LANGUAGE SUPPORT FOR RELIABLE, EXTENSIBLE LARGE-SCALE SOFTWARE SYSTEMS

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

Xin Qi

January 2010

LANGUAGE SUPPORT FOR RELIABLE, EXTENSIBLE LARGE-SCALE SOFTWARE SYSTEMS

Xin Qi, Ph.D.

Cornell University 2010

Large software systems, which often consist of many interacting components, are hard to develop. For example, a compiler may contain tens of components modeling abstract syntax tree (AST) nodes, and various components for compiler passes, and the implementation of each component is entangled with that of many other components, as suggested in Wadler's *expression problem* [84]. This dissertation describes language-based mechanisms to improve the reliability and extensibility of large software.

Accessing uninitialized data is a significant source of software unreliability, causing unpredictable or exceptional behavior. Existing object-oriented languages do not guarantee that objects are correctly initialized before use. This dissertation presents *masked types* to ensure the soundness of object initialization, even with class inheritance and cyclic data structures. The type system tracks initialization in a fine-grained, modular way, and embeds a graph theoretic algorithm for reasoning about the construction of complex data structures.

Class inheritance is an important way to reuse code in object-oriented languages, but it has two limitations when applied to large software systems. First, a family of interacting classes cannot be extended together while preserving their relationships, and second, new functionality cannot be added to existing objects in a modular way. The dissertation presents two solutions: *class sharing* and *family sharing*, both addressing the two limitations at once. Class sharing is *heterogeneous*, which allows two families of classes to share some of their members, but at the price of complex language mechanisms. Family sharing is *homogeneous*: two shared families always share all of their member classes. *Shadow classes* are introduced to ensure type safety, and provide the additional expressiveness of *open families*.

Finally, the dissertation presents implementation techniques that make the sharing mechanisms practical.

BIOGRAPHICAL SKETCH

Xin Qi was born in the coastal city of Tianjin in China, in 1981. He attended Tsinghua University in Beijing, China, and earned a Bachelor of Engineering degree in Computer Science in 2003. Although he became fluent in several programming languages shortly after getting his first personal computer at the age of twelve, Xin had never thought of doing serious research in the area of programming languages before entering Cornell University in August 2003. Since then, he has been studying for his doctorate degree, while enjoying the weather in Ithaca, New York.

To my wife Ruijie Wang.

ACKNOWLEDGMENTS

First of all, I would like to thank sincerely my advisor, Andrew Myers, who has always provided me with great advice, kind encouragement, and insightful criticism. I have learned from him so many things, including how to do research, how to present work, and how to play Illuminati, just to name a few.

I would also like to thank the other members of my thesis committee: David Williamson and Dexter Kozen. I have benefited a lot from their teaching and by talking to them about my research.

During my six years at Cornell, many friends and fellow students at the Computer Science department have helped make the process of getting a Ph.D. easier and more fun. Nate Nystrom, Steve Chong, and Michael Clarkson have done a lot of great work that this thesis is built upon, and have provided very helpful feedback and suggestions on my work. Michael George, Jed Liu, K. Vikram, Lucas Waye, Fan Yang, Lantian Zheng, and Xin Zheng have been wonderful colleagues, who I have always enjoyed working with. Sigmund Cherem, Maya Haridasan, Mingsheng Hong, Kan Li, Huijia Lin, Hongzhou Liu, Wojciech Moczydlowski, Krzysztof Ostrowski, Hui Tan, Dustin Tseng, Jonathan Winter, Xinyang Zhang, and Changxi Zheng have been good friends and have made my life as a graduate student much more interesting. I would also like to thank Becky Stewart and Bill Hogan for helping me filling out numerous forms and support letters and for answering all kinds of questions.

Finally, I would like to thank my parents for always being supportive during my graduate study, and to thank especially my wife Ruijie for her ccompany, understanding, and for providing the best motivation for getting my Ph.D.

TABLE OF CONTENTS

Biographical Sketch iii
Dedication iv
Acknowledgments v
Table of Contents vi
List of Tables ix
List of Figures x

1 Introduction **1**
1.1 Software as systems of interacting components 2
1.2 Object initialization and null-pointer exceptions 4
1.3 Extending and adapting class families 5
1.4 Efficient implementation 7
1.5 Outline 8

2 Sound Object Initialization **11**
2.1 Masked types 11
2.1.1 Types for initialization state 13
2.1.2 Mask effects 14
2.1.3 Must-masks 16
2.1.4 Reinitialization 18
2.1.5 Initializing cyclic data structures 20
2.2 Abstract masks 24
2.2.1 Modular checking of abstract masks 26
2.2.2 Mask algebra 28
2.3 Formal semantics and soundness 30
2.3.1 Grammar 30
2.3.2 Class member lookup 33
2.3.3 Subtyping 33
2.3.4 Expression typing 36
2.3.5 Program typing 38
2.3.6 Decidability of type checking 39
2.3.7 Operational semantics 40
2.3.8 Type safety 42
2.3.9 Proof of soundness 42
2.4 Implementation 67
2.4.1 Inserting default effects 68
2.4.2 Transforming initialization code 69
2.4.3 Type checking 69
2.4.4 Inner classes 70
2.5 Experience 70
2.5.1 LinkedList 71
2.5.2 HashMap 73

2.5.3 TreeMap . . . 73
2.5.4 Summary . . . 73
2.6 Related work . . . 74

3 Improving Family Extensibility with Class Sharing 80
3.1 Sharing classes between families . . . 81
3.1.1 Family inheritance . . . 81
3.1.2 Sharing declarations . . . 84
3.1.3 Views and view changes . . . 87
3.1.4 Dynamic object evolution via view change . . . 90
3.1.5 Sharing constraints . . . 93
3.2 Protecting unshared state with masked types . . . 95
3.2.1 Unshared fields . . . 95
3.2.2 In-place translation with unshared classes . . . 97
3.2.3 Translation from the base family to the derived family . . 100
3.3 Formal semantics and soundness . . . 101
3.3.1 Grammar . . . 101
3.3.2 Auxiliary definitions . . . 103
3.3.3 Class lookup . . . 103
3.3.4 Subclassing . . . 105
3.3.5 Prefix types . . . 105
3.3.6 Member lookup . . . 106
3.3.7 Final access paths and exactness . . . 106
3.3.8 Type well-formedness . . . 108
3.3.9 Non-dependent bounding types . . . 109
3.3.10 Type substitution . . . 109
3.3.11 Sharing relationships . . . 110
3.3.12 Typing rules . . . 111
3.3.13 Subtyping . . . 112
3.3.14 Program typing . . . 113
3.3.15 Operational semantics . . . 113
3.3.16 Runtime typing environments . . . 116
3.3.17 Well-formed configurations . . . 118
3.3.18 Soundness . . . 118
3.4 Experience . . . 133
3.4.1 Lambda compiler . . . 133
3.4.2 CorONA . . . 134
3.5 Related work . . . 136

4 Homogeneous Family Sharing 141
4.1 Class sharing without families . . . 141
4.1.1 From inheritance to sharing . . . 141
4.1.2 Views and view changes . . . 143
4.1.3 Sharing relationships . . . 144

4.2 Homogeneous family sharing 145
4.2.1 Family inheritance 145
4.2.2 Family sharing 146
4.2.3 Shadow classes and shadow methods 150
4.2.4 Open families 153
4.3 Formal semantics and soundness 156
4.3.1 Syntax 157
4.3.2 Lookup functions 158
4.3.3 Sharing and subclassing 160
4.3.4 Prefix types 160
4.3.5 Type substitution 161
4.3.6 Static semantics 161
4.3.7 Operational semantics 165
4.3.8 Soundness 167
4.4 Experience 181

5 Efficient Implementation of Family Extensibility and Class Sharing 185
5.1 Background on implementing family inheritance 187
5.1.1 Static translation 188
5.1.2 Dynamic representation with data structures 192
5.2 Classloader-based implementation of family inheritance 196
5.2.1 Compiler translation 197
5.2.2 Classloader-based run-time system 200
5.2.3 Load-time optimizations 202
5.2.4 Java compatiblity 203
5.3 Implementing sharing 204
5.3.1 Translating classes 205
5.3.2 Supporting views and view changes 205
5.3.3 Implementing heterogeneous class sharing 207
5.3.4 Implementing homogeneous family sharing 209
5.4 Results 211
5.4.1 Microbenchmarks for sharing 211
5.4.2 Jolden benchmarks 212
5.4.3 Tree traversal 213
5.5 More efficient implementation 214

6 Conclusions 217

Bibliography 220

LIST OF TABLES

5.1 Microbenchmarks: average time per operation, in ns. 211
5.2 Results for the jolden benchmarks. Average time over ten runs, in seconds. 212
5.3 Comparing view changes with explicit translation. Average time over ten runs, in seconds. 214

LIST OF FIGURES

2.1 Code with an initialization bug 12
2.2 Object recycling 19
2.3 Initialization of a tree with parent pointers 21
2.4 Mask dependencies 22
2.5 The tree example with abstract masks 25
2.6 Code that needs mask constraints 27
2.7 Grammar 30
2.8 Class member lookup 33
2.9 Static semantics 34
2.10 Auxiliary definitions 35
2.11 Well-formed expressions 38
2.12 Program typing 39
2.13 Small-step operational semantics 41
2.14 Well-formed heaps 41

3.1 An expression family and a GUI family 82
3.2 Mixing display into expressions, with nested inheritance 82
3.3 Using class sharing to adapt `Exp` to `TreeDisplay` 86
3.4 Evolution of a network service package 92
3.5 Shared classes with unshared fields 95
3.6 Lambda calculus and pair compiler extension 98
3.7 In-place translation of the pair language 99
3.8 Grammar of the J&$_s$ calculus 102
3.9 Classes 104
3.10 Static semantics 107
3.11 Paths and exactness 108
3.12 Type well-formedness 109
3.13 Type bounds 109
3.14 Type substitution 110
3.15 Program typing 114
3.16 Definitions for operational semantics 115
3.17 Small-step operational semantics 117
3.18 Runtime typing environments 117
3.19 Well-formed configurations 118
3.20 Lambda compiler structure. Translator and some AST nodes not shown. 133

4.1 A GUI button class and its extension 142
4.2 A base GUI family and an extension 145
4.3 Shadow classes and shadow methods 150
4.4 Sketch of the IMP compiler structure. Shadow classes in dashed boxes. 154

4.5 In-place translation for loop statements 155
4.6 Syntax of JHS 157
4.7 Lookup functions 159
4.8 Sharing and subclassing 160
4.9 Auxiliary definitions 162
4.10 Static semantics: auxiliary judgments 163
4.11 Static semantics: sharing and subtyping 164
4.12 Static semantics: expressiong typing 164
4.13 Program typing 165
4.14 Definitions for operational semantics 166
4.15 Small-step operational semantics 166
4.16 The view function 167
4.17 Runtime typing environments 168
4.18 Runtime configuration well-formedness 168
4.19 Compiler structure with heterogeneous sharing. Translator and some AST nodes not shown. 182
4.20 Compiler structure with homogeneous sharing. Translator and some AST nodes not shown. 182

5.1 Static class translation 189
5.2 Updated class representation 197
5.3 An example with a Java constructor 198
5.4 Illustration of translation of code in Figure 5.3. Name mangling not shown for clarity. 199

CHAPTER 1

INTRODUCTION

It has long been observed that much of the difficulty of building reliable software systems arises from their size and complexity [6]. Large software systems are challenging to develop because these systems contain many components that interact in complex ways. Modular programming and object-oriented programming are intended to help programmers manage interactions by allowing related functions and data to be grouped together in modules or classes. However, existing language mechanisms do not scale up to large systems containing many classes, in which these classes interact both by reference and by inheritance. As a result, practitioners have been relying on various conventions and design patterns in order to improve the reliability and extensibility of software systems. However, these mechanisms are usually tedious to implement and sometimes only suitable for special cases, and the lack of language support means that correct usage of these mechanisms is not statically enforced.

Therefore, new language mechanisms are desirable for developing reliable and extensible software. This dissertation focuses on two problems that arise with large collections of interacting classes:

- Accessing uninitialized data is a significant source of software unreliability. Existing object-oriented programming languages do not guarantee that objects are correctly initialized before use; object initialization is therefore unsound. Accessing uninitialized data leads either to unpredictable behavior (e.g., in C++) or to null-pointer exceptions (e.g., in Java).
- Large collections of interacting classes are hard to extend, evolve, and

reuse. Ordinary inheritance aims to support extension and reuse, but has two important limitations. First, a *family* of interacting classes cannot be extended at the same time while preserving their relationships, and second, new functionality cannot be added to the objects of an existing class without modifying the class.

This work addresses the above problems by extending existing languages with light-weight, type-based mechanisms that help programmers initialize and evolve interacting classes and their objects. Such a language-based solution has the following benefits:

- Type safety. The compiler statically prevents the software from crashing at run time due to incorrect usage of language mechanisms.
- Static reasoning. Types in source code capture valuable information about invariants and contexts, and help programmers reason about code locally.
- Clean code. By eliminating complex, error-prone coding conventions and design patterns, the source code becomes easier to understand and maintain.

1.1 Software as systems of interacting components

Large software systems often contain many interacting data structures, algorithms, and interfaces. While working on these systems, developers must understand these interactions and ensure that as the code evolves, these interactions are correctly preserved.

Programming language support has proved to be useful for helping programmers deal with the complexity of software systems. Many modern programming languages provide modularity mechanisms that help programmers engage in *local reasoning* about software components, so they can focus on one component at a time.

Object-oriented programming has become a popular way to obtain local reasoning. It supports modularity through classes. Further, it supports reuse and extensibility through class inheritance, which has many uses. In particular, inheritance makes possible software libraries whose classes are designed to be conveniently inherited, and whose methods are intended to be overridden by user-defined subclasses. In applications using these libraries, subclasses coexist with—and sometimes replace—the superclasses they inherit from.

Unfortunately, these existing mechanisms for modularity and extensibility do not scale up to large software systems. The problem is that large software systems contain many classes, which interact with each other in complex ways, creating dependencies that are hard to reason about. Classes interact by *reference* when objects of one class refer to objects of another, in which case the referring class depends on the publicly exposed interface of the referenced class. Classes can also interact by *inheritance*, in which case the subclass depends on the *specialization interface* [43]—and often on unspecified details of the superclass implementation.

In large software systems, classes typically interact through both reference and inheritance. The resulting complex dependencies among classes, and among families of classes, are challenging for programmers who are designing, maintaining, or reusing these software systems. This makes large software systems expensive, unreliable, and even insecure.

1.2 Object initialization and null-pointer exceptions

In most object-oriented languages, objects are initialized via constructors. During initialization of one object, its constructor may in turn create and initialize other referenced objects, and also invoke the object's superclass constructor. But in a system with many interacting classes, these other object initializations may invoke arbitrarily complex code. There is therefore a danger that the original object may be used before its initialization finishes.

In fact, in current object-oriented languages, object initialization is *unsound*: fields of objects can be accessed before they are initialized. There is no simple fix to this problem; object-oriented programs often need to create complex cyclic data structures where the need to manipulate partially initialized objects is unavoidable.

Unsound initialization leads to unreliable or even insecure software. In C++, accessing uninitialized fields has unpredictable results and may create security vulnerabilities. The Java and C# languages avoid security problems by performing a *default initialization* on all fields, setting the values of all (object-typed) fields to the special value `null`. However, if these `null` values are used in places of the correctly initialized value, execution of the program is likely to halt with a null-pointer exception.

The connection between unsound object initialization and null-pointer exceptions goes deeper than this argument suggests. In order to allow default initialization, languages like Java and C# must treat the value `null` as a valid member of every object type. Since every object can be `null`, almost every operation on objects—invoking methods, reading or writing fields—must check for `null` and potentially raise an exception that will crash the program. Null pointer exceptions can happen almost anywhere in the code, and there is little the compiler can do to help. This significantly hurts software reliability [10].

Recently there has been interest in controlling null-pointer exceptions through *non-null annotations* and other means [25, 10, 42, 20]. But non-null annotations by themselves do not solve the problem of object initialization; in fact, they make it all the more important because all non-null fields must be explicitly initialized before use.

To solve the initialization problem, this dissertation proposes new language mechanisms that make object initialization sound and that eliminate the possibility of null-pointer exceptions.

1.3 Extending and adapting class families

The code of software systems needs to be extended, evolved, and reused. However, existing programming paradigms are not *scalable* in the following sense: the programming effort required to make a change to code is not proportional to the degree of change. Making a change to a large system often involves making edits to many places throughout the system.

Through the mechanism of inheritance, object-oriented programming supports reuse, evolution, and extensibility. The advantage of inheritance is that it is a relatively lightweight extensibility mechanism. Every object method implicitly serves as an interposition point where new behavior can be added through overriding. This contrasts with more heavyweight approaches to extensibility based on explicit parameterization of code, such as parameterized types [45], parameterized mixins [5], and functors [54].

In our experience with the Polyglot extensible compiler framework [57], we found that object-oriented inheritance does not adequately support extensibility for systems containing interacting classes. To work around the limitations of inheritance, we invented new design patterns and wrote Polyglot in a stylized way using these design patterns. However, the design patterns were verbose, error-prone, and not type-safe.

We (and others) have found that inheritance has two notable limitations. The first limitation is that inheritance operates on one class at a time. Therefore, interacting classes can only be extended individually, which does not support coordinated changes that span multiple classes. For example, suppose there is an inheritance hierarchy of classes rooted at class A. The programmer wants to reuse this code by creating a second, similar hierarchy of classes with the difference that every class has an additional field `f`. There is no way to use inheritance to achieve this in a scalable, modular way. Inheritance is also inadequate for extending a set of classes whose objects interact according to some protocol, a pattern that occurs in many domains ranging from compilers to user interface toolkits.

A second limitation of inheritance is that new behavior or state cannot be added to objects of an existing class without modifying the class definition. We refer to this ability as *adaptation*, because the Adapter design pattern partially addresses this problem. Adaptation differs from inheritance, where only objects of new classes have the new behaviors. With adaptation, new behavior can be added to instances of objects that are created by an existing library, or that are deserialized from a file.

These limitations have been addressed separately by two lines of research. In the first, the ability to extend classes as a group is provided by *family inheritance* mechanisms, developed by the investigator and others [51, 24, 12, 71, 38, 56, 58]. The second line of research has explored language support for adaptation, allowing modular addition of functionality to an existing class [13, 85, 3, 52]. However, no prior work has been done to fully integrate these two distinct, but closely related approaches for software extensibility.

This dissertation presents *class sharing*, which I introduced in [65, 63], to provide the first full integration of the two kinds of extensibility, and to enable new ways to develop extensible software systems.

1.4 Efficient implementation

Being expressive and type-safe does not automatically grant success to a language mechanism. An efficient implementation would demonstrate that the mechanism may work well in practice, and promote the adoption of the mechanism.

For nested inheritance and nested intersection, the original implementations presented in [56, 58] are not very satisfactory. The implementation in [56] does not support scalable compilation, since it generates target code for implicit classes that do not appear in the source code, which makes the compilation time not proportional to the size of the source code. For example, compilation of a small extension (less than 100 lines of code) of the Polyglot compiler framework would take more than 30 minutes (on a 2004 laptop computer) and larger than 1GB of memory. On the other hand, the compiler in [58] is scalable, generating code that is proportional to the size of the source code. However, the scalability is achieved with a complex translation scheme that at run time uses various data structures to handle operations like method dispatches and field accesses. Therefore, the performance overhead is very large, with almost 6x slowdown compared to Java [65]. Improvements on the implementation would reduce the cost of applying these family extensibility mechanisms, and make these mechanisms more appealing.

This dissertation demonstrates how to implement family inheritance in a way that is both efficient and scalable, using a custom classloader on top of a commercial Java virtual machine. The implementation is also extended to support class sharing.

1.5 Outline

The rest of the dissertation is organized as follows. Chapter 2 presents a type system that enforces sound object initialization, using masked types [64]. Language mechanisms like mask effects, abstract masks, and conditional masks make initialization safe even with interacting classes that are related through class inheritance and mutual references. The soundness of the type system is formally proved. Masked types are implemented in the J\mask language, a mostly backward compatible extension to Java with no run-time overhead. Experience with the implementation shows that masked types introduce low annotation burden for the programmer, suggesting their utilities with large software systems.

Chapter 3 and 4 describe two ways of integrating in-place extensibility and family extensibility: (*heterogeneous*) class sharing [65] and *homogeneous* family sharing [63]. Heterogeneous sharing supports sharing declarations between individual classes from different families, and allows families to share some of their corresponding nested classes. Some nested classes might be left unshared, either on purpose, or because of new subclasses introduced in one of the shared families. Language mechanisms like sharing constraints and masked types are used to ensure type safety, in the face of challenges created by these unshared classes. On the contrary, homogeneous sharing lifts sharing to the level of families: sharing is declared between families, and is automatically applied to all the nested classes. Shadow classes ensure that entire families are shared, even with new subclasses introduced. The soundness of the type system is much easier to establish, and sharing constraints and masked types are not needed. Therefore, homogeneous family sharing presents a clean, safe, expressive, and scalable solution to the problem of integrating in-place extensibility and family extensibility. Soundness proofs for the type systems and experiences with the languages are also presented.

Chapter 5 presents an efficient and scalable translation scheme for family inheritance, and its extension to support class sharing as well. The key idea is to use a custom classloader to synthesize code for implicit classes at run time. Extensions for heterogeneous and homogeneous sharing are also included. The performance is practical, especially considering the expressive power provided by the language. Finally, this chapter describes ideas that may further reduce the performance overhead.

Finally, Chapter 6 concludes the dissertation.

CHAPTER 2

SOUND OBJECT INITIALIZATION

Object initialization remains an unsatisfactory aspect of object-oriented programming. In the usual approach, objects of a given class are created and initialized only by class constructors. Therefore, when implementing class methods, the programmer can assume that object fields satisfy an invariant established by the constructors. However, in the presence of inheritance or mutual references that form cycles, the methods of partly initialized objects may be invoked before the invariants has been established, which may cause unexpected or unwanted behavior—e.g., throwing null-pointer exceptions—and make reasoning about object initialization non-modular.

This chapter presents a new solution to the object initialization problem, based on a new type mechanism, *masked types*, implemented in the J\mask language. The J\mask type system conservatively tracks initialization state of every object, so a partially initialized object cannot be used where a fully initialized object is expected. Mechanisms like mask effects and conditional masks make initialization safe with large-scale software systems with complicated inheritance and referencing relationships between classes.

2.1 Masked types

Figure 2.1 illustrates a bug that can easily happen in an object-oriented language like Java. In the class `Point`, representing a 2D point, the constructor calls a virtual method `display` that prints the coordinates of the point. The two fields `x`

```
class Point {
  int x; int y;
  Point(int x, int y) {
    this.x = x;
    this.y = y;
    display();
  }
  void display() {
    System.out.println(x + " " + y);
  }
}

class CPoint extends Point {
  Color c;
  CPoint(int x, int y, Color c) {
    super(x, y);
    this.c = c;
  }
  void display() {
    System.out.println(x + " " + y + " " + c.name());
  }
}
```

Figure 2.1: Code with an initialization bug

and `y` are properly initialized before `display` is called. However, in the subclass `CPoint` representing a colored point, the `display` method has been overridden in a way that causes the added `c` field to be read before it is initialized, resulting in a null pointer exception.

This example is simple, but in general, initialization bugs are difficult to prevent in an automatic way. It would be too restrictive to rule out virtual method calls on partially constructed objects. Further, the bug involves the interaction of code from two different classes (`Point` and `CPoint`). An implementer of `CPoint` might not have access to the code of `Point` and would not realize the danger of overriding the `display` method in this seemingly reasonable way.

Our goal is to prevent code like that of class `Point` from type-checking, but to allow complex, legitimate initialization patterns. The key observation is that before the call to `display` on line 6, the fields in `Point` are initialized, but fields of subclasses of `Point` are not. However, the type of the method `display` does not prevent the partially initialized receiver from being passed to an overridden version of the method that reads uninitialized fields, as in `CPoint`.

2.1.1 Types for initialization state

A masked type $T\backslash M$, where M is a *mask* that denotes some object fields, is the type T but without read access to the denoted fields. Masked types are a completely static mechanism, so a J\mask program is compiled by erasing masks. No run-time penalty is paid for safe object initialization.

The simplest form of a mask is just the name of a field. For example, an object of type `CPoint\c` is an instance of the `CPoint` class whose field `c` cannot be read, perhaps because it has not been initialized. We say that the field `c` is *masked* in this type.

A type with no mask means that the object is fully initialized. In typical programming practice, this would be the ordinary state of the object, in which its invariant is already established.

On entry to a constructor such as `Point()`, the newly created object has all its fields masked. The actual class of the new object might be a subclass (for example, `CPoint`), so on exit, subclass fields remain to be initialized. A *subclass mask*, written $C.\mathsf{sub}$, is used to mask all fields introduced in subclasses of C, not including those of C itself. Therefore, just before line 4 in Figure 2.1, the object being constructed has type `Point\x\y\Point.sub`. (While this type looks complicated, it can be inferred automatically.)

When a field is initialized by assigning to it, the corresponding mask is removed from the type of the object. For example, line 4 in Figure 2.1 assigns to field `x`, so the type of `this` becomes `Point\y\Point.sub`. After the assignment to `y` on the next line, the type of `this` becomes `Point\Point.sub`. Thus, the initialization of various fields is recorded in the changing type of `this`. Because variables may have different types at different program points, J\mask has a *flow-sensitive* type system.

Subclass masks such as `Point.sub` can be removed when the exact run-time class of an object is known, because there are no subclass fields left to initialize. The type of a `new` expression is known exactly, as is the type of a value of any class known not to have a subclass (in Java, a "final" class).

J\mask has a special mask $*$ as a convenient shorthand for masking all fields, including those masked by the subclass mask. On entry to the `CPoint` constructor, the object can be given type `CPoint\*`, which is equivalent to `CPoint\x\y\c\CPoint.sub`.

2.1.2 Mask effects

In J\mask, methods and constructors can have *effects* [48] that propagate mask information across calls. For example, the J\mask signatures for the `Point` constructor and the `display` method can be annotated explicitly with effect clauses:

```
    Point(int x, int y) effect * -> Point.sub
    void display() effect {} -> {}
```

The effect of this `Point` constructor says that at entry to the constructor, all fields are uninitialized (precondition mask `*`) and therefore unreadable; at the end of the constructor, only fields introduced by subclasses of `Point` remain uninitialized (postcondition mask `Point.sub`). Because the initial and final masks of the `display` method are both `{}`, denoting the absence of any mask, the method can be called only with a fully initialized object, and it leaves the object fully initialized.

With these effects, the bug in Figure 2.1 would be caught statically. The method `display` cannot be invoked on line 6, because there the type of `this` is `Point\Point.sub`, which does not satisfy the precondition of `display`. The J\mask compiler detects this unsafe call without inspecting any subclass of `Point`.

This example suggests how mask effects make the J\mask type system modular. Mask effects explicitly represent the contract on initialization states that a method is guaranteed to follow. This explicit contract allows the compiler to type-check programs one class at a time, and also enables programmers to reason about initialization locally.

Indeed, masked types and mask effects capture changes to initialization state with enough precision that constructors in J\mask are essentially ordinary methods that remove masks from the receiver. However, for convenience and backward compatibility, the J\mask language still has constructors.

To reduce the annotation burden, the J\mask language provides default effects for methods and constructors. Programmers do not normally have to annotate code with effects or masks. For ordinary methods, the default is `{} -> {}`; for constructors, the default effect is close to that shown above (see Section 2.1.3).

The effects shown capture changes to the initialization state of the parameter `this`, the receiver object. J\mask also supports effects on other parameters, as shown in Section 2.1.5.

For simplicity, exceptions, which are rarely thrown during initialization anyway, have been ignored in this paper. However, exceptions can be supported by providing a postcondition for each exceptional exit path in the effect clause.

2.1.3 Must-masks

All the masks shown in Section 2.1.1 are *simple* masks. A simple mask S, e.g., f, $*$, or $C.\mathtt{sub}$, means that the fields it describe *may* be uninitialized. Thus, there is a subtyping relationship $T \leq T\backslash S$, because it is safe to treat an initialized field as one that may be uninitialized.

However, when an object is created, it is known that all the fields *must* be uninitialized. J\mask uses *must-masks*, written $S!$, to describe fields that must definitely be uninitialized. A must-masked type $T\backslash S!$ is also a subtype of $T\backslash S$, but T is not a subtype of $T\backslash S!$.

One use of must-masks is for initialization of "final" fields, which is only allowed when the field is must-masked, ensuring that the field is initialized exactly once. Must-masks and the absence of masks roughly correspond to the notions of *definite unassignment* and *definite assignment* in the Java Language Specification [31]. However, J\mask ensures that a final field cannot be read before it is initialized, while Java does not. J\mask also lifts the limitation in Java that final fields can only be initialized in a constructor or an initializer.

Must-masks are also used to express the default effect of a constructor of class `C`, which is `*! -> C.sub!`. Objects start with all fields definitely uninitialized, which is represented with the initial mask `*!`. Constructors usually do not initialize fields declared in subclasses, so the default postcondition mask is `C.sub!`.

Must-masks impose restrictions on how an object can be aliased: if there is a reference with a must-masked type, it must be the only reference through which the object may be accessed; otherwise, the must-masked field might be initialized through another reference to the object, invalidating the must-mask. This does *not* preclude aliasing, but implies rather that other references have to be through fields that are themselves masked.

J\mask uses typestate to keep track of initialization state. A problem with most previous typestate mechanisms is that they require reasoning about potential aliasing, to prevent aliases to the same object that disagree about the current state. Aliasing makes it notoriously difficult to check whether clients and implementations are compliant with protocols specified with typestate [4], and much previous work on typestates requires complicated aliasing annotations or linear types. J\mask is designed to work with no extra aliasing control mechanism, which provides the added benefit of soundness in a multi-threading setting, since operations on an object through aliases from other threads do not invalidate typestates in the current thread.

The key to avoiding reasoning about aliasing is that if an assignment creates an unmasked alias, then must-masks on both sides are conservatively converted to corresponding simple ("may") masks. For example, after the following code, the type of both `x` and `y` is the simply masked type `C\f`:

```
C\f! x = ...;
C\f! y = x;
```

Similarly the following code also removes the must annotation from the type binding of variable `x`, because `z.g` becomes an alias and the field `g` is not masked in the type `D` of variable `z`:

```
C\f! x = ...;
D z = ...;
z.g = x;
```

The non-aliasing requirement on must-masks might seem restrictive, but it is usually not a problem: must-masks typically appear near allocation sites, where no alias has been created.

2.1.4 Reinitialization

Beyond initialization, masked types can help reasoning about *reinitialization*. A mask can represent not only an uninitialized field, but also a field that must be reassigned before further read accesses. To enforce reinitialization, a mask can be introduced on the field, via the subtyping rule $T \le T\backslash f$.

For example, Figure 2.2 illustrates a custom memory management system that manages a pool of recycled objects of the class `Node`. Actively used objects are not in the pool and store data in their `d` fields. Objects in the pool are threaded into a freelist using their `next` fields. When a `Node` object is no longer used, it is put into a pool by calling the `recycle` method; when a new instance of `Node` is needed, the `getNode` method returns an object from the pool, if there is any. Masked types can help ensure that the field `d` is reinitialized whenever a `Node` object is retrieved from the pool and gets a second life. Of course, like most custom memory management systems, the code in this example does not guarantee that no alias exists after an object is recycled. Masked types are not intended to enforce this kind of general correctness.

```
1  class Node {
2    Data d;
3    Node\d next;
4  }
5
6  class Pool {
7    Node\d head;
8    ...
9    Node\d\next getNode() {
10     if (head != sentinel) {
11       Node\d\next result = head;
12       head = head.next;
13       return result;
14     } else
15       return new Node();
16   }
17   void recycle(Node\next n) {
18     n.next = head;
19     head = n;
20   }
21 }
```

Figure 2.2: Object recycling

The type `Node` is a subtype of `Node\d`, and therefore the second assignment (line 19) in method `recycle` type-checks, causing `Node` objects in the pool to "forget" about the data stored in field `d`.

Masked types provide an additional benefit here. Objects in active use have type `Node\next`, preventing traversal of the freelist from outside the `Pool` class.

2.1.5 Initializing cyclic data structures

Many data structures that arise in practice contain circular references: for example, doubly linked lists and trees whose nodes have parent pointers. Safe initialization of these cyclic data structures poses a challenge. In object-oriented languages, storing a reference to a partially initialized object is normally required, with no guarantee that the object is fully initialized before use.

J\mask explicitly tracks fields that point to partially initialized objects with *conditionally masked types*, written $T\backslash f[x_1.g_1, \ldots, x_n.g_n]$. The conditional mask $f[x_1.g_1, \ldots, x_n.g_n]$ describes a field f referencing a partially initialized object, which will become fully initialized when all fields $x_i.g_i$ are initialized. In other words, the removal of the mask on f is conditioned on the removal of all masks on $x_i.g_i$.

