typing derivations ending in a rule other than T-SUB.

We consider e by cases depending on the reduction rule used. First, note that since $\lfloor H \rfloor$ contains no $x : T$, and since $\lfloor H \rfloor \vdash e : T$, e contains no free variables. Also, note that by Lemma 7.8, $\lfloor H' \rfloor$ extends $\lfloor H \rfloor$.

For the cases below where $e = E[e_0]$ and R-CONG applies, to show that $\vdash e, H'$, we need only show that the typing derivation for e includes $\lfloor H \rfloor \vdash e_0 : T_0$. Then, by the induction hypothesis, $\vdash e'_0, H'$, and by Lemma 7.40, we can derive $\vdash E[e'_0], H'$. For the cases below where $e = NE$, R-NULL applies and $r = \mathsf{NullError}$.

Case $e = v$:

Vacuously true since v cannot take a step.

Case $e = x$:

Vacuously true since e contains no free variables.

Case $e = e_0.f$:

Case $e = \ell.f_i$:

Then R-GET is the only rule that can apply, $H' = H$, and $r = v_i = H(\ell)[f_i]$ where $H(\ell) = S\ \{\overline{f} = \overline{v}\}$. Besides T-SUB, handled above, there are two cases for the derivation of $\lfloor H \rfloor \vdash \ell.f_i : T$.

Case T-FIN:

Then $T = \ell.f_i.\texttt{class}$ and f_i is a final field By the definition of $\lfloor H \rfloor$, since $H(\ell) = S\ \{\overline{f} = \overline{v}\}$, it must that $\ell.f_i.\texttt{class} = v_i.\texttt{class} \in \lfloor H \rfloor$. Thus, by S-ALIAS, $\lfloor H \rfloor \vdash v_i.\texttt{class} \leq \ell.f_i.\texttt{class}$. Thus, by T-SUB, $\lfloor H \rfloor \vdash v_i : \ell.f_i.\texttt{class}$. Note that this is the place where we use the fact that fields are final. If f_i is not final, $\ell.f_i.\texttt{class} = v_i.\texttt{class}$ will not be in $\lfloor H \rfloor$. Since $\lfloor H' \rfloor$ extends $\lfloor H \rfloor$, By Lemma 7.4 we have $\lfloor H \rfloor \vdash v_i : \ell.f_i.\texttt{class}$.

Case T-GET:

By F-LOC and T-FIN, $\lfloor H \rfloor \vdash \ell : \ell.\texttt{class}$. Let $\mathsf{ftype}(\lfloor H \rfloor, \ell.\texttt{class}, f_i) = T_f$. By T-GET, $T_f = T$ and we can derive $\lfloor H \rfloor \vdash \ell.f_i : T$. Since $\vdash H$, and $H(\ell)[f_i] = v_i$, we have by H-LOC, $\lfloor H \rfloor \vdash v_i : T_f$.

Case $e = \texttt{null}.f$:

Then R-NULL is the only rule that can apply.

Case $e = e_0.f$ where $e_0 \neq v$:

Then R-CONG is the only rule that can apply and $e_0, H \longrightarrow e_0', H'$. Again, there are two cases for the derivation of $\lfloor H \rfloor \vdash e_0.f_i : T$.

Case T-FIN:

Then $e_0 = p$ and $e_0' = p'$ and $T = p.f.\texttt{class}$. By T-FIN, $\lfloor H \rfloor \vdash p.f : T_p$ final. By Lemma 7.36 and Lemma 7.37, $H = H'$, and $\lfloor H \rfloor \vdash p'.f : T_p'$ final, and $\lfloor H \rfloor \vdash p.f = p'.f$. Thus, we can derive by T-FIN, $\lfloor H \rfloor \vdash p'.f : p'.f.\texttt{class}$, and by S-ALIAS, $\lfloor H \rfloor \vdash p.f.\texttt{class} \approx p'.f.\texttt{class}$. and by S-SUB, $\lfloor H \rfloor \vdash p'.f : p.f.\texttt{class}$.

Case T-GET:

Then $\lfloor H \rfloor \vdash e_0 : T_0$ and $\mathsf{ftype}(\lfloor H \rfloor, T_0, f) = T_f = T$. Since $\lfloor H \rfloor \vdash e_0 : T_0$, by the induction hypothesis, $\lfloor H' \rfloor \vdash e_0' : T_0$. By Lemma 7.4, since $\lfloor H' \rfloor$ extends $\lfloor H \rfloor$, we have $\mathsf{ftype}(\lfloor H' \rfloor, T_0, f) = T_f$. Thus, we can derive by T-GET, $\lfloor H' \rfloor \vdash e_0'.f : T$.

Case $e = e_0.f = e_1$:

Case $e = \texttt{null}.f = e_1$:

Then R-NULL is the only rule that can apply.

Case $e = \ell.f = v$:

Then R-SET is the only rule that can apply and $e' = v$ and $H'(\ell)[f] = v$. The judgment $\lfloor H \rfloor \vdash v : T$ follows trivially from T-SET. Let $H(\ell) = S\ \{\overline{f} = \overline{v}\}$. Since $\vdash e, H$, we have $\vdash H$ and $H \vdash \overline{v} : \mathsf{loc}$ and also $H \vdash v : \mathsf{loc}$. By F-LOC and T-FIN, $\lfloor H' \rfloor \vdash \ell : \ell.\texttt{class}$. Let $\mathsf{ftype}(\lfloor H \rfloor, \ell.\texttt{class}, f) = T_f$. To show that H' is well-formed, we need to show that $\lfloor H' \rfloor \vdash v : T_f$. By T-SET, $T = T_f$ and therefore $\lfloor H \rfloor \vdash v : T$. Therefore by Lemma 7.4, $\lfloor H' \rfloor \vdash v : T$. Since H' is equal to H except for the value stored in $H'(\ell)[f]$, namely v, and since both $\lfloor H \rfloor \vdash v : T$ and $\lfloor H' \rfloor \vdash v : T$, and since $H \vdash v : \mathsf{loc}$, it must be that $\vdash H'$.

Case $e = \ell.f = e_1$ where $e_1 \neq v$:

Then R-CONG is the only rule that can apply and $e_1, H \longrightarrow e_1', H'$. By T-SET, $\lfloor H \rfloor \vdash \ell : T_0$, $\mathsf{ftype}(\lfloor H \rfloor, T_0, f) = T_f = T$, and $\lfloor H \rfloor \vdash e_1 : T$. By Lemma 7.4, since $\lfloor H' \rfloor$ extends $\lfloor H \rfloor$, we have $\mathsf{ftype}(\lfloor H' \rfloor, T_0, f) = T_f = T$ and $\lfloor H' \rfloor \vdash \ell : T_0$. By the induction hypothesis $\lfloor H' \rfloor \vdash e_1' : T$. Thus we can derive by T-SET, $\lfloor H' \rfloor \vdash e' : T$.

Case $e = e_0.f = e_1$ where $e_0 \neq v$:

Then R-CONG is the only rule that can apply and $e_0, H \longrightarrow e_0', H'$. By T-SET, $\lfloor H \rfloor \vdash e_0 : T_0$, $\mathsf{ftype}(\lfloor H \rfloor, T_0, f) = T_f = T$, and $\lfloor H \rfloor \vdash e_1 : T$. By the induction

hypothesis $\lfloor H' \rfloor \vdash e'_0 : T$. By Lemma 7.4, since $\lfloor H' \rfloor$ extends $\lfloor H \rfloor$, we have $\mathsf{ftype}(\lfloor H' \rfloor, T_0, f) = T_f = T$ and $\lfloor H' \rfloor \vdash e_1 : T$. Thus we can derive by T-SET, $\lfloor H' \rfloor \vdash e' : T$.

Case $e = e_0.m(\overline{e})$:

By T-CALL, all of the following hold:

- $\lfloor H \rfloor \vdash e_0 : T_0^0$
- $\mathsf{mtype}(\lfloor H \rfloor, T_0^0, m) = (\overline{x} : \overline{T^0}) \to T_{n+1}^0$
- $x_0 = \texttt{this}$
- $\forall i = 1, \ldots, n+1.\ \forall j = 1, \ldots, i.\ T_i^{j-1} \{\!\{ \lfloor H \rfloor;\ T_{j-1}^{j-1} / x_{j-1} \}\!\} = T_i^j$
- $\forall i = 1, \ldots, n.\ \forall j = 1, \ldots, i.\ \mathsf{prefixExact}(T_i^{j-1}, k) \Rightarrow \mathsf{prefixExact}(T_i^j, k)$
- $\forall i = 1, \ldots, n.\ \forall j = 1, \ldots, i.\ p.f \in \mathsf{paths}(T_i^{j-1}) \Rightarrow p\{e_{j-1}/x_{j-1}\}.f \in \mathsf{paths}(T_i^j)$
- $\forall i = 1, \ldots, n.\ \lfloor H \rfloor \vdash e_i : T_i^i$.
- $T = T_{n+1}^{n+1}$.

We consider e by cases.

Case $e = \texttt{null}.m(\overline{e})$:

Then R-NULL is the only rule that can apply.

Case $e = \ell.m(\overline{v})$:

Then R-CALL is the only rule that can apply and $H = H'$. By R-CALL, $\lfloor H \rfloor \vdash T_0^0 \lhd S$, and $\mathsf{mbody}(S, m) = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e_m\}$. By M-OK, $\Gamma \vdash e_m : T_{n+1}$ where $\Gamma = \texttt{this} : P, \overline{x} : \overline{T}$ for some $P \in \mathsf{supers}(S)$. By Lemma 7.4, $(\lfloor H \rfloor, \Gamma) \vdash e_m : T_{n+1}$. Let $e_0 = e_m$ and $T_e^0 = T_{n+1}$, and let $e_1 = e_m\{\ell/\texttt{this}\}$ and $T_e^1 = T_{n+1}\{\ell/\texttt{this}\}$, and for $j = 1, \ldots, n$, let $e_{j+1} = e_j\{v_j/x_j\}$ and $T_e^{j+1} = T_e^j\{v_j/x_j\}$. Note $e' = e_{n+1}$.

We want to show that $\lfloor H \rfloor \vdash e_{n+1} : T_{n+1}^{n+1}$. We do this in two steps. First, we show (1) by Lemma 7.31, $\lfloor H \rfloor \vdash e_{n+1} : T_e^{n+1}$. Then we show (2) by Lemma 7.32, $\lfloor H \rfloor \vdash T_e^{n+1} \leq T_{n+1}^{n+1}$. By T-SUB, $\lfloor H \rfloor \vdash e_{n+1} : T_{n+1}^{n+1}$.

To apply the two lemmas, we need to show that the types of the actual values are subtypes of the (substituted) declared formal types; that is, when the lemmas are applied to a substitution of v for x in some Γ, if $x : T_x \in \Gamma$ and $\Gamma\{v/x\} \vdash v : T_v$, we must have $\Gamma\{v/x\} \vdash T_v \leq T_x$. Specifically, we need to show:

1. $\lfloor H \rfloor \vdash T_0^0 \leq P$
2. for $i = 1, \ldots, n$ and $j = 1, \ldots, i$, $\lfloor H \rfloor \vdash T_i^j \leq T_i^{j-1}\{v_{j-1}/x_{j-1}\}$, with $x_0 = \texttt{this}$ and $v_0 = \ell$.

We first prove (1). Since by T-CALL, $\lfloor H \rfloor \vdash \ell : T_0^0$, we need to show that $\lfloor H \rfloor \vdash T_0^0 \leq P$. We do so as follows: Since $\lfloor H \rfloor \vdash T_0^0 \lhd S$, we have $\lfloor H \rfloor \vdash T_0^0 \leq S$ by S-BOUND. Since $\vdash S \sqsubset^* P$, by Lemma 7.34, $\emptyset \vdash S \leq P$. Therefore, by S-TRANS, $\lfloor H \rfloor \vdash T_0^0 \leq P$. This proves (1).

To prove (2), we fix i and j. The proof is by structural induction on T_i^{j-1}.

Case $T_i^{j-1} = \circ$:

Then $T_i^j = T_i^{j-1}\{v_{j-1}/x_{j-1}\} = T_i^{j-1}$.

Case $T_i^{j-1} = T_i'.C$:

Follows from the induction hypothesis and S-NEST.

Case $T_i^{j-1} = p.\texttt{class}$:

We consider p by cases.

If $p = v$ or $p = x \neq x_{j-1}$, then $T_i^j = T_i^{j-1}\{v_{j-1}/x_{j-1}\} = T_i^{j-1}$.

If $p = x_{j-1}$, then $T_i^{j-1} = x_{j-1}.\texttt{class}$, and $T_i^j = T_i^{j-1}\{\!\{ \lfloor H \rfloor ;\ T_{j-1}^{j-1}/x_{j-1} \}\!\} = T_{j-1}^{j-1}$ and $T_i^{j-1}\{v_{j-1}/x_{j-1}\} = v_{j-1}.\texttt{class}$. Since $\mathsf{exact}(T_i^{j-1})$, by T-CALL we have $\mathsf{exact}(T_i^j)$, and hence $\mathsf{exact}(T_{j-1}^{j-1})$. Thus, since $\lfloor H \rfloor \vdash v_{j-1}.\texttt{class} \leq T_{j-1}^{j-1}$, we have $\lfloor H \rfloor \vdash T_{j-1}^{j-1} \leq v_{j-1}.\texttt{class}$ by Lemma 7.41.