Conditional masks are normally introduced by an assignment to a must-masked field f, when the right-hand side of the assignment has more masks than the declared field type. Consider, for example, a field assignment `x.f = y`, where `x` has type $T\backslash f!$, `y` has type $T'\backslash g$, and the field `f` of class T has type T'. Note that $T'\backslash g$ is not a subtype of T'. J\mask makes this assignment safe by changing the type of `x` to $T\backslash f[y.g]$ after the assignment, showing that the field x.f is still masked, but its mask should be removed upon the removal of the mask on y.g.

```
class Node {
  Node parent;
  Node() effect *! -> *! { }
}

final class Leaf extends Node {
  Leaf() effect *! -> parent! { }
}

final class Binary extends Node {
  Node left, right;
  Binary(
    Node\parent!\Node.sub[l.parent] -> *[this.parent] l,
    Node\parent!\Node.sub[r.parent] -> *[this.parent] r)
  effect *! -> parent!, left[this.parent],
               right[this.parent] {
    this.left = l;
    this.right = r;
    l.parent = this;
    r.parent = this;
  }
}

Leaf\parent! l = new Leaf();
Leaf\parent! r = new Leaf();
Binary\parent!\left[root.parent]\right[root.parent]
    root = new Binary(l, r);
root.parent = root; // Now root has type Binary.
```

Figure 2.3: Initialization of a tree with parent pointers

Figure 2.3 shows how to safely initialize a binary tree with parent pointers. For convenience, we assume all local variables, including formal parameters, are final. (Section 2.4 discusses how to relax this.)

Figure 2.3 also demonstrates effects on parameters other than the receiver this: the parameters l and r of the Binary constructor both have the type `Node\*[this.parent]` upon the exit of the constructor.

In this example, initialization is bottom-up, as it would be, for example, in a shift-reduce parser. Child nodes are created, initialized, and then used to construct their parent node. However, child nodes cannot be fully initialized before their `parent` fields are set, and moreover, they cannot even be considered fully initialized before the fields of all the objects that are transitively reachable are set. (Top-down initialization of this data structure creates similar issues.)

The `parent` field of a node will eventually point to an object that is created later and that contains child pointers pointing back to the current node, creating parent–child cycles. Of course, the `parent` field of the root of the tree must point to something special. For example, it can point to the root itself, as shown on line 28, or to a sentinel node.

The dependencies between masks after line 20 in Figure 2.3 are summarized in Figure 2.4, where the mask at the tail of an arrow is removed when the mask at its head is removed. The masks on `this.left` and `this.right` after line 20 transitively depend on the mask on `this.parent`.

$$\begin{tikzcd}
 & & \texttt{this.parent} & & \\
\texttt{l.Node.sub} \arrow[r] & \texttt{l.parent} \arrow[ur] \arrow[d, bend right] & & \texttt{r.parent} \arrow[ul] \arrow[d, bend left] & \texttt{r.Node.sub} \arrow[l] \\
 & \texttt{this.left} \arrow[u] & & \texttt{this.right} \arrow[u] &
\end{tikzcd}$$

Figure 2.4: Mask dependencies

The postcondition in the effect of the `Binary` constructor summarizes the dependencies: parameters `l` and `r` both have mask $\star$[`this.parent`], which means that all their fields are conditionally masked, and `this` has type `Binary\parent!\left[this.parent]\right[this.parent]`, which is compatible with the parameter type of the `Binary` constructor. Therefore, the construction can proceed to build higher trees. Finally, the tree is fully initialized when the `parent` field of the root is initialized, because removing its mask enables removing all the masks in Figure 2.4.

In general, a field *f* should be unreadable unless every object transitively reachable through *f* has been appropriately initialized. That is, its masks have been removed at least to the level according to the type of the field through which the object is referenced.

Therefore, there are three ways to remove a conditional mask on field *f*:

- Like other kinds of masks, the conditional mask can be removed by directly initializing the field *f*.
- As shown in Figure 2.3, on line 28, conditional masks on `root.left` and `root.right` are removed by removing the mask `root.parent` they (transitively) depend on.
- The last way to remove a conditional mask is by creating cyclic dependencies. For example, the following code creates cyclic dependencies between `x.f` and `y.g`, which cancel each other.

  ```
  // x starts with type C\f!, and y starts with D\g!
  x.f = y; // now x has type C\f[y.g]
  y.g = x; // now y has type D\g[x.f]
           // x can be typed C, and y can be typed D
  ```

  In general, if some dependencies form a strongly connected component in which no mask depends on a mask outside the component, they can all be removed together.

Subtyping generalizes to conditionally masked types: $T \leq T\backslash f[x_1.g_1, \ldots, x_n.g_n] \leq T\backslash f$. In fact, a type T with unmasked field f can be viewed as a type that has empty conditions for the mask on f, that is, $T\backslash f[\,]$, and a simply masked type $T\backslash f$ can be seen as having an unsatisfiable condition on f, because a simple mask cannot be removed by removing other masks.

Conditional masks and simple masks do not impose any restriction on aliasing, because mask subtyping ensures that they cannot be invalidated by any future change to the object. This property has been called heap monotonicity [26].

Conditional masks also provide a way to create temporarily unreadable aliases for must-masked objects. Because the aliases are unreadable, the must annotations need not be removed. In Figure 2.3, for example, the assignment on line 17 creates an alias `this.left` for the left child object stored in variable `l`, but `l` remains of type `Node\parent!`, since the field `this.left` is masked with the conditional mask `left[l.parent]` after line 17. Not losing the must information means the initialization state of `l` is tracked more precisely.

For simplicity, fields currently must be declared with unmasked or simply masked types; no conditional masks or must-masks are allowed. It should be straightforward to add support for conditionally masked field types, but this is left for future work.

2.2 Abstract masks

With the exception of $*$ and $C.\mathtt{sub}$, the masks we have seen so far are *concrete*, explicitly naming instance fields. Concrete masks create difficulties for data abstraction, because the fields might not be visible where the masks are needed. For example, in Figure 2.3, if the two fields `left` and `right` of class `Binary` were private, it would be impossible to declare the local variable `root` as shown on line 26, because its type mentions the names of the fields outside the class definition.

```
1  class Node {
2    mask Children;
3    ...
4  }
5
6  final class Binary extends Node {
7    private Node left, right;
8    mask Children += left, right;
9    Binary(...)
10   effect *! -> parent!,
11                Children[this.parent] { ... }
12   ...
13 }
14 ...
15 Binary\parent!\Children[root.parent]
16     root = new Binary(l, r);
17 root.parent = root;
```

Figure 2.5: The tree example with abstract masks

Therefore J\mask introduces *abstract masks* that abstract over sets of concrete fields, providing a way to write types that mask fields that are not visible. Figure 2.5 shows an updated version of the code from Figure 2.3, where the two fields `left` and `right` are now private, and an abstract mask `Children` is introduced to mask them outside the class `Binary`. The `Children` mask is first declared in class `Node` (line 2), with an empty set of fields, and is *overridden* in `Binary` (line 8) to include the two children of a binary node. J\mask currently allows abstract masks to be overridden only to include more fields; more complex overriding is left to future work.

The $*$ mask, introduced in Section 2.1.1, is not much different from any other abstract mask, except that it is built-in, and is automatically overridden in every class to include all the fields declared in that class.

2.2.1 Modular checking of abstract masks

Subclass masks. The `Point/CPoint` example in Section 2.1.1 showed that unsafe calls to overridden methods could be caught in a modular way with the help of the subclass mask `Point.sub`. The mask `Point.sub` can be connected to the abstract mask $*$ through the equivalence of the two types `Point\*` and `Point\x\y\Point.sub`. Any type with an abstract mask can be similarly expanded. For example, given the code in Figure 2.5, the masked type `Binary\Children` is equivalent to `Binary\left\right\Binary.Children.sub`, where `Binary.Children.sub` represents all the concrete masks that are added into overriding declarations of `Children` in subclasses of `Binary`, excluding `Binary` itself. The set that consists of masks `left`, `right`, and `Binary.Children.sub` is the *interpretation* of `Children` in the context of `Binary`.

In general, $C.M.\mathsf{sub}$ represents the subclass mask of abstract mask M with respect to class C, and the interpretation of M in the context of C is a set consisting of all the concrete masks added into M in C and its superclasses, together with subclass mask $C.M.\mathsf{sub}$. Before type checking, the J\mask compiler internally expands all abstract masks into their interpretations.

Subclass masks are important for modular type checking, because they make it possible to distinguish the current definition of an abstract mask and overriding definitions in subclasses, which are generally unavailable in a modular setting.

```
1  class C {
2    T f;
3    mask M += f;
4    void initM() effect M -> {} {
5      this.f = ...;
6    }
7  }
8
9  class D extends C {
10   T g;
11   mask M += g;
12   void initM() effect M -> {} {
13     this.g = ...;
14     super.initM();
15   }
16 }
```

Figure 2.6: Code that needs mask constraints

Mask constraints. Subclass masks help prevent unsafe calls, but since they describe fields that are generally not known in the current class, safely removing them by initialization requires some additional mechanism. Figure 2.6 illustrates an initialization helper method `initM`, which is intended to remove the abstract mask `M` from its receiver. It is properly overridden in the subclass `D` to handle the overridden abstract mask `M`. However, the `initM` method would not type-check as written in Figure 2.6, because right after line 5, the type of `this` is actually `C\C.M.sub`, rather than the unmasked type `C`.

J\mask uses *mask constraints* to solve this problem. Every J\mask method can declare a mask constraint of the form `captures` $M_1, \ldots, M_n$, where $M_1, \ldots, M_n$ are abstract masks. This constraint means that the body of the method is type-checked assuming that the masks M_i are the same as their concrete definition in the class where the method is defined, with no subclass masks.

For example, the signature of `initM` on lines 4 and 12 can be updated with a mask constraint:

```
void initM() effect M -> {} captures M
```

The example then type-checks, because at the entries to `initM` in classes `C` and `D`, the type of `this` becomes `C\f` and `D\f\g` respectively, rather than `C\f\C.M.sub` and `D\f\g\D.M.sub`.

However, when type-checking callers against the public signature of the method, the abstract mask should still be interpreted to include the subclass mask.

A method defined in class C with a mask constraint on an abstract mask M depends on the set of fields that M denotes in C. It would be unsound to allow that method to be inherited by a subclass that overrides the abstract mask. Therefore, the type system requires such methods to be overridden when the masks they depend on are overridden. Consequently, constructors, final methods, and static methods cannot have mask constraints, because they cannot be overridden in subclasses.

2.2.2 Mask algebra

J\mask supports two algebraic operations on masks that make abstract masks more useful: $(M_1 + M_2)$ and $(M_1 - M_2)$.

An abstract mask can be interpreted as a set of concrete masks on fields and possibly a subclass mask. The two operators on masks correspond to the set union (+) and set difference (−) of the interpretations of the abstract masks. Concrete masks can appear in algebraic masks, where they are interpreted as singleton sets.

Algebraic masks enable the programmer to express initialization state abstractly, without knowing all the fields masked by an abstract mask. For example, suppose there is a local variable x, starting with the type $T\backslash M$ where M is an abstract mask, and field $x.f$ is initialized:

```
T\M x = ...;
x.f = ...;     // The type of x is now T\(M - f)
```

Here, one needs not know which concrete masks are included in M, nor even whether M includes f.

Mask algebra also helps programmers compose masks to keep the typestates in J\mask compact. For example, if a class has n fields, each of which might independently be initialized or uninitialized, it would require 2^n different typestates to represent all possible initialization states, were there no mask algebra. With mask algebra, one can simply use the “sum” of the masks corresponding to all the uninitialized fields.

J\mask currently only supports these two algebraic operations on masks, but they seem to suffice. Richer operators on masks are left to future work.

programs	Pr	$::= < \overline{L}, e >$
class declarations	L	$::= \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overline{F}\ \overline{Mt}\}$
field declarations	F	$::= T\ f$
method declarations	Mt	$::= T\ m(\overline{T}\ \overline{x})\ \texttt{effect}\ \overline{M_1} \leadsto \overline{M_2}\ \{e\}$
simple masks	S	$::= f \mid \texttt{sub}_C$
masks	M	$::= S \mid S\,! \mid S\,[\overline{p.S_p}]$
paths	p	$::= \ell \mid x$
unmasked types	U	$::= \circ \mid C \mid C!$
types	T	$::= U \mid T\backslash M$
expressions	e	$::= (T\ p) \mid \texttt{new}\ C \mid e_1;\ e_2 \mid e.f$
		$\mid (T_1\ p_1).f = (T_2\ p_2) \mid (T_0\ p_0).m((\overline{T}\ \overline{p}))$
		$\mid \texttt{let}\ T\ x = e_1\ \texttt{in}\ e_2$
typing environments	Γ	$::= \emptyset \mid \Gamma, x\!:\!T \mid \Gamma, \ell\!:\!T$
heaps	H	$::= \emptyset \mid H, \ell \mapsto o$
objects	o	$::= C!\backslash\overline{M}\{\overline{f} = \overline{\ell}\}$
evaluation contexts	E	$::= [\cdot] \mid E.f \mid E;\ e \mid \texttt{let}\ T\ x = E\ \texttt{in}\ e$

Figure 2.7: Grammar

2.3 Formal semantics and soundness

We now formalize masked types as part of a simple object calculus. Unfortunately, previous object calculi are not suitable for modeling masked types.

2.3.1 Grammar

Figure 2.7 shows the grammar of the core J\mask calculus. We use the notation $\overline{a}$ for both the list $a_1, \ldots, a_n$ and the set $\{a_1, \ldots, a_n\}$, for $n \geq 0$. We abbreviate terms with list subterms in the obvious way, e.g., $\overline{T}\ \overline{x}$ stands for $T_1\ x_1, \ldots, T_n\ x_n$, $T\backslash\overline{M}$ stands for $T\backslash M_1\backslash\ldots\backslash M_n$, and $p.\overline{S}$ stands for $p.S_1, \ldots, p.S_n$.

A program Pr is a pair $< \overline{L}, e >$ of a set of class declarations L and an expression e (the `main` method). Each class C is declared with a superclass C', a set of field declarations $\overline{F}$ and a set of method declarations $\overline{Mt}$. To simplify presentation, all the class declarations are assumed to be global information.

J\mask only supports single inheritance. The root of the class hierarchy is denoted by $\circ$. We write $C \sqsubset C'$ to mean that class C is a direct subclass of C', and the relation $\sqsubset^*$ is the reflexive and transitive closure of $\sqsubset$.

Notably, there is no `null` value in the language, because none is needed for object initialization.

There are three kinds of masks: simple masks S, must-masks $S\,!$, and conditional masks $S\,[\overline{p.S_p}]$. The auxiliary function simple elides the must annotation and conditions of a mask.

$$\begin{aligned}\mathsf{simple}(S) &= S\\ \mathsf{simple}(S\,!) &= S\\ \mathsf{simple}(S\,[\overline{p.S_p}]) &= S\end{aligned}$$

There are two kinds of simple masks: concrete field masks f, and subclass masks sub_C, that is, `C.sub` in the J\mask language. The calculus does not explicitly model the abstract mask $\star$, because it can be expanded into a collection of field masks and a subclass mask. For the simplicity of the semantics, other abstract masks and mask constraints are omitted.

We require that in a well-formed type, no two masks mention the same field, and every variable appearing in a condition is in the domain of the typing environment. The order of masks in a type does not matter, so $T\backslash f_1\backslash f_2 = T\backslash f_2\backslash f_1$.

An unmasked type U is either a normal class type C or an *exact* class type $C!$. An object of $C!$ must be an instance of class C, and not of any proper subclass of C. (This overloads the "!" symbol, which is also used for must-masks.)

The source of exactly typed values is object creation, because the expression `new` C has type $C!$. Exact types are useful because they make removal of subclass masks possible, as discussed in Section 2.1.1.

An object is created with expression `new` C, which adds a fresh memory location to the heap, with all fields uninitialized. Uninitialized fields are not represented in the heap, so there is no need for `null`. Initialization is done by calling appropriate methods.

To simplify presentation of the semantics and the proof of soundness, we allow only paths p (local variables x at compile time, or heap locations ℓ at run time) to appear in field assignments and method calls. This does not restrict expressiveness, because of `let` expressions.

Every read through a path p is represented as an expression $(T\ p)$, where the annotation T is a statically known type. The annotation is primarily to make the proof of soundness easier; in the actual J\mask implementation, T is inferred by the compiler.

Typing environments Γ contain type bindings for both variables x and heap locations ℓ. Bindings for locations are extracted from the heap and are used to type-check expressions during evaluation.

The J\mask calculus models the heap as a function from memory locations l to objects o. The formalization attaches a type to every object on the heap, in addition to value bindings for the fields. The object type is always based on some exact class type, which is known at run time. The type might also have masks, and since the base class is always exact, no subclass mask may appear on the heap. Masks in the operational semantics are included only for

$$\frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \{\overline{F}\ \overline{Mt}\}}{\begin{array}{c}\mathsf{ownFields}(C) = \overline{F}\\ \mathsf{ownMethods}(C) = \overline{Mt}\end{array}}$$

$$\begin{aligned}\mathsf{fields}(C) &= \bigcup_{C'\,:\,C\sqsubset^* C'} \mathsf{ownFields}(C')\\ \mathsf{methods}(C) &= \bigcup_{C'\,:\,C\sqsubset^* C'} \mathsf{ownMethods}(C')\end{aligned}$$

$$\frac{\overline{F} = \overline{U}\ \overline{f}}{\mathsf{fnames}(\overline{F}) = \overline{f}}$$

Figure 2.8: Class member lookup

the soundness proof and can be erased in the implementation.

2.3.2 Class member lookup

Figure 4.7 shows auxiliary functions for looking up class members. For a class C, $\mathsf{ownFields}(C)$ and $\mathsf{ownMethods}(C)$ are the set of fields and methods declared in C itself, and $\mathsf{fields}(C)$ and $\mathsf{methods}(C)$ also collect those declared in all the superclasses of C. $\mathsf{fnames}(\overline{F})$ is the set of all the field names in field declarations $\overline{F}$. For simplicity, we assume no two fields have the same name.

2.3.3 Subtyping

Subtyping rules are defined in Figure 2.9. The judgment $\Gamma \vdash T_1 \leq T_2$ states that type T_1 is a subtype of T_2 in context Γ. The judgment $\Gamma \vdash T_1 \approx T_2$ is sugar for the pair of judgments $\Gamma \vdash T_1 \leq T_2$ and $\Gamma \vdash T_2 \leq T_1$.

Most subtyping rules are intuitive. S-COND-SUB states that adding condi-

$\boxed{\Gamma \vdash T \leq T'}$

$$\Gamma \vdash T \leq T \;(\text{S-REFL}) \qquad \frac{\Gamma \vdash T_1 \leq T_2 \qquad \Gamma \vdash T_2 \leq T_3}{\Gamma \vdash T_1 \leq T_3} \;(\text{S-TRANS}) \qquad \frac{\vdash C \sqsubset C'}{\Gamma \vdash C \leq C'} \;(\text{S-SUP}) \qquad \Gamma \vdash C! \leq C \;(\text{S-EXACT})$$

$$\frac{\Gamma \vdash T_1 \leq T_2}{\Gamma \vdash T_1 \backslash M \leq T_2 \backslash M} \;(\text{S-MASK}) \qquad \Gamma \vdash T \backslash S\,[\,] \approx T \;(\text{S-EMPTY-COND}) \qquad \Gamma \vdash T \backslash S\,[\overline{p}.\overline{S_p}] \leq T \backslash S\,[\overline{p}.\overline{S_p}, p'.S'] \;(\text{S-COND-SUB})$$

$$\frac{S = \mathsf{simple}(M)}{\Gamma \vdash T \backslash M \leq T \backslash S} \;(\text{S-SIMPLE}) \qquad \frac{\mathsf{sub}_C = \mathsf{simple}(M)}{\Gamma \vdash C! \backslash M \approx C!} \;(\text{S-EXACT-MASK}) \qquad \frac{p' : C! \backslash \overline{M} \in \Gamma}{\Gamma \vdash T \backslash S\,[\overline{p}.\overline{S_p}, p'.\mathsf{sub}_C] \approx T \backslash S\,[\overline{p}.\overline{S_p}]} \;(\text{S-EXACT-COND})$$

$$\frac{\vdash C \sqsubset C' \qquad \mathsf{fnames}(\mathsf{ownFields}(C)) = \overline{f} \qquad \mathsf{sub}_{C'} = \mathsf{simple}(M)}{\Gamma \vdash T \backslash M \approx T \backslash \mathsf{expand}(M, \{\overline{f}, \mathsf{sub}_C\})} \;(\text{S-SUBMASK})$$

$$\frac{\vdash C \sqsubset C' \qquad \mathsf{fnames}(\mathsf{ownFields}(C)) = \overline{f}}{\Gamma \vdash T \backslash M[p.\mathsf{sub}_{C'}, \overline{p'}.\overline{S}] \approx T \backslash M[p.\overline{f}, p.\mathsf{sub}_C, \overline{p'}.\overline{S}]} \;(\text{S-SUBMASK-COND})$$

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

$$\frac{p : T \in \Gamma}{\Gamma \vdash p : T} \;(\text{TP-PATH}) \qquad \frac{\Gamma \vdash \ell : T_1 \qquad \Gamma \vdash T_1 \leq T_2}{\Gamma \vdash \ell : T_2} \;(\text{TP-SUB}) \qquad \frac{\Gamma \vdash p : T \backslash f[p.f, \overline{p'}.\overline{S}]}{\Gamma \vdash p : T \backslash f[\overline{p'}.\overline{S}]} \;(\text{TP-COND-CYCLE})$$

$$\frac{\begin{array}{c}\Gamma \vdash p : T \backslash S\,[p'.f, \overline{p''}.\overline{S'}] \\ \Gamma \vdash p' : T' \qquad f \notin \mathsf{masked}(T')\end{array}}{\Gamma \vdash p : T \backslash S\,[\overline{p''}.\overline{S''}]} \;(\text{TP-COND-ELIM}) \qquad \frac{\begin{array}{c}\Gamma \vdash p : T \backslash S\,[p'.S', \overline{p''}.\overline{S''}] \\ \Gamma \vdash p' : T' \backslash S'[\overline{p'''}.\overline{S'''}]\end{array}}{\Gamma \vdash p : T \backslash S\,[\overline{p''}.\overline{S''}, \overline{p'''}.\overline{S'''}]} \;(\text{TP-COND-TRANS})$$

$\boxed{\Gamma \vdash_R e : T, \Gamma'}$

$$\frac{\begin{array}{c}\Gamma \vdash x : T \qquad x : T_x \in \Gamma \\ \Gamma' = \Gamma\{\!\{x : \mathsf{noMust}(T_x)\}\!\} \qquad \Gamma' \vdash \mathsf{noMust}(T) \leq T'\end{array}}{\Gamma \vdash_R (T\ x) : T', \Gamma'} \;(\text{TR-VAR}) \qquad \frac{\begin{array}{c}\Gamma \vdash \ell : T \qquad \Gamma \vdash T \leq T' \qquad \ell : T_\ell \in \Gamma \\ \Gamma' = \Gamma\{\!\{\ell : \mathsf{noMust}(T_\ell)\}\!\}\end{array}}{\Gamma \vdash_R (T\ \ell) : T', \Gamma'} \;(\text{TR-LOC})$$

$$\frac{\Gamma \vdash e_1 : T_1, \Gamma_1 \qquad \Gamma_1 \vdash_R e_2 : T_2, \Gamma_2}{\Gamma \vdash_R e_1;\ e_2 : T_2, \Gamma_2} \;(\text{TR-SEQ}) \qquad \frac{\Gamma \vdash e : T, \Gamma' \qquad e \neq (T\ x) \wedge e \neq e_1;\ e_2}{\Gamma \vdash_R e : T, \Gamma'} \;(\text{TR-OTHER})$$

$\boxed{\Gamma \vdash e : T, \Gamma'}$

$$\frac{\begin{array}{c}\Gamma \vdash e : T_1, \Gamma' \\ \Gamma \vdash T_1 \leq T_2\end{array}}{\Gamma \vdash e : T_2, \Gamma'} \;(\text{T-SUB}) \qquad \frac{\Gamma \vdash p : T}{\Gamma \vdash (T\ p) : T, \Gamma} \;(\text{T-PATH}) \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : T_1, \Gamma_1 \\ \Gamma_1 \vdash e_2 : T_2, \Gamma_2\end{array}}{\Gamma \vdash e_1;\ e_2 : T_2, \Gamma_2} \;(\text{T-SEQ}) \qquad \frac{\overline{f} = \mathsf{fnames}(\mathsf{fields}(C))}{\Gamma \vdash \mathtt{new}\ C : C! \backslash \overline{f}!, \Gamma} \;(\text{T-NEW})$$

$$\frac{\begin{array}{c}\Gamma \vdash_R e_1 : T, \Gamma_1 \qquad x \notin \mathsf{dom}(\Gamma_1) \\ \Gamma_1, x : T \vdash e_2 : T_2, \Gamma_2 \\ \Gamma_2 = \Gamma_2', x : T' \\ \Gamma_2'' = \mathsf{remove}(\Gamma_2', x)\end{array}}{\Gamma \vdash \mathtt{let}\ T\ x = e_1\ \mathtt{in}\ e_2 : T_2, \Gamma_2''} \;(\text{T-LET}) \qquad \frac{\begin{array}{c}\Gamma \vdash e : T, \Gamma' \\ T_f = \mathsf{ftype}(T, f)\end{array}}{\Gamma \vdash e.f : T_f, \Gamma'} \;(\text{T-GET}) \qquad \frac{\begin{array}{c}\Gamma \vdash (T_1\ p_1) : T_1, \Gamma \qquad T_1 \neq T_1' \backslash f! \\ \Gamma \vdash_R (T_2\ p_2) : \mathsf{ftype}(\mathsf{grant}(T_1, f), f), \Gamma' \\ p_1 : T \in \Gamma' \qquad \Gamma'' = \Gamma'\{\!\{p_1 : \mathsf{grant}(T, f)\}\!\}\end{array}}{\Gamma \vdash (T_1\ p_1).f = (T_2\ p_2) : \circ \backslash \mathsf{sub}_\circ, \Gamma''} \;(\text{T-SET})$$

$$\frac{\begin{array}{c}\Gamma \vdash (T_1 \backslash f!\ p_1) : T_1 \backslash f!, \Gamma \\ \Gamma \vdash (T_2\ p_2) : T_2, \Gamma \qquad T_2 = U_2 \backslash \overline{M} \\ \mathsf{ftype}(T_1, f) = U_f \backslash \overline{S_f} \qquad \Gamma \vdash U_2 \leq U_f \\ \overline{S} = \{S \mid S \in \mathsf{simple}(\overline{M}) \wedge (S! \in \overline{M} \vee S \notin \overline{S_f})\} \\ p_1 : T \backslash f! \in \Gamma \qquad \Gamma' = \Gamma\{\!\{p_1 : T \backslash f[p_2.\overline{S}]\}\!\}\end{array}}{\Gamma \vdash (T_1 \backslash f!\ p_1).f = (T_2\ p_2) : \circ \backslash \mathsf{sub}_\circ, \Gamma'} \;(\text{T-SET-COND})$$

$$\frac{\begin{array}{c}\Gamma \vdash (T_0\ p_0) : T_0, \Gamma \qquad T_0 = U \backslash \overline{M} \qquad p_0 : U_0 \backslash \overline{M'} \in \Gamma \\ \mathsf{mbody}(T_0, m) = T'_{n+1}\ m(\overline{T'}\ \overline{x})\ \mathtt{effect}\ \overline{M_1} \rightsquigarrow \overline{M_2}\ \{e\} \\ \Gamma \vdash T_0 \leq U \backslash \overline{M_1}\{p_0/\mathtt{this}\}\{\overline{p}/\overline{x}\} \\ \forall i \in 1..n+1.\ T_i'' = T_i'\{p_0/\mathtt{this}\}\{\overline{p}/\overline{x}\} \\ \forall i \in 1..n.\ \Gamma \vdash (T_i\ p_i) : T_i'', \Gamma \\ \forall i \in 0..n.\ T_i = T''' \backslash S! \Rightarrow (T_i'' = T'''' \backslash S! \wedge \forall j \neq i.\ p_i \neq p_j) \\ \Gamma' = \Gamma\{\!\{p_0 : \mathsf{update}(p_0, \overline{M'}, U_0 \backslash \overline{M_2}\{p_0/\mathtt{this}\}\{\overline{p}/\overline{x}\})\}\!\}\end{array}}{\Gamma \vdash (T_0\ p_0).m((\overline{T}\ \overline{p})) : T'_{n+1}, \Gamma'} \;(\text{T-CALL})$$

Figure 2.9: Static semantics

$$\begin{array}{rcl}
\text{masked}(U) &=& \emptyset \\
\text{masked}(T \backslash S\,!) &=& \text{masked}(T \backslash S) \\
\text{masked}(T \backslash S\,[\overline{p.S_p}]) &=& \text{masked}(T \backslash S) \\
\text{masked}(T \backslash f) &=& \{f\} \cup \text{masked}(T) \\
\text{masked}(T \backslash \text{sub}_C) &=& \text{masked}(T)
\end{array}
\qquad
\begin{array}{rcl}
\text{class}(C) &=& C \\
\text{class}(C!) &=& C \\
\text{class}(T \backslash M) &=& \text{class}(T)
\end{array}$$

$$\frac{\begin{array}{c} C = \text{class}(T) \\ f \notin \text{masked}(T) \\ \text{fields}(C) = \overline{F} \\ F_i = T_f\ f \end{array}}{\text{ftype}(T, f) = T_f}
\qquad
\frac{\begin{array}{c} C = \text{class}(T) \qquad C \sqsubset C' \\ Mt = \ldots\ m(\ldots)\ \ldots \\ \left(\begin{array}{l} Mt \in \text{ownMethods}(C) \vee \\ Mt \notin \text{ownMethods}(C) \wedge \\ \text{mbody}(C', m) = Mt \end{array}\right) \end{array}}{\text{mbody}(T, m) = Mt}$$

$$\begin{array}{rcl}
\text{noMust}(U) &=& U \\
\text{noMust}(T \backslash M) &=& \begin{cases} \text{noMust}(T) \backslash S & \text{if } M = S\,! \\ \text{noMust}(T) \backslash M & \text{otherwise} \end{cases}
\end{array}
\qquad
\text{grant}(T, f) = \begin{cases} T' & \text{if } T = T' \backslash f \\ T' & \text{if } T = T' \backslash f[\overline{p.S}] \\ T' & \text{if } T = T' \backslash f! \\ T & \text{otherwise} \end{cases}$$

$$\begin{array}{rcl}
\text{remove}(\emptyset, x) &=& \emptyset \\
\text{remove}((\Gamma, p : T), x) &=& \text{remove}(\Gamma, x), p : \text{remove}(T, x) \\
\text{remove}(U, x) &=& U \\
\text{remove}(T \backslash S\,[x.S_x, \ldots], x) &=& \text{remove}(T, x) \backslash S
\end{array}
\qquad
\begin{array}{rcl}
\text{update}(x, \overline{M}, T) &=& T \\
\text{update}(\ell, \overline{M}, U) &=& U \\
\text{update}(\ell, \overline{M}, T \backslash M') &=& \begin{cases} \text{update}(\ell, \overline{M}, T) \backslash M' & \text{if } M_i = \text{simple}(M')! \\ \text{update}(\ell, \overline{M}, T) \backslash M_i & \text{if simple}(M_i) = \text{simple}(M') \\ \text{update}(\ell, \overline{M}, T) & \text{otherwise} \end{cases}
\end{array}$$

Figure 2.10: Auxiliary definitions

tions makes a conditional mask more conservative. S-SIMPLE states that a type with a must-mask or a conditional mask is a subtype of the corresponding simply masked type.

The subtyping rule S-SUBMASK uses an auxiliary function expand, which expands a mask S into a set of masks $\overline{S'}$, while preserving any annotation on S:

$$\begin{array}{rcl}
\text{expand}(S, \overline{S'}) &=& \overline{S'} \\
\text{expand}(S\,!, \overline{S'}) &=& \overline{S'}! \\
\text{expand}(S\,[\overline{p.S_p}], \overline{S'}) &=& \overline{S'}[\overline{p.S_p}]
\end{array}$$

As shown in Figure 2.9, there are often a number of different ways of writing equivalent types. The five type equivalence rules (S-EMPTY-COND, S-EXACT-MASK, S-EXACT-COND, S-SUBMASK, and S-SUBMASK-COND) can be read as normalization rules, where the types on the left-hand side of $\approx$ are reduced to those on the right-hand side. Note that in each of the five rules, the type on the right-hand side is either syntactically simpler than that on the left-hand side, or converts an occurrence of a class on the left-hand side to its subclass. This ensures type normalization terminates. Normalized types have the following characteristics:

- A type $C\backslash\overline{M}$ has at most one subclass mask, which must be sub_C. A type $C!\backslash\overline{M}$ has no subclass mask.
- The condition $p.\mathsf{sub}_C$ does not show up if the path p has an exact type.
- Conditional masks have non-empty conditions.

For convenience of presentation, from now on, types are assumed to be in normal form, unless otherwise noted.

2.3.4 Expression typing

In the J\mask language, the evaluation of an expression might update some type bindings. For example, initializing a field removes the mask on that field, if there is one. Therefore, typing judgments, shown in Figure 2.9, are of the form $\Gamma \vdash e : T, \Gamma'$, where Γ' is the typing environment after evaluating e. We write $\Gamma\{\!\{p : T\}\!\}$ for environment Γ with the type binding of p updated to T.

There are two other kinds of judgments in Figure 2.9. The judgment $\Gamma \vdash p : T$ types a path p without updating the typing environment. The subsumption rule TP-SUB is limited to locations l, not any variables x, to ensure that the expression $(T\ \ x)$ has the most precise type annotation T (see T-PATH and TR-VAR).

The judgment $\Gamma \vdash_R e : T, \Gamma'$ is used in T-LET and T-SET for typing the right-hand side of assignment, and in M-OK for typing the return expression (see Section 2.3.5). It avoids creating aliases for variables with type bindings that have must-masks. However, aliases are allowed if they are created with conditional masks, as shown in T-SET-COND, where no TR- rule is used.