Finally, assume $p = p_0.f$ and let $T_p = p_0.\texttt{class}\{\!\{ \lfloor H \rfloor ;\ T_{j-1}^{j-1}/x_{j-1} \}\!\}$. If T_p is not a path type, then $T_i^j = \mathsf{ftype}(\lfloor H \rfloor, T_p, f)$, which is not exact by F-OK. Hence, this case holds vacuously. Otherwise, if $T_p = p_0'.\texttt{class}$, then $T_i^j =$

$p_0'.f.\texttt{class}$. We need to show $\lfloor H \rfloor \vdash p_0'.f.\texttt{class} \leq p_0\{v_{j-1}/x_{j-1}\}.f.\texttt{class}$. Since T-CALL requires field paths be preserved and since $p_0.f \in \mathsf{paths}(T_i^{j-1})$, we must have $p' \in \mathsf{paths}(T_i^j)$ where $\lfloor H \rfloor \vdash p' = p_0\{v_{j-1}/x_{j-1}\}.f$. By S-ALIAS, $\lfloor H \rfloor \vdash p_0'.f.\texttt{class} \leq p_0\{v_{j-1}/x_{j-1}\}.f.\texttt{class}$.

Case $T_i^{j-1} = P[T_i']$:

Follows from the induction hypothesis and S-PRE-1.

Case $T_i^{j-1} = \&\overline{T}$:

Follows from the induction hypothesis and S-MEET-G.

Therefore, for all $i = 1,\ldots,n$ and for all $j = 1,\ldots,j$, we have $\lfloor H \rfloor \vdash T_i^j \leq T_i^{j-1}\{v_{j-1}/x_{j-1}\}$. This proves (2).

Now that we have proved (1) and (2), a simple application of Lemma 7.31 and Lemma 7.32 gives us $\lfloor H \rfloor \vdash e' : T_{n+1}$. Thus, by T-SUB, $\lfloor H \rfloor \vdash e_m\{\ell, \overline{v}/\texttt{this}, \overline{x}\} : T$.

Case $e = \ell.m(\overline{e})$ where some $e_i \neq v$:

Then R-CONG is the only rule that can apply. WLOG let e_i be the first e_i that is not a value. Then, $e_i, H \longrightarrow e_i', H'$. By the induction hypothesis, $\lfloor H' \rfloor \vdash e_i' : T_i^i$.

By applying Lemma 7.4 to all other subexpressions, we have for all $j \neq i$, $\lfloor H' \rfloor \vdash e_j : T_j^j$ and $\lfloor H' \rfloor \vdash \ell : T_0^0$. By Lemma 7.4, since $\lfloor H' \rfloor$ extends $\lfloor H \rfloor$, we have $\mathsf{mtype}(\lfloor H' \rfloor, T_0^0, m) = (\overline{x} : \overline{T^0}) \to T_{n+1}^0$. Also, by Lemma 7.4, for all $j = 1,\ldots,n+1$ and all $k \leq j$, $T_j^{k-1}\{\!\{\lfloor H' \rfloor;\, x_k/T_k^k\}\!\} = T_j^k$.

Since the types of all $\overline{e}$ are preserved, and since $\mathsf{prefixExact}(T_i^{j-1}, k)$ if and only if $\mathsf{prefixExact}(T_i^j, k)$ before the step, then this property also holds after the step.

Since the types of all $\overline{e}$ are preserved, $\mathsf{paths}(T_i^{j-1})$ and $\mathsf{paths}(T_i^j)$ are also preserved. Thus, we can derive by T-CALL $\lfloor H' \rfloor \vdash e' : T$.

Case $e = e_0.m(\overline{e})$ where $e_0 \neq v$:

Then R-CONG is the only rule that can apply and $e_0, H \longrightarrow e_0', H'$. By the induction hypothesis, $\lfloor H' \rfloor \vdash e_0' : T_0^0$. By Lemma 7.4, we have for all $i \geq 0$ $\lfloor H' \rfloor \vdash e_i : T_i^i$. By Lemma 7.4, since $\lfloor H' \rfloor$ extends $\lfloor H \rfloor$, we have $\mathsf{mtype}(\lfloor H' \rfloor, T_0^0, m) = (\overline{x} : \overline{T^0}) \to T_{n+1}^0$. Also, by Lemma 7.4, for all $j = 1, \ldots, n+1$ and all $k \leq j$, $T_j^{k-1}\{\!\{\lfloor H' \rfloor;\, x_k / T_k^k\}\!\} = T_j^k$.

Since the types of all $\overline{e}$ are preserved, and since $\mathsf{prefixExact}(T_i^{j-1}, k)$ if and only if $\mathsf{prefixExact}(T_i^j, k)$ before the step, this property also holds after the step.

Since the types of all $\overline{e}$ are preserved, $\mathsf{paths}(T_i^{j-1})$ and $\mathsf{paths}(T_i^j)$ are also preserved. Thus, we can derive by T-CALL $\lfloor H' \rfloor \vdash e' : T$.

Case $e = \mathtt{new}\ T(\overline{f = e})$:

Case $e = \mathtt{new}\ U(\overline{f = v})$:

Then R-NEW and R-ALLOC are the only rules that can apply. Let $\lfloor H \rfloor \vdash U \lhd S$.

- If $|\mathsf{fields}(S)| < |\overline{f}|$, then R-NEW is the only rule that can apply and $e' = \mathtt{new}\ U(\overline{f = v}, \overline{f' = e'})$ and $H = H'$ and $T = U$. By the definition of fields, for all $f_i' \in \overline{f'}$, we have $\mathsf{ftype}(\lfloor H \rfloor, U, f_i') = [\mathtt{final}]\ T_i'$ By F-OK, for all $f_i' \in \overline{f'}$, we have $\emptyset \vdash e_i' : T_i'$. By Lemma 7.4, for all i, $\lfloor H' \rfloor \vdash e_i' : T_i'$. Thus, we can derive by T-NEW we have $\lfloor H \rfloor \vdash e' : T$.
- If $|\mathsf{fields}(S)| = |\overline{f}|$, then R-ALLOC is the only rule that can apply and $e' = \ell$ and $H' = H, \ell \mapsto S\ \{\overline{f = v}\}$. Since $H'(\ell) = S\ \{\overline{f = v}\}$, $\ell : S \in \lfloor H' \rfloor$. Therefore, by F-LOC, $\lfloor H' \rfloor \vdash \ell : S$ final, and by T-FIN, $\lfloor H' \rfloor \vdash \ell : \ell.\mathtt{class}$. Since $\lfloor H' \rfloor \vdash \ell.\mathtt{class} \lhd S$, we have by S-EVAL, $\lfloor H' \rfloor \vdash \ell.\mathtt{class} \leq U$. Therefore, by S-SUB, $\lfloor H' \rfloor \vdash e' : U$. Since $\vdash e, H$, we have $\vdash H$. Thus, $H \vdash \ell' : \mathsf{loc}$ for all $\ell' \in \mathrm{dom}(H)$. Since the only new location is ℓ, we just need to show that $H' \vdash \ell : \mathsf{loc}$. By R-ALLOC, we have $H'(\ell) = S\ \{\overline{f = v}\}$. Since $\vdash e, H$, all $\mathsf{locs}(e) \subseteq \mathrm{dom}(H)$.

Therefore $\overline{v} \subseteq \mathrm{dom}(H) \cup \{\texttt{null}\}$. By T-NEW, for all i, $\mathsf{ftype}(\lfloor H \rfloor, U, f_i) = T_i$ and $\lfloor H \rfloor \vdash v_i : T_i$.

By Lemma 7.4, for all i, $\lfloor H' \rfloor \vdash v_i : T_i$. Thus, we can derive $H' \vdash \ell : \mathsf{loc}$ by H-LOC. Since $\ell \in \mathrm{dom}(H')$, and $e' = \ell$, we have $\mathsf{locs}(e') \subseteq \mathrm{dom}(H')$. Therefore, we can derive by CONFIG, $\vdash e', H'$.

Case $e = \texttt{new}\ U(\overline{f} = \overline{e})$ where some $e_i \neq v$:

Then R-CONG is the only rule that can apply. WLOG let e_i be the first e_i that is not a value. Then, $e_i, H \longrightarrow e_i', H'$. By T-NEW, $\mathsf{ftype}(\lfloor H \rfloor, U, \overline{f}) = \overline{T}$. By Lemma 7.4, since $\lfloor H' \rfloor$ extends $\lfloor H \rfloor$, we have $\mathsf{ftype}(\lfloor H' \rfloor, U, \overline{f}) = \overline{T}$. By T-NEW, $\lfloor H \rfloor \vdash e_i : T_i$. Therefore, by the induction hypothesis, $\lfloor H' \rfloor \vdash e_i' : T_i$. With this judgment and by Lemma 7.4 for all other subexpressions, we have $\lfloor H' \rfloor \vdash \overline{e} : \overline{T}$. Thus, by T-NEW, we can derive $\lfloor H' \rfloor \vdash e' : T$.

Case $e = \texttt{new}\ TE[\texttt{null}](\overline{f} = \overline{e})$:

Then R-NULL is the only rule that can apply.

Case $e = \texttt{new}\ TE[p](\overline{f} = \overline{e})$ where $p \neq \texttt{null}$ and $TE[p] \neq U$:

Then R-CONG is the only rule that can apply and $p, H \longrightarrow p', H$. By T-NEW, we have $\lfloor H \rfloor \vdash \overline{e} : \overline{T}$. By Lemma 7.4, we have $\lfloor H' \rfloor \vdash \overline{e} : \overline{T}$. Since $\lfloor H \rfloor \vdash TE[p] : \mathsf{type}$, by Lemma 7.39, $\lfloor H \rfloor \vdash TE[p'] : \mathsf{type}$. Thus, by T-NEW, we can derive $\lfloor H' \rfloor \vdash e' : TE[p']$.

Case $e = e_1;\ e_2$:

Case $e = v_1;\ e_2$:

Then R-SEQ is the only rule that can apply, and $H = H'$ and $r = e_2$. By T-SEQ, since $\lfloor H \rfloor \vdash v_1;\ e_2 : T$, we have $\lfloor H \rfloor \vdash e_2 : T$. Since $H = H'$, $\vdash e_2, H$.

Case $e = e_1;\ e_2$ where $e_1 \neq v$:

Then R-CONG is the only rule that can apply and $r = e_1'; \ e_2$. By T-SEQ, since $\lfloor H \rfloor \vdash e_1; \ e_2 : T$, we have $\lfloor H \rfloor \vdash e_1 : T_1$ and $\lfloor H \rfloor \vdash e_2 : T$. By the induction hypothesis, $\lfloor H' \rfloor \vdash e_1' : T_1$. By Lemma 7.4, $\lfloor H' \rfloor \vdash e_2 : T$. Thus we can derive, by T-SEQ, $\lfloor H' \rfloor \vdash e_1'; \ e_2 : T$. $\square$

7.10 Progress

The progress lemma states that for any well-formed configuration e, H, either e is a value or e, H steps to a new configuration r, H'.

Lemma 7.3 *(Progress)* If $\vdash e, H$ and $\lfloor H \rfloor \vdash e : T$, then either $e = v$, or there is an r and an H' such that $e, H \longrightarrow r, H'$.

Proof. The proof is by structural induction on e.

Case $e = \texttt{null}$:

Trivial since e is a value.

Case $e = \ell$:

Trivial since e is a value.

Case $e = x$:

Vacuous since x is not in $\mathrm{dom}(\lfloor H \rfloor)$.

Case $e = e_0.f$:

- If $e_0 = \texttt{null}$, then the configuration can take a step by R-NULL.
- If $e_0 = \ell$, then since $\vdash e, H, H(\ell) = S\,\{\overline{f} = \overline{v}\}$ and $f \in \overline{f}$, and so the configuration can take a step by R-GET.
- Otherwise, e can take a step by R-CONG.

Case $e = e_0.f = e_1$:

- If $e_0 = \texttt{null}$, then the configuration can take a step by R-NULL.
- If $e_0 = \ell$ and $e_1 = v$, then since $\vdash e, H$, $H(\ell) = S\ \{\overline{f} = \overline{v}\}$ and $f \in \overline{f}$, and so the configuration can take a step by R-SET.
- Otherwise, e can take a step by R-CONG.

Case $e = e_0.m(\overline{e})$:

- If $e_0 = \texttt{null}$, then the configuration can take a step by R-NULL.
- Assume $e_0 = \ell$ and $\overline{e}$ are all values. Since $\vdash e, H$, $\ell : S \in \lfloor H \rfloor$ for some S. Therefore, $\lfloor H \rfloor \vdash \ell : S$ by F-LOC and T-FIN. Since $\lfloor H \rfloor \vdash e : T$, by T-CALL we have $\mathsf{mtype}(\lfloor H \rfloor, S, m)$ is defined. Since $\emptyset \vdash S : \mathsf{type}$, $\mathsf{mtype}(\emptyset, S, m) = \mathsf{mtype}(\lfloor H \rfloor, S, m)$. Hence, by Lemma 7.35, $\mathsf{mbody}(S, m)$ is defined and, therefore, a step can be taken by R-CALL.
- Otherwise, e can take a step by R-CONG.

Case $e = \texttt{new}\ T(\overline{f} = \overline{e})$:

- If $T = U$, and $\overline{e}$ are all values, then since $\vdash e, H$, there is an S such that $\lfloor H \rfloor \vdash U \lhd S$. If $|\mathsf{fields}(S)| = |\overline{f}|$, then a step can be taken by R-ALLOC; otherwise, if $|\mathsf{fields}(S)| < |\overline{f}|$, then a step can be taken by R-NEW.
- Otherwise, e can take a step by R-CONG.