Figure 4.9 defines auxiliary functions used in the typing rules. Most of them are self-explanatory. The function update, used in T-CALL, updates the type binding of the receiver according to the effect, and ensures monotonicity if the receiver is a location.

J\mask has several expression well-formedness rules, written $\vdash e$ `wf`, shown in Figure 2.11. The important rule is LET-WF, which imposes two requirements on `let` expressions:

- A `let` expression cannot end with a variable bound outside the scope of the `let`. For example, one cannot write **let** $T\ x = e_1$ **in** $(e_2;\ y)$ where y is free in the `let` expression, but rather the equivalent expression (**let** $T\ x = e_1$ **in** e_2); y. This helps simplify type-checking of right-hand sides of assignments ($\Gamma \vdash_R e : T, \Gamma'$), so that a separate TR-LET is not necessary.
- If the variable x is bound to a location already in the scope of the `let` expression, the declared type of x cannot have any must-mask. This prevents x from being an alias with must-masks.

The expression well-formedness rules help simplify the proof of the substitution lemma (Lemma 2.3.10), without limiting the expressiveness of the calculus.

$$\frac{\begin{array}{c}\vdash e_1\ \texttt{wf} \qquad \vdash e_2\ \texttt{wf} \\ \forall x' \in \mathsf{FV}(\texttt{let}\ T\ x = e_1\ \texttt{in}\ e_2).\ e_2 \neq x' \wedge e_2 \neq e';\ x' \\ ((e_1 = (T_\ell\ \ell) \vee e_1 = e'';\ (T_\ell\ \ell)) \wedge \ell \in \mathsf{locs}(e_2)) \Rightarrow T \neq T'\backslash S\,!\end{array}}{\vdash \texttt{let}\ T\ x = e_1\ \texttt{in}\ e_2\ \texttt{wf}} \quad \text{(LET-WF)}$$

$$\frac{\vdash e_1\ \texttt{wf} \qquad \vdash e_2\ \texttt{wf}}{\vdash e_1;\ e_2\ \texttt{wf}} \quad \text{(SEQ-WF)}$$

$$\frac{\vdash e\ \texttt{wf}}{\vdash e.f\ \texttt{wf}} \quad \text{(GET-WF)}$$

$$\frac{e \neq \texttt{let}\ T\ x = e_1\ \texttt{in}\ e_2 \qquad e \neq e_1;\ e_2 \qquad e \neq e'.f}{\vdash e\ \texttt{wf}} \quad \text{(OTHER-WF)}$$

Figure 2.11: Well-formed expressions

2.3.5 Program typing

Figure 2.12 shows the rules for checking the well-formedness of field and method declarations in a class C.

For a field declaration, the declared type may not use must-masks or conditional masks.

For a method declaration, the special variable `this` is assumed to have the precondition masks $\overline{M_1}$ at the entry point of the method, and it must be typable with the postcondition masks $\overline{M_2}$ when the method exits. Method parameters other than the receiver should remain typable with the same types at the entry. J\mask permits effects on other parameters, but for simplicity, the calculus does not support this feature. M-OK also specifies some constraints on the method effect: it cannot introduce must-masks, which is only allowed with the `new` expression; a mask in the precondition that is not a must-mask can only be replaced with a corresponding mask that is more conservative.

$$\frac{T = U\backslash\overline{S}}{C \vdash T\ f\ \mathsf{ok}} \tag{F-OK}$$

$$\frac{\begin{array}{c} \vdash e\ \mathtt{wf} \\ \Gamma = \mathtt{this}\!:\!C\backslash\overline{M_1}, \overline{x}\!:\!\overline{T} \\ \Gamma \vdash_R e\!:\!T_r, \Gamma_r \\ \Gamma_r \vdash \mathtt{this}\!:\!C\backslash\overline{M_2} \\ \Gamma_r \vdash \overline{x}\!:\!\overline{T} \\ S\,! \in \overline{M_2} \Rightarrow S\,! \in \overline{M_1} \\ \left(\begin{array}{l} M \in \overline{M_1} \wedge M' \in \overline{M_2} \wedge M \neq S\,! \\ \wedge \mathsf{simple}(M) = \mathsf{simple}(M') \end{array}\right) \Rightarrow\ \ \vdash C\backslash M \leq C\backslash M' \end{array}}{C \vdash T_r\ m(\overline{T}\ \overline{x})\ \mathtt{effect}\ \overline{M_1} \rightsquigarrow \overline{M_2}\ \{e\}\ \mathsf{ok}} \tag{M-OK}$$

Figure 2.12: Program typing

2.3.6 Decidability of type checking

The type system of J\mask is decidable:

- For T-SUB and TP-SUB, we disallow the use of reflexivity of subtyping, and require all the rules about type equivalence ($\approx$) to be used in the direction of normalization (see Section 2.3.3).
- The three rules TP-COND-CYCLE, TP-COND-ELIM, and TP-COND-TRANS actually characterize a graph-theoretic reachability problem on the dependency graph (such as in Figure 2.4), which can be solved with depth-first search.

All other rules are syntax-directed. Therefore, type checking is decidable for J\mask.

2.3.7 Operational semantics

Figure 2.13 shows the judgments for the small-step operational semantics of J\mask, where $e, H \longrightarrow e', H'$ means that expression e and heap H step to expression e' and heap H'.

Most of the rules in Figure 2.13 are standard, and the notable ones are those for field assignments (R-SET and R-SET-COND), which are similar to the corresponding expression typing rules (T-SET and T-SET-COND).

In the operational semantics and in the soundness proof, typing environments are extracted from the heap, represented as $\lfloor H \rfloor$:

$$\lfloor \emptyset \rfloor = \emptyset$$
$$\lfloor H, \ell \mapsto T\ \{\overline{f} = \overline{\ell}\} \rfloor = \lfloor H \rfloor, \ell : T$$

The notation $H\{\!\{\ell := o\}\!\}$ means that the value binding of ℓ in the heap H is updated to another object o.

Figure 2.14 shows the heap typing rules. A heap H is well-formed, written $\vdash H$, if every field that is not masked in its container's type is bound to a location, and that location can be given a type compatible with the declared type of the field.

In H-LOC, $H(\ell, f)$ refers to the value binding of the field f of the object stored in $H(\ell)$.

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

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

$$\mathtt{let}\ T\ x = (T_\ell\ \ell)\ \mathtt{in}\ e, H \longrightarrow e\{\ell/x\}, H \qquad \text{(R-LET)}$$

$$\frac{H(\ell) = T\ \{\overline{f} = \overline{\ell}\} \qquad T_i = \mathsf{ftype}(T, f_i)}{(T_\ell\ \ell).f_i, H \longrightarrow (T_i\ \ell_i), H} \qquad \text{(R-GET)}$$

$$\frac{\begin{array}{c} H(\ell) = T\ \{\overline{f} = \overline{\ell}\} \qquad T_\ell \neq T' \backslash f! \\ H' = H\{\!\{\ell := \mathsf{grant}(T, f)\ \{\ldots, f = \ell'\}\}\!\} \end{array}}{(T_\ell\ \ell).f = (T'_\ell\ \ell'), H \longrightarrow (\circ \backslash \mathsf{sub}_\circ\ \ell'), H'} \qquad \text{(R-SET)}$$

$$\frac{\begin{array}{c} H(\ell) = T \backslash f!\ \{\overline{f} = \overline{\ell}\} \qquad \mathsf{ftype}(T, f) = U_f \backslash \overline{S_f} \\ \overline{S} = \{S \mid S \in \mathsf{simple}(\overline{M}) \wedge (S! \in \overline{M} \vee S \notin \overline{S_f})\} \\ H' = H\{\!\{\ell := T \backslash f[\ell'.\overline{S}]\ \{\ldots, f = \ell'\}\}\!\} \end{array}}{(T_\ell \backslash f!\ \ell).f = (U \backslash \overline{M}\ \ell'), H \longrightarrow (\circ \backslash \mathsf{sub}_\circ\ \ell'), H'} \qquad \text{(R-SET-COND)}$$

$$\frac{\mathsf{mbody}(T_0, m) = T_r\ m(\overline{T_x}\ \overline{x})\ \ldots\ \{e\}}{(T_0\ \ell_0).m(\overline{(T\ \ell)}), H \longrightarrow e\{\ell_0/\mathtt{this}\}\{\overline{\ell}/\overline{x}\}, H} \qquad \text{(R-CALL)}$$

$$\frac{\begin{array}{c} \ell \notin \mathrm{dom}(H) \qquad \mathsf{fnames}(\mathsf{fields}(C)) = \overline{f} \\ H' = H, \ell \mapsto C! \backslash \overline{f}!\{\} \end{array}}{\mathtt{new}\ C, H \longrightarrow (C! \backslash \overline{f}!\ \ell), H'} \qquad \text{(R-ALLOC)}$$

$$(T\ \ell);\ e, H \longrightarrow e, H \qquad \text{(R-SEQ)}$$

Figure 2.13: Small-step operational semantics

$$\frac{\begin{array}{c} \ell : C! \backslash \overline{M} \in \lfloor H \rfloor \\ \overline{f} = \mathsf{fnames}(\mathsf{fields}(C)) \\ \lfloor H \rfloor \vdash \ell : T \\ \forall f \in \overline{f}. \left(\begin{array}{l} f \notin \mathsf{masked}(T) \Rightarrow \\ H(\ell, f) = \ell' \wedge \lfloor H \rfloor \vdash \ell' : \mathsf{ftype}(T, f) \end{array} \right) \end{array}}{H \vdash \ell} \qquad \text{(H-LOC)}$$

$$\frac{\forall \ell \in \mathrm{dom}(H).\ H \vdash \ell}{\vdash H} \qquad \text{(HEAP-WF)}$$

Figure 2.14: Well-formed heaps

2.3.8 Type safety

The soundness theorem of the J\mask calculus states that if an expression e is well-typed, and it can reduce to a value $(T_\ell\ \ell)$, then $(T_\ell\ \ell)$ has the same type as e. A corollary of this theorem is that object initialization is sound in the sense used elsewhere in the paper: if a program tried to read an uninitialized field, the evaluation would get stuck according to R-GET.

Theorem 2.3.1 *(Soundness) If $\vdash e\ \texttt{wf}$, and $\vdash e : T$, and $e, \emptyset \rightarrow^* (T_\ell\ \ell), H$, then $\lfloor H \rfloor \vdash (T_\ell\ \ell) : T$.*

The proof uses the standard technique of proving subject reduction and progress [87], shown in Section 2.3.

2.3.9 Proof of soundness

Extensions of typing environments. The definition of extensions is given in Section 2.3.8. In order to prove Lemma 2.3.4, we first prove the following simple lemma:

Lemma 2.3.2 *If Γ_2 is an extension of Γ_1, and $\Gamma_1 \vdash T_1 \leq T_2$, then $\Gamma_2 \vdash T_1 \leq T_2$.*

PROOF: The proof is by induction on the derivation of $\Gamma_1 \vdash T_1 \leq T_2$. Most cases are simple, because all the subtyping rules, except for T-EXACT-COND, do not depend on the typing environment.

For S-EXACT-COND, $T_1 = T\backslash S[\overline{p}.\overline{S_p}, p'.\mathsf{sub}_C]$, $T_2 = T\backslash S[\overline{p}.\overline{S_p}]$, and $p' : C!\backslash\overline{M} \in \Gamma_1$. By the definition of environment extensions, $\Gamma_2 \vdash p' : C!\backslash\overline{M}$, and therefore $p' : C!\backslash\overline{M'} \in \Gamma_2$, because of the exactness of $C!$. Thus S-EXACT-COND can apply, and $\Gamma_2 \vdash T_1 \leq T_2$. □

Lemma 2.3.3 *If Γ_2 is an extension of Γ_1, and $\Gamma_1 \vdash p : T$, then $\Gamma_2 \vdash p : T$.*

PROOF: By induction on the derivation of $\Gamma_1 \vdash p : T$.

- TP-PATH

  Then $\Gamma_1 \vdash p : T$. By the definition of extensions, $\Gamma_2 \vdash p : T$. So $\Gamma_2 \vdash p : T$ by TP-PATH.

- TP-SUB

  Then $p = \ell$, and $\Gamma_1 \vdash p : T'$, and $\Gamma_1 \vdash T' \leq T$. By the induction hypothesis, $\Gamma_2 \vdash p : T'$. By Lemma 2.3.2, $\Gamma_2 \vdash T' \leq T$. Thus by TP-SUB, $\Gamma_2 \vdash p : T$.

- TP-COND-CYCLE, TP-COND-ELIM, and TP-COND-TRANS

  Apply the induction hypothesis on the premises. Then the respective path typing rules can apply to Γ_2.

□

Lemma 2.3.4 *If Γ_2 is an extension of Γ_1, and $\Gamma_1 \vdash e : T, \Gamma_1'$, then $\Gamma_2 \vdash e : T, \Gamma_2'$, and Γ_2' is an extension of Γ_1'.*

PROOF: By induction on the derivation of $\Gamma_1 \vdash e : T, \Gamma_1'$.

- T-SUB

  Then there is a type T' such that $\Gamma_1 \vdash e : T', \Gamma_1'$, and $\Gamma_1 \vdash T' \leq T$. By the induction hypothesis, $\Gamma_2 \vdash e : T', \Gamma_2'$, and Γ_2' is an extension of Γ_1'. By Lemma 2.3.2, $\Gamma_2 \vdash T' \leq T$. Then it follows that $\Gamma_2 \vdash e : T, \Gamma_2'$.

- T-PATH

  Then $e = (T\ p)$, and $\Gamma_1 \vdash p : T$, and $\Gamma_1' = \Gamma_1$. By Lemma 2.3.3, $\Gamma_2 \vdash p : T$, Thus $\Gamma_2 \vdash (T\ p) : T, \Gamma_2$, by T-PATH.

- T-NEW

  Trivial since the rule does not depend on the typing environment.

- T-SEQ

  Then $e = e_1;\ e_2$, and $\Gamma_1 \vdash e_1 : T_1, \Gamma_1''$, and $\Gamma_1'' \vdash e_2 : T, \Gamma_1'$. By the induction hypothesis, $\Gamma_2 \vdash e_1 : T_1, \Gamma_2''$, and Γ_2'' is an extension of Γ_1''. Again, by the induction hypothesis, $\Gamma_2'' \vdash e_2 : T, \Gamma_2'$, and Γ_2' is an extension of Γ_1'. By T-SEQ, $\Gamma_2 \vdash e : T, \Gamma_2'$.

- T-GET

  Then $e = e_1.f$, and $\Gamma_1 \vdash e_1 : T_1, \Gamma_1'$, and $T = \mathsf{ftype}(T_1, f)$. By the induction hypothesis, $\Gamma_2 \vdash e_1 : T_1, \Gamma_2'$, and Γ_2' is an extension of Γ_1'. Thus by T-GET, $\Gamma_2 \vdash e : T, \Gamma_2'$.

- T-SET

  Then $e = (T_1\ p_1).f = (T_2\ p_2)$, and $\Gamma_1 \vdash (T_1\ p_1) : T_1, \Gamma_1$, and $\Gamma_1 \vdash_R (T_2\ p_2) : \mathsf{ftype}(\mathsf{grant}(T_1, f), f), \Gamma_1''$, and $\Gamma_1' = \Gamma_1''\{\!\{p_1 : \mathsf{grant}(T_{p_1}, f)\}\!\}$ where $p_1 : T_{p_1} \in \Gamma_1''$, and $T = \circ\backslash\mathsf{sub}_\circ$. By the induction hypothesis, $\Gamma_2 \vdash (T_1\ p_1) : T_1, \Gamma_2$. It only remains to prove $\Gamma_2 \vdash_R (T_2\ p_2) : \mathsf{ftype}(\mathsf{grant}(T_1, f), f), \Gamma_2''$, and Γ_2'' is an extension of Γ_1''. Then it follows easily that $\Gamma_2' = \Gamma_2''\{\!\{p_1 : \mathsf{grant}(T_{p_1}', f)\}\!\}$ is an extension of Γ_1', where $p_1 : T_{p_1}' \in \Gamma_2''$. There are two cases for the derivation of $\Gamma_1 \vdash_R (T_2\ p_2) : \mathsf{ftype}(\mathsf{grant}(T_1, f), f), \Gamma_1''$:

- TR-VAR

  Then $p_2 = x$, and $\Gamma_1 \vdash x : T_2$. By Lemma 2.3.3, $\Gamma_2 \vdash x : T_2$. By the definition of extensions, the type binding of x is the same in Γ_1 and Γ_2. Therefore $\Gamma_2'' = \Gamma_2\{\!\{x : \mathsf{noMust}(T_2)\}\!\}$ is still an extension of $\Gamma_1'' = \Gamma_1\{\!\{x : \mathsf{noMust}(T_2)\}\!\}$. Then the proof easily follows.

- TR-LOC

  The proof follows from Lemma 2.3.3, and the fact that removing must annotations from Γ_1 and Γ_2 does not change the extension relationship.

• T-SET-COND

Then $e = (T_1\backslash f!\ p_1).f = (T_2\ p_2)$, and $T_2 = U_2\backslash\overline{M}$, and $\mathsf{ftype}(T_1, f) = U_f\backslash\overline{S_f}$, and $\Gamma_1 \vdash U_2 \leq U_f$, and $\overline{S} = \{S | S \in \mathsf{simple}(\overline{M}) \wedge (S! \in \overline{M} \vee S \notin \overline{S_f})\}$, and $\Gamma_1' = \Gamma_1\{\!\{p_1 : T'\backslash f[p_2.\overline{S}]\}\!\}$, where $p_1 : T'\backslash f! \in \Gamma_1$. We can apply the induction hypothesis to all the typing judgments in the premises. By Lemma 2.3.2, $\Gamma_2 \vdash U_2 \leq U_f$. Note that the set $\overline{S}$ does not depend on the typing environment. Now it only remains to prove that Γ_2' is an extension of Γ_1'. There are again two cases:

- $p_1 = x$

  Then by the definition of extensions, $x : T'\backslash f! \in \Gamma_2$. $\Gamma_2' = \Gamma_2\{\!\{x : T'\backslash f[p_2.\overline{S}]\}\!\}$. Thus Γ_2' is an extension of Γ_1'.

- $p_1 = \ell$

Then by the definition of extension, $\ell : T'' \in \Gamma_2$, and $\Gamma_2 \vdash T'' \leq T'\backslash f!$. Then it must be the case that $T'' = T'''\backslash f!$ for some T''' such that $\Gamma_2 \vdash T''' \leq T'$, and therefore $\Gamma_2' = \Gamma_2\{\!\{\ell : T'''\backslash f[p_2.\overline{S}]\}\!\}$ and $\Gamma_2' \vdash T''' \leq T'$. Then it follows that $\Gamma_2' \vdash T'''\backslash f[p_2.\overline{S}] \leq T'\backslash f[p_2.\overline{S}]$. Thus Γ_2' is an extension of Γ_1'.

- T-CALL

  Since the premise only uses typing judgments with the same typing environment, we can just apply the induction hypothesis, and apply T-CALL again on Γ_2. It only remains to prove that Γ_2' is an extension of Γ_1', and there are two cases for the receiver ($T_0\ p_0$):

  - $p_0 = x$

    By the definition of extensions, the variable x has the same type binding in Γ_1 and Γ_2. Then according to the definition of update, it still has the same type binding in Γ_1' and Γ_2', and therefore Γ_2' is an extension of Γ_1'.

  - $p_0 = \ell$

    Suppose $\ell : T_1 \in \Gamma_1$, and $\ell : T_2 \in \Gamma_2$, then $\Gamma_2 \vdash T_2 \leq T_1$ by the definition of extensions. Then according to the definition of update, some masks are removed and some must-masks are updated, simultaneously for the type bindings T_1' and T_2' of ℓ in Γ_1' and Γ_2'. By inspecting all the subtyping rules, we can see that $\Gamma_2' \vdash T_2' \leq T_1'$. Then Γ_2' is still an extension of Γ_1'.

- T-LET

  Then $e = \texttt{let}\ T_x\ x = e_1\ \texttt{in}\ e_2$, and $\Gamma_1 \vdash_R e_1 : T_x, \Gamma_1''$, and $\Gamma_1'', x : T_x \vdash e_2 : T, \Gamma_1'''$, and $\Gamma_1''' = \Gamma_1'''', x : T_x'$, and $\Gamma_1' = \mathsf{remove}(\Gamma_1'''', x)$. The only hard part is to prove $\Gamma_2 \vdash_R e_1 : T_x, \Gamma_2''$, and Γ_2'' is an extension of Γ_1''. The rest can be proved simply by using the induction hypothesis. The proof is by induction on the derivation of $\Gamma_1 \vdash_R e_1 : T_x, \Gamma_1''$.

– TR-VAR

  The proof is similar to the corresponding sub-case for T-SET.

– TR-LOC

  The proof is similar to the corresponding sub-case for T-SET.

– TR-SEQ

  Then $e_1 = e_1';\ e_2'$. Simply apply the outer induction hypothesis with e_1', and the inner induction hypothesis with e_2', and then the proof follows.

– TR-OTHER

  Then $\Gamma_1 \vdash e_1 : T_x, \Gamma_1''$, and by the outer induction hypothesis, $\Gamma_2 \vdash e_1 : T_x, \Gamma_2''$, and Γ_2'' is an extension of Γ_1''. Then by TR-OTHER, $\Gamma_2 \vdash_R e_1 : T_x, \Gamma_2''$.

□

Preservation of subtyping.

Lemma 2.3.5 *If $\lfloor H \rfloor \vdash T_1 \leq T_2$, and $e, H \longrightarrow e', H'$, then $\lfloor H' \rfloor \vdash T_1 \leq T_2$.*

PROOF: The proof is by induction on the derivation of $\lfloor H \rfloor \vdash T_1 \leq T_2$. Most of the cases are obvious, because all the subtyping rules, except S-EXACT-COND, do not depend on the typing environment. For the case of S-EXACT-COND, it only requires that $\lfloor H' \rfloor$ preserves the exact base class of each location in $\lfloor H \rfloor$, which is obvious since the rules in the operational semantics can only change masks of the type bindings. □

Location typing.

Lemma 2.3.6 *If $\Gamma \vdash e : T, \Gamma'$, and $\ell : T_\ell \in \Gamma$, and $\ell : T'_\ell \in \Gamma'$, and $T_\ell \neq T' \backslash S\,!$ for any T', then $T'_\ell \neq T' \backslash S\,!$ for any T'.*

PROOF: The proof is by induction on the derivation of $\Gamma \vdash e : T, \Gamma'$. In any of the typing rules, must annotations (or even the mask itself) in the typing environment can only be removed, e.g., TR-VAR, TR-LOC, T-SET, and T-SET-COND. The only notable case is T-CALL, where the type binding of the receiver is updated according to the effect clause. However, by M-OK, the effect clause cannot introduce new must-masks. □

Lemma 2.3.7 *If $\ell : T_\ell \in \Gamma$, and $\Gamma \vdash \ell : T$, then*

- $\Gamma\{\!\{\ell : \mathsf{grant}(T_\ell, f)\}\!\} \vdash \ell : \mathsf{grant}(T, f)$;
- $\Gamma\{\!\{\ell : \mathsf{noMust}(T_\ell)\}\!\} \vdash \ell : \mathsf{noMust}(T)$;
- $\mathsf{remove}(\Gamma, x) \vdash \ell : \mathsf{remove}(T, x)$;
- $(T_\ell = T'_\ell \backslash M \wedge \forall T'.\ T \neq T' \backslash \mathsf{simple}(M)!) \Rightarrow \Gamma\{\!\{\ell : T'_\ell\}\!\} \vdash \ell : T$;
- $(T_\ell = T'_\ell \backslash S\,! \wedge \forall T'.\ T \neq T' \backslash S\,!) \Rightarrow (T = T'' \backslash S \wedge \Gamma\{\!\{\ell : T'_\ell \backslash S[\ldots]\}\!\} \vdash \ell : T \wedge \Gamma\{\!\{\ell : T'_\ell \backslash f\}\!\} \vdash \ell : T)$;

- $(T_\ell = T'_\ell \backslash S\,! \land T = T' \backslash S\,! \land S = \mathsf{simple}(M)) \Rightarrow \Gamma\{\!\{\ell : T'_\ell \backslash M\}\!\} \vdash \ell : T' \backslash M$.

PROOF: The proof is by induction on the derivation of $\Gamma \vdash \ell : T$. □

Substitutions.

Lemma 2.3.8 *If* $\Gamma = \Gamma', \ell : T_\ell, x : T_x$, *and* $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash \ell : T_\ell\{\ell/x\}$, *and* $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash \ell : T_x\{\ell/x\}$, *and* $\Gamma \vdash T_1 \leq T_2$, *then* $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash T_1\{\ell/x\} \leq T_2\{\ell/x\}$.

PROOF: The proof is by induction on the derivation of $\Gamma \vdash T_1 \leq T_2$. The only notable case is S-EXACT-COND: if $T_\ell = C!\backslash\overline{M_\ell}$ or $T_x = C!\backslash\overline{M_x}$, then it is easy to see that $T'_\ell = C!\backslash\overline{M'_\ell}$, and S-EXACT-COND can still apply after the substitution. □

Lemma 2.3.9 *If* $\Gamma = \Gamma', \ell : T_\ell, x : T_x$, *and* $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash \ell : T_\ell\{\ell/x\}$, *and* $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash \ell : T_x\{\ell/x\}$, *and* $\Gamma \vdash p : T$, *then* $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash p\{\ell/x\} : T\{\ell/x\}$.

PROOF: By induction on the derivation of $\Gamma \vdash p : T$.

- TP-PATH

  Then $p : T \in \Gamma$. Consider the following three cases:

  - $p = x$

    Then $T_x = T$ and $\ell = p\{\ell/x\}$. Therefore by the assumption, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash p\{\ell/x\} : T\{\ell/x\}$.

- $p = \ell$

  Then $T_\ell = T$ and $\ell = p\{\ell/x\}$. Therefore by the assumption, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash p\{\ell/x\} : T\{\ell/x\}$.

- $p \neq x$ and $p \neq \ell$

  Then $p : T \in \Gamma'$, and therefore $p : T\{\ell/x\} \in \Gamma'\{\ell/x\}$. By TP-PATH, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash p : T\{\ell/x\}$.

- TP-SUB

  Then $p = \ell'$, which is not necessarily the same as ℓ, and there exists a type T' such that $\Gamma \vdash \ell' : T'$ and $\Gamma \vdash T' \leq T$. By the induction hypothesis, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash p\{\ell/x\} : T'\{\ell/x\}$. By Lemma 2.3.8, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash T'\{\ell/x\} \leq T\{\ell/x\}$. Thus by TP-SUB, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash p\{\ell/x\} : T\{\ell/x\}$.

- TP-COND-CYCLE

  Then $T = T' \backslash f[\overline{p'.S}]$, and $\Gamma \vdash p : T' \backslash f[p.f, \overline{p'.S}]$, and by the induction hypothesis, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash p\{\ell/x\} : T' \backslash f[p.f, \overline{p'.S}]\{\ell/x\}$. Thus by TP-COND-CYCLE, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash p\{\ell/x\} : T\{\ell/x\}$.

- TP-COND-ELIM

  Then $T = T' \backslash S[\overline{p''.S''}]$, and $\Gamma \vdash p : T' \backslash S[p'.f, \overline{p''.S''}]$, and $\Gamma \vdash p' : T''$, where $f \notin \mathsf{masked}(T'')$. By the induction hypothesis, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash p\{\ell/x\} : T' \backslash S[p'.f, \overline{p''.S''}]\{\ell/x\}$, and $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash p'\{\ell/x\} : T''\{\ell/x\}$. Note that $T' \backslash S[p'.f, \overline{p''.S''}]\{\ell/x\} = T'\{\ell/x\} \backslash S[p'\{\ell/x\}.f, \overline{p''\{\ell/x\}.S''}]$, and $f \notin \mathsf{masked}(T''\{\ell/x\})$. Thus by TP-COND-ELIM, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash p\{\ell/x\} : T\{\ell/x\}$.

- TP-COND-TRANS

  Similar to the proof of the above case for TP-COND-ELIM.

□

Lemma 2.3.10 shows that substituting a location for a variable preserves typing. It is used in the proof of Lemma 2.3.13 for method calls and `let` expressions. Before stating the substitution lemma, we first define substitution for typing environments:

An environment Γ' is the result of substituting a location ℓ of type T for a variable x in Γ, written $\Gamma' = \Gamma\{\!\{\ell/x;\ \ell : T\}\!\}$, if $\Gamma = \Gamma'', \ell : T_\ell, x : T_x$, and $\Gamma' = \Gamma''\{\ell/x\}, \ell : T$, and $\Gamma' \vdash \ell : T_\ell\{\ell/x\}$, and $\Gamma' \vdash \ell : T_x\{\ell/x\}$.

Lemma 2.3.10 *If* $\Gamma = \Gamma', \ell : T_\ell, x : T_x$, *and* $\Gamma \vdash e : T, \Gamma_r$, *and* $T_\ell \neq T'\backslash S\,!$, *and* $T_x \neq T'\backslash S\,!$ *when* $\ell \in \mathsf{locs}(e)$, *then* $\Gamma\{\!\{\ell/x;\ \ell : T'_\ell\}\!\} \vdash e\{\ell/x\} : T\{\ell/x\}, \Gamma_r\{\!\{\ell/x;\ \ell : T''_\ell\}\!\}$ *for some* T''_ℓ.

The following lemma is essentially the same as Lemma 2.3.10, with the definition of substitutions of typing environments expanded.

Lemma 2.3.11 *If* $\Gamma = \Gamma', \ell : T_\ell, x : T_x$, *and* $\Gamma \vdash e : T, \Gamma_r$, *and* $T_\ell \neq T'\backslash S\,!$, *and* $T_x \neq T'\backslash S\,!$ *when* $\ell \in \mathsf{locs}(e)$, *and* $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash \ell : T_\ell\{\ell/x\}$, *and* $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash \ell : T_x\{\ell/x\}$, *and* $\Gamma_r = \Gamma'_r, \ell : T^r_\ell, x : T^r_x$, *then* $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash e\{\ell/x\} : T\{\ell/x\}, \Gamma''_r$, *where* $\Gamma''_r = \Gamma'_r\{\ell/x\}, \ell : T''_\ell$, *and* $\Gamma''_r \vdash \ell : T^r_\ell\{\ell/x\}$, *and* $\Gamma''_r \vdash \ell : T^r_x\{\ell/x\}$.

PROOF: By induction on the derivation of $\Gamma \vdash e : T, \Gamma_r$.

- T-SUB

  Then $\Gamma \vdash e : T_1, \Gamma_r$, and $\Gamma \vdash T_1 \leq T$. Apply the induction hypothesis to $\Gamma \vdash e : T_1, \Gamma_r$, and by Lemma 2.3.8, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash T_1\{\ell/x\} \leq T\{\ell/x\}$, and then the proof follows.

- T-PATH

  Then $e = (T\ p)$, and $\Gamma \vdash p : T$. By Lemma 2.3.9, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash p\{\ell/x\} : T\{\ell/x\}$. Thus $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash e\{\ell/x\} : T\{\ell/x\}, (\Gamma'\{\ell/x\}, \ell : T'_\ell)$ by T-PATH.

- T-SEQ

  Then $e = e_1;\ e_2$, and $\Gamma \vdash e_1 : T_1, \Gamma_1$, and $\Gamma_1 \vdash e_2 : T, \Gamma_r$. Let $\Gamma_1 = \Gamma'_1, \ell : T^1_\ell, x : T^1_x$. By Lemma 2.3.6, $T^1_\ell \neq T'\backslash S!$ for any T' and any S. By the induction hypothesis, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash e_1\{\ell/x\} : T_1\{\ell/x\}, (\Gamma'_1\{\ell/x\}, \ell : T'''_\ell)$, and $\Gamma'_1\{\ell/x\}, \ell : T'''_\ell \vdash \ell : T^1_\ell\{\ell/x\}$, and $\Gamma'_1\{\ell/x\}, \ell : T'''_\ell \vdash \ell : T^1_x\{\ell/x\}$. Also by the induction hypothesis, $\Gamma'_1\{\ell/x\}, \ell : T'''_\ell \vdash e_2\{\ell/x\} : T\{\ell/x\}, \Gamma''_r$, and $\Gamma''_r = \Gamma'_r\{\ell/x\}, \ell : T''_\ell$, and $\Gamma''_r \vdash \ell : T^r_\ell\{\ell/x\}$, and $\Gamma''_r \vdash \ell : T^r_x\{\ell/x\}$. Thus T-SEQ applies, and $\Gamma'\{\ell/x\} \vdash e\{\ell/x\} : T\{\ell/x\}, \Gamma''_r$.

- T-NEW

  Trivial since the typing of a `new` expression is not affected by the substitution.

- T-GET

  Then $e = e_1.f$, and $\Gamma \vdash e_1 : T_1, \Gamma_r$, and $T = \mathsf{ftype}(T_1, f)$. By the definition of ftype, $f \notin \mathsf{masked}(T_1)$. It is easy to see that $f \notin \mathsf{masked}(T_1\{\ell/x\})$, so $\mathsf{ftype}(T_1\{\ell/x\}, f)$ is well defined. Also, $T\{\ell/x\} = T$ since T is the declared field type. By the induction hypothesis, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash e_1\{\ell/x\} : T_1\{\ell/x\}, \Gamma''_r$. Then T-GET applies, and $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash e\{\ell/x\} : T\{\ell/x\}, \Gamma''_r$.