Case $e = e_1;\ e_2$:

If $e_1 = v$, a step can be taken by R-SEQ. Otherwise, e can take a step by R-CONG. □

7.11 Soundness

Soundness follows directly from the subject reduction and progress lemmas.

Theorem 7.1 *(Soundness)* If $\vdash \langle \overline{L}, e \rangle$ ok, and $\emptyset \vdash e : T$, and $e, \emptyset \longrightarrow^* r, H$ where r is in normal form, and either $r = v$ and $\lfloor H \rfloor \vdash v : T$ or $r = \mathsf{NullError}$.

Proof. Follows from Lemma 7.2 and Lemma 7.3 by induction on the number of steps. □

Chapter 8

Implementation

This chapter describes two alternative implementations of J&. Both were implemented in Java using the Polyglot framework [84]. The two compilers translate J& to Java and share parsing and semantic checking code. The compilers differ only in their translation strategies.

The main difference between the two implementations is how implicit classes are translated. A class is *implicit* if it is inherited from another namespace, but not further bound. An *explicit class* is a class declared in the source program. In the *static implicit class (SIC) translation* class declarations are generated for both explicit and implicit classes. In the *dynamic implicit class (DIC) translation* no code is generated for implicit classes; data structures for method dispatching and run-time type discrimination for these classes are constructed on demand at run time.

The static implicit class translation has better run-time performance and a smaller memory footprint, but generates code proportional to the number of classes, explicit and implicit, in the source code. The main advantage of the dynamic implicit class translation is that it generates code proportional only to the number of explicit classes in the source code. The generated code is therefore much smaller, but there is a performance cost. Neither translation duplicates code to implement inheritance.

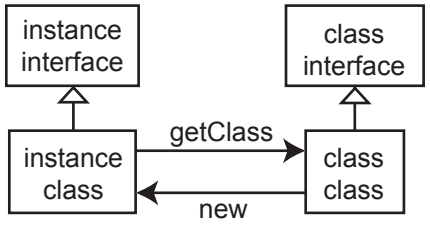

Figure 8.1: Static implicit class translation

8.1 Static implicit class translation

8.1.1 Translating classes

As illustrated in Figure 8.1, each J& class is represented by four classes: an *instance class*, an *instance interface*, a *class class*, and a *class interface*.

References to a class or interface T are translated to references to T's *instance interface*, $\mathsf{II}(T)$. The interface contains signatures for all instance methods of T as well as field getters and setters to allow access to fields from contexts where the actual run-time class is unknown. $\mathsf{II}(T)$ extends the instance interfaces of each of T's supertypes. Dependent classes, prefix types, and static virtual types are translated to the instance interface of their most precise statically known non-dependent supertype.

At run time, an object of the J& class T is represented as a single object of the *instance class*, $\mathsf{IC}(T)$, which implements the instance interface $\mathsf{II}(T)$. Instance classes are generated for implicit classes and intersection classes as well as for explicit classes. For a non-intersection class T with explicit superclass T' (which may be an intersection class), the instance class $\mathsf{IC}(T)$ extends $\mathsf{IC}(T')$. If the least common ancestor of the classes T_1 and T_2 is T_{lca}, then the instance class $\mathsf{IC}(T_1 \,\&\, T_2)$ extends $\mathsf{IC}(T_{\mathsf{lca}})$. Note that since all classes are subclasses of `Object`, the least common ancestor must exist. $\mathsf{IC}(T)$ contains, or inherits from its superclass, all fields declared in T or inherited from any of the superclasses of T.

For every J& class, there is a *class class*, $\mathsf{CC}(T)$. The class class is a singleton object instantiated at run time. The instance class $\mathsf{IC}(T)$ contains a reference to its class class $\mathsf{CC}(T)$. Static methods of T are translated to instance methods of $\mathsf{CC}(T)$ to allow static methods to be invoked on dependent types, where the actual run-time class is unknown. To support `super` calls in the presence of multiple inheritance, instance methods of T are also translated to methods of the $\mathsf{CC}(T)$. The instance class $\mathsf{IC}(T)$ contains short one-line methods to dispatch to the implementation of the method in the appropriate class class. The class class also provides functions for accessing run-time type information to implement `instanceof` and casts, for constructing instances of the class, and for accessing the class class of prefixes and members classes, including static virtual types. The code generated for expressions that dispatch on a dependent class (e.g., `new A[x.class].B()`) evaluates the dependent class's access path to locate the class class for the type. For prefix types, the class class is used to navigate to the class class of the prefix.

The class class $\mathsf{CC}(T)$ implements the *class interface* $\mathsf{CI}(T_i)$ of each of T's supertypes $T_1, \ldots, T_n$ (including T itself). The class interface contains signatures for all static methods of the class and also a factory method for each constructor.

8.1.2 Method and constructor dispatching

Method declarations are translated in two steps. First, an instance method declaration `m(...)` of a J& class T is transformed to a static method `m(`T `self, ...)`, by adding a parameter `self` that points to `this`. The `this` reference is translated to a reference to the instance class object. Then, like all static methods, the method is translated into an instance method of the class class. Method calls are dispatched to the implementation of the appropriate class class.

Normally a reference to J& type T is translated to $\mathsf{II}(S)$ for T's most precise non-dependent supertype S. However, for `this.class`, the non-dependent bound varies with the enclosing class. To implement correct method overriding, when `this.class` occurs in a method signature, it is translated to the instance interface of the class that introduced the method. This ensures that all overriding methods are translated to methods with the same signature. The method body casts the `this` reference to the instance interface of the actual enclosing class.

Constructors are similarly translated to become methods of the class class. All instance field initializations in a J& class are collected to form a field initialization method of the class class. The field initialization method is called immediately after invoking the translated superclass constructor.

8.1.3 Translating packages

To support package inheritance and composition, the representation of a package `p` includes a *package interface* and a *package class* that implements the interface. The package interface and package class are analogous to the class interface and class class. The package class provides type information about the package at run time and access to the class class or package class singletons of its members and prefixes. Both the package class and package interface of `p` are members of package `p`; packages have no instance classes or instance interfaces.

8.1.4 Java compatibility

Since J& is translated to Java, the generated code can only use single inheritance. To interact with Java code, a J& class may have only one most-specific Java superclass. The generated instance class is a subclass of this Java class. Because the instance interface is

```
package pair_and_sum extends pair & sum;

// Resolve conflicting versions of main
class Compiler {
    void main() {
        Exp e = parse();
        e.accept(new TypeChecker());
        e = e.accept(new TranslatePairs());
        e = e.accept(new TranslateSums());
        e.accept(new Emitter());
    }
}
```

Figure 8.2: Example J& source code

not a subtype of any Java class (except `Object`), when passing J& objects to a method expecting a Java class, the object must be cast from the instance interface type to the expected Java supertype.

8.2 Dynamic implicit class translation

The compiler is a 2700-LOC (lines of code, excluding blank and comment lines) extension of the Jx compiler [83], itself a 22-kLOC extension of the Polyglot base Java compiler.

8.2.1 Translating classes

Each explicit J& class is translated into four classes: an instance class, a subobject class, a class class, and a method interface. Recall the `pair&sum` compiler from Chapter 1. The composed compiler is shown again in Figure 8.2. Figure 8.3 shows a simplified fragment of the translation of the code in Figure 8.2. Several optimizations, discussed below, are not shown.

At run time, each instance of a J& class T is represented as an instance of T's instance class, $\mathsf{IC}(T)$. Each explicit class has its own instance class. The instance class of an implicit class or intersection class is the instance class of one of its explicit superclasses. An instance of $\mathsf{IC}(T)$ contains a reference to an instance of the *class class* of T, $\mathsf{CC}(T)$. The class class contains method and constructor implementations, static fields, and type information needed to implement `instanceof`, prefix types, and type selection from dependent classes. If J& were implemented natively or had virtual machine support, rather than being translated to Java, then the reference to $\mathsf{CC}(T)$ could be implemented more efficiently as part of $\mathsf{IC}(T)$'s method dispatch table. All instance classes implement the interface `JetInst`.

8.2.2 Subobject classes and field accesses

Each instance of $\mathsf{IC}(T)$ contains a *subobject* for each explicit superclass of T, including T itself if it is explicit. The subobject class for an explicit class T contains all instance fields declared in T; it does not contain inherited fields. The instance class maintains a map from each explicit superclass of T to the subobject for that superclass. The static `view` method in the subobject class implements the map lookup function for that particular subobject. If J& were implemented natively, the subobjects could be inlined into the instance class and implemented more efficiently.

To get or set a field of an object, the `view` method is used to lookup the subobject for the superclass that declared the field. The field can then be accessed directly from the subobject. The `view` method could be inlined at each field access, but this would make the generated code more difficult to read and debug.

```
package base;

// method interfaces for Exp
interface Exp$methods {
    interface Accept { JetInst accept(JetInst self, JetInst v); }
}

// class class of Exp
class Exp$class implements Exp$methods.Accept {
    JetInst accept(JetInst self, JetInst v) {
        // abstract method: cannot be called
    }
    static JetInst accept$disp(JetClass c, JetInst self, JetInst v) {
        JetClass r = ... // find the class class with the
                         // most specific implementation
        return ((Exp$methods.Accept) r).accept(self, v);
    } ...
}

// class class of Abs
class Abs$class implements Exp$methods.Accept {
    JetInst accept(JetInst self, JetInst v) {
        Abs$ext.view(self).e =
            Exp$class.accept$disp(null, Abs$ext.view(self).e, v);
        return Visitor$class.visitAbs$disp(null, v, self);
    } ...
}

// instance class of Abs
class Abs implements JetInst {
    JetSubobjectMap extMap; // subobject map
    JetClass jetGetClass() {
        // get the class class instance
    } ...
}

// subobject class of Abs
class Abs$ext {
    String x; JetInst e;
    static Abs$ext view(JetInst self) {
        // find the subobject for Abs in self.extMap
    }
}
```

Figure 8.3: Fragment of translation of code in Figure 3.6

8.2.3 Class classes and method dispatch

For each J& class, there is a singleton class class object that is instantiated when the class is first used. A class class declaration is created for each explicit J& class. For an implicit or intersection class T, $\mathsf{CC}(T)$ is the runtime system class `JetClass`; the instance of `JetClass` contains a reference to the class class object of each immediate superclass of T.

The class class provides functions for accessing run-time type information to implement `instanceof` and casts, for constructing instances of the class, and for accessing the class class object of prefix types and member types, including static virtual types. The code generated for expressions that dispatch on a dependent class (a `new x.class()` expression, for example) evaluates the dependent class's access path (i.e., `x`) and uses the method `jetGetClass()` to locate the class class object for the type.

All methods, including static methods, are translated to instance methods of the class class. This allows static methods to be invoked on dependent types, where the actual run-time class is statically unknown. Nonvirtual `super` calls are implemented by invoking the method in the appropriate class class instance.

Each method has an interface nested in the *method interface* of the J& class that first introduced the method. The class class implements the corresponding interfaces for all methods it declares or overrides. The class class of the J& class that introduces a method `m` also contains a method `m$disp`, responsible for method dispatching. The receiver and method arguments as well as a class class are passed into the dispatch method. The class class argument is used to implement nonvirtual `super` calls; for virtual calls, `null` is passed in (to prevent the receiver from being evaluated more than once) and the receiver's class class is used.

Single-method interfaces allow us to generate code only for those methods that appear in the corresponding J& class. An alternative, an interface containing all methods declared for each class, would require class classes to implement trampoline methods to dispatch methods they inherit but do not override, greatly increasing the size of the generated code.

As shown in Figure 8.3, all references to J& objects are of type `JetInst`. The translation mangles method names handle overloading. To improve readability, name mangling is not shown in Figure 8.3.

8.2.4 Allocation

A factory method in the class class is generated for each constructor in the source class. The factory method for a J& class T first creates an instance of the appropriate instance class, and then initializes the subobject map for T's explicit superclasses, including T itself. Because constructors in J& can be inherited and overridden, constructors are dispatched similarly to methods.

Initialization code in constructors and initializers are factored out into initialization methods in the class class and are invoked by the factory method. A superclass-constructor call is translated into a call to the appropriate initialization method of the superclass's class class.

8.2.5 Translating packages

To support package inheritance and composition, a package `p` is represented as a *package class*, analogous to the class class. The package class provides type information about the package at run time and access to the class class or package class instances of its member types. The package class of `p` is a member of package `p`. Since packages

cannot be instantiated and contain no methods, package classes have no analogue to instance classes, subobject classes, or method interfaces.

8.2.6 Java compatibility

To leverage existing software and libraries, J& classes can inherit from Java classes. The compiler ensures that every J& class has exactly one most specific Java superclass. When the J& class is instantiated, there is only one `super` constructor call to some constructor of this Java superclass.

In the translated code, the instance class $\mathsf{IC}(T)$ is a subclass of the most specific Java superclass of T. When assigning into a variable or parameter that expects a Java class or interface, the instance of $\mathsf{IC}(T)$ can be used directly. A cast may need to be inserted because references to $\mathsf{IC}(T)$ are of type `JetInst`, which may not be a subtype of the expected Java type; these inserted casts always succeed. The instance class also overrides methods inherited from Java superclasses to dispatch through the appropriate class class dispatch method.

8.2.7 Optimizations

One problem with the translation described above is that a single J& object is represented by multiple objects at run time: an instance class object and several subobjects. This slows down allocation and garbage collection.