- T-SET

  Then $e = (T_1\ p_1).f = (T_2\ p_2)$, and $T = \circ\backslash\mathsf{sub}_\circ$, and $T_1 \neq T''_1\backslash f!$, and $\Gamma \vdash (T_1\ p_1) : T_1, \Gamma$, and $\Gamma \vdash_R (T_2\ p_2) : \mathsf{ftype}(\mathsf{grant}(T_1, f), f), \Gamma_2$, and $\Gamma_r = \Gamma_2\{\!\{p_1 : \mathsf{grant}(T'_1, f)\}\!\}$ where $p_1 : T'_1 \in \Gamma_2$. By the induction hypothesis, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash (T_1\ p_1)\{\ell/x\} : T_1\{\ell/x\}, (\Gamma'\{\ell/x\}, \ell : T'_\ell)$. Note that $\mathsf{ftype}(\mathsf{grant}(T_1, f)\{\ell/x\}, f)$ is well-defined, and it is not changed by the substitution, i.e., $\mathsf{ftype}(\mathsf{grant}(T_1, f)\{\ell/x\}, f) = \mathsf{ftype}(\mathsf{grant}(T_1, f), f)\{\ell/x\} = \mathsf{ftype}(\mathsf{grant}(T_1, f), f)$. Let $T_f = \mathsf{ftype}(\mathsf{grant}(T_1, f), f)$. There are several cases for p_2:

- $p_2 = x$

  By TR-VAR, $\Gamma \vdash x : T_2$, and $\Gamma_2 = \Gamma\{\!\{x : \mathsf{noMust}(T_x)\}\!\}$, that is, $\Gamma_2 = \Gamma', \ell : T_\ell, x : \mathsf{noMust}(T_x)$, and $\Gamma_2 \vdash \mathsf{noMust}(T_2) \leq T_f$. By the definition of noMust and S-SIMPLE, $\Gamma_2 \vdash T_2 \leq \mathsf{noMust}(T_2)$, and then by S-TRANS, $\Gamma_2 \vdash T_2 \leq T_f$. Similarly $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash T_x\{\ell/x\} \leq \mathsf{noMust}(T_x\{\ell/x\})$, and therefore $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash \ell : \mathsf{noMust}(T_x\{\ell/x\})$. By Lemma 2.3.9, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash \ell : T_2\{\ell/x\}$. By Lemma 2.3.8, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash T_2\{\ell/x\} \leq T_f$. By TR-LOC, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash_R (T_2\ p_2)\{\ell/x\} : T_f, (\Gamma'\{\ell/x\}, \ell : \mathsf{noMust}(T'_\ell))$. Then by T-SET, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash e\{\ell/x\} : T\{\ell/x\}, \Gamma''_r$. There are again several cases for p_1:

  * $p_1 = x$

    Then $\Gamma''_r = \Gamma'\{\ell/x\}, \ell : \mathsf{grant}(\mathsf{noMust}(T'_\ell), f)$, and $\Gamma_r = \Gamma', \ell : T_\ell, x : \mathsf{grant}(\mathsf{noMust}(T_x), f)$. By Lemma 2.3.7, $\Gamma'\{\ell/x\}, \ell : \mathsf{noMust}(T'_\ell) \vdash \ell : T_\ell\{\ell/x\}$ since T_ℓ contains no must-masks. Also, $\Gamma''_r \vdash \mathsf{grant}(\mathsf{noMust}(T'_\ell), f) \leq \mathsf{noMust}(T'_\ell)$, and therefore Γ''_r is an extension of $\Gamma'\{\ell/x\}, \ell : \mathsf{noMust}(T'_\ell)$. Thus we have $\Gamma''_r \vdash \ell : T_\ell\{\ell/x\}$.

  * $p_1 = \ell$

    Then $\Gamma''_r = \Gamma'\{\ell/x\}, \ell : \mathsf{grant}(\mathsf{noMust}(T'_\ell), f)$, and $\Gamma_r = \Gamma', \ell : \mathsf{grant}(T_\ell, f), x : \mathsf{noMust}(T_x)$. By Lemma 2.3.7, $\Gamma''_r \vdash \ell : \mathsf{grant}(T_\ell\{\ell/x\}, f)$, and $\Gamma'\{\ell/x\}, \ell : \mathsf{noMust}(T'_\ell) \vdash \ell : \mathsf{noMust}(T_x\{\ell/x\})$. Then $\Gamma''_r \vdash \ell : \mathsf{noMust}(T_x\{\ell/x\})$ since Γ''_r is an extension of $\Gamma'\{\ell/x\}, \ell : \mathsf{noMust}(T'_\ell)$.

  * $p_1 \neq x$ and $p_1 \neq \ell$

Then $\Gamma_r'' = \Gamma''\{\ell/x\}, \ell : \mathsf{noMust}(T_\ell')$, and $\Gamma_r = \Gamma'', \ell : T_\ell, x : \mathsf{noMust}(T_x)$, where $\Gamma'' = \Gamma'\{\!\{p_1 : \mathsf{grant}(T_p, f)\}\!\}$ and $p_1 : T_p \in \Gamma'$. Consider the environment $\Gamma''' = \Gamma'\{\!\{p_1 : \mathsf{noMust}(T_p)\}\!\}$, and we can see that $\Gamma'''\{\ell/x\}, \ell : T_\ell' \vdash \ell : T_\ell\{\ell/x\}$ and $\Gamma'''\{\ell/x\}, \ell : T_\ell' \vdash \ell : T_x\{\ell/x\}$, because $p_1 \neq x$ and $p_1 \neq \ell$. By Lemma 2.3.7, $\Gamma'''\{\ell/x\}, \ell : \mathsf{noMust}(T_\ell') \vdash \ell : \mathsf{noMust}(T_x\{\ell/x\})$ and $\Gamma'''\{\ell/x\}, \ell : \mathsf{noMust}(T_\ell') \vdash T_\ell\{\ell/x\})$, since T_ℓ has no must-masks. Note that Γ_r'' is an extension of $\Gamma'''\{\ell/x\}, \ell : \mathsf{noMust}(T_\ell')$. By Lemma 2.3.3, $\Gamma_r'' \vdash \ell : T_\ell\{\ell/x\}$, and $\Gamma_r'' \vdash \ell : \mathsf{noMust}(T_x\{\ell/x\})$.

– $p_2 = \ell$

By TR-LOC, $\Gamma \vdash \ell : T_2$, and $\Gamma \vdash T_2 \leq T_f$, and $\Gamma_2 = \Gamma$ since $\mathsf{noMust}(T_\ell) = T_\ell$. By Lemma 2.3.9, $\Gamma'\{\ell/x\}, \ell : T_\ell' \vdash \ell : T_2\{\ell/x\}$, and by Lemma 2.3.8, $\Gamma'\{\ell/x\}, \ell : T_\ell' \vdash T_2\{\ell/x\} \leq T_f$. Therefore by TR-LOC, $\Gamma'\{\ell/x\}, \ell : T_\ell' \vdash_R (T_2\ p_2)\{\ell/x\} : T_f, (\Gamma'\{\ell/x\}, \ell : \mathsf{noMust}(T_\ell'))$. Since $p_2 = \ell$, i.e., $\ell \in \mathsf{locs}(e)$, we have $T_x \neq T'\backslash S\,!$. There are several cases for p_1:

∗ $p_1 = x$

Then $\Gamma_r'' = \Gamma'\{\ell/x\}, \ell : \mathsf{grant}(\mathsf{noMust}(T_\ell'), f)$, and $\Gamma_r = \Gamma', \ell : T_\ell, x : \mathsf{grant}(T_x, f)$. By Lemma 2.3.7, $\Gamma_r'' \vdash \ell : \mathsf{grant}(T_x\{\ell/x\}, f)$ where $T_x = \mathsf{noMust}(T_x)$, and $\Gamma'\{\ell/x\}, \ell : \mathsf{noMust}(T_\ell') \vdash \ell : T_\ell\{\ell/x\}$. Since Γ_r'' is an extension of $\Gamma'\{\ell/x\}, \ell : \mathsf{noMust}(T_\ell')$, we have $\Gamma_r'' \vdash \ell : T_\ell\{\ell/x\}$.

∗ $p_1 = \ell$

Then $\Gamma_r'' = \Gamma'\{\ell/x\}, \ell : \mathsf{grant}(\mathsf{noMust}(T_\ell'), f)$, and $\Gamma_r = \Gamma', \ell : \mathsf{grant}(T_\ell, f), x : T_x$. By Lemma 2.3.7, $\Gamma_r'' \vdash \ell : \mathsf{grant}(T_\ell\{\ell/x\}, f)$, and $\Gamma'\{\ell/x\}, \ell : \mathsf{noMust}(T_\ell') \vdash \ell : T_x\{\ell/x\}$ since $T_x = \mathsf{noMust}(T_x)$. Finally $\Gamma_r'' \vdash \ell : T_x\{\ell/x\}$ since Γ_r'' is an extension of $\Gamma'\{\ell/x\}, \ell : \mathsf{noMust}(T_\ell')$.

∗ $p_1 \neq x$ and $p_1 \neq \ell$

Then $\Gamma''_r = \Gamma''\{\ell/x\}, \ell : \mathsf{noMust}(T'_\ell)$, and $\Gamma_r = \Gamma'', \ell : T_\ell, x : T_x$, where $\Gamma'' = \Gamma'\{\!\{p_1 : \mathsf{grant}(T_p, f)\}\!\}$ and $p_1 : T_p \in \Gamma'$. Note that $T_x = \mathsf{noMust}(T_x)$. The proof is the same as that for the above case $p_2 = x$ and $p_1 \neq x$ and $p_1 \neq \ell$. We can get $\Gamma''_r \vdash \ell : T_\ell\{\ell/x\}$, and $\Gamma''_r \vdash \ell : T_x\{\ell/x\}$.

– $p_2 = x'$ and $x' \neq x$

By TR-VAR, $\Gamma \vdash x' : T_2$, and $\Gamma_r \vdash \mathsf{noMust}(T_2) \leq T_f$. By Lemma 2.3.9, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash x' : T_2\{\ell/x\}$. By Lemma 2.3.8, $\Gamma'_r\{\ell/x\}, \ell : T'_\ell \vdash T_2\{\ell/x\} \leq T_f$. Obviously changing the masks on x' is not affected by the substitution, so TR-VAR can apply, and $\Gamma'_r\{\ell/x\}, \ell : T'_\ell \vdash_R (T_2\ x')\{\ell/x\} : T_f, \Gamma''_r$. There are several cases for p_1:

∗ $p_1 = x$

Then $\Gamma''_r = \Gamma''\{\ell/x\}, \ell : \mathsf{grant}(T'_\ell, f)$, and $\Gamma_r = \Gamma'', \ell : T_\ell, x : \mathsf{grant}(T_x, f)$, where $\Gamma'' = \Gamma'\{\!\{p_2 : \mathsf{noMust}(T_p)\}\!\}$ and $p_2 : T_p \in \Gamma'$. We should have $\Gamma''\{\ell/x\}, \ell : T'_\ell \vdash \ell : T_\ell\{\ell/x\}$ and $\Gamma''\{\ell/x\}, \ell : T'_\ell \vdash \ell : T_x\{\ell/x\}$, because removing must annotations on p_2 does not change the dependency graph. By Lemma 2.3.7, $\Gamma''_r \vdash \ell : \mathsf{grant}(T_x\{\ell/x\}, f)$, and $\Gamma''_r \vdash \ell : \mathsf{grant}(T_\ell\{\ell/x\}, f)$. Moreover, $\Gamma''_r \vdash \mathsf{grant}(T_\ell\{\ell/x\}, f) \leq T_\ell\{\ell/x\}$, because T_ℓ has no must-mask. Therefore $\Gamma''_r \vdash \ell : T_\ell\{\ell/x\}$ by TP-SUB.

∗ $p_1 = \ell$

Then $\Gamma''_r = \Gamma''\{\ell/x\}, \ell : \mathsf{grant}(T'_\ell, f)$, and $\Gamma_r = \Gamma'', \ell : \mathsf{grant}(T_\ell, f), x : T_x$, where $\Gamma'' = \Gamma'\{\!\{p_2 : \mathsf{noMust}(T_p)\}\!\}$ and $p_2 : T_p \in \Gamma'$. Similar to the above case, $\Gamma''\{\ell/x\}, \ell : T'_\ell \vdash \ell : T_\ell\{\ell/x\}$ and $\Gamma''\{\ell/x\}, \ell : T'_\ell \vdash \ell : T_x\{\ell/x\}$. Since $p_1 = \ell$, $\ell \in \mathsf{locs}(e)$, and therefore $T_x = \mathsf{noMust}(T_x)$. Then $\Gamma''_r \vdash \mathsf{grant}(T_x\{\ell/x\}, f) \leq T_x\{\ell/x\}$. By Lemma 2.3.7, $\Gamma''_r \vdash \ell : \mathsf{grant}(T_\ell\{\ell/x\}, f)$

and $\Gamma''_r \vdash \ell\!:\!\mathsf{grant}(T_x\{\ell/x\}, f)$. By TP-SUB, $\Gamma''_r \vdash \ell\!:\!T_x\{\ell/x\}$.

  * $p_1 \neq x$ and $p_1 \neq \ell$

    Suppose $p_1 : T_p \in \Gamma'$ and $p_2 : T'_p \in \Gamma'$. Consider the environment $\Gamma''' = \Gamma'\{\!\{p_1\!:\!\mathsf{noMust}(T_p)\}\!\}\{\!\{p_2 : \mathsf{noMust}(T'_p)\}\!\}$, and we can see that $\Gamma'''\{\ell/x\}, \ell\!:\!T'_\ell \vdash \ell\!:\!T_x\{\ell/x\}$ and $\Gamma'''\{\ell/x\}, \ell\!:\!T'_\ell \vdash \ell\!:\!T_\ell\{\ell/x\}$. Note that Γ''_r is an extension of $\Gamma''', \ell\!:\!T'_\ell$. Then the proof follows.

- $p_2 = \ell'$ and $\ell' \neq \ell$

  Similar to the case above, with TR-VAR replaced by TR-LOC.

- T-SET-COND

Then $e = (T_1\backslash f!\ p_1).f = (T_2\ p_2)$. By the induction hypothesis, $\Gamma'\{\ell/x\}, \ell\!:\!T'_\ell \vdash (T_1\backslash f!\ p_1)\{\ell/x\} : T_1\backslash f!\{\ell/x\}, (\Gamma'\{\ell/x\}, \ell : T'_\ell)$, and $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash (T_2\ p_2)\{\ell/x\} : T_2\{\ell/x\}, (\Gamma'\{\ell/x\}, \ell\!:\!T'_\ell)$. Note that the set $\overline{S}$ in T-SET-COND is not affected by the substitution, and therefore, $\Gamma'\{\ell/x\}, \ell\!:\!T'_\ell \vdash e\{\ell/x\}\!:\!\circ\backslash\mathsf{sub}_\circ, \Gamma''_r$, where $\Gamma''_r = (\Gamma'\{\ell/x\}, \ell\!:\!T'_\ell)\{\!\{p_1\{\ell/x\}\!:\!T_p\backslash f[p_2\{\ell/x\}.\overline{S}]\}\!\}$, and $p_1\{\ell/x\}\!:\!T_p\backslash f! \in (\Gamma'\{\ell/x\}, \ell\!:\!T'_\ell)$. Also note that $\Gamma_r = \Gamma\{\!\{p_1 : T'_p\backslash f[p_2.\overline{S}]\}\!\}$ where $p_1 : T'_p\backslash f! \in \Gamma$. There are several cases for p_1:

- $p_1 = x$

  Then $T'_\ell = T_p\backslash f!$, and $T_x = T'_p\backslash f!$. It is then obvious that $\Gamma''_r \vdash \ell\!:\!T_x\{\ell/x\}$. By Lemma 2.3.7, $\Gamma''_r \vdash \ell\!:\!T_\ell\{\ell/x\}$ since T_ℓ contains no must-masks.

- $p_1 = \ell$

  Then $T'_\ell = T_p\backslash f!$, and $T_\ell = T'_p\backslash f!$, and T_x has no must-masks by the assumption. By Lemma 2.3.7, $\Gamma''_r \vdash \ell : T_x\{\ell/x\}$. Also, it is obvious $\Gamma''_r \vdash \ell\!:\!T^r_\ell\{\ell/x\}$, where $T^r_\ell = T'_p\backslash f[p_2.\overline{S}]$.

- $p_1 \neq x$ and $p_1 \neq \ell$

This case is easy, because the type bindings of x and ℓ are not changed, and the dependency graph only has more edges than before.

- T-CALL

  Apply the induction hypothesis to the typing judgments in the premises, and Lemma 2.3.8 to the subtyping judgments, and finally apply T-CALL again. Note that when the substitution uses a location that is already in the parameter list, both T_x and T_ℓ contain no must masks, and therefore the corresponding formal types contain no must-masks. It only remains to prove that $\Gamma''_r \vdash \ell : T^r_\ell\{\ell/x\}$ and $\Gamma''_r \vdash \ell : T^r_x\{\ell/x\}$. There are several cases for p_0:

  - $p_0 = x$

    Then $T^r_\ell = T_\ell$, and T^r_x is obtained from T_x by replacing all the masks with $\overline{M_2}$. By the definition of update, in order to get T''_ℓ, a mask in T'_ℓ might be removed, or if it is a must-mask, it might be converted to a conditional mask or a simple mask. Therefore, by Lemma 2.3.7, $\Gamma''_r \vdash \ell : T_\ell\{\ell/x\}$, because T_ℓ has no must-mask.

    Now it remains to prove that $\Gamma''_r \vdash \ell : T^r_x\{\ell/x\}$. Note that $T^r_x = U_x\backslash\overline{M_2}$ and $\Gamma \vdash x : U_x\backslash\overline{M_1}$ where $T_x = U_x\backslash\overline{M_x}$. By Lemma 2.3.9, $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash \ell : U_x\backslash\overline{M_1}\{\ell/x\}$. Now let us consider the change from T'_ℓ to T''_ℓ and that from $U_x\backslash\overline{M_1}$ to $U_x\backslash\overline{M_2}$, according to the definition of update and M-OK, and prove that the typing of ℓ preserves:

    * A mask, but not a must-mask, is added to $\overline{M_2}$, or a mask in $\overline{M_1}$ is replaced with a more conservative mask. By TP-SUB, the typing preserves.

    * Corresponding masks are removed from both $\overline{M_1}$ and T'_ℓ, i.e.,

grant is applied to both types. By Lemma 2.3.7, the typing preserves.

  * Both $\overline{M_1}$ and T'_ℓ have a mask $S\,!$, which is replaced with a simple mask S or a conditional mask $S\,[\ldots]$. By Lemma 2.3.7, the typing of ℓ preserves.

  Therefore $\Gamma''_r \vdash \ell : T^r_x\{\ell/x\}$.

- $p_0 = \ell$

  Then $T^r_x = T_x$, and both T_ℓ and T_x contain no must-masks by the assumption. Then by the definition of update, we can see $\Gamma''_r \vdash \ell : T^r_\ell\{\ell/x\}$, because T''_ℓ and $T^r_\ell\{\ell/x\}$ are obtained from T'_ℓ and T_ℓ respectively, calling update with the same target masks.

  Now it remains to prove that $\Gamma''_r \vdash \ell : T_x\{\ell/x\}$. Note that $\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash \ell : T_x\{\ell/x\}$. Let us inspect the change from T'_ℓ to T''_ℓ according to the definition of update, and prove that the typing of ℓ preserves:

  * A mask is removed from T'_ℓ. By Lemma 2.3.7, the typing preserves, because T_x has no must-mask.
  * T'_ℓ has a must-mask $S\,!$, which is replaced with a simple mask S or a conditional mask $S\,[\ldots]$. By Lemma 2.3.7, the typing preserves, because T_x has no must-mask.

  Therefore $\Gamma''_r \vdash \ell : T_x\{\ell/x\}$, that is, $\Gamma''_r \vdash \ell : T^r_x\{\ell/x\}$.

- $p_0 \neq x$ and $p_0 \neq \ell$

  Follows from that typing of ℓ and x is not changed.

- T-LET

  Then $e = \texttt{let}\ T'\ x' = e_1\ \texttt{in}\ e_2$ where $x' \neq x$, and $\Gamma \vdash_R e_1 : T', \Gamma_1$, and $\Gamma_1, x' : T' \vdash e_2 : T, \Gamma_2$, and $\Gamma_2 = \Gamma_3, x' : T''$, and $\Gamma_r = \mathsf{remove}(\Gamma_3, x')$. We first prove

$\Gamma'\{\ell/x\}, \ell : T'_\ell \vdash_R e_1\{\ell/x\} : T'\{\ell/x\}, (\Gamma'_1\{\ell/x\}, \ell : T'''_\ell)$ where $\Gamma_1 = \Gamma'_1, \ell : T^1_\ell, x : T^1_x$, and $\Gamma'_1\{\ell/x\}, \ell : T'''_\ell \vdash \ell : T^1_\ell\{\ell/x\}$, and $\Gamma'_1\{\ell/x\}, \ell : T'''_\ell \vdash \ell : T^1_x\{\ell/x\}$. The proof is by induction on the derivation of $\Gamma \vdash_R e_1 : T', \Gamma_1$:

- TR-VAR

  Similar to the proof of the case for T-SET, we can show that $\Gamma'_1\{\ell/x\}, \ell : T'''_\ell \vdash \ell : T^1_\ell\{\ell/x\}$. There are then two cases:

  * $e_1 = (T_1\ x)$

    Then $T^1_\ell = T_\ell$, and $T^1_x = \mathsf{noMust}(T_x)$, and $T'''_\ell = \mathsf{noMust}(T'_\ell)$. Note that $T^1_\ell = \mathsf{noMust}(T^1_\ell)$ since it contains no must-masks. The proof follows by Lemma 2.3.7.

  * $e_1 = (T_1\ x'')$ and $x'' \neq x$

    The proof follows by the outer induction hypothesis with $\Gamma_1, x' : T' \vdash e_2 : T, \Gamma_2$.

- TR-LOC

  Similar to the proof of the case for T-SET, we can show that $\Gamma'_1\{\ell/x\}, \ell : T'''_\ell \vdash \ell : T^1_\ell\{\ell/x\}$. There are again two cases:

  * $e_1 = (T_1\ \ell)$

    Then $T^1_\ell = \mathsf{noMust}(T_\ell)$, and $T^1_x = T_x$, and $T'''_\ell = \mathsf{noMust}(T'_\ell)$. Note that $T^1_x = \mathsf{noMust}(T^1_x)$ by the assumption. Therefore by Lemma 2.3.7, $\Gamma'_1\{\ell/x\}, \ell : T'''_\ell \vdash \ell : T^1_\ell\{\ell/x\}$, and $\Gamma'_1\{\ell/x\}, \ell : T'''_\ell \vdash \ell : T^1_x\{\ell/x\}$.

  * $e_1 = (T_1\ \ell')$ and $\ell' \neq \ell$

    The proof follows by the outer induction hypothesis with $\Gamma_1, x' : T' \vdash e_2 : T, \Gamma_2$.

- TR-SEQ

Then $e_1 = e'_1;\ e'_2$. The proof is by application of the outer induction hypothesis on e'_1 and the inner induction hypothesis on e'_2.

– TR-OTHER

Then just apply the outer induction hypothesis.

Then the proof follows by the induction hypothesis on e_2 and by Lemma 2.3.7.

□

Progress. Now we prove the progress lemma.

Lemma 2.3.12 *(Progress) If $\vdash H$, and $\lfloor H \rfloor \vdash e : T, \Gamma$ then either $e = (T_\ell\ \ell)$ or there is an expression e' and a heap H' such that $e, H \longrightarrow e', H'$.*

PROOF: The proof is by structural induction on the expression e.

According to the definition of $\lfloor H \rfloor$, there is no type bindings for variables in $\lfloor H \rfloor$, and therefore the expression e does not have any free variable.

T-SUB is the only non-syntax-directed typing rule that might be used for $\lfloor H \rfloor \vdash e : T, \Gamma$. However, it does not yield a subexpression of e, or a different typing environment, and therefore the derivation $\lfloor H \rfloor \vdash e : T, \Gamma$ must contain an application of a rule other than T-SUB. Thus for the remainder of the proof, only syntax-directed typing rules are considered for typing e.

- $e = (T_\ell\ \ell)$

Trivial.

- $e = \texttt{new}\ C$

  Then R-ALLOC applies, and therefore $e' = (C!\backslash\overline{f}!\ \ell)$, where $\ell \notin \mathrm{dom}(H)$, and $H' = H, \ell \mapsto C!\backslash\overline{f}!\ \{\}$.

- $e = e_1;\ e_2$

  If $e_1 = (T_\ell\ \ell)$, then R-SEQ applies, and therefore $e' = e_2$ and $H' = H$; otherwise, by the induction hypothesis, there exists e_1' and H_1' such that $e_1, H \longrightarrow e_1', H_1'$, and R-CONG applies.

- $e = e_1.f$

  If $e_1 = (T_\ell\ \ell)$, then T-GET applies. Therefore $\lfloor H \rfloor \vdash \ell : T_\ell$ and $\mathsf{ftype}(T_\ell, f)$ is well-defined. By the definition of ftype, $f \notin \mathsf{masked}(T_\ell)$. According to the definition of $\vdash H$, there exists ℓ' such that $H(\ell, f) = \ell'$. So R-GET applies, and $e' = (\mathsf{ftype}(T_\ell, f)\ \ell')$.

  If $e_1 \neq (T_\ell\ \ell)$, then by the induction hypothesis, there exists e_1' and H_1' such that $e_1, H \longrightarrow e_1', H_1'$, and R-CONG applies.

- $e = (T_1\ \ell_1).f = (T_2\ \ell_2)$

  There are two cases, depending on which of the two typing rules of field assignments applies to $\lfloor H \rfloor \vdash e : T, \Gamma$.

  - T-SET

    Then $T_1 \neq T_1'\backslash f!$. Therefore R-SET can apply, and the evaluation can progress.

  - T-SET-COND

    Then $T_1 = T_1'\backslash f!$, and $\lfloor H \rfloor \vdash \ell_1 : T_1'\backslash f!$ by T-PATH. It is easy to see that $\ell_1 : T_1''\backslash f! \in \lfloor H \rfloor$, or otherwise it would contradict the must-mask on f in T_1. Therefore R-SET-COND can apply.

- $e = (T_0\ \ell_0).m((\overline{T}\ \overline{\ell}))$

  Then T-CALL applies, and therefore $\mathsf{mbody}(T_0, m)$ is well-defined. Thus R-CALL can apply, and the evaluation can make progress.

- $e = \texttt{let}\ T_x\ x = e_1\ \texttt{in}\ e_2$

  If $e_1 = (T_\ell\ \ell)$, then R-LET can apply; otherwise, by the induction hypothesis, there exists e_1' and H_1' such that $e_1, H \longrightarrow e_1', H_1'$, and R-CONG can apply.

□

Subject reduction. Next, we prove subject reduction.

Lemma 2.3.13 *(Subject reduction) If $\vdash e\ \texttt{wf}$, and $\vdash H$, and $\lfloor H \rfloor \vdash e : T, \Gamma$, and $e, H \longrightarrow e', H'$, then $\vdash e'\ \texttt{wf}$, and $\vdash H'$, and $\lfloor H' \rfloor \vdash e' : T, \Gamma'$, and Γ' is an extension of Γ.*

PROOF: We first show expression well-formedness $\vdash e\ \texttt{wf}$ is preserved by evaluation. For any `let` subexpression $\texttt{let}\ T_x\ x = e_1\ \texttt{in}\ e_2$ contained in e, the only possibility for e_2 to have a new location ℓ is through a substitution of another variable x' that is free in the `let` expression, since e_2 is not in an evaluation context. Then if $e_1 = x'$ or $e_1 = e_1';\ x'$, according to TR-VAR, T_x has no must-masks.

The remaining proof is by induction on the derivation of $\lfloor H \rfloor \vdash e : T, \Gamma$.

- T-SUB

  Then $\lfloor H \rfloor \vdash e : T', \Gamma$ and $\lfloor H \rfloor \vdash T' \leq T$. By the induction hypothesis, $\vdash H'$, and $\lfloor H' \rfloor \vdash e : T', \Gamma'$, and Γ' is an extension of Γ. By Lemma 2.3.5, $\lfloor H' \rfloor \vdash T' \leq T$. Thus $\lfloor H' \rfloor \vdash e : T, \Gamma'$, by T-SUB.

- T-PATH

  Vacuously true since e cannot have any free variable, and $(T\ \ell)$ cannot take a step.

- T-NEW

  Then $T = C!\backslash\overline{f}!$ and $\Gamma = \lfloor H \rfloor$. By R-ALLOC, $e' = (C!\backslash\overline{f}!\ \ell)$, and $H' = H, \ell \mapsto C!\backslash\overline{f}!\ \{\}$. Therefore, $\lfloor H' \rfloor \vdash \ell : C!\backslash\overline{f}!$, and by T-LOC, $\lfloor H' \rfloor \vdash e' : T, \lfloor H' \rfloor$. Obviously $\lfloor H' \rfloor$ is an extension of $\lfloor H \rfloor$. H' is still well-formed, because ℓ does not appear in H, and no field of ℓ is bound in H'.

- T-SEQ

  Then $e = e_1;\ e_2$. There are two cases for e_1.

  - $e_1 = (T_\ell\ \ell)$

    By T-SEQ, $\lfloor H \rfloor \vdash e_2 : T, \Gamma$. By R-SEQ, $e' = e_2$, and $H' = H$. Then it is obvious that $\lfloor H' \rfloor \vdash e' : T, \Gamma'$ where $\Gamma' = \Gamma$.

  - $e_1 \neq (T_\ell\ \ell)$

    Then R-CONG applies: $e_1, H \longrightarrow e_1', H'$, and $e' = e_1';\ e_2$. By T-SEQ, $\lfloor H \rfloor \vdash e_1 : T_1, \Gamma_1$, and $\Gamma_1 \vdash e_2 : T, \Gamma$. By the induction hypothesis, $\lfloor H' \rfloor \vdash e_1' : T_1, \Gamma_1'$, where Γ_1' is an extension of Γ_1. By Lemma 2.3.4, $\Gamma_1' \vdash e_2 : T, \Gamma'$, and Γ' is an extension of Γ.

- T-GET

  Then $e = e_1.f$, and $\lfloor H \rfloor \vdash e_1 : T_1, \Gamma$, and $T = \mathsf{ftype}(T, f)$. There are two cases for e_1.

  - $e_1 = (T_\ell\ \ell)$

    Then $\Gamma = \lfloor H \rfloor$. By the definition of ftype, $f \notin \mathsf{masked}(T_1)$. By H-LOC, $H(\ell, f) = \ell'$ and $\lfloor H \rfloor \vdash \ell' : T$. Finally, $e' = (\mathsf{ftype}(T_1, f)\ \ell')$ and $H' = H$, by R-GET.

– $e_1 \neq (T_\ell\ \ell)$

Then R-CONG applies: $e_1, H \longrightarrow e_1', H'$. By the induction hypothesis, $\vdash H'$ and $\lfloor H' \rfloor \vdash e_1' : T_1, \Gamma'$ where Γ' is an extension of Γ. Then T-GET applies, and therefore $\lfloor H' \rfloor \vdash e_1'.f : T, \Gamma'$.

- T-SET

Then $e = (T_1\ \ell_1).f = (T_2\ \ell_2)$, and $T = \circ\backslash\mathsf{sub}_\circ$, and $\lfloor H \rfloor \vdash_R (T_2\ \ell_2) : \mathsf{ftype}(\mathsf{grant}(T_1, f), f), \Gamma_2$, and $\Gamma_2 = \lfloor H \rfloor\{\!\{\ell_2 : \mathsf{noMust}(T_2')\}\!\}$ where $\ell_2 : T_2' \in \lfloor H \rfloor$, and $\Gamma = \Gamma_2\{\!\{\ell_1 : \mathsf{grant}(T_1', f)\}\!\}$ where $\ell_1 : T_1' \in \lfloor H \rfloor$, and $T_1 \neq T'\backslash f!$ for any T'. By R-SET, $e, H \longrightarrow (\circ\backslash\mathsf{sub}_\circ\ \ell_2), H'$, and $H' = H\{\!\{\ell_1 := \mathsf{grant}(T_1', f)\,\{\ldots, f = \ell_2\}\}\!\}$. It is easy to see that $\lfloor H' \rfloor \vdash (\circ\backslash\mathsf{sub}_\circ\ \ell_2) : \circ\backslash\mathsf{sub}_\circ, \lfloor H' \rfloor$. Note that $\Gamma' = \lfloor H' \rfloor = \lfloor H \rfloor\{\!\{\ell_1 : \mathsf{grant}(T_1', f)\}\!\}$, and $\Gamma = \lfloor H \rfloor\{\!\{\ell_2 : \mathsf{noMust}(T_2')\}\!\}\{\!\{\ell_1 : \mathsf{grant}(T_1', f)\}\!\}$. Therefore Γ' is an extension of Γ, because ℓ_1 has the same type binding in Γ and Γ', and ℓ_2 has a more conservative type binding in Γ. By TR-LOC, $\lfloor H \rfloor \vdash \ell_2 : T_2$, and $\lfloor H \rfloor \vdash T_2 \leq \mathsf{ftype}(\mathsf{grant}(T_1, f), f)$. Then $\lfloor H \rfloor \vdash \ell_2 : \mathsf{ftype}(\mathsf{grant}(T_1, f), f)$. Consider the typing environment $\Gamma_3 = \lfloor H \rfloor\{\!\{\ell_1 : \mathsf{noMust}(T_1')\}\!\}\{\!\{\ell_2 : \mathsf{noMust}(T_2')\}\!\}$. We still have $\Gamma_3 \vdash \ell_2 : \mathsf{ftype}(\mathsf{grant}(T_1, f), f)$, because $\mathsf{ftype}(\mathsf{grant}(T_1, f), f)$ has no must-masks, and removing must annotations does not affect the dependency graph. It is easy to see that $\lfloor H' \rfloor$ is an extension of Γ_3, and therefore $\lfloor H' \rfloor \vdash \ell_2 : \mathsf{ftype}(\mathsf{grant}(T_1, f), f)$. Thus we have $\vdash H'$.