A simple optimization is not to create subobjects for those J& classes that do not introduce instance fields. The instance class of explicit J& class T can inline the subobjects into $\mathsf{IC}(T)$. Thus, at run time, an instance of an explicit J& class can be represented by a single object; an instance of an implicit class or intersection class is

represented by an instance class object and subobjects for superclasses not merged into the instance class object. We expect this optimization to improve efficiency greatly.

8.3 Performance results

To compare the performance of the two translations, we implemented several microbenchmarks. The results are presented in Table 8.1. All benchmarks were run on the Java HotSpot Client VM 1.5.0 running under Mac OS X 10.4.7 on an Apple iMac G5 with a 1.8 GHz PowerPC and 1 GB RAM. The VM was run with a 512MB heap. Each benchmark consisted of executing a single operation in a loop. The loop was executed 10 million times for the allocation benchmarks and 100 million times for all other benchmarks. The Java method `System.currentTimeMillis` was used to time the runs. Each benchmark was run 15 times; outliers more than 1.5 standard deviations from the mean were discarded, resulting in 10–13 data points per benchmark. Table 8.1 shows the mean run time in nanoseconds and the standard deviation for each operation after outliers were discarded. For the non-discarded runs, the standard deviation was within 3% of the mean.

All J& benchmarks were run on the classes shown in Figure 8.4. The Java benchmarks were run on the same classes, but with `A2.B2` made explicit and with its superclasses linearized `A2.B2`, `A2.B1`, `A1.B2`, `A1.B1`.

The explicit class allocation microbenchmark allocates an instance of `A2.B1` in a loop with 10 million iterations. The implicit class allocation benchmark allocates an instance of `A2.B2`. The times include invoking the constructor and initializing the object. The object is dead immediately after allocation, so garbage collection time is also included. HotSpot uses a generational garbage collector with a copying collector for the nursery,

Table 8.1: Microbenchmark results (nanoseconds)

Benchmark	Java	Jx SIC		Jx DIC	
	Time	Time	Jx/Java	Time	Jx/Java
Allocation (explicit class)	39.4 ± 1.09	45.3 ± 0.67	1.2	1402.3 ± 2.26	35.6
Allocation (implicit class)	39.4 ± 0.97	51.6 ± 0.85	1.3	2113.0 ± 21.33	53.6
Virtual call (cache hit)	8.05 ± 0.16	46.23 ± 0.06	5.7	57.82 ± 0.15	7.2
Virtual call (cache miss)	3.83 ± 0.05			118.63 ± 2.18	31.0
Static call (explicit class)	8.08 ± 0.06	4.01 ± 0.04	0.50	14.27 ± 0.12	1.8
Static call (implicit class)	8.19 ± 0.07	4.65 ± 0.08	0.57	104.17 ± 0.19	12.7
Field write	7.64 ± 0.05	23.11 ± 0.05	3.0	19.54 ± 0.24	2.6
Field read	7.49 ± 0.07	13.85 ± 0.10	1.8	15.98 ± 0.18	2.1

```
class A1 {
    static class B1 {
        int x;
        void m() { }
        static void s() { }
    }
    static class B2 extends B1 { }
}

class A2 extends A1 {
    static class B1 {
        int y;
        void m() { }
    }
}
```

Figure 8.4: Microbenchmark classes

the youngest generation. For most collections in the loop, the collector should only need to swap the to- and from-space pointers since there are no live objects in the nursery.

The SIC constructor implementation has a 20–30% overhead versus Java due to extra method dispatching in the object initialization code. The translation of a constructor for a class `C` does not itself contain the translation of the constructor body. Instead, it first invokes an instance field initialization method in the class class of each of `C`'s superclasses, and then invokes a method of `C`'s class class containing the constructor body.

The DIC implementation has a much larger slowdown. The translated constructor allocates not only an instance of the instance class for the J& object, but also a subobject map and subobjects for each superclass. The implicit class has an even larger overhead because it has more superclasses than the explicit class: to allocate a single J& `A2.B2`, the DIC implementation allocates a subobject map (implemented as two objects) and four subobjects.

Virtual calls in both implementations are 5–7 times slower than Java. This is the overhead of dispatching to the implementation in the class class. The DIC implementation has a higher overhead because it does a method map lookup to determine the appropriate class class. The overhead is larger (31x) when there is a method map cache miss. The Java time for the cache miss benchmark differs from the cache hit benchmark because different code is run in the loop body to force a cache miss in the translated code.

Static calls, surprisingly, are faster in the SIC implementation than with Java. Static calls are translated to virtual calls on the class class. We conjecture that these calls are inlined in the SIC translation, whereas the static Java calls are not inlined.

In the DIC translation, the calls are not inlined and the extra call indirection accounts for the slowdown. Calling a static method of an implicit class has a larger slowdown

(12.7x) because of a table lookup to locate the class class, which is passed to the method to allow it to access elements of `thisclass`.

Field accesses in both translation schemes are implemented as method calls, resulting in a 2-3x slowdown.

The results show that J& can be implemented reasonably efficiently with the SIC implementation. Explicit control over memory layout would enable large performance improvements. This is discussed in more detail in Section 11.5.

Chapter 9

Experience

To demonstrate the utility of nested inheritance and nested intersection, we ported two extensible Java frameworks to J&: the Polyglot compiler framework and the FreePastry peer-to-peer networking system. Both ported frameworks support composition of extensions. For example, two compilers adding different, domain-specific features to Java can be composed to obtain a compiler for a language that supports both sets of features.

9.1 Polyglot

Following the approach described in Chapter 5, we ported the Polyglot compiler framework and several Polyglot-based extensions, all written in Java, to J&. The Polyglot base compiler is a 31.9 kLOC program that performs semantic checking on Java source code and outputs equivalent Java source code. Special design patterns make Polyglot highly extensible [83]; more than a dozen research projects have used Polyglot to implement various extensions to Java (e.g., JPred [76], JMatch [66], as well as Jx and J&). We ported six extensions ranging in size from 200 to 3000 LOC.

The extensions are summarized in Table 9.1. The parsers for the base compiler, extensions, and compositions were generated from CUP [49] or Polyglot parser genera-

Table 9.1: Ported Polyglot extensions

Name	Extends Java 1.4 ...	LOC original	LOC ported	% original
`polyglot`	with nothing	31888	27984	87.8
`param`	with infrastructure for parameterized types	513	540	105.3
`coffer`	with resource management facilities similar to Vault [32]	2965	2642	89.1
`j0`	with pedagogical features	679	436	64.2
`pao`	to treat primitives as objects	415	347	83.6
`carray`	with constant arrays	217	122	56.2
`covarRet`	to allow covariant method return types	228	214	93.9

tor (PPG) [84] grammar files. Because PPG supports only single grammar inheritance, grammars were composed manually; line counts do not include parser code.

The port of the base compiler was our first attempt to port a large program to J&, and was completed by one of the authors within a few days, excluding time to fix bugs in the J& compiler. Porting of each of the extensions took from one hour to a few days. Much of the porting effort could be automated, with most files requiring only modification of `import` statements. Porting issues are described below.

The ported base compiler is 28.0 kLOC. The code becomes shorter because it eliminates factory methods and other extension patterns which were needed to make the Java version extensible, but which are not needed in J&. We eliminated only extension patterns that were obviously unnecessary, and could remove additional code with more effort.

The number of type downcasts in each compiler extension is reduced in J&. For example, `coffer` went from 192 to 102 downcasts. The reduction is due to (1) use of dependent types, obviating the need for casts to access methods and fields introduced in extensions, and (2) removal of old extension pattern code. Receivers of calls to

Table 9.2: Polyglot composition results: lines of code

	j0	pao	carray	covarRet
coffer	63	86	34	66
j0		46	34	37
pao			34	53
carray				31

conflicting methods sometimes needed to be upcast to resolve the ambiguities; there are 19 such upcasts in the port of `coffer`.

Table 9.2 shows lines of code needed to compose each pair of extensions, producing working compilers that implemented a composed language. The `param` extension was not composed because it is an *abstract extension* containing infrastructure for parameterized types, and it does not change the language semantics; however, `coffer` extends the `param` extension.

The data show that all the compositions can be implemented with very little code; further, most added code straightforwardly resolves trivial name conflicts, such as between the methods that return the name and version of the compiler. Only three of ten compositions (`coffer&pao`, `coffer&covarRet`, and `pao&covarRet`) required resolution of nontrivial conflicts, for example, resolving conflicting code for checking method overrides. The code to resolve these conflicts is no more 10 lines in each case.

9.2 Pastry

We also ported the FreePastry peer-to-peer framework [99] version 1.2 to J& and composed a few Pastry applications. The sizes of the original and ported Pastry extensions are shown in Table 9.3. Excluding bundled applications, FreePastry is 7100 LOC.

Host nodes in Pastry exchange messages that can be handled in an application-specific manner. In FreePastry, network message dispatching is implemented with

`instanceof` statements and casts. We changed this code to use more straightforward method dispatch instead, thus making dispatch extensible and eliminating several downcasts. Messages are dispatched to several protocol-specific handlers. For example, there is a handler for the routing protocol, another for the join protocol, and others for any applications built on top of the framework. The Pastry framework allows applications to choose to use one of three different messaging layer implementations: an RMI layer, a wire layer that uses sockets or datagrams, and an in-memory layer in which nodes of the distributed system are simulated in a single JVM. Family polymorphism enforced by the J& type system statically ensures that messages associated with a given handler are not delivered to another handler and that objects associated with a given transport layer are not used by code for a different layer implementation.

Pastry implements a distributed hash table. Beehive and PC-Pastry extend Pastry with caching functionality [95]. PC-Pastry uses a simple passive caching algorithm, where lookups are cached on nodes along the route from the requesting node to a node containing a value for the key. Beehive actively replicates objects throughout the network according to their popularity. We introduced a package ("`cache`") containing functionality in common between Beehive and PC-Pastry; the CorONA RSS feed aggregation service [94] was modified to extend the `cache` package rather than Beehive.

Using nested intersection, the modified CorONA was composed first with Beehive, and then with PC-Pastry, creating two applications providing the CorONA RSS aggregation service but using different caching algorithms. Each composition of CorONA and a caching extension contains a single `main` method and some configuration constants to initialize the cache manager data structures. The CorONA–Beehive composition also overrides some CorONA message handlers to keep track of each cached object's popularity. We also implemented and composed test drivers for the CorONA extension, but line counts for these are not included since the original Java code did not include them.

Table 9.3: Ported Pastry extensions and compositions

Name	LOC original	LOC ported
Pastry	7082	7363
Beehive	3686	3634
PC-Pastry	695	630
CorONA	626	591
`cache`	N/A	140
CorONA–Beehive	N/A	68
CorONA–PC-Pastry	N/A	28

The J& code for FreePastry is 7400 LOC, 300 lines longer than the original Java code. The additional code consists primarily of interfaces introduced to implement network message dispatching. The Pastry extensions had similar message dispatching overhead; since code in common between Beehive and PC-Pastry was factored out into the `cache` extension, the size of the ported extensions is smaller. The size reduction in CorONA is partially attributable to moving code from the CorONA extension to the CorONA–Beehive composition.

9.3 Porting Java to J&

Porting Java code to J& was usually straightforward, but certain common issues are worth discussing.

9.3.1 Type names

In J&, unqualified type names are syntactic sugar for members of `this.class` or a prefix of `this.class`, e.g., `Visitor` might be sugar for `base[this.class].Visitor`. In Java, unqualified type names are sugar for fully qualified names; thus, `Visitor` would

resolve to `base.Visitor`. To take full advantage of the extensibility provided by J&, fully qualified type names sometimes must be changed to be only partially qualified.

In particular, `import` statements in most compilation units are rewritten to allow names of other classes to resolve to dependent types. For example, in Polyglot the import statement `import polyglot.ast.*;` was changed to `import ast.*;` so that imported classes resolve to classes in `polyglot[this.class].ast` rather than in `polyglot.ast`.

9.3.2 Final access paths

To make some expressions pass the type checker, it was necessary to declare some variables final so they could coerced to dependent classes. In many cases, non-final access paths used in method calls could be coerced automatically by the compiler, as described in Section 4.4. However, non-final field accesses were not coerced automatically because the field might be updated (possibly by another thread) between evaluation and method entry. The common workaround is to save non-final fields in a final local variable and then to use that variable in the call.

This issue was not as problematic as originally expected. In fact, in 30 kLOC of ported Polyglot code, only three such calls needed to be modified. In most other cases, the actual method receiver type was of the form P`[`p`.class].`Q and the formal parameter types were of the form P`[this.class].`R. Even if an actual argument were updated between its evaluation and method entry, the type system ensures its new value is a class enclosed by the same run-time namespace P`[`p`.class]` as the receiver, ensuring that the call is safe.

To illustrate why most calls do not need to be modified, consider the following typical call in the Polyglot source code:

```
this.ts.canOverride(this, mj);
```

The relevant context for the call is:

```
this    : polyglot.ext.jl.types.MethodInstance_c
this.ts : polyglot[this.class].types.TypeSystem
mj      : polyglot[this.class].types.MethodInstance
```

The signature of `canOverride` is:

```
boolean
canOverride(polyglot[this.class].types.MethodInstance mi,
            polyglot[this.class].types.MethodInstance mj)
```

Even though the receiver is a non-final field `this.ts`, and the formal parameters depend on the receiver, the call is safe because the `this.ts`'s declared type is a dependent type. The formal parameter `mj` depends only on the `polyglot` prefix of the receiver's run-time class, not on the run-time class itself. Since the `ts` field can only be updated with a value of `polyglot[this.class].types.TypeSystem`, the `polyglot` prefix of the run-time class of the field cannot be changed to another package.