- T-SET-COND

Then $e = (T_1\backslash f!\ \ell_1).f = (U_2\backslash\overline{M}\ \ell_2)$, and $\mathsf{ftype}(T_1, f) = U_f\backslash\overline{S_f}$, and $\overline{S} = \{S | S \in \mathsf{simple}(\overline{M}) \wedge (S! \in \overline{M} \vee S \notin \overline{S_f})\}$, and $\ell_1 : T_1'\backslash f! \in \lfloor H \rfloor$, and $\Gamma = \lfloor H \rfloor\{\!\{\ell_1 : T_1'\backslash f[\ell_2.\overline{S}]\}\!\}$, and $T = \circ\backslash\mathsf{sub}_\circ$. By R-SET-COND, $e, H \longrightarrow$

$(\circ\backslash\mathsf{sub}_\circ\ \ell_2), H'$, where $\lfloor H' \rfloor = \Gamma$. By T-PATH, $\lfloor H' \rfloor \vdash (\circ\backslash\mathsf{sub}_\circ\ \ell_2) : \circ\backslash\mathsf{sub}_\circ, \lfloor H' \rfloor$, i.e., $\Gamma' = \lfloor H' \rfloor = \Gamma$. It remains to show $\vdash H'$: suppose there exists a type T_1'' such that $\lfloor H' \rfloor \vdash \ell_1 : T_1''$ and $f \notin \mathsf{masked}(T_1'')$, then we have $\lfloor H' \rfloor \vdash \ell_2 : \mathsf{ftype}(T_1'', f)$, that is, not masked by any of $\overline{S}$, since the type binding of ℓ_1 in $\lfloor H' \rfloor$ has a mask $f[\ell_2.\overline{S}]$.

- T-CALL

  Then $e = (T_0\ \ell_0).m(\overline{(T'\ \ell)})$, and $\lfloor H \rfloor \vdash \ell_0 : T_0$, and $\mathsf{mbody}(T_0, m) = T_{n+1}\ m(\overline{T}\ \overline{x})\ \mathtt{effect}\ \overline{M_1} \rightsquigarrow \overline{M_2}\ \{e_m\}$, and $T = T_{n+1}\{\ell_0/\mathtt{this}\}\{\overline{\ell}/\overline{x}\}$, and $\Gamma = \lfloor H \rfloor \{\!\{\ell_0 : \mathsf{update}(\ell_0, \overline{M}, U_0\backslash\overline{M_2}\{\ell_0/\mathtt{this}\}\{\overline{\ell}/\overline{x}\})\}\!\}$, where $\ell_0 : U_0\backslash\overline{M} \in \lfloor H \rfloor$. By R-CALL, $e' = e_m\{\ell_0/\mathtt{this}\}\{\overline{\ell}/\overline{x}\}$, and $H' = H$. Let $\Gamma_m = \mathtt{this} : C\backslash\overline{M_1}, \overline{x} : \overline{T}$ where C is the class that m is found, and then by M-OK, $\Gamma_m \vdash_R e_m : T_{n+1}, \Gamma_r$, where Γ_r only contains type bindings for $\mathtt{this}$ and $\overline{x}$. It is obvious that e_m contains no locations, and we can prove that $\Gamma_m \vdash e_m : T_{n+1}, \Gamma_r'$, and $\Gamma_r' \vdash \mathtt{this} : C\backslash\overline{M_2}$, and $\Gamma_r' \vdash \overline{x} : \overline{T}$, (by induction on TR-VAR, TR-SEQ, and TR-OTHER). Since $\lfloor H \rfloor, \Gamma_m$ is an extension of Γ_m, we have $\lfloor H \rfloor, \Gamma_m \vdash e_m : T_{n+1}, \Gamma_r''$, and Γ_r'' is an extension of Γ_r', by Lemma 2.3.4. Then we can apply Lemma 2.3.11 on $\mathtt{this}$ and $\overline{x}$, one after the other, and the proof follows.

- T-LET

  Then $e = \mathtt{let}\ T_x\ x = e_1\ \mathtt{in}\ e_2$. Note that e_1 cannot have a free variable. There are two cases for e_1:

  - $e_1 = (T_\ell\ \ell)$

    Then R-LET applies, and therefore $e' = e_2\{\ell/x\}$ and $H' = H$. Suppose $\ell : T_\ell' \in \lfloor H \rfloor$. By T-LET and TR-LOC, $\lfloor H \rfloor \vdash_R e_1 : T_x, \lfloor H \rfloor \{\!\{\ell : \mathsf{noMust}(T_\ell')\}\!\}$, and $\lfloor H \rfloor \{\!\{\ell : \mathsf{noMust}(T_\ell')\}\!\}, x : T_x \vdash e_2 : T, \Gamma_2$, and $\Gamma_2 = \Gamma_2', x : T_x'$, and $\Gamma = \mathsf{remove}(\Gamma_2', x)$. We have $\lfloor H \rfloor\{\ell/x\} = \lfloor H \rfloor$, $T_x\{\ell/x\} = T_x$, and

$T\{\ell/x\} = T$, because $\lfloor H \rfloor$ contains no type bindings for variables. Also, according to the well-formedness of e, if $\ell \in \mathsf{locs}(e_2)$, T_x contains no must-masks. Then by Lemma 2.3.11, $\lfloor H \rfloor \vdash e_2\{\ell/x\} : T, \Gamma'$, and Γ' is an extension of $\Gamma'_2\{\ell/x\}$. By the definition of remove, Γ' is an extension of Γ, because of S-SIMPLE and the fact that Γ contains no type binding for any variable.

– $e_1 \neq (T_\ell\ \ell)$

  Then R-CONG applies, and $e_1, H \longrightarrow e'_1, H'$. By T-LET, $\lfloor H \rfloor \vdash_R e_1 : T_x, \Gamma_1$, and $\Gamma_1, x : T_x \vdash e_2 : T, \Gamma_2$, and $\Gamma_2 = \Gamma'_2, x : T'_x$ where $\Gamma = \mathsf{remove}(\Gamma'_2, x)$. By the induction hypothesis, $\vdash H'$. Now we need to show $\lfloor H' \rfloor \vdash_R e'_1 : T_x, \Gamma'_1$, and Γ'_1 is an extension of Γ_1. Consider all the cases for $\lfloor H \rfloor \vdash_R e_1 : T_x, \Gamma_1$:

  * TR-VAR

    Impossible since e_1 contains no free variable.

  * TR-LOC

    Impossible.

  * TR-SEQ

    Then $e_1 = e''_1;\ e''_2$, and $\lfloor H \rfloor \vdash e''_1 : T''_1, \Gamma''_1$, and $\Gamma''_1 \vdash_R e''_2 : T_x, \Gamma_1$. There are again two cases for e''_1:

    · $e''_1 = (T'_\ell\ \ell)$

      Then $e'_1 = e''_2$, and $H' = H$, and $\Gamma'_1 = \Gamma_1$.

    · $e''_1, H \longrightarrow e'''_1, H'$

      Simply use the outer induction hypothesis, and then the proof is similar to that of the case for T-LET in Lemma 2.3.4.

  * TR-OTHER

    Follows from the outer induction hypothesis.

Therefore $\Gamma'_1, x : T_x$ is an extension of $\Gamma_1, x : T_x$, and by Lemma 2.3.4, $\Gamma'_1, x : T_x \vdash e_2 : T, \Gamma''_2$, where Γ''_2 is an extension of Γ_2. Γ''_2 must contain a type binding for x, that is, $\Gamma''_2 = \Gamma'''_2, x : T''_x$, and $\Gamma' = \mathsf{remove}(\Gamma'''_2, x)$. Obviously Γ'''_2 is an extension of Γ'_2, and according to the definition of remove, Γ' is an extension of Γ.

□

Soundness. With progress and subject reduction proved, we now state the soundness theorem.

Theorem 2.3.14 *(Soundness) If $\vdash e$ $\mathtt{wf}$, and $\vdash e : T$, and $e, \emptyset \rightarrow^* (T_\ell\ \ell), H$, then $\lfloor H \rfloor \vdash (T_\ell\ \ell) : T$.*

PROOF: Follows from Lemma 2.3.12 and Lemma 2.3.13. □

2.4 Implementation

We have implemented a prototype compiler of J\mask as an extension in the Polyglot framework [57]. The extension code has about 3,700 lines of code, excluding blank lines and comments.

J\mask is implemented as a translation to Java. The translation is mostly by erasure, that is, by erasing all the masks, effects, and mask constraints from the code.

The compiler also applies several transformations to the J\mask source code, before erasing masks. Default effects are inserted for constructors and methods that do not have them already. To simplify type checking, initialization code, including initializers, constructors, and new expressions, is also transformed.

J\mask requires that in a conditionally masked type $T \backslash f[\overline{x.g}]$, every x_i, including `this`, is a final local variable. However, the compiler uses a simple analysis to automatically insert the `final` modifier for local variables that are assigned only once, and for formal parameters that are never reassigned.

2.4.1 Inserting default effects

For a constructor of class `C`, the default effect is `*! -> C.sub!`, which describes the behavior of most constructors. The constructor starts with all the fields uninitialized, and it initializes all the fields inherited from superclasses of `C`—by calling the super constructor—and the fields declared by `C`, leaving the fields in subclasses of `C` uninitialized.

The default effect for a virtual method is `{} -> {}` because virtual methods normally work on fully initialized objects.

In our experience with using J\mask (see Section 2.5), these default effects work well. Programmers only have to annotate code that uses interesting initialization patterns.

2.4.2 Transforming initialization code

Java field declarations can include initialization expressions that are implicitly called from constructors in the same order that they appear in the class body. The J\mask compiler collects all these initializers and inserts them directly in constructors, right after super constructor calls. This initializer code is type-checked in the same way as any other constructor code.

A constructor in J\mask is just an initialization method that is called after an object is allocated on the heap. The J\mask compiler converts every constructor in the source code to a final method with the same name as the class. The transformed constructor can then be type-checked just as any other method. The compiler also inserts an empty default constructor in the generated Java code.

Every new expression `new C(...)` is split into a call to the empty default constructor to allocate the memory on the heap, and then a call to the initialization method generated from the corresponding constructor, as shown in the following piece of code:

```
final C!\(* - C.sub)! temp = new C();
temp.C(...);
```

Then the fresh local variable `temp` replaces the original expression.

2.4.3 Type checking

Flow sensitivity in the J\mask type system shows up only on masks, and not on any of the classes appearing in masked types. Therefore, each method is type-checked in two phases. The first phase is just normal Java type checking of the erased method code; the second phase, built upon the dataflow analysis framework provided in Polyglot, is flow-sensitive, and uses the result of the first phase as its starting point.

Once type checking is complete, masks are erased to generate Java code. This works because resolution of method overloading does not depend on parameter masks.

2.4.4 Inner classes

A (nonstatic) inner class is a class that is nested in the body of another class and contains an implicit reference to an instance (the *outer instance*) of the enclosing class. Every constructor of an inner class has an implicit formal parameter for the outer instance. J\mask assumes that the type of the outer instance has no masks, that is, the outer instance has been fully initialized before an instance of the inner class is created. If an inner class with a partially initialized outer instance is really needed, a transformation as described in [37] can be applied to make the outer instance explicit. J\mask currently does not directly support local classes and anonymous classes, which are inner classes nested in method bodies, although these could be converted to normal inner classes.

2.5 Experience

The language was evaluated by porting several classes in the Java Collection Framework (Java SDK version 1.4.2) to J\mask. The ported classes are

`ArrayList`, `HashMap`, `LinkedList`, `TreeMap`, and `Vector`, together with all the classes and interfaces that they depend on. There are in total 29 source files, comprising 18,000 lines of J\mask code (exclusive of empty lines and comments).

Porting these classes to J\mask was not difficult. It was completed by one of the authors within a couple of days, including time to debug the compiler. Only 11 constructors and methods required annotation with effects or mask constraints, thanks to the default effects provided by the compiler (Section 2.4.1). Besides effects and mask constraints, only 11 other masked types were needed, a very small number compared to the size of the code.

The port of this code eliminated all `null`s used as placeholders for initialization. However, some `null`s were not removed:

- Java allows storing the `null` value into collections and maps.
- Some method parameters and local variables can be intentionally set to `null`, indicating that they are not available.

Among the classes we ported, the following three exhibited nontrivial initialization patterns:

2.5.1 LinkedList

The `LinkedList` class implements a doubly-linked cyclic list. When an instance of `LinkedList` is constructed, a sentinel node, which is an instance of the nested class `Entry`, needs to be created with its `previous` and `next` fields both pointing to itself.

The Java code first constructs an instance of `Entry` with its `previous` and `next` fields set to `null`, and then initializes the two fields with the header node itself. The following code is extracted from the constructor of `LinkedList`, where `header` is the field pointing to the sentinel node:

```
header = new Entry(null, null, null);
header.previous = header.next = header;
```

With masked types, the two fields cannot be read before they are initialized. In the constructor of the ported `LinkedList` class, the field `header` is initialized as follows:

```
header = createHeader();
```

The method `createHeader` is shown below:

```
private static Entry createHeader() {
  Entry\(* - Entry.sub)! h = new Entry();
  h.element = dummyElement;
  h.next = h;
  h.previous = h;
  return h;
}
```

The static field `dummyElement` points to an object of `java.lang.Object` because the header node does not store any real data element. Therefore, there is no need to use `null`.

2.5.2 HashMap

The `HashMap` class has an empty method `init`, which, according to comments in the source code, is an "initialization hook for subclasses". When a subclass of `HashMap` is created, it should override the `init` method to initialize any new subclass fields, but Java has no way to enforce this. With effects and mask constraints, the J\mask version of `HashMap` can explicitly express the contract in the signature of the method `init`:

```
  void init() effect HashMap.sub -> {} captures *
```

2.5.3 TreeMap

`TreeMap` implements a map as a red-black tree where elements are sorted according to their keys. Each node in the tree contains fields for the left and right children, and a field pointing to its parent. A method `buildFromSorted` is used to build the tree from the bottom up, similarly to the example shown in Figure 2.3. Masked types support sound initialization of `TreeMap` nodes without using `null`.

2.5.4 Summary

Our experience is that J\mask is expressive, since it was easy to port classes with the various initialization patterns found in the Java Collection Framework. The explicit annotations in the ported code are infrequent and seem easy to understand, suggesting masked types are a natural way for programmers to enforce proper initialization of objects.

2.6 Related work

Non-null types. The importance of distinguishing non-null references from possibly-null references at the type level has long been recognized. Many languages, including CLU [46], Theta [47], Moby [29], Eiffel [39], ML [54], and Haskell [40], support some form of non-null and possibly-null types in their type system. In the context of Java, several proposals [10, 42, 20] have been made to support non-null types.

With non-null types, sound object initialization is usually accomplished by severely restricting expressiveness. Most existing languages with non-null types restrict how objects can be initialized; for example, some require all (non-null) fields to be initialized at once [29, 47]. This means fields and methods of an object under construction cannot be used. Further, cyclic data structures are impossible to initialize without using a placeholder value such as `null`.

Masked types are different from non-null types: when a field is masked, it is potentially uninitialized and unreadable, and therefore reading that field is statically disallowed; with non-null types, a field is always accessible regardless of how it is declared.

Fähndrich and Leino [25] make use of *raw types* to represent objects that are in the middle of being constructed, that is, objects with some non-null fields containing nulls. Methods can be declared to expect raw objects, and therefore can be called from within the constructors. Delayed types [27], extended from [25], provide a solution to the problem of safely initializing cyclic data structures, by introducing labels on object types, which represent the time by which an object is fully initialized. Delay times are associated with scopes, and form a stack at run time. Objects created with a delay time remain raw until execution exits the corresponding scope. Initialization of cyclic structures is supported by giving objects the same delay, and they become initialized together at once.

Compared to raw types, masked types provide a finer-grained representation of objects under construction. Conditional masks and delayed types are both means to track dependencies between objects under construction. However, delay times are an indirect way to represent dependencies, whereas conditional masks capture dependencies directly and explicitly. Therefore, programmers may have better control of the initialization process with masked types. Masked types also have richer subtyping relationships, which can be used to enforce reinitialization.

Typestates. In most object-oriented programming languages, an object has the same type for its entire lifetime. However, objects often evolve over time, that is, having different states at different times. Typestates [76] abstractly describe object states, and when an object is updated, its typestate may also change.

Typestates have been used to express and verify various protocols [76, 17, 18, 4, 28]. Typestates have been interpreted as abstract states in finite state machines and as predicates over objects.

Masked types are not intended for checking general protocols, but rather just focus on safe object initialization. However, masks cannot be easily encoded in terms of previous typestate mechanisms. Algebraic masks, for instance, provide compact representations of partial initialization states without requiring abstract states potentially exponential in the number of fields. Conditional masks represent dependencies generated at use sites, rather than being fixed at declaration sites of predicates. Mask subtyping enriches the state space, and previous work on typestates does not appear to have anything like it.

J\mask uses subclass masks and mask constraints to ensure modular type checking. These techniques are related to rest typestates and sliding methods in Fugue [18]. However, Fugue requires that sliding methods are overridden in every subclass, whereas mask constraints in J\mask force methods to be overridden only when their watched abstract masks are overridden.

Aliasing has always been a hard problem for any typestate mechanism: first, it is not easy to maintain correct typestate information in the presence of aliasing; second, although there are typing mechanisms like linear types that help keep track of aliases, they are inconvenient for ordinary programmers. Previous work on typestates has proposed various treatments to the aliasing problem: Nil [76] completely rules out aliasing; Vault [17] and Fugue disallow further state changes once an object becomes aliased unless the changes are temporary; Bierhoff and Aldrich [4] refine the two aliasing annotations "not aliased" and "maybe aliased" in Fugue to a richer set of permissions; Fähndrich and Leino [26] also identify a kind of typestates that are heap-monotonic and work without aliasing information; Fink et al. [28] conduct whole-program verification and rely on a global alias analysis. The treatment of the aliasing problem in J\mask is inspired by [26]: simple masks and conditional masks are heap-monotonic, and must-masks, though not heap-monotonic, are associated with newly created objects whose aliasing information is easy to track. We believe J\mask achieves a good trade-off between expressiveness and simplicity for the aliasing problem in the context of object initialization.

Masked types are reminiscent of type-based access control mechanisms that statically restrict access to individual fields or methods, e.g., [41, 69]. However, masked types are very different; they are designed for reasoning about initialization, and access is "granted" by the act of assignment to the resource, which makes little sense as an access control feature.

Static analysis. J\mask, similar to other typestate mechanisms, has a flow-sensitive type system, which can be viewed as a dataflow analysis. An alternative to masked types is an interprocedural def-use analysis, but this would lose many of the advantages of masked types. Java already has an intraprocedural analysis [78] to ensure that every local variable is definitely assigned before it is used. However, Java cannot safely prevent reading from uninitialized fields. There has been work on interprocedural def-use analysis in the context of object-oriented languages [74, 73], with varying cost and precision. This prior work detects initialization bugs on fields, but requires non-modular whole-program def-use analyses and is subject to the typically limited accuracy of whole-program alias/points-to analyses. By contrast, type checking in J\mask is modular and therefore scalable. Masked types bring another benefit because they specify the initialization contracts of methods, helping programmers reason about the code. Explicitly capturing this aspect of programmer intent seems valuable.

FindBugs [33] contains an analysis [34] that is designed specifically to detect null-pointer bugs. The analysis is neither sound nor complete, but focuses on improving accuracy. The basic analysis is interprocedural, but extensions are proposed in which non-null annotations are inserted into method signatures to represent contracts.

Shape analyses are aimed at extracting heap invariants that describe the "shape" of recursive data structures [86]. Conditional masks capture some part of the shape information of the data structure under construction. However, conditional masks are not concerned with initialized fields, and also are more about dependencies than the shape of references, and therefore have transitivity and cycle cancellation. Shape analyses are normally built upon alias analyses, and contain explicit representation of heap locations, neither of which is present in the J\mask language. J\mask only tracks mask changes on local variables, which gives it a flavor of local reasoning somewhat similar to the analysis in [11].

Because they summarize a set of concrete fields, abstract masks have some similarity to data groups [44], a mechanism used for modular program verification. Data groups do not have the equivalent of mask algebra. Moreover, masked types are about more than just abstracting fields; must-masks and conditional masks are new mechanisms that enable sound initialization of complicated data structures.

Other kinds of languages. The initialization problem is not unique to object-oriented languages. In a purely functional programming style, values are constructed all at once, avoiding the creation of partially initialized values. However, functional languages typically do not easily support the construction of cyclic data structures well, though it can be achieved in some cases with *value recursion* [80]. The typed assembly language in [55] supports initialization flags that are similar to the simple masks in J\mask.

CHAPTER 3
IMPROVING FAMILY EXTENSIBILITY WITH CLASS SHARING

Relationships (subclassing, referencing, etc.) between classes posit challenges not only for initializing objects, but also for modularly extending existing code with new behavior. Inheritance and adaptation are the two notable ways of extending software modularly. Inheritance creates new classes enhanced with new functionality, and adaptation adds new functionality to existing objects without modifying the original class definition. However, both inheritance and adaptation operate on individual classes and do not support coordinated changes that span *families* of related, interacting classes. Therefore, extensibility provided by these mechanisms does not scale well to large software systems.

In the context of inheritance, recent work on *family inheritance*, e.g., [51, 22, 56, 58, 38], supports coordinated extensions to families of classes. This chapter presents *class sharing* as the first language mechanism that integrates family inheritance and in-place, bidirectional adaptation. Class sharing is implemented in the J&$_s$ language, which is an extension to the J& language presented in [58] that supports family inheritance. Class sharing in this chapter is *heterogeneous*: families might not share all the classes. Masked types, presented in 2, are used to ensure the type safety of J&$_s$. Chapter 4 shows a *homogeneous* version of sharing mechanism, which is more straightforward and more scalable.

3.1 Sharing classes between families

The new class sharing mechanism in J&$_s$ is a safe, modular mechanism that relaxes the disjointness of class families. A family of classes may not only inherit another family (and hence all its nested classes), but may also *share* some of the classes from the family it inherits from. This enables new kinds of extensibility.

3.1.1 Family inheritance

J&$_s$ builds on the family inheritance mechanisms introduced by Nystrom et al.: nested inheritance [56] and nested intersection [58].

Nested inheritance is inheritance at the granularity of a *namespace* (a package or a class), which defines a family in which related classes are grouped. Nested inheritance supports coordinated changes that span the entire family. When a namespace inherits from another (base) namespace, not only are fields and methods inherited, but also namespaces nested in the base namespace. In addition, the derived namespace can *override*, or *further bind* [50] inherited namespaces, changing nested class declarations. Nested inheritance does not provide adaptation, however. Nested classes in the inheriting namespace, even those not overridden (*implicit classes*), are different classes than those of the same name in the base namespace.

Nested intersection supports *composing* families with generalized *intersection types* [68, 14]. Given classes S and T, their intersection $S\&T$ inherits all members of S and T. When two namespaces are intersected, their common nested namespaces are themselves intersected, i.e., $(S\&T).C = (S.C)\&(T.C)$.

```
class AST {
  class Exp {...}
  class Value
     extends Exp {...}
  class Binary
     extends Exp
  { Exp l, r; }
  ...
}
```

```
class TreeDisplay {
  class Node {
    void display() {...}
  }
  class Composite extends Node {
    Node getChild(int i) {...}
  }
  class Leaf extends Node {...}
}
```

Figure 3.1: An expression family and a GUI family

```
class ASTDisplay extends AST & TreeDisplay {
  class Exp extends Node { void display() { ... } }
  class Value extends Exp & Leaf { ... }
  class Binary extends Exp & Composite
    { void display() { ... l.display(); ... } }
  void show(Exp e) { e.display(); }
}
```

Figure 3.2: Mixing display into expressions, with nested inheritance

Figure 3.1 shows an example with two families of classes. Class AST contains a family of classes for representing expressions, and class TreeDisplay contains a family of classes for graphically visualizing trees. Figure 3.2 shows the skeleton of code that implements the functionality of displaying an expression as a tree—without changing the existing families. ASTDisplay inherits both AST and TreeDisplay, and therefore inherits all nested classes from both of them. The GUI classes are implicit in ASTDisplay, and expression classes are further bound to inherit GUI classes in addition to their original superclasses, and to override GUI methods with appropriate rendering code. As the show method in Figure 3.2 demonstrates, expression classes in the new family support the added display method.

Two mechanisms are essential for nested inheritance to work: *late binding of type names* and *exact types*. These mechanisms are also important for class sharing.

Late binding of type names. Late binding of type names ensures relationships between classes are preserved in the derived family.

When the name of a class is inherited into a new namespace, the name is interpreted in the context of the new namespace. For example, inside the family `ASTDisplay`, the type `Exp` refers to `Exp` in `ASTDisplay`. Consider the field `l`, which is declared in `AST.Binary` with type `Exp`, and then inherited by the class `ASTDisplay.Binary`. When inherited, its type is the `Exp` of `ASTDisplay`, *not* the original type. This late binding makes the call `l.display()` on line 5 legal. Similarly, the superclass of `ASTDisplay.Binary` is `ASTDisplay.Exp`, not `AST.Exp`.

Two mechanisms make the late binding of type names type-safe: *dependent classes* and *prefix types*. The dependent class *p*.`class` represents the run-time class of the object referred to by the *final access path p*. A final access path is either a final local variable, including `this` and final formal parameters, or a field access *p*.`f`, where *p* is a final access path and `f` is a final field. In general, the class represented by *p*.`class` is statically unknown, but fixed. A prefix type *P*`[`*T*`]` represents the enclosing namespace of the class or interface *T* that is a subtype of the namespace *P*, i.e., the family at the level of *P* that contains *T* [56]. In Figure 3.2, if one writes `AST[this.class]`, it refers to either `AST` or `ASTDisplay`, depending on the run-time class of the value stored in `this`.

Type names that are not fully qualified are sugar for members of prefix types that depend on the current class **this**.`class`, and in an inheriting family, they will be reinterpreted. In Figure 3.1, the type `Exp` inside class `AST` is sugar for `AST[this.class].Exp`.

Exact types. Both dependent classes and prefix types of dependent classes are *exact types* [7]: all instances of these types must have the same run-time class.

Simple types may be exact too. If A is a class, the exact type $A!$ represents values of the run-time class A. Even if class B inherits from A, neither B nor $B!$ is a subtype of $A!$. Exactness applies to the entire type preceding "!", so supertypes of a simple exact type can be obtained by shifting the exactness outward. For example, `ASTDisplay.Exp!` is not a subtype of `AST.Exp!`, but it is a subtype of `ASTDisplay!.Exp`, which is a subtype of `ASTDisplay.Exp`.

Exact types also restrict the subtyping relationships of nested types. For example, `ASTDisplay!.Binary` is not a subtype of `AST!.Binary`, even though `ASTDisplay.Binary` is a subtype of `AST.Binary`. Therefore exact types can mark the boundary of a family: classes nested in `ASTDisplay!` form one family and those nested in `AST!` form another. Non-dependent exact types provide a *locally closed world*: at compile time, one can enumerate *all* classes that are subtypes of $A!.C$, without inspecting subclasses of A. Exact types are important for the modularity of J&$_s$ (Section 3.1.5).

3.1.2 Sharing declarations

J&$_s$ introduces shared classes with *sharing declarations* in the derived family, such as the declaration "`shares A.C`" in this code:

```
class A { class C ... } // the base family
class B extends A {     // the derived family
  class C extends D shares A.C { ... }
}
```

This sharing declaration establishes a *sharing relationship* between classes `A.C` and `B.C`. Sharing declarations induce an equivalence relation on classes that is the reflexive, symmetric, and transitive closure of the declared sharing relationships. If two classes have a sharing relationship, they have the same set of object instances. However, subclasses of the two shared classes are not automatically shared, unless the subclasses also have appropriate sharing declarations. Therefore, the sharing relationship established in the above example can be represented as a relationship between two exact types, written $\mathtt{A.C!} \leftrightarrow \mathtt{B.C!}$.

J&$_s$ requires that only an overriding class in a derived family (e.g., `B.C`) may declare a sharing relationship with the overridden class in a base family (e.g., `A.C`). This restriction helps keep shared classes similar to each other.

Sharing vs. subtyping. The J&$_s$ language keeps subtyping largely separate from sharing. Adding a sharing relationship does not change the subtyping relation. In the above example, `B.C` is a subtype of `A.C`, and `B!.C` is not a subtype of `A!.C` due to the exactness, whether there is a sharing declaration or not. The sharing relationship does not make `A.C` a subtype of `B.C`, nor does it create any subtyping relationship between `A!.C` and `B!.C`.

Since sharing is not subtyping, it is in general not allowed to directly treat an object of a shared class as an instance of the other class in the sharing relationship; an explicit *view change* may be required (see Section 3.1.3 for more details).

Family adaptation. Sharing solves the problem of *object adaptation* [85], in which the goal is to augment existing objects with new behavior or state. Adap-

```
class ASTDisplay extends AST & TreeDisplay {
  class Exp extends ... shares AST.Exp { ... }
  class Value extends ... shares AST.Value { ... }
  class Binary extends ... shares AST.Binary
    { void display() { ... l.display(); ... } }
  void show(AST!.Exp e) sharing AST!.Exp = Exp {
    Exp temp = (view Exp)e;
    temp.display();
  }
}
```

Figure 3.3: Using class sharing to adapt `Exp` to `TreeDisplay`

tation is different from inheritance, where only objects of new classes have the new behaviors. J&$_s$ is the first language to fully integrate family inheritance with in-place adaptation that preserves object identity. Moreover, because sharing is an equivalence relation on classes, J&$_s$ supports bidirectional, transitive adaptation.

Adaptation can improve the example code from Figures 3.1 and 3.2. Although `ASTDisplay` in Figure 3.2 provides expression classes extended with the ability to display themselves, this new functionality is not available for instances of the original classes in `AST`. This is unfortunate, because instances of the original classes might be created by existing library code or deserialized from a file. We can avoid this limitation by using class sharing as shown in Figure 3.3. Here, `shares` clauses cause the two families `ASTDisplay` and `AST` to share all the expression classes. Instances of `AST` expression classes are also instances of corresponding `ASTDisplay` expression classes.

Because of sharing, expression objects from the `AST` family can possess GUI display operations, even if the implementer of `AST` was not aware of `TreeDisplay` or `ASTDisplay`; client code—for example, a visualization toolkit, which expects `TreeDisplay` objects—would obtain the ability to handle existing AST objects. Thus every expression class in the ASTDisplay family becomes an *adapter* [30] for the corresponding class in AST; the ASTDisplay family provides *family adaptation* for AST.

Given a tree of expression objects from AST, family adaptation ensures that the whole tree is safely adapted to ASTDisplay. The adaptation preserves the original tree structure, and the relationships between objects in the tree. This contrasts with prior adaptation mechanisms that work on individual objects, which do not guarantee safety or the preservation of object relationships. With prior mechanisms such as the adapter design pattern [30], one might forget to adapt the left child of a Binary object, and the call l.display() on line 5 would fail.

In Figure 3.3, every expression class has a sharing declaration, which could be tedious to write. J&$_s$ provides the adapts clause as a shorthand for adding sharing declarations to all inherited member classes. For example, ASTDisplay may be declared with the following class header, without any individual sharing declarations:

```
class ASTDisplay extends ... adapts AST { ... }
```

3.1.3 Views and view changes

Views. If two classes are shared, a single object might be treated as an instance of either one. Each class is a distinct *view* of that object. A J&$_s$ object can have any number of views, all equally valid. This contrasts with an ordinary object-oriented language like Java (or even J&), where an object has exactly one view:

its run-time class.

In J&$_s$, an object reference is not just a heap location; it is essentially a pair $< \ell, T >$ of a heap location ℓ and a type T, where T is the view, represented as a non-dependent exact type. The view T determines the behavior of the object when accessed through that reference.

For example, the method `display` in Figure 3.3 cannot be directly called on an object created as an instance of the class `AST.Binary`, because the reference has the view `AST.Binary!`. However, when the object obtains a new reference—for example, by storing the object in a local variable—with the view `ASTDisplay.Binary!`, it also obtains a new behavior—the method `display` becomes available through the new reference. Moreover, methods that are available through the original reference might behave differently when they are called through the new reference. See Section 3.1.4 for an example.

View changes. J&$_s$ has a *view change* operation (`view` T)e, which generates a new reference with the same heap location but a different view. On line 7 in Figure 3.3, the method `show` contains a view change expression (`view Exp`)`e`, the result of which is a reference that still points to the same object as `e` but with a new view that is a subtype of `Exp`. (Recall from Section 3.1.1 that within `ASTDisplay`, `Exp` is sugar for `ASTDisplay[this.class].Exp`).

View changes support late binding. Although the expression (`view` T)e has a statically known type T, the actual run-time view of the result is a subtype of T that is shared with the run-time view of the value of e. For example, in line 7 of Figure 3.3, if `e` evaluates to a value with the view `AST.Value!`, then within the family `ASTDisplay`, the expression (`view Exp`)`e` produces a value with the view `ASTDisplay.Value!`, which is a subtype of `ASTDisplay!.Exp`, and shared with `AST.Value`.