9.3.3 Path aliasing

The port of Pastry and its extensions made more extensive use of field-dependent classes (e.g., `this.thePastryNode.class`) than the Polyglot port. Several casts needed to be inserted in the J& code for Pastry to allow a type dependent upon one access path to be coerced to a type dependent upon another path. Often, the two paths refer to the same object, ensuring the cast will always succeed. Implementing a simple local alias analysis should eliminate the need for many of these casts.

9.3.4 Inheriting constructors

To support allocation of instances of dependent classes, where the class being allocated is statically unknown, J& requires that a subclass implement constructors with the same

signatures as its superclasses' constructors, unless the superclass constructor is declared `nonvirtual`. When porting the base compiler to J&, 36 constructors out of 278 were declared `nonvirtual`.

Chapter 10

Related Work

Many language features and design patterns have been proposed to enable more effective code reuse. As discussed in Chapter 1, neither of the two programming models described by Reynolds [96] supports scalable, orthogonal extensions with both data types and new operations: Data-oriented programming typical of traditional object-oriented programming languages permits extension with new data types, but not scalable extension with new operations; operation-directed programming as used in functional languages permits scalable extension with new operations, but not with new data types.

10.1 The expression problem

Much of the recent work on supported extensibility in programming languages was prompted by Phil Wadler's *expression problem* [117]. The problem is to extend a system with both new data types and new operations in a statically type safe language that supports separate compilation. Wadler originally suggested that Java extended with parameterized types [15] was expressive enough to solve the expression problem, but soon realized this solution was insufficient. Subsequent papers proposed various

solutions. The expression problem emphasizes the challenge of supporting orthogonal extension in a type safe, modular way.

10.2 Mixins

One approach to enhancing the scalability of class-based inheritance is mixins [14, 40]. A mixin, or *abstract subclass*, is a class with an unspecified or parameterized superclass. By instantiating on multiple different superclasses, a mixin can provide uniform extension, adding new fields or methods, to a large number of classes.

Mixins can be simulated using explicit multiple inheritance. Because mixins themselves provide a form of multiple inheritance, instantiating a mixin can introduce name conflicts.

J& provides additional mixin-like functionality through virtual superclasses [34]. Additionally, nested inheritance allows the implicit subclasses of the new base class to be instantiated without writing any additional code. Mixins have no analogous mechanism.

Mixins are composed linearly; that is, an instantiated mixin's superclass is the class (possibly an instantiated mixin itself) on which it was instantiated. An instantiated mixin may not be able to access a member of a given superclass because the member is overridden by another mixin. Explicit multiple inheritance such as in J& imposes no ordering on composition of superclasses.

Mixins originated as a coding convention in the Common Lisp Object System (CLOS) [33]. The CLOS implementation of multiple inheritance uses a linearization algorithm that violated the principle of encapsulation [105]. One of the earliest implementations of mixins was in the language Jigsaw [14, 13]. Recent work has extended Java with mixin functionality [71, 4, 2].

A design for mixins in Java appeared in the language MixedJava [40]. MixedJava replaces classes with mixins, allows mixins to both extend and implement interfaces, and allows composition of mixins. Extension of an interface allows the mixin to use the `super` keyword to access members of its abstract superclass. The method name conflict problem is solved by maintaining a run-time `view` of the object that is a tail of the full chain of mixins that defines the object. The view is used to select which method to dispatch. Type soundness is proved for MixedJava, but no implementation was produced. The authors speculate that an implementation would require double-wide references for objects, one for the object pointer and one for the run-time view.

Jam [4] is an extension of Java with mixins. It is implemented as source-to-source translation to Java 1.0. Mixins can implement interfaces, but do not extend any types, nor can they be composed. Jam allows mixins to require that their abstract superclass contain certain members, thus making it possible to refer to the superclass member from within the mixin. Jam resolves the name conflict problem by always overriding, even when the conflict is unexpected, thus introducing the potential for the generated Java code to be illegal. Jam uses a heterogeneous translation [86] that produces a new Java class for each unique mixin instantiation. This can cause a large increase in the size of the generated class files, similar to the code bloat problem with C++ templates.

MixGen [2] extends the Java type system with mixins. In MixGen, mixins are implemented as parameterized superclasses, similar to the example code in Figure 10.1. Because nested inheritance has no type parametricity, it cannot provide a mixin that can be applied to many different, unrelated classes.

The treatment of the `super` keyword in J& is reminiscent of (though much simpler than) the CLOS [33] algorithm for linearizing superclasses in the presence of multiple inheritance.

```
class M<T> extends T {
    class A extends T.A { ... }
    class B extends T.B { ... }
}

class C {
    class A { ... }
    class B { ... }
}
```

Figure 10.1: Mixin layers example

A problem with mixins is that instantiating multiple superclasses on a mixin is not scalable and can quickly become cumbersome [113, 114]. Mixin layers [102, 103] are a design pattern that address this scalability problem by using *nested mixins*, that is, mixins nested within a mixin. The pattern is illustrated using MixGen-like syntax. in Figure 10.1, in which `M` is a mixin with superclass parameter `T`, containing nested mixins `A` and `B`, each of which extends a nested class of `T`. Instantiating the outer mixin `M` with the superclass `C` simultaneously instantiates all enclosed mixins on nested classes of `C`; that is, `M<C>.A` extends `C.A`, and `M<C>.B` extends `C.B`. Nevertheless, while mixin layers do address the scalability issue, they sacrifice separate compilation and are not modular. The semantics of layer composition is still an open problem.

10.3 Open classes

The language MultiJava [28] provides *open classes*. An open class is a class to which new methods can be added without needing to edit the class directly or recompile code that depends on the class. Classes that inherit from the augmented class inherit the new methods. Open classes thus provide a mechanism for scalable extension with new operations. Nested inheritance provides similar functionality through class overriding in

an extended container. However, open classes do not enable scalable extension with new state: new fields cannot be added to an open class in the same way as new methods can.

Open classes are modular: code that referenced the unmodified class does not need to be recompiled after the changes. However, since open classes *modify* existing class hierarchies, extension of open classes is destructive. The original behavior of an open class to clients of the augmented class. In contrast, nested inheritance creates a new class hierarchy by extending the container of the classes in the hierarchy, permitting use of the original hierarchy in conjunction with the new one.

Nested inheritance provides additional extensibility that open classes do not, such as the "virtual" behavior of constructors, and the ability to extend an existing class with new fields that are automatically inherited by its subclasses.

Similar to open classes, *expanders* [118] are a mechanism for extending existing classes. They address some of the limitations of open classes by enabling classes to be updated not only with new methods, but also with new fields and superinterfaces. Like open classes, expanders do not change the behavior of existing clients of the classes being extended.

Existing classes are extended with new state using wrapper objects. One limitation of this approach is that object identity is not preserved, which may cause run-time type checks to return incorrect results.

10.4 Virtual classes

Virtual classes [68, 69, 38]. are a language-based extensibility mechanism that can provide functionality similar to open classes for both methods and fields. They were originally introduced in the language BETA [68] as a mechanism for supporting genericity. Virtual classes are similar to nested classes in J& in that they can be further bound in

a subclass; that is, a subclass can refine a virtual class inherited from its superclass by extending it with new members.

Virtual classes in BETA are not statically type safe. BETA allows covariant method parameter types, which can lead to an unsound type system. BETA performs run-time type checks on method entry to avoid program crashes and permit the program to continue running after it has recovered from the error. Recent work on type-safe variants of virtual classes has limited method parameter types to be invariant [111] or uses *self types* [20], discussed below in Section 10.12.

Erik Ernst's generalized BETA (gbeta) language [34, 35] uses path-dependent types, similar to dependent classes in J&, to ensure static type safety. Type-safe virtual classes using path-dependent types were formalized by Ernst et al. in the *vc* calculus [38].

A virtual class is nested within an object, the *enclosing instance*: given an expression *e* of an object type, *e*.*C* is a virtual class nested within *e*. The implementation of *e*.*C* is determined at run time from the value of *e*. In contrast, nested classes in J& are nested within their enclosing class. Late binding of types is achieved by using dependent classes: the implementation of *e*.`class`.*C* is determined at run time from the value of *e*. Each virtual class may only have one enclosing instance. For this reason, a virtual class can extend only other classes nested within the same object; it may not extend a more deeply nested virtual class. This can limit the ability to extend components of a larger system. J& does not have this limitation. Because it is unique, the enclosing instance of a virtual class can be referred to unambiguously with an `out` path: `this.out` is the enclosing instance of `this`'s class. J& uses prefix types to refer to enclosing classes of dependent classes.

Recent implementations of virtual classes such as gbeta [34] support scalable, orthogonal extension using *virtual superclasses* [34], subclassing another virtual class nested within the same object. As illustrated in Figure 10.2, virtual superclasses provide

```
class C {
    class A { ... }
    class B extends this.A { ... }
}

class D extends C {
    class A { int f; ... }
    // this.B inherits this.A's f field
}
```

Figure 10.2: Virtual superclasses example

open-class-like extensibility: when new members are added to a further-bound virtual superclass, its inherited subclasses inherit the new members.

The enclosing instance contains a hierarchy of classes that can be refined by subclassing the containing object's class [37]. When the enclosing instance's class is extended via inheritance, the derived namespace replicates the class hierarchy of the original namespace, forming a *higher-order hierarchy* [37]. Unlike in J&, because virtual classes are contained in an object rather than in a class, there is no subtyping relationship between classes in the original hierarchy and further bound classes in the derived hierarchy. There is an induced subclass relationship, however.

Virtual classes can also be multiply inherited [34, 35]; however, all superclasses of a given class must be contained within the same object: a virtual class $e.C$ can only extend other virtual classes in e. This restriction limits the compositional power of virtual classes. As in J&, commonly named virtual classes inherited into a class are themselves composed [35]. However, multiple inheritance is limited to other classes nested within the same enclosing instance.

Unlike in J&, in the *vc* calculus, new fields cannot be added into a further bound class, limiting the ability to add state into a non-leaf class of a hierarchy.

In gbeta, each object defines a *family* of classes: the collection of mutually dependent virtual classes nested within it. Virtual classes in gbeta support *family polymor-*

phism [36]: two virtual classes enclosed by distinct objects cannot be statically confused. When a containing namespace is extended, family polymorphism ensures the static type safety of the classes in the derived family by preventing it from treating classes belonging to the base family as if they belonged to the extension. Because nested classes in J& are attributes of their enclosing class, rather than an enclosing object, J& supports nested inheritance supports what Clarke et al. [27] call *class-based family polymorphism*. With virtual classes, all members of the family are named from a single "family object", which must be accessible throughout the system. In contrast, with class-based family polymorphism, each dependent class defines a family. By using prefix types, any member of the family can be used to name the family.

Delegation layers [91] use virtual classes and delegation to provide family polymorphism, solving many of the problems of mixin layers. With normal inheritance and virtual classes, when a method is not implemented by a class, the call is dispatched to the superclass. With delegation, the superclass view of an object may be implemented by another *object*. Methods are dispatched through a chain of delegate objects rather than through the class hierarchy. Delegation layers provide much of the same power as nested inheritance. Since delegates are associated with objects at run-time rather than at compile-time, delegation allows objects to be composed more flexibly than with mixins or with nested inheritance. No formal semantics has been given for delegation layers.

10.5 Virtual types

Virtual types, also introduced in BETA [68], are similar to virtual classes. A virtual type is a type binding nested within an enclosing instance. Virtual types are illustrated in Figure 10.3. In the figure, the class `List` introduces a virtual type `T`. `T` is not bound to a particular type, but is declared to extend `Object`. Subclasses of `List` may further bind

```
abstract class List {
    type T extends Object;
    void add(T x) { ... }
    ...
}

class IntList extends List {
    final type T = Integer;
}
```

Figure 10.3: Virtual types

`T` to a subclass of `T`'s bounding type, `Object`. The subclass `IntList` *final binds* `T` to the class `Integer`. Final binding prevents subclasses of `IntList` from further binding `T`. Thus, if *e* is an `IntList` (or a subclass), *e*.`T` and `Integer` are aliases. Because `T` is bound to `Integer` in `IntList`, only instances of `Integer` can be passed to `IntList`'s `add` method. In contrast, a virtual class may only be declared a subtype of another type (via the class's `extends` clause), not *equal* to another type.

As can be seen from the example, virtual types may be used to provide genericity. Indeed, Thorup [110] proposed extending Java with virtual types, with final binding, as a genericity mechanism.

To ensure inheritance relationships can be determined statically, a virtual type in BETA may be inherited from only if it is final bound. J& does not permit inheritance from dependent classes, ensuring a static inheritance hierarchy.

Igarashi and Pierce [52] model the semantics of virtual types and several variants in a typed lambda-calculus with subtyping and dependent types.

10.6 Tribe

Tribe [27] is another language that provides a variant of virtual classes. By treating a final access path *p* as a type, nested classes in Tribe can be considered attributes of an

enclosing class as in J& or as attributes of an enclosing instance as in BETA and its derivatives. This flexibility allows a further bound class to be a subtype of the class it overrides, like in J& but unlike with virtual classes. Tribe also supports multiple inheritance. However, superclasses of a Tribe class must be nested within the same enclosing class, limiting extensibility. This restriction allows the enclosing type to be named using an `owner` attribute: T.`owner` is the enclosing class of T.