A view change (`view` T)e looks syntactically like a type cast (T)e, and it does have the same static target type T, but it is actually quite different. First, a type cast might fail at run time, but view changes that type-check always succeed. Second, no matter whether it is an upcast or a downcast, an ordinary cast only works if the target type is a supertype of the *run-time* class of the object. However, a view change has in general a target type that is neither a supertype nor a subtype of the current view of the object, but from another family. For example, if families `A` and `B` share the class `C`, together with all its subclasses within each family, the following code is legal:

```
A!.C a = new A.C();
B!.C b = (view B!.C)a;
```

The initial view of the created object is `A.C!`. The target type `B!.C` in the view change is neither a supertype nor a subtype of `A.C!`. Of course, if the type in a view change is indeed a supertype of the current view of the instance, the view change is a no-op.

Third, a type cast checks run-time typing information, but has no other effects at run time. By contrast, a view change can affect the behavior of the object.

View-dependent types. Nested inheritance uses the dependent class p`.class` to indicate the family that the object referenced by p belongs to. J&$_s$ generalizes p`.class` to be *view-dependent*, since an object may be a member of multiple families. For example, suppose `ASTDisplay` contained the following code:

```
AST!.Binary a = new AST.Binary();
Binary b = (view Binary)a;
```

At run time, a.class would denote AST.Binary!, and b.class would denote ASTDisplay.Binary!. This example shows that in J&$_s$, the dependent classes associated with different aliases (a and b in this example) are not necessarily equal; they are interpreted as the views associated with the respective references.

Prefix types of dependent classes are also view-dependent, ensuring that late-bound type names belong to consistent families. Consider the left child of class Binary, stored in the field l. The type of the field is Exp, sugar for AST[this.class].Exp, which depends on the view of the object that contains the field. Accessing the left child with a.l returns an object in the family AST, whereas accessing it with b.l returns the same object, but with a view in the family ASTDisplay. In either case, the left child object and the containing Binary object have views that are in the same family. This means that for a tree of objects in one family, a single explicit view change on the root object effectively moves the whole tree to another family, implicitly triggering view changes on child objects as they are accessed.

3.1.4 Dynamic object evolution via view change

View-based dynamic method dispatching. In J&$_s$, method calls are dispatched on the current *view* of the receiver object, rather than on the receiver itself as in Java. When two references to the same receiver object have different views, the same method call may invoke different code. Therefore, when a J&$_s$

class overrides a method inherited from its declared shared class, both versions of the method become available to the object, and the choice is made at run time, based on the view associated with the reference.

Method dispatching in J&$_s$ differs from that of nonvirtual methods in C++ [77], and from the statically scoped adaptation in *expanders* [85]. For non-virtual methods in C++, the static type of the receiver acts as a static view that selects the method to call. By contrast, the view change operation in J&$_s$ affects method dispatching, but still allows late binding, since the type T in a view change (`view` T)e is a supertype of the statically unknown run-time view.

Expanders dispatch methods within the same family dynamically, but unlike with class sharing, the choice between original code and expander code is static. Expander methods can be invoked only when expanders are in scope, and therefore the behavior of existing code written before the expanders cannot change. Since expansion is *not* inheritance, expander methods can only *overload* original methods rather than *overriding* them.

Dynamic object evolution. View-based method dispatching enables a new form of dynamic object evolution, in which existing objects are updated with different behavior without breaking running code. It is more powerful than object adaptation, which generally only adds new behavior to existing objects, whereas evolution can also make objects change behavior, even in a context that does not mention the updated classes. J&$_s$ supports evolution at the family level, evolving interacting objects consistently to the new family.

For example, Figure 3.4 shows a package `service` that implements several network services, and a dispatcher that calls different services based on the kind

```
package service;
class SomeService {
  void handle(Packet p) {...}
}
...
class Dispatcher {
  SomeService s;
  void dispatch(Packet p) {
    switch (p.kind) {
      case 0: s.handle(p);
      ...
    }
  }
}
```

```
package logService extends
  service;
class SomeService shares
  service.SomeService {...}
...
class Logger {...}
class Dispatcher shares
  service.Dispatcher {
  Logger logger;
  void dispatch(Packet p) {
    logger.log(...);
    ...
  }
}
```

Figure 3.4: Evolution of a network service package

of the received packet. The server code has a static field storing the dispatcher, and an event loop:

```
static service.Dispatcher disp;
...
while (true) { ... disp.dispatch(p); ... }
```

Suppose the system implemented in the `service` family has started running, and then an updated package `logService` is developed that extends `service` with logging at various places (Figure 3.4 only shows the additional logging in the `dispatch` method). The goal is to update the system with logging ability, without having to stop it from running.

To evolve the system from `service` to `logService`, the view of the dispatcher object stored in the static field `disp` needs to be changed. This could be done in initialization code in the extended package, as follows:

```
  service!.Dispatcher d =
    (service!.Dispatcher)Server.disp; // cast
  Server.disp = (view Dispatcher)d;   // view change
```

After this view change, the `dispatch` method overridden in the extended package will be called.

More importantly, although just a single explicit view change is applied to the dispatcher object, all the other objects transitively reachable from the dispatcher, such as through the field `s` of type `SomeService`, will also obtain new views when they are accessed, resulting in using the versions of their methods that have logging enabled. Thus, a single explicit view change causes a consistent evolution of many objects to the new family. This kind of evolution is simpler to implement and likely to be more efficient than going through all the objects and updating them individually.

J&$_s$ allows the old version before the update to coexist with the new version, because each object can have multiple views at the same time. This is useful, for example, when an Internet service system is upgraded, to allow existing user sessions to continue communicating with the old version, without any service disruption or inconsistent behavior caused by the upgrade.

3.1.5 Sharing constraints

Sharing relationships between types are not always preserved by derived families, either by design or as the result of changed class hierarchy in the derived family. For example, in a family inherited from `ASTDisplay`, one might choose not to share any class, or to share classes from `TreeDisplay`, and in either case, the new family no longer shares classes with the `AST` family.

Therefore, a view-change operation that works in the base family might not make sense in the derived family. J&$_s$ does not try to check all inherited method code for inapplicable view changes, but rather makes checking modular via *sharing constraints*. A J&$_s$ method can have sharing constraints of the form `sharing` $T_1 = T_2$, which means that any value of type T_1 can be viewed as of type T_2, and vice versa; in other words, the sharing relationship $T_1 \leftrightarrow T_2$ may be assumed in the method body. A view change can only appear in a method with an enabling sharing constraint. Outside the scope of sharing constraints, the type checker (and programmer) need not be concerned with sharing. Therefore, reasoning about class sharing is local.

For example, in Figure 3.3, the method `show` has a sharing constraint `AST!.Exp = Exp` (line 6), which allows the view change (`view Exp`)`e` to be applied to the variable `e` of static type `AST!.Exp`.

To know statically that the view change (`view Exp`)`e` will succeed, we must know that every subclass of `AST!.Exp` has a corresponding shared subclass under `ASTDisplay!.Exp`. J&$_s$ requires that some prefix of each type in the constraint is exact, and either non-dependent or only dependent on the path `this`; thus, we can check all the subclasses in a locally closed world (Section 3.1.1), without a whole-program analysis. In this example, the exact prefixes in question are `AST!` and `ASTDisplay[this.class]`.

The type checker verifies that sharing constraints in the base family still hold in the derived family; base family methods whose sharing constraints do not

```
class A1 {
  class B { }
  class C { D g; }
  class D { }
}
class A2 extends A1 {
  class B shares A1.B {
    T f;                    // a new field
  }
  class C shares A1.C\g { } // shared with a mask
  class E extends D { }     // a new subclass of D
}
```

Figure 3.5: Shared classes with unshared fields

hold must be overridden.

Although sharing constraints support modular type checking, they do introduce an annotation burden for the programmer. Our experience suggests that the annotation burden is manageable. While it appears possible to automatically infer sharing constraints, by inspecting the type of the source expression and the target type of every view change operation in the method body, we leave this to future work.

3.2 Protecting unshared state with masked types

3.2.1 Unshared fields

In J&$_s$, shared classes do not necessarily share all their fields. Unshared fields are important for greater extensibility, but they pose challenges for the safety of the language. J&$_s$ uses *masked types* [64] and duplicate fields to ensure safety in the presence of unshared fields.

Figure 3.5 illustrates the two kinds of unshared fields. First, new object fields may be introduced by shared classes in the derived family. Because these fields do not exist in the base family, they cannot be shared. In Figure 3.5, classes `A1.B` and `A2.B` are shared, but `A2.B` introduces a new field `f` that does not exist in `A1.B`. When a view change from the base family to the derived family—for example, from `A1!.B` to `A2!.B`—is applied, the new field (e.g., the field `f` in Figure 3.5) would be uninitialized, which may be unexpected to the code in the derived family.

J&$_s$ uses masked types to prevent the possibly uninitialized new field from being read. A masked type, written $T \backslash f$, is the type T without read access to the field f. We say that the field f is *masked* in $T \backslash f$. The mask on a field can be removed with an assignment to that field. Masked types introduce the subtyping relationship $T \leq T \backslash f$. For the example in Figure 3.5, a view change from `A1!.B` to `A2!.B` must have a mask on the target type:

```
A1!.B b1 = new A1.B();
A2!.B\f b2 = (view A2!.B\f)b1;
```

Therefore, after this view change, the field `f` of `b2` cannot be read before it is initialized.

The second kind of unshared field is those with unshared types. In Figure 3.5, the field `g` has type `D`, and it cannot be shared, because in the `A2` family, `g` might store an object of class `E`. When a view change is applied from the derived family to the base family (for example, from `A2!.C` to `A1!.C`), an `A2.E` object stored in `g` would not have a view compatible with the base family.

Therefore, J&$_s$ requires that fields with unshared types are masked in the sharing declaration. A duplicate field with the same name for the shared class is also generated automatically in the derived family. For example, in Figure 3.5, it is as if the class `A2.C` has its own implicit declaration of field `g`:

```
class C shares A1.C\g {
  D g;
}
```

An instance of `A1.C` and `A2.C` contains two copies of the field `g`, each appearing as a "new" field to the other class. Which copy is accessed depends on the current view of the instance through which the field `g` is accessed. Field duplication prevents objects of an unshared class from being accidentally accessed in other families. Therefore, interacting objects always have consistent views.

When shared classes have duplicate fields, it is up to the programmer how to keep the copies in sync, or even whether to keep them in sync. The programmer may choose to construct a corresponding object in the target family, storing it in the duplicate field, as in the example in Section 3.2.2, or just to leave the field masked in the target family.

3.2.2 In-place translation with unshared classes

J&$_s$ supports in-place translation of data structures between families in which not all the classes are shared (translation is trivial if all classes are shared). As the data structure is translated, some objects in the structure may remain the same, with only a view change, and other objects, particularly those of unshared types, are explicitly translated. One use of this kind of translation is for compilers,

```
package base;
abstract class Exp
{ ... }
class Var extends Exp {
  String x;
  ...
}
class Abs extends Exp {
  String x;
  Exp e;
  ...
}
```

```
package pair extends base;
abstract class Exp
  { abstract base!.Exp translate
    (Translator v); }
class Pair extends Exp
  { Exp fst, snd; ...}
class Translator {
  base!.Abs reconstructAbs
    (Abs old, String x,
     base!.Exp exp) { ... }
  ...
}
```

Figure 3.6: Lambda calculus and pair compiler extension

where the data structure is an abstract syntax tree, and many parts of the AST do not need to change during a given compiler pass.

Figure 3.6 shows the skeleton of a simple compiler implemented with family inheritance but without class sharing, modeled on the Polyglot extensible compiler framework [57]. This example translates the λ-calculus extended with pairs, into the ordinary λ-calculus. The package `base` defines the target language through class declarations for AST nodes of the λ-calculus (e.g., `Abs` for λ abstractions); the package `pair` extends the source language with one additional AST node `Pair`. Classes inherited from `base` are further bound in `pair` with `translate` methods that recursively translate an AST from `pair` to `base`. The class `Translator` provides the methods AST classes use to (re)construct nodes in the target family using translated versions of their child nodes; the method `reconstructAbs` does this for λ abstractions. However, without sharing, the translation from `pair` to `base` has to recreate the whole AST, even for trivial cases like `Var`, because the two families `base` and `pair` are completely disjoint despite their structural similarity.

By contrast, Figure 3.7 shows the J&$_s$ code that does in-place translation. The

```
1  package pair extends base;
2  abstract class Exp shares base.Exp {
3    abstract base!.Exp translate(Translator v);
4  }
5  class Var extends Exp shares base.Var { ... }
6  class Abs extends Exp shares base.Abs\e {
7    base!.Exp translate(Translator v) {
8      base!.Exp exp = e.translate(v);
9      return v.reconstructAbs(this, x, exp);
10   }
11 }
12 class Pair extends Exp {
13   Exp fst, snd;
14   base!.Exp translate(Translator v) {
15     return new ...; // (λx.λy.λf. f x y)⟦fst⟧⟦snd⟧
16   }
17 }
18 class Translator {
19   base!.Abs reconstructAbs(Abs old, String x,
20                            base!.Exp exp)
21   sharing Abs\e = base!.Abs\e {
22     if (old.x == x && old.e == exp) {
23       base!.Abs\e temp = (view base!.Abs\e)old;
24       temp.e = exp;
25       return temp;
26     }
27     else return new base.Abs(x, exp);
28   }
29   ...
30 }
```

Figure 3.7: In-place translation of the pair language

`base` family remains the same as in Figure 3.6, and is omitted. Classes `Var` and `Abs` in `pair` are declared to share corresponding classes in `base` (lines 5–6). `Pair` is not shared, because it does not exist in `base`.

Consider the two shared classes `pair.Abs` and `base.Abs`. The type of their field `e`, which is `base[this.class].Exp`, is interpreted as `pair!.Exp` and `base!.Exp` in the two families. The two interpreted field types are not shared, because a value of type `pair!.Exp` might have run-time class `pair.Pair`, which has no corresponding shared class in the `base` family. Therefore the two `Abs` classes each have their own copies of the field `e`, and the sharing declaration on line 6 has a mask on `e`.

Similarly, the sharing constraint on line 21 also has masks, and so does the view change operation on line 23, where the `Abs` instance is reused if all its subexpressions have been translated in place. The sharing constraint, together with the corresponding view change operation, has a mask on the field `e`, in case it points to an instance of an unshared subclass of `Exp`, such as `Pair`. The mask is removed on line 24, after which the type of the variable `temp` becomes `base!.Abs`.

As shown in the above example, J&$_s$ uses masked types to prevent objects of unshared types, such as `Pair`, from being leaked into an incompatible family (here, `base`). Masked types do introduce some annotation burden. However, class sharing is expected to be used in practice between families that are similar to each other, where extensibility and reuse are needed and make sense. In that case, there should not be many masks.

3.2.3 Translation from the base family to the derived family

Translations in different directions are not always of the same difficulty. Section 3.2.2 proposes a solution for in-place translation from the derived family (`pair`) to the base family (`base`). The complexity of the solution arises mostly because the objects of the `Pair` class must be translated away. However, as noted in [15], in-place translation in the other direction, that is, from `base` to `pair`, should be almost trivial, by treating an AST in `base` as an AST in `pair`.

To capture the asymmetry, J&$_s$ supports *directional* sharing relationships between types that are possibly masked, represented as $T_1 \rightsquigarrow T_2$, which means that an object of static type T_1 may be applied a view change with target type T_2. In Figure 3.7, the J&$_s$ compiler may infer the sharing relationship `base!.Exp` $\rightsquigarrow$ `pair!.Exp`, (the other direction `pair!.Exp` $\rightsquigarrow$ `base!.Exp` does not hold, because of the class `Pair`), and allows a constant-time in-place translation from `base` to `pair`, by a view change from `base!.Exp` to `pair!.Exp`.

Note that in this case, the sharing declaration on line 6 in Figure 3.7 actually induces the following two directional sharing relationships:

`base.Abs!` $\rightsquigarrow$ `pair.Abs!`

`pair.Abs!\e` $\rightsquigarrow$ `base.Abs!\e`

rather than just one bidirectional sharing relationship `base.Abs!\e` $\leftrightarrow$ `pair.Abs!\e`.

3.3 Formal semantics and soundness

3.3.1 Grammar

The grammar of the language is shown in Figure 3.8. 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. Depending on context, $\overline{a}$ sometimes denotes the set containing all list members. Types with multiple masked fields are written as $T \backslash \overline{f}$.

Every class has a `shares` clause. If a class C does not share any other class, the type that appears in its `shares` clause is C itself.

programs	Pr	$::= < \overline{L}, e >$
class declarations	L	$::= \texttt{class}\ C\ \texttt{extends}\ T_1\ \texttt{shares}\ T_2\ \{\overline{L}\ \overline{F}\ \overline{M}\}$
field declarations	F	$::= [\texttt{final}]\ T\ f = e$
method declarations	M	$::= T\ m(\overline{T}\ \overline{x})\ \texttt{sharing}\ \overline{Q}\ \{e\}$
sharing constraints	Q	$::= T_1 \rightsquigarrow T_2$
pure types	PT	$::= \circ \mid PT.C \mid p.\texttt{class} \mid P\texttt{[}PT\texttt{]} \mid \texttt{\&}\overline{PT} \mid PT!$
types	T	$::= PT \mid PT\backslash\overline{f}$
pure non-dependent types	PS	$::= \circ \mid PS.C \mid P\texttt{[}PS\texttt{]} \mid \texttt{\&}\overline{PS} \mid PS\texttt{!}$
non-dependent types	S	$::= PS \mid PS\backslash\overline{f}$
classes	P	$::= \circ \mid P.C$
values	v	$::= < \ell, S >$
access paths	p	$::= v \mid x \mid p.f$
expressions	e	$::= v \mid x \mid e.f \mid x.f = e \mid e_0.m(\overline{e}) \mid e_1;\ e_2$
		$\mid \texttt{new}\ T \mid (\texttt{view}\ T)e \mid \texttt{final}\ T\ x = e_1;\ e_2$
typing environments	Γ	$::= \emptyset \mid \Gamma, x{:}T \mid \Gamma, p_1 = p_2 \mid \Gamma, Q$

Figure 3.8: Grammar of the J&$_s$ calculus

Unlike Featherweight Java [36] or the J& calculus [59], object allocation does not provide any initial values for fields. Every field, in its declaration, specifies an initial value.

Methods have sharing constraints $\overline{Q}$, which are used to type-check view change operations, and to control method overriding. For simplicity, the formal semantics assumes all the classes are known, and the modular type-checking of sharing constraints is not modeled.

In J&$_s$, a value v, that is, a reference, is a pair of a heap location ℓ and a view S, which is a non-dependent exact type that may include masks.

Expressions are mostly standard, with the addition of a view change operation $(\texttt{view}\ T)e$. Without loss of generality, the receiver of a field assignment is assumed to always be a variable.

The typing environment contains aliasing information about access paths. An entry $p_1 = p_2$ means p_1 and p_2 are aliases, which also have the same run-time view. As in [12], this kind of information is not needed in the static semantics, but just in the soundness proof.

3.3.2 Auxiliary definitions

Some auxiliary definitions that are straightforward are only informally explained here.

- $\mathsf{super}(P)$ and $\mathsf{share}(P)$ represent the declared supertype and shared type of P. The set of all superclasses of S is $\mathsf{supers}(S)$.
- Well-formedness judgments of types and environments are represented as $\Gamma \vdash T$ and $\vdash \Gamma\ \mathsf{ok}$.
- $\mathsf{FV}(e)$ and $\mathsf{refs}(e)$ are the sets of free variables and free references in expression e.
- $\mathsf{pure}(T)$ strips all the masks from type T.

3.3.3 Class lookup

The class table $CT(P)$, defined in Figure 3.9, returns the declaration for *explicit* class P, or $\perp$ otherwise. In CT-OUT, the premise $Pr = < \overline{L}, e >$ indicates that the program consists of a set of top-level class declarations $\overline{L}$ and a "main" expression e, and the "outermost" class $\circ$ contains all the class declarations in $\overline{L}$.

Besides CT, there is also an extended class table CT', which includes implicit classes (but not any top-level intersection types). The class table CT' is needed

$\boxed{CT(P)}$

$$\frac{Pr =< \overline{L}, e >}{CT(\circ) = \texttt{class}\ \circ\ \texttt{extends}\ \&\mathsf{nil}\ \texttt{shares}\ \circ\ \{\overline{L}\}}\ (\text{CT-OUT}) \qquad \frac{\begin{array}{c} CT(P) = \texttt{class}\ C'\ \texttt{extends}\ T'_e\ \texttt{shares}\ T'_s\ \{\overline{L'}\ \overline{F'}\ \overline{M'}\} \\ \texttt{class}\ C\ \texttt{extends}\ T_e\ \texttt{shares}\ T_s\ \{\ldots\} \in \overline{L'} \end{array}}{CT(P.C) = \texttt{class}\ C\ \texttt{extends}\ T_e\ \texttt{shares}\ T_s\ \{\ldots\}}\ (\text{CT-EXP})$$

$\boxed{CT'(P)}$

$$\frac{CT(P) \neq \bot}{CT'(P) = CT(P)}\ (\text{CT}'\text{-EXP}) \qquad \frac{\begin{array}{c} CT'(P) \neq \bot \qquad CT(P.C) = \bot \qquad \vdash P \sqsubset^{*} P' \qquad CT'(P'.C) \neq \bot \\ T_e = \&_{P'' \sqsupset_{\mathsf{fb}} P.C}\ \mathsf{super}(P'') \qquad T_s = P.C \end{array}}{CT'(P.C) = \texttt{class}\ C\ \texttt{extends}\ T_e\ \texttt{shares}\ T_s\ \{\}}\ (\text{CT}'\text{-IMP})$$

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

$$\frac{CT'(P) \neq \bot}{\mathsf{mem}(P) = \{P\}} \qquad \frac{D = \{P_i \in \mathsf{mem}(PS) \mid CT'(P_i.C) \neq \bot\}}{\mathsf{mem}(PS.C) = \bigcup_{P_i \in D}\{P_i.C\}}$$

$$\mathsf{mem}(P[PS]) = \mathsf{prefix}(P, PS) \qquad \mathsf{mem}(\&\overline{PS}) = \bigcup_{PS_i \in \overline{PS}} \mathsf{mem}(PS_i) \qquad \mathsf{mem}(PS\,!) = \mathsf{mem}(PS)$$

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

$$\frac{\begin{array}{c} \vdash P_1 \sqsubset^{*} P \\ CT'(P.C) = \texttt{class}\ C\ \texttt{extends}\ T\ \ldots \\ T\{\!\{\emptyset;\ P_1/\texttt{this}\}\!\} = PS \\ P_2 \in \mathsf{mem}(PS) \end{array}}{\vdash P_1.C \sqsubset_{\mathsf{sc}} P_2}\ (\text{SC}) \qquad \frac{\vdash P_1 \sqsubset P_2}{\vdash P_1.C \sqsubset_{\mathsf{fb}} P_2.C}\ (\text{FB}) \qquad \frac{\vdash P_1 \sqsubset_{\mathsf{sc}} P_2}{\vdash P_1 \sqsubset P_2}\ (\text{INH-SC}) \qquad \frac{\vdash P_1 \sqsubset_{\mathsf{fb}} P_2}{\vdash P_1 \sqsubset P_2}\ (\text{INH-FB})$$

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

$$\frac{\begin{array}{c} \vdash P_1.C \sqsubset_{\mathsf{fb}} P_0.C \\ \vdash P_2.C \sqsubset_{\mathsf{fb}} P_0.C \end{array}}{\vdash P_1 \sim P_2}\ (\text{REL-FB}) \qquad \vdash P \sim P\ (\text{REL-REFL}) \qquad \frac{\vdash P_1 \sim P_2}{\vdash P_2 \sim P_1}\ (\text{REL-SYM}) \qquad \frac{\begin{array}{c} \vdash P_1 \sim P_2 \\ \vdash P_2 \sim P_3 \end{array}}{\vdash P_1 \sim P_3}\ (\text{REL-TRANS})$$

$\boxed{\text{Member lookup}}$

$$\frac{CT'(P) = \texttt{class}\ C\ \ldots\ \{\overline{L}\ \overline{F}\ \overline{M}\}}{\begin{array}{c} \mathsf{ownFields}(P) = \overline{F} \\ \mathsf{ownMethods}(P) = \overline{M} \end{array}} \qquad \frac{\overline{F} = [\texttt{final}]\ \overline{T}\ \overline{f} = \overline{e}}{\mathsf{fnames}(\overline{F}) = \overline{f}} \qquad \frac{\begin{array}{c} \Gamma \vdash T \trianglelefteq PS \\ \mathsf{methods}(PS) = \overline{M} \\ M_i = T_{n+1}\ m(\overline{T}\ \overline{x})\ \ldots\ \{e\} \end{array}}{\mathsf{mtype}(\Gamma, T, m) = (\overline{x}\!:\!\overline{T}) \rightarrow T_{n+1}}$$

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

$$\mathsf{fields}(S) = \bigcup_{P_i \in \mathsf{supers}(S)} \mathsf{ownFields}(P_i) \qquad \frac{\begin{array}{c} \mathsf{methods}(S) = \overline{M} \\ M_i = T_{n+1}\ m(\overline{T}\ \overline{x})\ \ldots\ \{e\} \end{array}}{\mathsf{mbody}(S, m) = M_i}$$

$$\mathsf{methods}(S) = \bigcup_{P_i \in \mathsf{supers}(S)} \mathsf{ownMethods}(P_i) \qquad \frac{\begin{array}{c} T_f^{decl} = \mathsf{ftype}_{decl}(\Gamma, T, f) \\ T_f = T_f^{decl}\{\!\{\Gamma;\ T/\texttt{this}\}\!\} \qquad T \neq T''\backslash f \end{array}}{\mathsf{ftype}(\Gamma, T, f) = T_f}$$

Figure 3.9: Classes

because sharing constraints must also be checked for implicit classes. In the rule CT′-IMP, a declaration of an implicit class $P.C$ is created: the declared supertype T_e is the intersection of the supertypes of all the classes that $P.C$ further binds; the shared type T_s just refers to the class itself with `this.class.`C. In CT′-IMP, $P'' \sqsupset_{\mathsf{fb}} P.C$ is a shorthand for $\vdash P.C \sqsubset_{\mathsf{fb}} P''$.

3.3.4 Subclassing

Inheritance among classes is defined in Figure 3.9. The judgment $\vdash P_1 \sqsubset_{\mathsf{sc}} P_2$ states that P_1 is a declared subclass of P_2, and $\vdash P_1.C \sqsubset_{\mathsf{fb}} P_2.C$ states that $P_1.C$ further binds $P_2.C$. We write $\vdash P_1 \sqsubset P_2$ if P_1 either subclasses or further binds P_2.

The definition uses the mem function, also shown in Figure 3.9, which returns the set of classes P comprising a pure non-dependent type PS.

3.3.5 Prefix types

The meaning of a non-dependent prefix type $P[PS]$ is defined using the function prefix(P, PS):

$$\mathsf{prefix}(P, PS) = \{P' \mid \exists\, C, C' .\ \vdash P \sim P' \wedge P.C \in \mathsf{supers}(PS) \wedge P'.C' \in \mathsf{supers}(PS)\}$$

If $\mathsf{prefixExact}_1(PS) = \mathsf{false}$ (see Section 3.3.7 for the definition of $\mathsf{prefixExact}_k(T)$), the P-prefix of a pure non-dependent type PS is the intersection of all classes P' where P and P' both inherit a common nested class—that is, P and P' are equivalent under the $\sim$ relation, shown in Figure 3.9—and PS extends nested classes of both P and P'. This definition ensures that if P is a subtype of P', then $P\,[\,PS\,]$ is equal to $P'\,[\,PS\,]$. If $\mathsf{prefixExact}_1(PS) = \mathsf{true}$, then $P\,[\,PS\,]$ is $(\&\mathsf{prefix}(P, PS))!$, which keeps the exactness of the index.

Note that the index PT of a prefix type $P\,[\,PT\,]$ can only be a pure type, as shown by WF-PRE in Figure 3.12. Also, as in [59], the J&$_s$ calculus only allow prefix types $P\,[\,T\,]$ where some supertype of T is immediately enclosed by a subclass of P, and more general prefix types can be encoded.

3.3.6 Member lookup

Figure 3.9 also shows auxiliary functions for looking up fields and methods: the functions $\mathsf{fields}(P)$ and $\mathsf{methods}(P)$ collect all the field and method declarations from P and its superclasses; $\mathsf{fnames}(\overline{F})$ is the set of all field names in field declarations $\overline{F}$; $\mathsf{ftype}_{decl}(\Gamma, T, f)$ returns the declared type of field f, which might be a type dependent on `this`; $\mathsf{ftype}(\Gamma, T, f)$ substitute the receiver type for `this.class`, assuming the receiver type has no mask on f; $\mathsf{mtype}(\Gamma, T, m)$ and $\mathsf{mbody}(S, m)$ look up the type and the declaration of a method m.

3.3.7 Final access paths and exactness

The judgment $\Gamma \vdash_{\mathsf{final}} p : T$, shown in Figure 3.10, gives the static type bound for final access path p. Note that F-REF does not need a premise, because a reference contains its own type.

The function $\mathsf{paths}(T)$ shown in Figure 3.11, returns the set of final access paths in the structure of type T.