10.7 Concord

Concord [55] also provides a type-safe variant of virtual classes. In Concord, mutually dependent classes are organized into *groups*, which can be extended via inheritance. References to other classes within a group are made using types dependent on the current group, `MyGrp`, similarly to how prefix types are used in J&. Relative supertype declarations provide functionality similar to virtual superclasses. Groups in Concord cannot be nested, nor can groups be multiply inherited.

10.8 Nested types

Nested classes originated with Simula [31], and have been implemented in many subsequent object-oriented programming languages such as Java [45] or C++ [107].

Igarashi and Pierce [54] present a formalization of Java's inner classes, using Featherweight Java [53]. An instance of a Java inner class holds a reference to its enclosing instance. Igarashi and Pierce present a translation that transforms inner classes into top-level classes. J& implements inner classes using a similar translation.

Odersky and Zenger [87] propose *nested types*, which combine the abstraction properties of ML-style modules with support, via encoding, for object-oriented constructs

like virtual types, self types, and covariant families of classes. Nested types provide *witness types*, similar to path-dependent types in Scala and dependent classes in J&.

10.9 Multiple inheritance

J& provides multiple inheritance through nested intersection. Intersection types were introduced by Reynolds in the language Forsythe [97] and were used by Compagnoni and Pierce to model multiple inheritance [29]. Cardelli [24] presents a formal semantics of multiple inheritance.

The distinction between name conflicts among methods introduced in a common base class and among methods introduced independently with possibly different semantics was made as early as 1982 by Borning and Ingalls [10]. Many languages, such as C++ [107] and Self [25], treat all name conflicts as ambiguities to be resolved by the caller. Jigsaw [13] provides merging operators that require the programmer to specify manually how to resolve name conflicts. Jigsaw, as well as other languages [72, 100], also allows methods to be renamed or aliased to resolve conflicts.

10.10 Traits

Traits [100] are collections of abstract and non-abstract methods that may be composed with state to form classes. Since traits do not have fields, many of the issues introduced by multiple inheritance (for example, whether to duplicate code inherited through more than one base trait) are avoided. The code reuse provided by traits is largely orthogonal to that provided by nested inheritance and could be integrated into J&.

Traits are parameterized on other methods, which must be provided to create a class using the trait. Using a trait-like mechanism to compose large collections of mutually-dependent classes or traits could lead to parameter explosion.

Unlike with mixins, the order of trait composition does not matter.

10.11 Scala

Scala [88] is another language that supports scalable extensibility and family polymorphism through a statically safe virtual type mechanism based on path-dependent types. However, Scala's path-dependent type *p*.`type` is a singleton type containing only the value named by access path *p*; in J&, *p*.`class` is not a singleton. For instance, `new x.class(...)` creates a new object of type `x.class` distinct from the object referred to by `x`. This difference gives J& more flexibility, while preserving type soundness. Scala provides virtual types, but not virtual classes. It has no analogue to prefix types, nor does it provide virtual superclasses, limiting the scalability of its extension mechanisms. Scala supports composition using traits. Since traits do not have fields, new state cannot be easily added into an existing class hierarchy.

10.12 Self types and matching

Bruce et al. [21, 18] introduce *matching* as an alternative to subtyping, with a *self type*, or `MyType`, representing the type of the method's receiver. The dependent class `this.class` is similar but represents only the class referred to by `this` and not its subclasses. Type systems with `MyType` decouple subtyping and subclassing; in PolyTOIL and LOOM, a subclass *matches* its base class but is not a subtype. With nested inheritance, subclasses are subtypes. Bruce and Vanderwaart [22, 19] propose

type groups as a means to aggregate and extend mutually dependent classes, similarly to Concord's group construct, but using matching rather than subtyping.

With `MyType`, an instance of a subtype of the `MyType` may be assigned to a variable of the `MyType`. A variable of type *p*.`class` may only contain instances of *exactly p*'s run-time class since no other type is a subtype of *p*.`class`, In addition, the use of dependent types is more flexible than `MyType` because it allows `this.class` to escape the body of its class by assigning `this.class` into another variable.

Bruce et al. [20] use matching to provide a statically safe virtual type mechanism.

10.13 Aspect-oriented programming

An *aspect* [60] is a unit of functionality that cuts across modular boundaries. *Aspect weaving* applies an aspect to a set of classes to produce executable code. Aspects modify existing class hierarchies, whereas nested inheritance creates a new class hierarchy, allowing the new hierarchy to be used alongside the old.

Caesar [73] is an aspect-oriented language that also supports family polymorphism, permitting application of aspects to mutually recursive nested types.

In AspectJ [59] weaving can be performed at compile time either on the source code or on binaries, or at load time. With compile-time weaving, aspects are applied to a whole program, outputting a new program. Separate compilation is not supported. With load-time weaving, aspects are applied when classes are loaded into the virtual machine. Since errors may not be detected until a class is loaded, the program may fail at run-time.

Nested inheritance provides limited aspect-like extensibility: an extension of an enclosing class or package may implement functionality that cuts across the class boundaries of the nested classes. However, nested inheritance provides only *static cross-cutting*, the ability to modify the static nature of the program, as opposed to *dynamic*

cross-cutting, the ability to change the way a program executes. Much of the recent work on aspect-oriented programming appears to focus on dynamic cross-cutting.

10.14 Program composition

It is undecidable to determine precisely whether two programs, including compilers, have conflicting semantics that prevent their composition [48].

Several conservative algorithms based on program slicing [120] have been proposed for integrating programs [48, 9, 70]. These algorithms merges two programs A and B into a program C that produces the same outputs as A and B for identical inputs. The two programs *interfere* and cannot be integrated when A and B produce different outputs for the same input.

Interprocedural program integration [9] requires the whole programs of A and B and it is unclear whether the algorithm can scale up to large programs. A type system similar to one used for information-flow checking [116, 82] may offer a way to achieve modular program integration at the expense of additional programmer annotations.

10.15 Class hierarchy composition

Ossher and Harrison [90] propose an approach in which extensions of a class hierarchy are written in separate sparse extension hierarchies containing only new functionality. Extension hierarchies can be merged and naming conflicts detected. However, semantic incompatibilities between extension hierarchies are not detected. Unlike with nested intersection, hierarchies do not nest and there is no subtyping relationship between classes in different hierarchies.

Tarr et al. [108] define a specification language for composing class hierarchies. Rules specify how to merge "concepts" in the hierarchies. Nested intersection supports composition with a rule analogous to merging concepts by name. These ideas were implemented for Java in the specification language Hyper/J [108], which allows the programmer to specify how to merge *units*: classes, interfaces, and methods. Unlike in J&, methods can be merged by ordering one after the other, or by selecting one implementation over another. In addition, differently named units can be merged.

Snelting and Tip [104] present an algorithm for composing class hierarchies and a semantic interference criterion. If the hierarchies are *interference-free*, the composed system preserves the original behavior of classes in the hierarchies. J& reports a conflict if composed class hierarchies have a *static interference*, but makes no effort to detect dynamic interference.

10.16 Algebraic datatypes

The functional language community has also addressed the extensibility problem. Much of this work focuses on making algebraic datatypes more extensible [122, 44].

Zenger and Odersky [122] describe a mechanism for extending algebraic datatypes. To ensure exhaustive pattern matching, all functions on extensible datatypes must provide a default case. Functions are not extensible: if a new datatype variant is added that requires overriding a function case, a new function must be created and callers must be modified to invoke the new function. Object-oriented languages avoid this problem through method overriding.

Garrigue [44] describes *polymorphic variants*, which are variants defined independently of a datatype. Functions are not extensible.

Millstein, Bleckner, and Chambers [77] describe EML, an extension of ML that supports scalable, orthogonal extensibility with modular type-checking. Unlike Zenger and Odersky's language, EML provides extensible functions. Both EML and MultiJava are based on Dubious [78, 79], an object calculus with multimethods.

10.17 Jiazzi

Jiazzi [71] is a module system for Java. Programmers define components similar to units [39], a module system in which programmers explicitly specify imports and exports for each unit; a linker wires up the units to produce a closed program. Jiazzi components may be Java classes or composites of several components. Components import and export classes and packages. To support separate compilation, programmers specify package signatures, and Jiazzi generates stubs for imported components to allow classes to be compiled with a standard Java compiler. An external linker checks Java classes and packages against their signatures and performs class file symbol rewriting to update references to imported classes. Jiazzi units are expressive enough to specify mixins naturally; open classes can be simulated with a design pattern. However, Jiazzi components do not provide scalable extensibility since the programmer must specify how components are linked together: if a component is extended, linking code needs to be written for all other components in the system the original component linked with.

10.18 Classboxes

A *classbox* [8] is a module-based reuse mechanism. Classes defined in one classbox may be imported into another classbox and refined to create a subclass of the imported class. By dispatching based on a dynamically chosen classbox, names of types and

methods occurring in imported code are late bound to refined versions of those types and methods. This feature provides similar functionality to the late binding of types provided by `this`-dependent classes and prefix types in J&. As presented, classboxes are dynamically typed; it is unclear if they can be defined in a statically type-safe manner.

Since reuse is based on import of classboxes rather than inheritance, classboxes do not support multiple inheritance, but they do allow multiple imports. When two classboxes that both refine the same class are imported, the classes are not composed like in J&. Instead, one of the classes is chosen over the other.

10.19 Software components

Component systems are a popular means of code reuse. Components are self-contained abstractions intended to be reused multiple times in different contexts. Examples of component systems include Microsoft's COM [98] and .NET [93], CORBA [89], Sun Microsystems' JavaBeans [74], and IBM's System Object Model [23].

Components are often language-neutral and may be distributed. Clients are not statically linked against components; instead, a reflection API allows clients to access a component's interface. Hence, run-time errors may occur when a component's interface is not used correctly.

10.20 Macro systems and preprocessors

Another approach to enhancing extensibility is to use macro systems [119, 12, 5, 6] and preprocessors [51, 61, 109]. These systems enable programmers to extend the syntax of the programming language. Semantic checking is typically not performed on the code written in the extended syntax. A syntax-directed translation transforms the extended

source into the base language, on which further syntactic and semantic checks are performed. This limits the ability to extend the static semantics of the base language.

The Extensible Java Preprocessor Kit (EPP) [51, 50] allows type system extensions by providing an extensible type checker [50]. Type system extensions are limited to preserve separate compilation. EPP has been used to extend Java with mixins with multiple inheritance [50].

Many preprocessors such as the C preprocessor (CPP) [58], are non-hygienic: expanded macros may not parse correctly or may evaluate an actual argument more than once, often causing the repetition of side effects and leading to unexpected behavior. By contrast, macros systems such as those for Lisp [106], Dylan [101], or Scheme [57] are hygienic. Recent preprocessors and macro systems for Java such as the Extensible Java Preprocessor Kit (EPP) [51], the Java Syntactic Extender (JSE) [5], and the Java Pre-processor (JPP) [61] are also hygienic.

Programmable syntax macros [119] provide non-hygienic macros for the C language. However, the macro syntax is restricted to ensure expanded macros are syntactically correct. Metamorphic syntax macros [12] improve on programmable syntax macros to allow more expressive macros.

Maya [6] is a generalization of macro systems that uses generic functions and multimethods to allow extension of Java syntax. Semantic actions can be defined as multimethods on those generic functions. It is not clear how these systems scale to support semantic checking for large extensions to the base language.

OpenJava [109] uses a meta-object protocol (MOP) similar to Java's reflection API [75] to allow manipulation of a program's structure. OpenJava allows very limited extension of syntax, but through its MOP exposes much of the semantic structure of the program. OpenC++ [26] is similar.

The Jakarta Tools Suite (JTS) [7] is a toolkit for implementing Java preprocessors to create domain-specific languages. Extensions of a base language are encapsulated as components that define the syntax and semantics of the extension. New compiler passes are added using the data-directed approach, implementing new methods in a node definition [103]. Mixin layers are used to achieve composability in JTS [102]. JTS is concerned primarily with syntactic analysis of the extension language, not with semantic analysis. This makes JTS more like a macro system in which the macros are defined by extending the compiler rather than by declaring them in the source code.

10.21 Plugins

Several recent applications are designed to be extended via plugin architectures. Plugins are linked into the application at load time.

Much of the Firefox web browser [42] is implemented in JavaScript and XUL, an XML-based UI description language. Browser extensions can be written in the same languages to extend or override the browser's behavior. Because JavaScript is an interpreted language, Firefox extensions may fail to load or run correctly. Extensions are not isolated from each other or from the base system. Interfering extensions can cause each other or the browser itself to behave incorrectly or to crash. Firefox supports binary plugins.

Eclipse [41, 47] is an extensible platform for building development environments. The system consists of a set of core plugins to bootstrap the system; most application functionality is provided by extension plugins. Plugins in Eclipse are dynamically linked components [11]. A plugin can provide *extension points* to enable other plugins to further extend it. A *contract* specifies the programmatic interface between the host plugin and the extender plugin at each extension point.

10.22 Extended visitor patterns

Data-oriented programming is the natural programming model in object-oriented languages. The Visitor pattern [43] is used to provide an operation-directed programming model, making the addition of new operations easy, but limiting the ability to add new data types. The Visitor pattern therefore enables a different kind of extensibility, but does not enable scalable, orthogonal extension. The original Visitor design pattern has led to many refinements.

The *Extensible Visitor* [62] pattern is a composite design pattern that uses the Visitor and the Factory Method patterns [43] to enable extension of both nodes and passes. A problem with the Visitor pattern is that it cannot explicitly use constructors (e.g., via a `new` expression) to create new visitors because the constructor call ties the new visitor to a particular implementation that may not be aware of new nodes added by an extension. Extensible Visitors solve this problem using factory methods. Nested inheritance deals with this problem through virtual dispatch of constructor calls. The pattern does not address extension of existing visitors with a callback for new nodes.

Staggered Visitors [115] use multiple inheritance to extend visitors with support for new nodes. When a new node class is added, a callback for the new node is added to existing visitors by writing a class containing the callback and then multiply inheriting that class with each existing visitor class. The amount of code that needs to be written is therefore proportional to the number of existing visitor classes.

Walkabouts [92] are a generalization of the Visitor pattern that uses reflection [75] to find all objects in the data structure being traversed. This solves the extensibility problem, but incurs a large runtime penalty due to the use of reflection.

Chapter 11

Future Directions

This chapter discusses future directions for research on scalable extension and composition.

11.1 Unanticipated reuse

Often existing code provides needed functionality, but because of how the code is organized, that functionality may not be readily available for reuse. Consequently, developers are forced to either copy code from the base system or refactor the base system to enable reuse. More effective mechanisms for unanticipated code reuse are needed.

11.1.1 Restructuring

In nested inheritance, the interaction between inheritance and containment can sometimes prevent effective code reuse. To preserve mutual dependencies among a set of classes as they are simultaneously extended, those classes must be nested within the same namespace. For example, for the Visitor design pattern to be type safe in an ex-

```
package visit;

class Visitor {
    void visitVar(ast.Var v) { ... }
    ...
}

class TypeChecker extends Visitor { ...  }
```

```
package ast;

class Node {
    void accept(visit.Visitor v) { ... }
}

class Var extends Node { ... }
```

Figure 11.1: A non-extensible compiler

tended compiler, both the visitor classes and the AST node classes must be contained within a common namespace that is inherited by the extended compiler.

To illustrate the problem, suppose a compiler is implemented as separate `visit` and `ast` packages, as shown in Figure 11.1. If the compiler is extended by adding a new AST node class in a derived package of `ast`, for example `pair_ast.Pair` in Figure 11.2, then a new callback method (`visitPair`) must be added to the `Visitor` class. To ensure the new method is inherited by subclasses of `Visitor` such as `TypeChecker`, the callback is added by further binding `Visitor` in a derived package of `visit` (`pair_visit`). However, `ast.Node.accept` is statically bound to the `visit.Visitor` class. Because it would be unsound, the J& type system does not allow subclasses of `ast.Node` like `pair_ast.Pair` to override `accept` to refer to the derived `pair_visit` package. Hence, the new AST class must downcast the visitor passed into its `accept` method in order to invoke the appropriate callback.

```
package pair_visit extends visit;

class Visitor {
    void visitPair(pair_ast.Pair v) { ... }
    ...
}
```

```
package pair_ast extends ast;

class Pair extends Node {
    void accept(visit.Visitor v) {
        ((pair_visit.Visitor) v).visitPair(this);
    }
}
```

Figure 11.2: Extending a non-extensible compiler

J& needs a mechanism for safely extending mutually dependent classes spanning multiple namespaces. One possible approach is to allow extensions to restructure the base system so that mutually dependent classes are nested within a common namespace that can be extended. This could be accomplished with a mechanism syntactically similar to static virtual types, as shown in Figure 11.3. The package `restructured_compiler` contains nested packages `my_ast` and `my_visit`; derived packages can further bind `my_ast` or `my_visit` or their nested classes. However, with static virtual types as currently designed, since the original `visit` and `ast` code used hard-coded names to refer to other types, those types are not late bound, but they need to be late bound to enable reuse. For example, when `Visitor` is further bound in `pair_compiler.my_visit`, the `accept` methods of `pair_compiler.my_ast.Node` continues to refer to `visit.Visitor`, not to the new version of `Visitor`.

```
package restructured_compiler;

package my_ast = ast;
package my_visit = visit;
```

```
package pair_compiler extends restructured_compiler;
```

```
package pair_compiler.my_visit;

// further bind restructured_compiler.my_visit.Visitor
class Visitor {
    void visitPair(my_ast.Pair p) { ... }
}
```

```
package pair_compiler.my_ast;

class Pair extends Node {
    void accept(my_visit.Visitor v) {
        v.visitPair(this);
    }
}
```

Figure 11.3: A restructured compiler

11.1.2 Reparameterization

As another example of unanticipated reuse, suppose a data structure is needed and a library provides code for a similar, but not identical, data structure. For example, a system might require a list of integers, and the library contains code for a list of strings. One could copy and adapt the existing code, but a better alternative would be to reuse it by providing code to adapt it to its new use. In this example, the extension needs to be able to identify string-specific code, including code dependent on string-specific code, and replace it with integer-specific code.

One approach might be for the extension to abstract the original code by adding explicit type parameters. In the example, the code for the list of strings is abstracted to parameterize it on the list element type, thus turning a list of strings into a list of T. The parameter can then be instantiated on a new type, integers in this case.

11.2 Multiple families

Nested inheritance supports family polymorphism, which ensures that classes indexed by different final access paths cannot be confused. This is essential for ensuring the J& type system is sound. However, as demonstrated with the extensible rewriting pattern described in Section 5.3, writing code that operates over more than one family can be onerous. When translating an AST from an extended language to its base language, all nodes in the extended AST must be copied to create a base language AST, even though the base AST classes are superclasses of the extended AST classes. It should be possible to simply implicitly coerce the extended nodes to base nodes without copying. The difficulty is that to provide the desired behavior as well as to ensure soundness, extended code cannot be executed for the coerced nodes. Traditional object-oriented method dispatch may violate soundness in this case.

One possible approach is to introduce multimethods to J&. This would allow the family (e.g., the compiler's object language) to be decoupled from the class within the family (e.g., an AST node or visitor class). Rather than reconstructing the entire AST in the target language, the same AST could be used, but methods dispatched on an object representing the source language as well as the AST node. Multimethods have the added benefit of obviating the need for the visitor design pattern.

11.3 Composition

The composition of Polyglot compiler extensions in described in Section 9.1 was relatively easy because the language extensions did not have many semantic conflicts. J& detects only naming conflicts between composed class hierarchies. However, semantic conflicts between classes may occur even in the absence of naming conflicts. As a simple example, the J& code in Figure 11.4 has a semantic conflict in the class `A3`. Both `A1` and `A2` expect the method `m2` to print "`1`"; however, `A3.m2` prints "`2`".

Greater programmer control over how classes and methods are composed and an analysis that detects and reports semantic conflicts would greatly help programmers compose large extensions. Several algorithms based on program slicing [120] have been proposed for integrating programs [48, 9, 70]. Because precise semantic conflict detection is undecidable, these algorithms are conservative and may therefore report false conflicts. In addition, interprocedural program integration [9] requires the whole program. It is unclear how well these algorithms perform in practice, particularly for large programs. Construction of a precise analysis that avoids most false conflicts, but that can also be implemented modularly is an open problem.

A type system might be used to make the analysis modular, but at the cost of precision. When there is a conflict, the type system would have to help the programmer

```
class A {
    int x;
    void run() {
        m1();
        m2();
    }
    void m1() { x = 0; }
    void m2() { print(x); }
}

class A1 extends A {
    void m1() { x = 1; }
}

class A2 extends A {
    void m2() { print(x+1); }
}

class A3 extends A1 & A2 {
    // no name conflict, but m2 prints 2,
    // which is not expected by either A1 or A2
}
```

Figure 11.4: A semantic conflict

identify code that depends on the conflict so it can be reconciled. The type system would also have to summarize the effects of methods precisely enough so that when a method is overridden without changing the effects, it is guaranteed that the behavior of the method's callers does not change.

In the example of Figure 11.4, it is the effects of methods `m1` and `m2` on the variable `x` that cause the semantic conflict. One possible effect system that could be used to detect semantic conflicts is to consider each access to label each method with the heap locations accessed by its statements. Thus, `m1` and `run` are labeled with the effect `writes x`, and `m2` and `run` are labeled with `reads x`, as shown in Figure 11.5. Detecting a possible semantic conflict between two classes can be performed by checking if the two classes both define methods with conflicting effects. In this case, `A1` and `A2` conflict because they both define methods that access `x`.

Since a method might access a variable only on certain paths through the method, or the conflicting methods might be called only in contexts where there is no dependency between them, the proposed effect system presented here is therefore too imprecise to be used in a practical setting. However, it does suggest an approach to detecting semantic conflicts. Tracking dependencies between methods might be achieved using a type system based on information flow [116, 82].

One problem with effects systems is that a subtype can refine the effects of a method only by removing effects: effects must be contravariant with respect to the subtyping relationship. However, in practice, subtypes often need to add effects, particularly to access new fields introduced by the subtype. Nested inheritance suggests a solution: dependent types. Dependent classes are used to allow method parameter types to be refined covariantly in subclasses rather than contravariantly. A similar solution, *virtual effects*, could be used to allow effects to be refined covariantly. A virtual effect is an effect contained in a class that can be further bound by subclasses, just as a nested class

```
class A {
    int x;
    void run() reads x, writes x {
        m1();
        m2();
    }
    void m1() writes x { x = 0; }
    void m2() reads x { print(x); }
}

class A1 extends A {
    void m1() writes x { x = 1; }
}

class A2 extends A {
    void m2() reads x { print(x+1); }
}

class A3 extends A1 & A2 {
    // A1 and A2 conflict since they both
    // define methods that access x
}
```

Figure 11.5: Semantic conflict with effects

```
class A {
    int x;
    effect E = writes x;
    void m() : this.class.E {
        x = 1;
    }
}

class B extends A {
    int y;
    effect E = super.class.E, writes y;
    void m() : this.class.E {
        x = 1;
        y = 2;
    }
}
```

Figure 11.6: Virtual effects example

in J& or a virtual type can be further bound. Figure 11.6 shows two classes with virtual effects. Class `B` refines the effect `E` declared in `A` to write the new field `y`. Intersecting two classes T_1 and T_2 unions the corresponding effects $T_1.E$ and $T_2.E$.

11.4 Programming language composition

As described in Chapter 5 and demonstrated with the port of Polyglot described in Chapter 9, J& can be used to compose compilers. The composed compiler implements a composition of the languages implemented by the constituent compilers. It should be possible to derive formal semantics for the composed language from the formal semantics of the constituent languages.

Several frameworks have been developed for constructing modular formal systems including type systems and structural operational semantics (e.g., [63, 81, 16, 17]). One of these frameworks might be adapted for describing the semantics of a programming

language constructed via nested intersection. The system should be capable of detecting semantic conflicts between the two languages. A useful property the framework should guarantee is that the composition of two sound programming languages is also sound. Work in this area may also help with the design of an analysis for statically detecting semantic conflicts in composed programs, as described in the previous section.

11.5 A J& virtual machine

The performance of the J& implementations described in Chapter 8 could be improved with greater control over the memory layout of objects. This can be achieved by implementing a J& virtual machine. One approach to implementing a J& VM is to port to J& an existing Java VM written in Java and then to extend the ported VM using nested inheritance to support J&-specific bytecode. A good candidate for a Java VM to port is Jikes RVM [3]. A translation of J& to J&-specific bytecode must be implemented.

A virtual machine written in J& would also offer another platform for investigating the effectiveness and usability of nested intersection as a mechanism for scalable extension and composition.

Rather than implementing a J& virtual machine, another approach to improving performance is to use bytecode rewriting. As with the virtual machine approach, J& is translated to J&-specific bytecode. But, instead of having the VM interpret the extended bytecode, a class loader could be installed to translate the J& bytecode to Java bytecode as it is loaded. With this approach, the compiler need only generate code for explicit classes, as with the dynamic implicit class translation. Bytecode for implicit classes could be generated at run-time, thus achieving performance comparable, if not better, than the static implicit class translation.

Chapter 12

Conclusions

This thesis describes mechanisms for scalably extending code with new data types and new operations. The design pattern approach used in Polyglot is effective, but requires care to use correctly, does not ensure type safety of extensions, and does not support composition of extensions.

Nested intersection is a more effective language mechanism for extending and composing large bodies of software. Extension and composition are scalable because new code needs to be written only to implement new functionality or to resolve conflicts between composed classes and packages. Novel features like prefix types and static virtual types offer important expressive power.

Nested intersection has been implemented in an extension of Java called J&. We have described the static and dynamic semantics of J& and presented a formal semantics and proof of soundness for a core calculus with nested intersection.

Using J&, we implemented a compiler framework for Java, and showed that different domain-specific compiler extensions can easily be composed, resulting in a way to construct compilers by choosing from available language implementation components. We demonstrated the utility of nested intersection outside the compiler domain by porting the FreePastry peer-to-peer system to J&. The effort required to port Java

programs to J& is not large. Ported programs were smaller, required fewer type casts, and supported more extensibility and composability.

Nested intersection is a powerful and convenient mechanism for building highly extensible software. We expect it to be useful for a wide variety of applications.

BIBLIOGRAPHY

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *The Structure and Interpretation of Computer Programs*. MIT Press, second edition, 1996.

[2] Eric Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In *Proc. OOPSLA '03*, pages 96–114, Anaheim, CA, October 2003.

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, , and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[4] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam: A smooth extension of Java with mixins. In *Proc. ECOOP '00*, LNCS 1850, pages 154–178, Cannes, France, 2000.

[5] Jonathan Bachrach and Keith Playford. The Java syntactic extender (JSE). In *Proceedings of the 2001 Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '01)*, pages 31–42, Tampa, FL, USA, 2001.

[6] Jason Baker and Wilson C. Hsieh. Maya: Multiple-dispatch syntax extension in Java. In *Proc. of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation (PLDI)*, pages 270–281, Berlin, Germany, June 2002.