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

$$\Gamma \vdash_{\mathsf{final}} {<}\,\ell, S\,{>} : S \ (\text{F-}\textsc{ref}) \qquad \frac{x : T \in \Gamma}{\Gamma \vdash_{\mathsf{final}} x : T}\ (\text{F-}\textsc{var}) \qquad \frac{\Gamma \vdash_{\mathsf{final}} p : T \qquad T_f = \mathsf{ftype}(\Gamma, T, f)}{\Gamma \vdash_{\mathsf{final}} p.f : T_f}\ (\text{F-}\textsc{get})$$

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

$$\frac{\begin{array}{c} p_1 = p_2 \in \Gamma \\ p_1 \neq x \\ p_2 \neq x \end{array}}{\Gamma \vdash p_1 = p_2}\ (\text{A-}\textsc{env}) \quad \frac{\begin{array}{c} \Gamma \vdash p_1 = p_2 \\ \Gamma \vdash_{\mathsf{final}} p_1.f : T_f \\ \Gamma \vdash_{\mathsf{final}} p_2.f : T_f \end{array}}{\Gamma \vdash p_1.f = p_2.f}\ (\text{A-}\textsc{field}) \quad \frac{\Gamma \vdash_{\mathsf{final}} p : T}{\Gamma \vdash p = p}\ (\text{A-}\textsc{refl}) \quad \frac{\Gamma \vdash p_2 = p_1}{\Gamma \vdash p_1 = p_2}\ (\text{A-}\textsc{sym}) \quad \frac{\begin{array}{c} \Gamma \vdash p_1 = p_2 \\ \Gamma \vdash p_2 = p_3 \end{array}}{\Gamma \vdash p_1 = p_3}\ (\text{A-}\textsc{trans})$$

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

$$\Gamma \vdash T \rightsquigarrow T \ (\text{SH-}\textsc{refl}) \qquad \frac{\begin{array}{c} \Gamma \vdash T_1 \rightsquigarrow T_2 \\ \Gamma \vdash T_2 \rightsquigarrow T_3 \end{array}}{\Gamma \vdash T_1 \rightsquigarrow T_3}\ (\text{SH-}\textsc{trans}) \qquad \frac{T_1 \rightsquigarrow T_2 \in \Gamma}{\Gamma \vdash T_1 \rightsquigarrow T_2}\ (\text{SH-}\textsc{env}) \qquad \frac{\Gamma \vdash T_1 \rightsquigarrow T_2}{\Gamma \vdash T_1 \backslash f \rightsquigarrow T_2 \backslash f}\ (\text{SH-}\textsc{mask})$$

$$\frac{\begin{array}{c} \mathsf{share}(P.C) = P' \backslash \overline{f} \\ \mathsf{fnames}(\mathsf{fields}(P.C) - \mathsf{fields}(P')) = \overline{f'} \end{array}}{\Gamma \vdash P.C! \backslash \overline{f} \backslash \overline{f'} = P'! \backslash \overline{f}}\ (\text{SH-}\textsc{decl}) \quad \frac{\begin{array}{c} PS_1 = P'_1!.\overline{C_1} \qquad PS_2 = P'_2!.\overline{C_2} \\ \forall P_1 . \left(\begin{array}{l} \Gamma \vdash P_1! \leq PS_1 \Rightarrow \\ \exists! P_2 \,.\, \exists \overline{f''} \supseteq \overline{f} \,.\, \Gamma \vdash P_2! \leq PS_2 \wedge \Gamma \vdash P_1! \backslash \overline{f''} \rightsquigarrow P_2! \backslash \overline{f'} \end{array} \right) \end{array}}{\Gamma \vdash PS_1 \backslash \overline{f} \rightsquigarrow PS_2 \backslash \overline{f'}}\ (\text{SH-}\textsc{cls})$$

$\boxed{\Gamma \vdash e : T, \Gamma'}$

$$\frac{\Gamma \vdash_{\mathsf{final}} p : T}{\Gamma \vdash p : \mathsf{ptype}(\Gamma, p), \Gamma}\ (\text{T-}\textsc{fin}) \quad \frac{\begin{array}{c} \Gamma \vdash e : T, \Gamma' \\ \Gamma \vdash T \leq T' \end{array}}{\Gamma \vdash e : T', \Gamma'}\ (\text{T-}\textsc{sub}) \quad \frac{\begin{array}{c} \Gamma \vdash e_1 : T_1, \Gamma_1 \\ \Gamma_1 \vdash e_2 : T_2, \Gamma_2 \end{array}}{\Gamma \vdash e_1 ; e_2 : T_2, \Gamma_2}\ (\text{T-}\textsc{seq}) \quad \Gamma \vdash \texttt{new}\ T : T!, \Gamma \ (\text{T-}\textsc{new})$$

$$\frac{\begin{array}{c} \Gamma \vdash e : T, \Gamma' \\ \mathsf{ftype}(\Gamma, T, f) = T_f \end{array}}{\Gamma \vdash e.f : T_f, \Gamma'}\ (\text{T-}\textsc{get}) \quad \frac{\begin{array}{c} x \notin \mathsf{dom}(\Gamma_1) \qquad \Gamma \vdash e_1 : T, \Gamma_1 \\ \Gamma_1, x : T \vdash e_2 : T_2, \Gamma_2 \qquad \Gamma_2 = \Gamma'_2, x : T' \end{array}}{\Gamma \vdash \texttt{final}\ T\ x = e_1;\ e_2 : T_2, \Gamma'_2}\ (\text{T-}\textsc{let}) \quad \frac{\begin{array}{c} \Gamma \vdash e : T', \Gamma' \\ \Gamma \vdash T' \rightsquigarrow T \end{array}}{\Gamma \vdash (\texttt{view}\ T) e : T, \Gamma'}\ (\text{T-}\textsc{view})$$

$$\frac{\begin{array}{c} \Gamma \vdash e : T_f, \Gamma' \qquad \Gamma' \vdash x : T \\ \mathsf{ftype}_{decl}(\Gamma', T', f) = T_f^{decl} \\ T_f^{decl} \{\!\{\Gamma';\ T/\texttt{this!}\}\!\} = T_f \end{array}}{\Gamma \vdash x.f = e : T_f, \mathsf{grant}(\Gamma', x.f)}\ (\text{T-}\textsc{set}) \quad \frac{\begin{array}{c} n = \#(\overline{e}) = \#(\overline{x}) \qquad x_0 = \texttt{this} \qquad \Gamma = \Gamma_0 \\ \mathsf{mtype}(\Gamma, T_0^0, m) = (\overline{x} : \overline{T^0}) \rightarrow T_{n+1}^0 \\ \forall i \in 1..n+1, j \in 1..i.\ T_i^{j-1} \{\!\{\Gamma_{i-1};\ T_{j-1}^{j-1} / x_{j-1}!\}\!\} = T_i^j \\ \forall i \in 0..n.\ \Gamma_i \vdash e_i : T_i^i, \Gamma'_{i+1} \end{array}}{\Gamma \vdash e_0.m(\overline{e}) : T_{n+1}^{n+1}, \Gamma_{n+1}}\ (\text{T-}\textsc{call})$$

$\boxed{\Gamma \vdash T \leq T'}$

$$\Gamma \vdash T \leq T \ (\text{S-}\textsc{refl}) \quad \frac{\begin{array}{c} \Gamma \vdash T_1 \leq T_2 \\ \Gamma \vdash T_2 \leq T_3 \end{array}}{\Gamma \vdash T_1 \leq T_3}\ (\text{S-}\textsc{trans}) \quad \Gamma \vdash T.C! \leq T!.C \ (\text{S-}\textsc{exact}) \quad \frac{\begin{array}{c} \Gamma \vdash T \leq P \\ \mathsf{super}(P.C) \{\!\{\Gamma;\ T/\texttt{this!}\}\!\} = T' \end{array}}{\Gamma \vdash T.C \leq T'}\ (\text{S-}\textsc{sup})$$

$$\frac{\begin{array}{c} \Gamma \vdash_{\mathsf{final}} p : T \\ \mathsf{pure}(T) = PT \end{array}}{\Gamma \vdash p.\texttt{class} \leq PT}\ (\text{S-}\textsc{fin}) \quad \frac{\begin{array}{c} \Gamma \vdash_{\mathsf{final}} p : T \\ \mathsf{pure}(T) = PT! \end{array}}{\Gamma \vdash p.\texttt{class} \approx PT!}\ (\text{S-}\textsc{fin-exact}) \quad \Gamma \vdash T \leq T \backslash f \ (\text{S-}\textsc{mask}) \quad \frac{\Gamma \vdash T_1 \leq T_2}{\Gamma \vdash T_1 \backslash f \leq T_2 \backslash f}\ (\text{S-}\textsc{sub-mask})$$

$$\frac{\Gamma \vdash p_1 = p_2}{\Gamma \vdash p_1.\texttt{class} \approx p_2.\texttt{class}}\ (\text{S-}\textsc{alias}) \quad \frac{\Gamma \vdash T \qquad \Gamma \vdash T \trianglelefteq S}{\Gamma \vdash T \leq S}\ (\text{S-}\textsc{bound}) \quad \frac{\Gamma \vdash T_1 \leq T_2 \qquad \Gamma \vdash T_2.C}{\Gamma \vdash T_1.C \leq T_2.C}\ (\text{S-}\textsc{nest})$$

$$\frac{\Gamma \vdash P\,\texttt{[}PT.C\texttt{]}}{\Gamma \vdash PT \approx P\,\texttt{[}PT.C\texttt{]}}\ (\text{S-}\textsc{pre-in}) \quad \frac{\Gamma \vdash PT \leq P.C}{\Gamma \vdash PT \leq P\,\texttt{[}PT\texttt{]}.C}\ (\text{S-}\textsc{pre-out}) \quad \frac{\Gamma \vdash PT_1 \leq PT_2 \qquad \Gamma \vdash P\,\texttt{[}PT_2\texttt{]}}{\Gamma \vdash P\,\texttt{[}PT_1\texttt{]} \leq P\,\texttt{[}PT_2\texttt{]}}\ (\text{S-}\textsc{pre-1})$$

$$\frac{\begin{array}{c} \vdash P_1 \sim P_2 \vee \vdash P_1 \sqsubset_{\mathsf{sc}} P_2 \\ \Gamma \vdash P_1\,\texttt{[}PT\texttt{]} \qquad \Gamma \vdash P_2\,\texttt{[}PT\texttt{]} \end{array}}{\Gamma \vdash P_1\,\texttt{[}PT\texttt{]} \approx P_2\,\texttt{[}PT\texttt{]}}\ (\text{S-}\textsc{pre-2}) \quad \Gamma \vdash \&\overline{T} \leq T_i \ (\text{S-}\textsc{meet-lb}) \quad \frac{\forall i.\ \Gamma \vdash T \leq T_i}{\Gamma \vdash T \leq \&\overline{T}}\ (\text{S-}\textsc{meet-g})$$

Figure 3.10: Static semantics

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

$$
\begin{aligned}
\mathsf{prefixExact}_k(\circ) &= \mathsf{false} \\
\mathsf{prefixExact}_k(T.C) &= \begin{cases} \mathsf{false} & \text{if } k = 0 \\ \mathsf{prefixExact}_{k-1}(T) & \text{otherwise} \end{cases} \\
\mathsf{prefixExact}_k(p.\texttt{class}) &= \mathsf{true} \\
\mathsf{prefixExact}_k(P\,\texttt{[}\,T\,\texttt{]}) &= \mathsf{prefixExact}_{k+1}(T) \\
\mathsf{prefixExact}_k(\&\overline{T}) &= \bigvee_{T_i \in \overline{T}} \mathsf{prefixExact}_k(T_i) \\
\mathsf{prefixExact}_k(T!) &= \mathsf{true} \\
\mathsf{exact}(T) &= \mathsf{prefixExact}_0(T)
\end{aligned}
$$

Figure 3.11: Paths and exactness

The function $\mathsf{prefixExact}_k(T)$ shown in Figure 3.11, is true if the kth prefix of T is an exact type for $k \geq 0$. If $\mathsf{prefixExact}_k(T)$, then $\mathsf{prefixExact}_{k+1}(T)$.

To type-check field accesses in the proof of soundness, the type system keeps track of aliases. The judgment $\Gamma \vdash p_1 = p_2$, defined in Figure 3.10, states that two final access paths are aliases, and the two references stored in p_1 and p_2 have the same view. Note that A-ENV does *not* allow a judgment of alias with a variable in it, although the environment Γ can have an element in the form of $x = v$ in it.

3.3.8 Type well-formedness

Type well-formedness is defined in Figure 3.12. The judgment $\Gamma \vdash T$ states that type T is well-formed in a context Γ. Most of the rules are similar to the J& calculus [59], except WF-EXACT, WF-REF, and WF-MASK, for new kinds of types in J&$_s$.

$$\frac{CT'(P) \neq \bot}{\Gamma \vdash P}\ (\text{WF-SIMP}) \qquad \frac{\Gamma \vdash_{\mathsf{final}} p : T}{\Gamma \vdash p.\texttt{class}}\ (\text{WF-FIN}) \qquad \frac{\Gamma \vdash T \qquad \Gamma \vdash T \trianglelefteq PS \qquad \mathsf{mem}(PS.C) \neq \emptyset}{\Gamma \vdash T.C}\ (\text{WF-NEST})$$

$$\frac{\begin{array}{c}\Gamma \vdash P \qquad \Gamma \vdash T \qquad \Gamma \vdash T \trianglelefteq PS \\ \mathsf{pure}(T) = T \qquad \mathsf{prefix}(P, PS) \neq \emptyset\end{array}}{\Gamma \vdash P[T]}\ (\text{WF-PRE}) \qquad \frac{\begin{array}{c}\Gamma \vdash T \qquad T \neq T'\backslash f \\ \Gamma \vdash T \trianglelefteq PS \qquad [\texttt{final}]\ T_f\ f \in \mathsf{fields}(PS)\end{array}}{\Gamma \vdash T\backslash f}\ (\text{WF-MASK})$$

$$\frac{\Gamma \vdash T}{\Gamma \vdash T!}\ (\text{WF-EXACT}) \qquad \frac{\begin{array}{c}\forall i.\ \Gamma \vdash T_i \qquad \forall p_i, p_j \in \mathsf{paths}(\&\overline{T}).\ \Gamma \vdash p_i = p_j \\ \forall T_i, T_j \in \overline{T}, k \geq 0.\ \mathsf{prefixExact}_k(T_i) \Leftrightarrow \mathsf{prefixExact}_k(T_j)\end{array}}{\Gamma \vdash \&\overline{T}}\ (\text{WF-MEET})$$

Figure 3.12: Type well-formedness

$$\Gamma \vdash PS \trianglelefteq PS\ \ (\text{BD-SIMP}) \qquad \frac{PS = \&\{PS' \mid \Gamma \vdash p = p' \wedge \Gamma \vdash_{\mathsf{final}} p' : T \wedge \Gamma \vdash T \trianglelefteq PS'\}}{\Gamma \vdash p.\texttt{class} \trianglelefteq PS}\ (\text{BD-FIN})$$

$$\frac{\Gamma \vdash T \trianglelefteq PS}{\Gamma \vdash T.C \trianglelefteq PS.C}\ (\text{BD-NEST}) \qquad \frac{\Gamma \vdash T \trianglelefteq PS}{\Gamma \vdash P[T] \trianglelefteq P[PS]}\ (\text{BD-PRE}) \qquad \frac{\forall i.\ \Gamma \vdash T_i \trianglelefteq PS_i}{\Gamma \vdash \&\overline{T} \trianglelefteq \&\overline{PS}}\ (\text{BD-MEET})$$

$$\frac{\Gamma \vdash T \trianglelefteq PS \qquad T \neq PS'}{\Gamma \vdash T! \trianglelefteq PS}\ (\text{BD-EXACT}) \qquad \frac{\Gamma \vdash \mathsf{pure}(T) \trianglelefteq PS}{\Gamma \vdash T \trianglelefteq PS}\ (\text{BD-PURE})$$

Figure 3.13: Type bounds

3.3.9 Non-dependent bounding types

The judgment $\Gamma \vdash T \trianglelefteq PS$, shown in Figure 3.13, states that type T has a static type bound PS that is non-dependent and pure, i.e., $\Gamma \vdash \mathsf{pure}(T) \leq PS$. Also, PS is the most specific static type bound of T in the context of Γ. The most interesting rule is BD-FIN, which ensures that the p_1.class and p_2.class have the same type bound, if p_1 and p_2 are aliases in the context of Γ. Therefore a type T might have different type bounds in different typing environments, but the bound is unique in the same environment.

3.3.10 Type substitution

The rules for type substitution are shown in Figure 3.14. Type substitution $T\{\!\{\Gamma;\ T_x/x\}\!\}$ substitutes $\mathsf{pure}(T_x)$ for x.class in T in the context of Γ. The typ-

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

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

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

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

$$x.\mathtt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = \mathsf{pure}(T_x)$$

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

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

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

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

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

$$\frac{T\{\!\{\Gamma;\ T_x/x\}\!\} = T'}{T\backslash f\{\!\{\Gamma;\ T_x/x\}\!\} = T'\backslash f}$$

Figure 3.14: Type substitution

ing 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 and a subtype of x's declared type.

For type safety, type substitutions on the right-hand side of a field assignment or on the parameters of method calls must preserve the exactness of the declared type. Therefore, only values from the family that is compatible with the receiver are assigned to fields or passed to method code. Exactness-preserving type substitution $T\{\!\{\Gamma;\ T_x/x!\}\!\}$ is defined as follows:

$$\frac{\begin{array}{c} T\{\!\{\Gamma;\ T_x/x\}\!\} = T' \\ \forall k.\ \mathsf{prefixExact}_k(T) \Rightarrow \mathsf{prefixExact}_k(T') \end{array}}{T\{\!\{\Gamma;\ T_x/x!\}\!\} = T'}$$

3.3.11 Sharing relationships

The directional sharing judgment $\Gamma \vdash T_1 \rightsquigarrow T_2$, shown in Figure 3.10, states that a value of type T_1 can be transformed to T_2 through a view change, that is, the sharing relationship $T_1 \rightsquigarrow T_2$ is valid. A bidirectional sharing judgment $\Gamma \vdash T_1 \leftrightarrow T_2$ is sugar for a pair of directional sharing judgments. SH-DECL collects sharing relationships from class declarations, and for simplicity, it does not model the inference of asymmetric sharing relationships, described in Section 3.2.3. SH-CLS states that for two types to be shared, every subclass of the source type must share a unique subclass of the target type, with appropriate masks. The uniqueness is necessary for the view function to set the type component in the generated reference. Moreover, the two types in SH-CLS have to be both nested in simple exact types, and therefore we can enumerate all their subclasses in the locally closed worlds.

3.3.12 Typing rules

Expression typing rules are shown in Figure 3.10. Evaluation of an expression can update type bindings of some variables. For example, assignment to a masked field removes the mask. Therefore, typing judgments are of the form $\Gamma \vdash e : T, \Gamma'$, stating that e has type T in context Γ, and that after evaluating e, an updated environment Γ' exists instead. Typing judgments are sometimes written as $\Gamma \vdash e : T$, if the updated typing environment is unused.

Masks can be introduced by T-SUB and S-MASK, and they can be removed by field assignments. The following auxiliary function grant removes the mask, that is, granting the access to a field: $\mathsf{grant}(\Gamma, e)$ generates an environment from Γ, where accesses to $x.f$ are enabled, for $e = x.f$.

$$\mathsf{grant}(\Gamma, e) = \begin{cases} \Gamma_1, x : T & \text{if } \Gamma = \Gamma_1, x : T \backslash f \wedge e = x.f \\ \Gamma & \text{otherwise} \end{cases}$$

The typing of a final access path p is complicated by the fact that $p.\texttt{class}$ is pure, and therefore p might not have $p.\texttt{class}$ as its type. The auxiliary function ptype, shown below, gives a proper type to p, and it also avoids dependent classes that have paths starting with a variable, if the typing environment contains the information about the variable on the stack.

$$\mathsf{ptype}(\Gamma, p) = \begin{cases} \mathsf{ptype}(\Gamma, v.\overline{f}) & \text{if } p = x.\overline{f} \wedge x = v \in \Gamma \\ p.\texttt{class}\backslash\overline{f} & \text{if } \Gamma \vdash_{\mathsf{final}} p : PT\backslash\overline{f} \end{cases}$$

In addition, A-ENV does not give aliasing information that includes variables. Therefore, in the proof of soundness, types that depend on paths starting with variables can be avoided all together, because the stack σ contains the value for every variable.

3.3.13 Subtyping

Subtyping rules are defined in Figure 3.10. The judgment $\Gamma \vdash T \leq T'$ states that T is a subtype of T' in context Γ, and type equivalence $\Gamma \vdash T \approx T'$ is sugar for a pair of subtyping judgments.

The J&$_s$ type system clearly distinguishes sharing relationships between types from subtyping relationships. Subtyping rules in J&$_s$ do not depend on any sharing judgment. In particular, $\Gamma \vdash T_1 \leadsto T_2$ does not imply $\Gamma \vdash T_1 \leq T_2$.

3.3.14 Program typing

Program typing rules are shown in Figure 3.15. P-OK is the rule for a program to be well-formed; L-OK, F-OK, and M-OK are the rules for a class, a field, and a method to be well-formed, respectively.

Q-OK gives the rule for a sharing constraint to be well-formed: the two types, interpreted in the context of the current containing class, should be shared. Also, whenever a sharing constraint in an inherited method no longer holds, the method should be overridden.

3.3.15 Operational semantics

A small-step semantics for J&$_s$ is shown in Figures 3.16 and 3.17. A stack σ is a function mapping variable names x to values v. A heap H is a function mapping tuples $< \ell, P, f >$ of memory locations, classes, and field names to values v. The J&$_s$ calculus follows the approach of J\mask [64], where uninitialized fields do not exist in the heap. A J&$_s$ object might have multiple versions of a field f, when the interpreted types of f are not shared for different views of the object. The class P in the domain of the heap to indicate which copy of the possibly unshared field f is used. Given a view $P'.C!\backslash\overline{f'}$ of ℓ and a field f, the indicator class P is obtained with the following auxiliary function fclass:

$$\mathsf{fclass}(P'.C, f) = \begin{cases} P'.C & \text{if } P'.C = \mathsf{share}(P'.C) \\ P'.C & \text{if } f \in \mathsf{fnames}(\mathsf{fields}(P'.C) - \mathsf{fields}(\mathsf{share}(P'.C))) \\ & \quad \vee \mathsf{share}(P'.C) = T\backslash f \\ \mathsf{fclass}(\mathsf{pure}(\mathsf{share}(P'.C)), f) & \text{otherwise} \end{cases}$$

$$\frac{\circ \vdash \overline{L}\ \mathsf{ok} \qquad \emptyset \vdash e : T \qquad \emptyset \vdash T \qquad \sqsubset^{+}\ \text{acyclic}}{\vdash <\overline{L}, e>\ \mathsf{ok}} \qquad \text{(P-OK)}$$

$$\frac{S_1 = T_1\{\!\{\emptyset;\ P!/\mathtt{this!}\}\!\} \qquad S_2 = T_2\{\!\{\emptyset;\ P!/\mathtt{this!}\}\!\} \qquad \emptyset \vdash S_1 \rightsquigarrow S_2}{P \vdash T_1 \rightsquigarrow T_2\ \mathsf{ok}} \qquad \text{(Q-OK)}$$

$$\frac{\begin{array}{c} \forall C'.\ CT'(P.C.C') \neq \bot \Rightarrow P.C \vdash CT'(P.C.C')\ \mathsf{ok} \\ P.C \vdash \overline{F}\ \mathsf{ok} \qquad P.C \vdash \overline{M}\ \mathsf{ok} \qquad P \vdash T\ \mathsf{super\ ok} \\ \forall P_i \in \mathsf{supers}(P.C)\backslash\{P.C\}.\ \vdash P.C\ \text{conforms to}\ P_i \\ T' = P'\backslash\overline{f} \qquad \vdash P.C \sqsubset^{*}_{\mathsf{fb}} P' \\ \forall f.\ \mathtt{final}\ T_f\ f\ \ldots \in \mathsf{fields}(P') \Rightarrow f \notin \overline{f} \\ \forall f.\ \big(\ldots\ T_f\ f\ \ldots \in \mathsf{fields}(P') \land f \notin \overline{f} \Rightarrow \vdash T_f\{\!\{\emptyset;\ P'!/\mathtt{this}\}\!\} \leftrightarrow T_f\{\!\{\emptyset;\ P.C!/\mathtt{this}\}\!\}\big) \end{array}}{P \vdash \mathtt{class}\ C\ \mathtt{extends}\ T\ \mathtt{shares}\ T'\ \{\overline{L}\ \overline{F}\ \overline{M}\}\ \mathsf{ok}} \qquad \text{(L-OK)}$$

$$\frac{T \neq \circ \qquad \mathtt{this} : P \vdash T \qquad \mathsf{paths}(T) \subseteq \{\mathtt{this}\} \qquad \neg\mathsf{exact}(T)}{P \vdash T\ \mathsf{super\ ok}}$$

$$\frac{\begin{array}{c} CT'(P) = \mathtt{class}\ C\ \mathtt{extends}\ T_e\ \mathtt{shares}\ T_s\ \{\overline{L}\ \overline{F}\ \overline{M}\} \\ CT'(P') = \mathtt{class}\ C'\ \mathtt{extends}\ T'_e\ \mathtt{shares}\ T'_s\ \{\overline{L'}\ \overline{F'}\ \overline{M'}\} \\ \forall i, j.\ \big(L_i = \mathtt{class}\ D\ \mathtt{extends}\ T_i\ \ldots\ \{\ldots\} \land L'_j = \mathtt{class}\ D\ \mathtt{extends}\ T'_j\ \ldots\ \{\ldots\}\big) \Rightarrow \mathtt{this} : P \vdash T_i \leq T'_j \\ (\mathsf{fnames}(\overline{F}) \cap \mathsf{fnames}(\overline{F'})) = \emptyset \\ \forall i, j.\ \big(M_i = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\} \land M'_j = T'_{n+1}\ m(\overline{T'}\ \overline{x'})\ \{e'\}\big) \Rightarrow P \vdash M_i\ \text{overrides}\ M'_j \\ \forall i.\ M'_i = \ldots\ \mathtt{sharing}\ \overline{Q}\ \ldots \land P \nvdash \overline{Q}\ \mathsf{ok} \Rightarrow \exists j.\ P \vdash M_j\ \text{overrides}\ M'_i \end{array}}{\vdash P\ \text{conforms to}\ P'}$$

$$\frac{\begin{array}{c} M = T_{n+1}\ m(\overline{T}\ \overline{x})\ \ldots\ \{e\} \\ M' = T'_{n+1}\ m(\overline{T'}\ \overline{x'})\ \ldots\ \{e'\} \\ \#(\overline{x}) = \#(\overline{x'}) = \#(\overline{y}) \qquad \overline{y} \cap (\overline{x} \cup \overline{x'}) = \emptyset \\ \Gamma = \mathtt{this} : P, \overline{y} : \overline{T}\{\overline{y}/\overline{x}\} \qquad \vdash \Gamma\ \mathsf{ok} \\ \Gamma \vdash \overline{T}\{\overline{y}/\overline{x}\} \approx \overline{T'}\{\overline{y}/\overline{x'}\} \qquad \Gamma \vdash T_{n+1}\{\overline{y}/\overline{x}\} \approx T'_{n+1}\{\overline{y}/\overline{x'}\} \end{array}}{P \vdash M\ \text{overrides}\ M'}$$

$$\frac{\begin{array}{c} \mathsf{fnames}(\mathsf{fields}(P)) = \overline{f} \qquad \Gamma = \mathtt{this} : P\backslash\overline{f} \\ \Gamma \vdash T \qquad \mathsf{paths}(T) \subseteq \{\mathtt{this}\} \qquad \mathsf{exact}(T) = \mathsf{false} \qquad \Gamma \vdash e : T, \Gamma \end{array}}{P \vdash [\mathtt{final}]\ T\ f = e\ \mathsf{ok}} \qquad \text{(F-OK)}$$

$$\frac{\begin{array}{c} P \vdash \overline{Q}\ \mathsf{ok} \qquad \Gamma = \mathtt{this} : P, \overline{x} : \overline{T} \\ \vdash \Gamma\ \mathsf{ok} \qquad \Gamma \vdash T_{n+1} \qquad \Gamma \vdash e : T_{n+1}, \Gamma_r \qquad \mathsf{FV}(e) \subseteq \{x_0, \overline{x}\} \\ n = \#(\overline{x}) \qquad x_0 = \mathtt{this} \\ \forall i \in 1..n+1.\ \mathsf{paths}(T_i) \in \{x_0, \ldots, x_{i-1}\} \end{array}}{P \vdash T_{n+1}\ m(\overline{T}\ \overline{x})\ \mathtt{sharing}\ \overline{Q}\ \{e\}\ \mathsf{ok}} \qquad \text{(M-OK)}$$

Figure 3.15: Program typing

$$
\begin{array}{lll}
\text{stacks} & \sigma & ::= \emptyset \mid \sigma, x \mapsto v \\
\text{heaps} & H & ::= \emptyset \mid H, <\ell, \overline{P}, f> \mapsto v \\
\text{reference sets} & R & ::= \emptyset \mid R, v \\
\text{evaluation contexts} & E & ::= [\cdot] \mid E.f \mid x.f = E \\
 & & \quad \mid x \mid E;\ e \mid (\texttt{view}\ TE)e \mid (\texttt{view}\ S)E \\
 & & \quad \mid E.m(\overline{e}) \mid v.m(\overline{v}, E, \overline{e}) \\
 & & \quad \mid \texttt{new}\ TE \mid \texttt{final}\ TE\ x = e_1;\ e_2 \mid \texttt{final}\ S\ x = E;\ e_2 \\
\text{type evaluation contexts} & TE & ::= TE.C \mid E.\texttt{class} \mid P\texttt{[}TE\texttt{]} \\
 & & \quad \mid \texttt{\&}(\overline{S}, TE, \overline{T}) \mid TE! \\
 & & \quad \mid TE\backslash\overline{f} \mid v.\texttt{class}
\end{array}
$$

Figure 3.16: Definitions for operational semantics

Stack updates and heap updates in J&$_s$ are represented as $\sigma[x := v]$ and $H[< \ell, P, f >:= v]$ respectively. We do not explicitly model popping off stack frames or garbage collection on the heap.

A reference set R, which contains all the references v that have been generated during evaluation, no matter whether they are reachable from the stack or not, is also part of the evaluation configuration e, σ, H, R. The set R is only for the proof of soundness: it prevents us from losing path equalities needed in the proof.

The evaluation rules (Figure 3.17) take the form $e, \sigma, H, R \longrightarrow e', \sigma', H', R'$. We overload the function grant to work on σ: it updates the type components of values $< \ell, S >$ stored in σ with appropriate annotations.

R-SET assigns a value to a field, and may remove the mask on field $x.f$. We overload the function grant, defined in Section 3.3.12, to work on σ: the type components of values $< \ell, S >$ stored in σ are updated with appropriate annotations.

R-ALLOC is the rule for `new` expressions, which initialize all the fields according to field initializers collected from class declarations. Note that other views' copies of unshared fields are not initialized right after the object is created. Masked types ensure that after a view change, those possibly uninitialized fields are not accessed.

The order of evaluation is captured by an evaluation context E. There is also a type evaluation context TE for evaluating dependent types. A fully evaluated type will always be non-dependent, since $<\ell, S>$`.class` can evaluate to S.

Since values can be viewed with different type S, the function view takes a value v and a non-dependent type S, and returns a different value consisting of the same location, with its view changed to be compatible with S. This function is used for field accesses and view changing operations, and the rule SH-CLS ensures that there is always a single view to change to. Note that the view in a reference is always an exact type.

$$\mathsf{view}(<\ell, P'!\backslash\overline{f'}>, PS\backslash\overline{f}) = \begin{cases} <\ell, P'!\backslash\overline{f}> & \vdash P'!\backslash\overline{f'} \le PS\backslash\overline{f} \\ <\ell, P!\backslash\overline{f}> & \exists! P\ .\ \exists\overline{f''} \supseteq \overline{f'}\ .\ \vdash P! \le PS \wedge \vdash P'!\backslash\overline{f''} \rightsquigarrow P!\backslash\overline{f} \end{cases}$$

In R-ALLOC, the judgment $H \vdash \ell$ `fresh` means that there is no tuple $<\ell, P, f>$ in the domain of H, for any P and any f.

3.3.16 Runtime typing environments

In the proof of soundness, run-time values are typed using a typing environment $\lfloor\sigma, H, R\rfloor$ constructed from the stack, the heap, and the reference set. Although references of different shared types might address the same heap location, the construction of $\lfloor\sigma, H, R\rfloor$ ensures that they are *not* included as aliases, which is necessary for S-ALIAS to hold. See Figure 3.18 for details.

$\boxed{e, \sigma, H, R \longrightarrow e', \sigma', H', R'}$

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

$$\frac{\sigma(x) = v}{x, \sigma, H, R \longrightarrow v, \sigma, H, R} \quad \text{(R-VAR)}$$

$$\frac{y \notin \mathrm{dom}(\sigma) \qquad \sigma' = \sigma[y := v_1]}{\texttt{final}\ S\ x = v_1;\ e_2, \sigma, H, R \longrightarrow e_2\{y/x\}, \sigma', H, R} \quad \text{(R-LET)}$$

$$\frac{H(\ell, \mathsf{fclass}(P, f), f) = v \qquad S = \mathsf{ftype}(\emptyset, P!\backslash\overline{f'}, f) \qquad R' = R, \mathsf{view}(v, S)}{< \ell, P!\backslash\overline{f'} > .f, \sigma, H, R \longrightarrow \mathsf{view}(v, S), \sigma, H, R'} \quad \text{(R-GET)}$$

$$\frac{\sigma(x) =< \ell, P!\backslash\overline{f'} > \qquad \sigma' = \mathsf{grant}(\sigma, x.f) \qquad H' = H[< \ell, \mathsf{fclass}(P, f), f >:= v] \qquad R' = R, < \ell, P!\backslash(\overline{f'} - f) >}{x.f = v, \sigma, H, R \longrightarrow v, \sigma', H', R'} \quad \text{(R-SET)}$$

$$\frac{\mathsf{mbody}(S, m) = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\} \qquad n = \#(\overline{v}) = \#(\overline{x}) \qquad y_0, y_1, \ldots, y_n \notin \mathrm{dom}(\sigma) \qquad \sigma' = \sigma[y_0 :=< \ell, S >, \overline{y} := \overline{v}]}{< \ell, S > .m(\overline{v}), \sigma, H, R \longrightarrow e\{y_0/\texttt{this}, \overline{y}/\overline{x}\}, \sigma', H, R} \quad \text{(R-CALL)}$$

$$\frac{H \vdash \ell\ \texttt{fresh} \qquad x \notin \mathrm{dom}(\sigma) \qquad \mathsf{fields}(S) = [\texttt{final}]\ \overline{T_f}\ \overline{f} = \overline{e} \qquad v =< \ell, S\,!\backslash\overline{f} > \qquad R' = R, v}{\texttt{new}\ S, \sigma, H, R \longrightarrow \texttt{final}\ S\,!\backslash\overline{f}\ x = v;\ x.\overline{f} = \overline{e}\{x/\texttt{this}\};\ x, \sigma, H, R'} \quad \text{(R-ALLOC)}$$

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

$$\frac{R' = R, \mathsf{view}(v, S)}{(\texttt{view}\ S)v, \sigma, H, R \longrightarrow \mathsf{view}(v, S), \sigma, H, R'} \quad \text{(R-VIEW)}$$

Figure 3.17: Small-step operational semantics

$$\frac{\sigma(x) =< \ell, S >}{x : S, x =< \ell, S > \in \lfloor \sigma, H, R \rfloor}$$

$$\frac{< \ell, PS\backslash\overline{f} > \in R \qquad < \ell, PS\backslash\overline{f'} > \in R}{< \ell, PS\backslash\overline{f} > =< \ell, PS\backslash\overline{f'} > \in \lfloor \sigma, H, R \rfloor}$$

$$\frac{\begin{array}{c} < \ell, S > \in R \qquad S = P!\backslash\overline{f'} \\ v = \mathsf{view}(H(\ell, \mathsf{fclass}(P, f), f), T_f) \\ \mathsf{ftype}(\emptyset, S, f) = T_f \qquad \texttt{final}\ T_f^{decl}\ f = e \in \mathsf{fields}(S) \end{array}}{< \ell, S > .f = v \in \lfloor \sigma, H, R \rfloor}$$

Figure 3.18: Runtime typing environments

$$\frac{\begin{array}{c}\mathsf{FV}(e) \subseteq \mathrm{dom}(\sigma) \qquad \mathsf{refs}(e) \subseteq R \\ \forall < \ell, P!\backslash\overline{f'} > \in R.\ \forall f \notin \overline{f'}.\ \exists \ell'\ .\ \exists S'\ .\ \left(\begin{array}{l} H(\ell, \mathsf{fclass}(P, f), f) = < \ell', S' > \wedge \\ (\vdash S' \leq \mathsf{ftype}(\emptyset, S, f) \vee \exists S''\ .\ \vdash S'' \leq \mathsf{ftype}(\emptyset, S, f) \wedge \vdash S' \rightsquigarrow S'') \end{array}\right)\end{array}}{\vdash e, \sigma, H, R}\ (\textsc{Config})$$

Figure 3.19: Well-formed configurations

3.3.17 Well-formed configurations

The well-formedness of a configuration e, σ, H, R is given in Figure 3.19.

A configuration e, σ, H, R is well-formed, if every value pointed to by an unmasked field has (or can be transformed to) a view compatible with its container. This ensures that field accesses always succeed. Also, all the free variables in e must be in the domain of σ, and all the references in e must be in R.