[7] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–53, Victoria, BC, Canada, 1998. IEEE.

[8] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proc. OOPSLA '05*, pages 177–189, San Diego, CA, USA, October 2005.

[9] David Binkley, Susan Horwitz, and Thomas Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(1):3–35, January 1995.

[10] Alan Borning and Daniel Ingalls. Multiple inheritance in Smalltalk-80. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 234–237, August 1982.

[11] Azad Boulour. Notes on the Eclipse plug-in architecture. http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html, July 2003.

[12] Claus Brabrand and Michael I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, pages 31–40, Portland, Oregon, 2002.

[13] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.

[14] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proc. OOPSLA '90*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[15] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. OOPSLA '98*, Vancouver, Canada, October 1998.

[16] Christiano Braga and José Meseguer. Modular rewriting semantics of programming languages. In *Proceedings of the 10th International Conference on Algebraic Methodology and Software Technology (AMAST)*, volume 3116 of *Lecture Notes in Computer Science*, pages 364–378, July 2004.

[17] Christiano Braga and Alberto Verdejo. Modular SOS with strategies. In *Proceedings of Structural Operational Semantics (SOS)*, August 2006.

[18] Kim B. Bruce. Safe static type checking with systems of mutually recursive classes and inheritance. Technical report, Williams College, 1997. http://cs.williams.edu/~kim/ftp/RecJava.ps.gz.

[19] Kim B. Bruce. Some challenging typing issues in object-oriented languages. *Electronic Notes in Theoretical Computer Science*, 82(8):1–29, October 2003.

[20] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *European Conference on Object-Oriented Programming (ECOOP)*, number 1445 in Lecture Notes in Computer Science, pages 523–549, Brussels, Belgium, July 1998. Springer-Verlag.

[21] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *European Conference on Object-*

Oriented Programming (ECOOP), number 952 in Lecture Notes in Computer Science, pages 27–51. Springer-Verlag, 1995.

[22] Kim B. Bruce and Joseph C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Mathematical Foundations of Programming Semantics (MFPS), Fifteenth Conference*, volume 20 of *Electronic Notes in Theoretical Computer Science*, pages 50–75, New Orleans, Louisiana, April 1999.

[23] F. R. Campognoni. IBM's system object model. *Dr. Dobb's*, pages 24–28, 1994. Winter 1994/1995.

[24] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Also in *Readings in Object-Oriented Database Systems,* S. Zdonik and D. Maier, eds., Morgan Kaufmann, 1990.

[25] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in Self. *Lisp and Symbolic Computation*, 4(3):207–222, June 1991.

[26] Shigeru Chiba. A metaobject protocol for C++. In *Proc. OOPSLA '95*, pages 285–299, October 1995.

[27] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: More types for virtual classes. Submitted for publication. Available at `http://slurp.doc.ic.ac.uk/pubs.html`, December 2005.

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

[29] Adriana B. Compagnoni and Benjamin C. Pierce. Higher order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, 1996.

[30] William R. Cook. Object-oriented programming versus abstract data types. In *Foundations of Object-Oriented Languages*, June 1990.

[31] O.-J. Dahl et al. The Simula 67 common base language. Publication No. S-22, Norwegian Computing Center, Oslo, 1970.

[32] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 59–69, June 2001.

[33] L. DeMichiel and R. Gabriel. The Common Lisp Object System: An overview. In *Proceedings of European Conference on Object-Oriented Programming*, 1987.

[34] Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[35] Erik Ernst. Propagating class and method combination. In *Proceedings of the Thirteenth European Conference on Object-Oriented Programming (ECOOP'99)*, number 1628 in Lecture Notes in Computer Science, pages 67–91. Springer-Verlag, June 1999.

[36] Erik Ernst. Family polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, LNCS 2072, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.

[37] Erik Ernst. Higher-order hierarchies. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *Lecture Notes in Computer Science*, pages 303–329, Heidelberg, Germany, July 2003. Springer-Verlag.

[38] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Proc. 33th ACM Symp. on Principles of Programming Languages (POPL)*, pages 270–282, Charleston, South Carolina, January 2006.

[39] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proc. 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 94–104, Baltimore, Maryland, USA, October 1998.

[40] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 171–183, San Diego, California, 1998.

[41] The Eclipse Foundation. Eclipse. `http://www.eclipse.org`, 2006.

[42] The Mozilla Foundation. Mozilla Firefox. `http://www.mozilla.com/firefox`, 2005.

[43] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1994.

[44] Jacques Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering (FOSE)*, Sasaguri, Japan, November 2000.

[45] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000. ISBN 0-201-31008-2.

[46] Carl Gunter and John C. Mitchell, editors. *Theoretical aspects of object-oriented programming*. MIT Press, 1994.

[47] Steve Holzner. *Eclipse*. O'Reilly Media, 2004.

[48] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[49] Scott E. Hudson, Frank Flannery, C. Scott Ananian, Dan Wang, and Andrew Appel. CUP LALR parser generator for Java, 1996. Software release. Located at `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[50] Yuuji Ichisugi. EPP: Extensible type system framework for a Java pre-processor. In *Proceedings of SPA'99*, March 1999. `http://staff.aist.go.jp/y-ichisugi/epp/edoc/epp-type-check.pdf`.

[51] Yuuji Ichisugi and Yves Roudier. The extensible Java preprocessor kit and a tiny data-parallel Java. In *Proc. ISCOPE '97*, LNCS 1343, pages 153–160. Springer, 1997.

[52] Atsushi Igarashi and Benjamin Pierce. Foundations for virtual types. In *Proceedings of the Thirteenth European Conference on Object-Oriented Programming (ECOOP'99)*, number 1628 in Lecture Notes in Computer Science, pages 161–185. Springer-Verlag, June 1999.

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

[54] Atsushi Igarashi and Benjamin C. Pierce. On inner classes. *Information and Computation*, 177(1):56–89, August 2002.

[55] Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. Simple dependent types: Concord. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP)*, Oslo, Norway, June 2004.

[56] Haskell 98: A non-strict, purely functional language, February 1999. Available at `http://www.haskell.org/onlinereport/`.

[57] Richard Kelsey, William Clinger, and Jonathan Rees (editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, October 1998.

[58] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., 2nd edition, 1988. ISBN 0-13-110362-8.

[59] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersen, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Berlin, Heidelberg, and New York, 2001. Springer-Verlag.

[60] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP'97)*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.

[61] Joseph R. Kiniry and Elaine Cheong. JPP: A Java pre-processor. Technical Report CS-TR-98-15, California Institute of Technology, Pasadena, CA, September 1998.

[62] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *Proc. ECOOP '98*, pages 91–113, 1998.

[63] Michael Y. Levin and Benjamin C. Pierce. TinkerType: A language for playing with formal systems. *Journal of Functional Programming*, 13(2), March 2003.

[64] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*, 1999.

[65] B. Liskov et al. CLU reference manual. In Goos and Hartmanis, editors, *Lecture Notes in Computer Science*, volume 114. Springer-Verlag, Berlin, 1981.

[66] Jed Liu and Andrew C. Myers. JMatch: Abstract iterable pattern matching for Java. In *Proc. 5th Int'l Symp. on Practical Aspects of Declarative Languages (PADL)*, pages 110–127, New Orleans, LA, January 2003.

[67] David MacQueen. Modules for Standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 198–204, August 1994.

[68] O. Lehrmann Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993.

[69] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism for object-oriented programming. In *Proc. OOPSLA '89*, pages 397–406, October 1989.

[70] Katsuhisa Maruyama and Ken-Ichi Shima. An automatic class generation mechanism by using method integration. *IEEE Transactions on Software Engineering*, 26(5):425–440, May 2000.

[71] Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proc. OOPSLA '01*, October 2001.

[72] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, 1988.

[73] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 90–100, Boston, Massachusetts, March 2003.

[74] Sun Microsystems. JavaBeans (version 1.0.1-a). `http://java.sun.com/products/javabeans/docs/spec.html`, August 1997.

[75] Sun Microsystems. Java reflection. `http://java.sun.com/j2se/1.3/guide/reflection`, 1999.

[76] Todd Millstein. Practical predicate dispatch. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, October 2004.

[77] Todd Millstein, Colin Bleckner, and Craig Chambers. Modular typechecking for hierarchically extensible datatypes and functions. *ACM Transactions on Programming Languages and Systems*, 26(5):836–889, September 2004.

[78] Todd Millstein and Craig Chambers. Modular statically typed multimethods. In *Proceedings of the Thirteenth European Conference on Object-Oriented Programming (ECOOP'99)*, number 1628 in Lecture Notes in Computer Science, pages 279–303. Springer-Verlag, June 1999.

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

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

[81] Peter D. Mosses. Foundations of modular SOS. *Mathematical Foundations of Computer Science*, pages 70–80, 1999.

[82] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.

[83] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 99–115, October 2004.

[84] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003*, number 2622 in Lecture Notes in Computer Science, pages 138–152, Warsaw, Poland, April 2003. Springer-Verlag.

[85] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software extension. In *Proceedings of the 2006 Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '06)*, Portland, OR, October 2006.

[86] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 146–159, Paris, France, January 1997.

[87] Martin Odersky and Christoph Zenger. Nested types. In *8th Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2001.

[88] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proc. OOPSLA '05*, pages 41–57, San Diego, CA, USA, October 2005.

[89] OMG. *The Common Object Request Broker: Architecture and Specification*, December 1991. OMG TC Document Number 91.12.1, Revision 1.1.

[90] Harold Ossher and William Harrison. Combination of inheritance hierarchies. In *Proc. OOPSLA '92*, pages 25–40, October 1992.

[91] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *Lecture Notes in Computer Science*, pages 89–110, Málaga, Spain, 2002. Springer-Verlag.

[92] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proceedings of COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, pages 9–15, Vienna, Austria, 1998.

[93] David S. Platt. *Introducing Microsoft .NET*. Microsoft Press, Redmond, WA, third edition, 2003.

[94] Venugopalan Ramasubramanian, Ryan Peterson, and Emin Gün Sirer. Corona: A high performance publish-subscribe system for the World Wide Web. In *Proceedings of Networked System Design and Implementation (NSDI)*, May 2006.

[95] Venugopalan Ramasubramanian and Emin Gün Sirer. Beehive: $O(1)$ lookup performance for power-law query distributions in peer-to-peer overlays. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[96] John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Stephen A. Schuman, editor, *New Directions in Algorithmic Languages*, pages 157–168. Institut de Recherche d'Informatique et d'Automatique, Le Chesnay, France, 1975. Reprinted in [46], pages 13–23.

[97] John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, June 1996.

[98] Dale Rogerson. *Inside COM*. Microsoft Press, Redmond, WA, 1997.

[99] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[100] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behavior. In Luca Cardelli, editor, *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, number 2743 in Lecture Notes in Computer Science, pages 248–274, Darmstadt, Germany, July 2003. Springer-Verlag.

[101] A. Shalit. *The Dylan Reference Manual*. Addison-Wesley, 1996.

[102] Yannis Smaragdakis and Don Batory. Implementing layered design with mixin layers. In Eric Jul, editor, *Proceedings ECOOP'98*, pages 550–570, Brussels, Belgium, 1998.

[103] Yannis Smaragdakis and Don Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255, April 2002.

[104] Gregor Snelting and Frank Tip. Semantics-based composition of class hierarchies. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *Lecture Notes in Computer Science*, pages 562–584, Málaga, Spain, 2002. Springer-Verlag.

[105] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, SIGPLAN Notices*, 21(11):38–45, November 1986. Published as *SIGPLAN Notices 21*, 11 (November 1986). Also in S. Zdonik and D. Maier, eds., *Readings in Object-Oriented Database Systems*.

[106] Guy Steele. *Common LISP: the Language*. Digital Press, second edition, 1990. ISBN 1-55558-041-6.

[107] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1987.

[108] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE)*, pages 107–119, May 1999.

[109] Michiaki Tatsubori, Shigeru Chiba, Marc-Oliver Killijian, and Kozo Itano. OpenJava: A class-based macro system for Java. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, LNCS 1826, pages 119–135. Springer-Verlag, July 2000.

[110] Kresten Krab Thorup. Genericity in Java with virtual types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in Lecture Notes in Computer Science, pages 444–471. Springer-Verlag, 1997.

[111] Mads Torgerson. Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 1998.

[112] Thomas VanDrunen and Jens Palsberg. Visitor-oriented programming. In *Eleventh International Workshop on Foundations of Object-Oriented Languages*, January 2004.

[113] Michael VanHilst and David Notkin. Using C++ templates to implement role-based designs. In *JSSST International Symposium on Object Technologies for Advanced Software*, pages 22–37. Springer-Verlag, 1996.

[114] Michael VanHilst and David Notkin. Using role components to implement collaboration-based designs. In *Proceedings OOPSLA '96 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 359–369, 1996.

[115] John Vlissides. Visitors in frameworks. *C++ Report*, 11(10), November 1999.

[116] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[117] Philip Wadler et al. The expression problem, December 1998. Discussion on Java-Genericity mailing list.

[118] Alessandro Warth, Milan Stanojević, and Todd Millstein. Statically scoped object adaptation with expanders. In *Proceedings of the 2006 Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '06)*, Portland, OR, October 2006.

[119] Daniel Weise and Roger F. Crew. Programmable syntax macros. In *Proc. of the '93 SIGPLAN Conference on Programming Language Design*, pages 156–165, 1993.

[120] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

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

[122] Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *Proc. 6th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Firenze, Italy, September 2001.