3.3.18 Soundness

We prove soundness using subject reduction and progress [87]. The proof of subject reduction is the hard part. There are two notable issues:

- The typing environment $\lfloor \sigma', H', R' \rfloor$ after an evaluation step is *not* a simple superset of the environment $\lfloor \sigma, H, R \rfloor$ before the step, because some field assignment might update variable typing (R-SET). However the updated type is always a subtype of the original type. Therefore, we define the environment extension as follows: Γ_2 extends Γ_1 if
  - For every $p_1 = p_2 \in \Gamma_1$, there is $p_1 = p_2 \in \Gamma_2$;
  - For every $T_1 \rightsquigarrow T_2 \in \Gamma_1$, there is $T_1 \rightsquigarrow T_2 \in \Gamma_2$;
  - For every $x : T \in \Gamma_1$, there is $x : T' \in \Gamma_2$ and $\Gamma_2 \vdash T' \leq T$.

- Method calls are evaluated by adding a set of fresh variable bindings (essentially, a new stack frame) into σ, rather than directly substituting actual parameters for formal arguments.

We first prove some lemmas about extensions of typing environments:

Lemma 3.3.1 *If* $e, \sigma, H, R \longrightarrow e', \sigma', H', R'$, *then* $\lfloor \sigma', H', R' \rfloor$ *is an extension of* $\lfloor \sigma, H, R \rfloor$.

PROOF: By induction on the operational semantic derivation.

- R-CONG

  By the induction hypothesis.

- R-VAR, and R-SEQ

  Trivial, because $\sigma' = \sigma$, $H' = H$, and $R' = R$ in these cases.

- R-LET

  Then $H' = H$, $R' = R$, and $\sigma' = \sigma[y := v_1]$ where y is fresh. We will have $\lfloor \sigma', H, R \rfloor = \lfloor \sigma, H, R \rfloor, y : S, y = v_1$, and $\lfloor \sigma', H, R \rfloor$ is an extension of $\lfloor \sigma, H, R \rfloor$.

- R-GET

  Then $\sigma' = \sigma$, $H' = H$, and R' might be a superset of R. By the definition of run-time typing environments, $\lfloor \sigma, H, R' \rfloor$ can only contain some aliasing information other than elements in $\lfloor \sigma, H, R \rfloor$. Thus $\lfloor \sigma, H, R' \rfloor$ is an extension of $\lfloor \sigma, H, R \rfloor$.

- R-SET

  Then $\sigma' = \mathsf{grant}(\sigma, x.f)$, $S = P!\backslash\overline{f'}$, $H' = H[< \ell, \mathsf{fclass}(P, f), f >:= v]$, and R' might be a superset of R. Compared to $\lfloor \sigma, H, R \rfloor$, the new environment

$\lfloor \sigma', H', R' \rfloor$ adds some aliasing relationships, and it might update the type binding of x from $T \backslash f$ to T for some type T. By S-MASK, $\lfloor \sigma', H', R' \rfloor \vdash T \leq T \backslash f$. Thus $\lfloor \sigma', H', R' \rfloor$ is an extension of $\lfloor \sigma, H, R \rfloor$.

- R-CALL

  Similar to the case of R-LET.

- R-ALLOC

  Similar to the case of R-GET.

- R-VIEW

  Similar to the case of R-GET.

□

Lemma 3.3.2 *If Γ_2 is an extension of Γ_1, then all of the following hold:*

- *If $\Gamma_1 \vdash T$, then $\Gamma_2 \vdash T$.*
- *If $\Gamma_1 \vdash T \trianglelefteq S$, then $\Gamma_2 \vdash T \trianglelefteq S$.*
- *If* $\mathsf{ftype}(\Gamma_1, T, f) = T_f$, *then* $\mathsf{ftype}(\Gamma_2, T, f) = T_f$.
- *If* $\mathsf{mtype}(\Gamma_1, T, m) = (\overline{x} : \overline{T}) \rightarrow T_{n+1}$, *then* $\mathsf{mtype}(\Gamma_2, T, m) = (\overline{x} : \overline{T}) \rightarrow T_{n+1}$.
- *If $\Gamma_1 \vdash p = p'$, then $\Gamma_2 \vdash p = p'$.*
- *if* $\mathsf{ptype}(\Gamma_1, p) = T$, *then* $\mathsf{ptype}(\Gamma_2, p) = T$.
- *If $T\{\!\{\Gamma_1;\ T_x/x\}\!\} = T'$, then $T\{\!\{\Gamma_2;\ T_x/x\}\!\} = T'$.*
- *If $\Gamma_1 \vdash T_1 \leq T_2$, then $\Gamma_2 \vdash T_1 \leq T_2$.*
- *If $\Gamma_1 \vdash v : T$, then $\Gamma_2 \vdash v : T$.*

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

Note that although Lemma 3.3.2 states that value typing does not change during the execution, it does not directly apply to more general expression typing, because variables can change their types during the execution.

The following lemma is about type substitution of dependent classes.

Lemma 3.3.3 *If* $x : T_x \in \Gamma$, $\mathsf{paths}(T_x) = \emptyset$, $\Gamma \vdash S_x! \leq T_x$, *and* $\Gamma \vdash e : T$, *then* $\Gamma\{\!\{\emptyset;\ S_x!/x\}\!\} \vdash e : T\{\!\{\emptyset;\ S_x!/x\}\!\}$.

PROOF: The proof is by induction on the typing derivation $\Gamma \vdash e : T$. □

Type substitution is generalized to the typing environment Γ, where substitution is applied to the type of every variable in Γ.

The following lemma establishes connections between type substitutions and subtyping relationships. It is generally used together with Lemma 3.3.3.

Lemma 3.3.4 *If* $\Gamma \vdash T_1 \leq T_2$, *then* $\Gamma \vdash T\{\!\{\Gamma;\ T_1/x\}\!\} \leq T\{\!\{\Gamma;\ T_2/x\}\!\}$.

PROOF: By induction on the structure of T. Note that there is no first-order function type, and therefore a type substitution does not occur at a contravariant position. □

Lemma 3.3.5 *If* $\vdash p, \sigma, H, R$, *and* $\lfloor \sigma, H, R \rfloor \vdash_{\mathsf{final}} p : T$, *and* $p, \sigma, H, R \longrightarrow p', \sigma', H', R'$, *and* $\lfloor \sigma', H', R' \rfloor \vdash_{\mathsf{final}} p' : T'$, *then* $\lfloor \sigma, H, R \rfloor \vdash \mathsf{ptype}(\lfloor \sigma, H, R \rfloor, p) \approx \mathsf{ptype}(\lfloor \sigma, H, R \rfloor, p')$.

PROOF: There are the following three cases:

- $p = x.\overline{f}$, where $\overline{f}$ may be empty

  Then by F-GET and F-VAR, $x : T_x \in \lfloor \sigma, H, R \rfloor$, which implies $\sigma(x) =< \ell, T_x >$ and $x =< \ell, T_x > \in \lfloor \sigma, H, R \rfloor$ by the definition of $\lfloor \sigma, H, R \rfloor$, and by R-VAR and R-CONG, $p' =< \ell, T_x > .\overline{f}$. By the definition of ptype, we have $\mathsf{ptype}(\lfloor \sigma, H, R \rfloor, p) = \mathsf{ptype}(\lfloor \sigma, H, R \rfloor, p')$. Therefore, by S-REFL, $\lfloor \sigma, H, R \rfloor \vdash \mathsf{ptype}(\lfloor \sigma, H, R \rfloor, p) \approx \mathsf{ptype}(\lfloor \sigma, H, R \rfloor, p')$.

- $p = v$

  Vacuously true, since it cannot make any progress.

- $p = v.f.\overline{f'}$, where $v =< \ell, P!\backslash\overline{f''} >$

  Then by R-GET and R-CONG, we have $p' = v'.\overline{f'}$, $H(\ell, \mathsf{fclass}(P, f), f) = v''$, and $v' = \mathsf{view}(v'', \mathsf{ftype}(\emptyset, P!\backslash\overline{f''}, f))$. Therefore, by the definition of $\lfloor \sigma, H, R \rfloor$, we have $v.f = v' \in \lfloor \sigma, H, R \rfloor$. By A-ENV, $\lfloor \sigma, H, R \rfloor \vdash v.f = v'$. By A-FIELD, $\lfloor \sigma, H, R \rfloor \vdash p = p'$. By S-ALIAS, $\lfloor \sigma, H, R \rfloor \vdash p.\texttt{class} \approx p'.\texttt{class}$. Now we need to show that T and T' have the same set of masks. There are again two cases:

  - $\overline{f'} = \emptyset$

    By the definition of the auxiliary function view, T' has the same set of masks as $\mathsf{ftype}(\emptyset, P!\backslash\overline{f''}, f)$, which is T.

  - $\overline{f'} \neq \emptyset$

    Then T and T' have the same set of masks, because they are the type of the same field (the last field in $\overline{f'}$).

□

In the proof of subject reduction, Lemma 3.3.2 proves the case where T-SUB is the last derivation; Lemma 3.3.3 is used to prove type preservation for the evaluation of a method call.

Lemma 3.3.6 *(Subject reduction)* *If* $\vdash e, \sigma, H, R$*, and* $\lfloor \sigma, H, R \rfloor \vdash e : T$*, and* $e, \sigma, H, R \longrightarrow e', \sigma', H', R'$*, then* $\vdash e', \sigma', H', R'$*, and* $\lfloor \sigma', H', R' \rfloor \vdash e' : T$*.*

PROOF: The proof is by induction on the typing derivation $\lfloor \sigma, H, R \rfloor \vdash e : T, \Gamma'$.

In order to handle the flow-sensitivity of the type system, the induction hypothesis is strengthened to:

If $\vdash e, \sigma, H, R$, and $\lfloor \sigma, H, R \rfloor \vdash e : T, \Gamma$, and $e, \sigma, H, R \longrightarrow e', \sigma', H', R'$, then $\vdash e', \sigma', H', R'$, and $\lfloor \sigma', H', R' \rfloor \vdash e' : T, \Gamma'$, and Γ' is an extension of Γ

In order to prove $\vdash e', \sigma', H', R'$, we only need to consider evaluation steps where $H' \neq H$ or $R' \neq R$. The congruence rule R-CONG can be proved by the induction hypothesis directly. The remaining cases are R-GET, R-SET, R-ALLOC, and R-VIEW, and the interesting cases are R-GET, R-VIEW, and R-SET, where R is updated with a value as the result of view or a new value with one of its masks removed. The proof is according to L-OK: when two classes are declared as being shared, their unmasked fields have shared types.

Now it remains to prove $\lfloor \sigma', H', R' \rfloor \vdash e' : T, \Gamma'$, where Γ' is an extension of Γ.

First, if the last derivation uses T-SUB, then it follows from the induction hypothesis and Lemma 3.3.2. Now it is only necessary to consider derivations that do not end with T-SUB. Consider different cases of e:

- $e = v$.

  Vacuously true, since evaluation cannot continue.

- $e = x$

  The evaluation is $x, \sigma, H, R \longrightarrow v, \sigma, H, R$ by R-VAR, where $\sigma(x) = v$. Suppose $v =< \ell, P!\backslash\overline{f} >$, and then by T-FIN and the definition of $\lfloor\sigma, H, R\rfloor$, we have $T = v.\texttt{class}\backslash\overline{f}$ and $\lfloor\sigma, H, R\rfloor \vdash x : T, \lfloor\sigma, H, R\rfloor$. Then by F-REF and T-FIN, $\lfloor\sigma, H, R\rfloor \vdash v : v.\texttt{class}\backslash\overline{f}, \lfloor\sigma, H, R\rfloor$.

- $e = e_0.f$

  - $e = v.f$

    Let $v =< \ell, S >$. There are also two cases for the derivation $\lfloor\sigma, H, R\rfloor \vdash v.f : T, \Gamma$: T-FIN and T-GET.

    * T-FIN

      Then $T = \mathsf{ptype}(\lfloor\sigma, H, R\rfloor, v.f)$, where f is a final field. The evaluation is $v.f, \sigma, H, R \longrightarrow v', \sigma, H, R'$, where $S = P!\backslash\overline{f'}$ and $H(\ell, \mathsf{fclass}(P, f), f) = v'$. According to Lemma 3.3.5, $\lfloor\sigma, H, R\rfloor \vdash \mathsf{ptype}(\lfloor\sigma, H, R\rfloor, v.f) \approx \mathsf{ptype}(\lfloor\sigma, H, R\rfloor, v')$. Then by T-FIN and T-SUB, $\lfloor\sigma, H, R\rfloor \vdash v' : \mathsf{ptype}(\lfloor\sigma, H, R\rfloor, v.f), \lfloor\sigma, H, R\rfloor$.

    * T-GET

      Let $\mathsf{ftype}(\lfloor\sigma, H, R\rfloor, S, f) = T_f$. By T-GET, $T = T_f$, and $\Gamma = \lfloor\sigma, H, R\rfloor$ since $e = v.f$. According to R-GET and the definition of the auxiliary function view, $\lfloor\sigma, H, R\rfloor \vdash v' : T, \lfloor\sigma, H, R\rfloor$. The applicability of view is based on the well-formedness of the heap.

  - $e = e_0.f$ where $e_0 \neq x$ and $e_0 \neq v$

Then R-CONG is the only rule that can apply, and $e_0, \sigma, H, R \longrightarrow e'_0, \sigma', H', R'$. There are also two cases for the derivation of $\lfloor \sigma, H, R \rfloor \vdash e_0.f : T, \Gamma$.

- $*$ T-FIN

  The proof is based on Lemma 3.3.5, which is similar to the corresponding case in the proof for $e = v.f$.

- $*$ T-GET

  Then $\lfloor \sigma, H, R \rfloor \vdash e_0 : T_0, \Gamma$ and $\mathsf{ftype}(\lfloor \sigma, H, R \rfloor, T_0, f) = T_f = T$. By the induction hypothesis, $\lfloor \sigma', H', R' \rfloor \vdash e'_0 : T_0, \Gamma'$, where Γ' is an extension of Γ. According to Lemma 3.3.2, $\mathsf{ftype}(\lfloor \sigma', H', R' \rfloor, T_0, f) = T_f$. Thus by T-GET, $\lfloor \sigma', H', R' \rfloor \vdash e'_0.f : T_f, \Gamma'$.

- $e = (x.f = e_0)$

  - $e = (x.f = v)$

    Then $x.f = v, \sigma, H, R \longrightarrow v, \sigma', H', R'$ where $\sigma' = \mathsf{grant}(\sigma, x.f)$, $\sigma(x) =< \ell, S >$, $H' = H[< \ell, \mathsf{fclass}(P, f), f >:= v]$ where $S = P!\backslash\overline{f'}$, and $R' = R, < P!\backslash(\overline{f'} - f) >$. Then T-SET applies to the derivation of $\lfloor \sigma, H, R \rfloor \vdash e : T, \Gamma$: $\Gamma = \mathsf{grant}(\lfloor \sigma, H, R \rfloor, x.f)$, and $\lfloor \sigma', H', R \rfloor = \mathsf{grant}(\lfloor \sigma, H, R \rfloor, x.f)$, so $\lfloor \sigma', H', R \rfloor$ is a trivial extension of Γ. According to the definition of runtime typing environment, $\lfloor \sigma', H', R' \rfloor$ may add to $\lfloor \sigma', H', R \rfloor$ several path equivalence relationships, and therefore an extension of $\lfloor \sigma', H', R \rfloor$, which means that $\lfloor \sigma', H', R' \rfloor$ is an extension of Γ.

  - $e = (x.f = e_0)$ where $e_0 \neq v$

    Then R-CONG can apply, and the induction hypothesis can be directly used.

- $e = (\texttt{final}\ S\ x = e_1; e_2)$

- $e = (\texttt{final}\ S\ \ x = v; e_2)$

  R-LET is the only rule that can apply: $e, \sigma, H, R \longrightarrow e_2, \sigma', H, R$ where $\sigma' = \sigma[x := v]$ and x is fresh variable. By T-LET and T-FIN, $\lfloor\sigma, H, R\rfloor \vdash v : S, \lfloor\sigma, H, R\rfloor$ and $\lfloor\sigma, H, R\rfloor, x : S \vdash e_2 : T, \Gamma_2$, and $\Gamma_2 = \Gamma, x : T'$ for some T'. Since $\sigma' = \sigma[x := v]$ and x is fresh, we have $\lfloor\sigma', H, R\rfloor = \lfloor\sigma, H, R\rfloor, x : T''$ for some T'', where $\vdash T' \leq S$, according to the definition of the run-time typing environment. Therefore $\lfloor\sigma', H, R\rfloor \vdash e_2 : T, \Gamma_2'$, where $\Gamma_2' = \Gamma', x : T'''$ for some T'''. It is then easy to see that Γ' is an extension of Γ.

- $e = (\texttt{final}\ S\ \ x = e_1; e_2)$ where $e_1 \neq v$

  Then R-CONG can apply, and therefore the induction hypothesis can be used.

• $e = e_0.m(\overline{e})$

- $e = < \ell, S > .m(\overline{v})$

  Then R-CALL is the only rule that can apply: $< \ell, S > .m(\overline{v}), \sigma, H, R \longrightarrow e_m\{y_0/\texttt{this}, \overline{y}/\overline{x}\}, \sigma', H, R$, where $\sigma' = \sigma[y_0 := < \ell, S >, \overline{y} := \overline{v}]$, and $y_0, y_1, \ldots, y_n \notin \mathrm{dom}(\sigma)$. Therefore, $\lfloor\sigma', H, R\rfloor$ is an extension of $\lfloor\sigma, H, R\rfloor$. According to M-OK and simple α renaming, $\lfloor\sigma, H, R\rfloor, y_0 : T_0, \overline{y} : \overline{T} \vdash e_m\{y_0/\texttt{this}, \overline{y}/\overline{x}\} : T_{n+1}$, where $\mathsf{mbody}(S, m) = T_{n+1}\ m(\overline{T}\ \ \overline{x})\ \{e_m\}$. Now applying Lemma 3.3.3 and Lemma 3.3.4, for $y_0, y_1, \ldots, y_n$, we can get $\lfloor\sigma', H, R\rfloor \vdash e_m : T_{n+1}^{n+1}$, where $T_i^0 = T_i$ and $T_i^j = T_i^{j-1}\{\!\{\lfloor\sigma', H, R\rfloor;\ T_{j-1}^{j-1}/y_{j-1}\}\!\}$. $T_{n+1}^{n+1} = T$ in T-CALL. Let $\lfloor\sigma, H, R\rfloor \vdash e : T, \Gamma$ and $\lfloor\sigma', H, R\rfloor \vdash e_m\{y_0/\texttt{this}, \overline{y}/\overline{x}\} : T, \Gamma'$. It is obvious that Γ' is an extension of Γ, because $\mathsf{FV}(e_m) \subseteq \{\texttt{this}, \overline{x}\}$.

- $e = e_0.m(\overline{e})$ where $e_i \neq v$ for some $i \in \{0, 1, \ldots, n\}$

Then R-CONG can apply. This case is easy to prove by the induction hypothesis.

- $e = \texttt{new}\, S$

  Then R-ALLOC can apply: $\texttt{new}\ S, \sigma, H, R \longrightarrow e', \sigma, H, R'$, where $e' = \texttt{final}\ S!\backslash\overline{f}\ x = v;\ x.\overline{f} = \overline{e}\{x/\texttt{this}\};\ x$, and $\mathsf{fields}(S) = [\texttt{final}]\ \overline{T_f}\ \overline{f} = \overline{e}$, and $v =< \ell, S!\backslash\overline{f} >$, and $R' = R, v$. It is easy to see that $\lfloor\sigma, H, R'\rfloor \vdash e' : S!$, because all the masks are going to be removed by the field initializers. It is also obvious that $\lfloor\sigma, H, R'\rfloor$ is an extension of $\lfloor\sigma, H, R\rfloor$ by the definition of run-time typing environments.

- $e = e_1;\ e_2$

  - $e = v;\ e_2$

    Then R-SEQ can apply: $e, \sigma, H, R \longrightarrow e_2, \sigma, H, R$. By T-SEQ, $\lfloor\sigma, H, R\rfloor \vdash v : T_1, \Gamma_1$ and $\Gamma_1 \vdash e_2 : T, \Gamma$. Also, it is easy to see that $\Gamma_1 = \lfloor\sigma, H, R\rfloor$. Therefore $\lfloor\sigma, H, R\rfloor \vdash e_2 : T, \Gamma$.

  - $e = e_1;\ e_2$ where $e_1 \neq v$

    Then R-CONG can apply, and the proof is by the induction hypothesis.

- $e = (\texttt{view}\ S)e_0$

  - $e = (\texttt{view}\ S)< \ell, S' >$

    Then R-VIEW is the only rule that can apply. By T-VIEW, $\lfloor\sigma, H, R\rfloor \vdash (\texttt{view}\ S)< \ell, S' > : S$, and there is a type T such that $\lfloor\sigma, H, R\rfloor \vdash < \ell, S' >: T$ and $\lfloor\sigma, H, R\rfloor \vdash T \rightsquigarrow S$. According to the definition of view, $\lfloor\sigma, H, R\rfloor \vdash \mathsf{view}(< \ell, S' >, S) : S$, and since $\lfloor\sigma, H, R'\rfloor$, where $R' = R, \mathsf{view}(< \ell, S' >, S)$, is an extension of $\lfloor\sigma, H, R\rfloor$, we have $\lfloor\sigma, H, R'\rfloor \vdash \mathsf{view}(< \ell, S' >, S) : S$.

Now it remains to prove that the application of the auxiliary function view is well-defined. If $S' \leq S$, it is obviously well-defined. Otherwise, let $S' = P'!\backslash\overline{f'}$, and $S = PS\backslash\overline{f}$, and we need to prove that $\exists! P \,.\, \exists\overline{f''} \supseteq \overline{f'} \,.\, \vdash P! \leq PS \wedge \vdash P'!\backslash\overline{f''} \rightsquigarrow P!\backslash\overline{f}$.

As shown above, there is a type T such that $\lfloor\sigma, H, R\rfloor \vdash T \rightsquigarrow S$, and therefore $\emptyset \vdash T \rightsquigarrow S$, according to the definition of $\lfloor\sigma, H, R\rfloor$. Then T has to be non-dependent. Also, it is obvious that $\vdash P'!\backslash\overline{f'} \leq T$, so $\overline{f'}$ must be a subset of the masks in T, which is $\overline{f''}$. Now the proof goes on by induction on the derivation of $\emptyset \vdash T \rightsquigarrow S$:

- SH-REFL

  Vacuously true, since it implies that $\vdash S' \leq S$, which is the first case of the definition of view.

- SH-TRANS

  By the inner induction hypothesis.

- SH-ENV

  Vacuously true.

- SH-MASK

  By the inner induction hypothesis.

- SH-DECL

  Then both T and S are exact types, and therefore $S = P!\backslash\overline{f}$ and $T = P'!\backslash\overline{f''}$, by the definition of exact types.

- SH-CLS

  The premise of SH-CLS is exactly what we want to prove.

– $e = (\texttt{view}\ S)e_0$ where $e_0 \neq v$

Then R-CONG can apply. This case can be easily proved using the induction hypothesis.

□

Lemma 3.3.7 *(Progress) If $\vdash e, \sigma, H, R$ and $\lfloor \sigma, H, R \rfloor \vdash e : T$, then $e = v$, or there is a configuration e', σ', H', R' such that $e, \sigma, H, R \longrightarrow e', \sigma', H', R'$.*

PROOF: The proof is by structural induction on e.

- $e = v$

  Trivial.

- $e = x$

  Then $\lfloor \sigma, H, R \rfloor \vdash x : T$, and by the definition of $\lfloor \sigma, H, R \rfloor$, $\sigma(x) =< \ell, T >$. Therefore R-VAR applies, and we have $e' =< \ell, T >$, $\sigma' = \sigma$, $H' = H$, and $R' = R$.

- $e = e_0.f$

  If $e_0 \neq v$, then it is easy to see that R-CONG applies, and the evaluation can make progress.

  If $e_0 = v$ where $v =< \ell, P!\backslash\overline{f'} >$, then by T-GET, and the definition of ftype, we know $f \notin \overline{f'}$. Then according to configuration well-formedness CONFIG, $H(\ell, \mathsf{fclass}(P), f) = v'$, and $\mathsf{view}(v', S)$ is well-defined ($S = \mathsf{ftype}(\emptyset, P!\backslash\overline{f'}, f)$). Therefore R-GET applies, and the evaluation can make progress.

- $e = x.f = e_0$

  If $e_0 \neq v$, then R-CONG applies.

Suppose $e_0 = v$. By T-SET and T-FIN, $\lfloor\sigma, H, R\rfloor \vdash v : T_v, \lfloor\sigma, H, R\rfloor$, and then by T-SET, $\lfloor\sigma, H, R\rfloor \vdash x : T_x$. By the definition of $\lfloor\sigma, H, R\rfloor$, we have $\sigma(x) =< \ell, T_x >$. It must be the case that $T_x = P!\backslash\overline{f'}$, for some P and $\overline{f'}$, because the view T_x is always a non-dependent exact type. Then R-SET applies, and $e' = v$, $\sigma' = \mathsf{grant}(\sigma, x.f)$, $H' = H[< \ell, \mathsf{fclass}(P, f), f >:= v]$, and $R' = R$.

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

  If e_0 or any e_i of $\overline{e}$ is not a value, R-CONG applies, and the evaluation can make progress.

  Otherwise, $e = v_0.m(\overline{v})$, where $v_0 =< \ell_0, S_0 >$. By T-CALL, and the definitions of mtype and mbody, method lookup succeeds and we have $\mathsf{mbody}(S, m) = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e_m\}$. Then R-CALL applies, and $e' = e_m\{y_0/\mathtt{this}, \overline{y}/\overline{x}\}$ where all the y_i are fresh variables, $\sigma' = \sigma[y_0 :=< \ell_0, S_0 >, \overline{y} = \overline{v}]$, $H' = H$, and $R' = R$.

- $e = e_1;\ e_2$

  If $e_1 = v$, then R-SEQ applies, and $e' = e_2$, $\sigma' = \sigma$, $H' = H$, and $R' = R$; otherwise, by the induction hypothesis, there exists e_1', σ', H', and R', such that $e_1, \sigma, H, R \longrightarrow e_1', \sigma', H', R'$. Then R-CONG applies.

- $e = \mathtt{new}\ T$

  If T is not a non-dependent type, then the type T can be further evaluated.

  Now suppose $T = S$. By R-ALLOC, $e' = \mathtt{final}\ S!\backslash\overline{f}\ x = v;\ x.\overline{f} = \overline{e}\{x/\mathtt{this}\};\ x$, where $\overline{f}$ is the set of all fields of S, x is a fresh variable, and $v =< \ell, S!\backslash\overline{f} >$ with a fresh location ℓ. We also have $\sigma' = \sigma$, $H' = H$, and $R' = R, v$.

- $e = (\mathtt{view}\ T)e_0$

  If $e_0 \neq v$ or $T \neq S$, then R-CONG can apply.

If $e_0 = v$ and $T = S$ where S is a non-dependent type, then $e' = \mathsf{view}(v, S)$, $\sigma' = \sigma, H' = H$, and $R' = R, \mathsf{view}(v, S)$. We only need to prove that $\mathsf{view}(v, S)$ is well-defined. Let $S = PS \backslash \overline{f}$ and $v = < \ell, P'!\backslash \overline{f'} >$. By T-FIN and T-VIEW, $\lfloor \sigma, H, R \rfloor \vdash P'!\backslash \overline{f''} \rightsquigarrow PS \backslash \overline{f}$ and $\overline{f'} \subseteq \overline{f''}$. Now the proof is by induction on the derivation of $\lfloor \sigma, H, R \rfloor \vdash P'!\backslash \overline{f''} \rightsquigarrow PS \backslash \overline{f}$:

- SH-REFL

  Then $P'!\backslash \overline{f''} = PS \backslash \overline{f}$, and $\mathsf{view}(v, S)$ is trivially well-defined.

- SH-TRANS

  Then there exists a non-dependent type T', such that $\lfloor \sigma, H, R \rfloor \vdash P'!\backslash \overline{f''} \rightsquigarrow T'$ and $\lfloor \sigma, H, R \rfloor \vdash T' \rightsquigarrow S$. Note that $\lfloor \sigma, H, R \rfloor$ contains no sharing relationships, which means that the T' has to be non-dependent, and $\lfloor \sigma, H, R \rfloor$ can actually be replaced with empty environment in the above two judgments. By the induction hypothesis, $\mathsf{view}(v, T') = < \ell, P''!\backslash \overline{f'''} >$ is well-formed, and $\mathsf{view}(< \ell, P''!\backslash \overline{f'''} >, S)$ is also well-formed. Now just observe that any combination of the two cases in the definition of the auxiliary function view implies the well-formedness of $\mathsf{view}(v, S)$.

- SH-ENV

  Vacuously true, because $\lfloor \sigma, H, R \rfloor$ does not contain any sharing relationships.

- SH-MASK

  It is easy to see that adding a mask on both types does not affect the applicability of view.

- SH-DECL

  Then S is an exact type, and is itself the unique target type in the result of the view function.

- SH-CLS

  The premise directly implies that $\mathsf{view}(v, S)$ is well-defined.

- $e = \texttt{final}\ T\ x = e_1;\ e_2$

  If $e_1 \neq v$ for any value v, then by T-LET, $\lfloor\sigma, H, R\rfloor \vdash e_1 : T_1, \Gamma$. Therefore, by the induction hypothesis, there exists e_1', σ', H', and R', such that $e_1, \sigma, H, R \longrightarrow e_1', \sigma', H', R'$. Then R-CONG applies.

  If $e_1 = v$, then R-LET applies, and $e' = e_2\{y/x\}$ where y is a fresh variable, $\sigma' = \sigma[y := v]$, $H' = H$, and $R' = R$.

□

Now we can prove the soundness theorem of the J&$_s$ calculus.

Theorem 3.3.8 *(Soundness) If* $\vdash < \overline{L}, e >\ \mathsf{ok}$*, and* $\emptyset \vdash e : T$*, and* $e, \emptyset, \emptyset, \emptyset \longrightarrow^* e', \sigma, H, R$*, then either* $e' = v$ *and* $\lfloor\sigma, H, R\rfloor \vdash v : T$*, or* $\exists e'', \sigma', H', R'\ .\ e', \sigma, H, R \longrightarrow e'', \sigma', H', R'$*.*

PROOF: Follows from Lemma 3.3.6 and Lemma 3.3.7. □

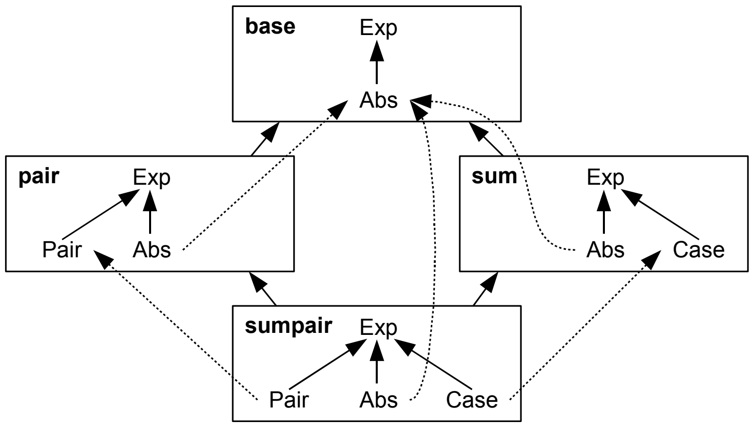

Figure 3.20: Lambda compiler structure. Translator and some AST nodes not shown.

3.4 Experience

3.4.1 Lambda compiler

The J&$_s$ implementation successfully compiles and executes the completed version of the lambda compiler shown in Figure 3.6, which compiles λ-calculus enhanced with sums and pairs down to simple λ-calculus.

The structure of the lambda compiler is illustrated in Figure 3.20. Solid arrows represent class and family inheritance, and dashed arrows represent sharing declarations.

The lambda compiler has a `base` family with classes representing AST nodes for simple λ-calculus. There are two families directly derived from the base family, extended with sums and pairs respectively. The `sum` family and the `pair` family each share AST classes from the `base` family, and each implement in-place translation to simple λ-calculus, as shown in Figure 3.6. The last derived family `sumpair` composes the `sum` family and the `pair` family, leading to a compiler that supports both sums and pairs. The `sumpair` family shares `Abs` and other AST classes from the simple λ-calculus with the `base` family, and by transitivity, with `sum` and `pair`. The code of `sumpair` just sets up the sharing relationships, without a single line of translation code. The in-place translation code from `sum` and `pair` is composed to translate away sums and pairs.

This program is about 250 lines long, but uses the features of J&$_s$ in a sophisticated way. For example, families have both shared and unshared classes, masked types are used to ensure objects of new AST classes (pair and sum) are translated away, and the two translations are composed to translate pairs and sums at once. The lambda compiler example is inspired by the Polyglot framework, and it encapsulates most of the interesting issues that arise in making Polyglot extensible. The results suggest Polyglot would be simpler in J&$_s$, but this is left for future work.

3.4.2 CorONA

Our second significant example shows that the adaptation capability of J&$_s$ can be used to seamlessly upgrade a running server and its existing state with entirely new functionality. We ported CorONA, an RSS feed aggregation system [66], to J&$_s$, and successfully used class sharing to update the system at run time with a new caching algorithm.

CorONA is originally an extension of Beehive [67], which is itself an extension of Pastry [70] that provides a distributed hash table. Beehive extends Pastry with active replication, whereas PC-Pastry [67] extends Pastry with passive caching. We refactored the ported CorONA, and composed it with PC-

Pastry and Beehive respectively, creating two applications (named PCCorONA and BeeCorONA) with different caching strategies. The system is tested by first running PCCorONA for a while, and then evolving the running system from passive caching to active caching by compiling and loading a new package, BeeCorONA.

Sharing declarations are added so that classes representing host nodes, data objects, and network addresses are shared between CorONA and its two derived families. Classes for network messages and cache management are not shared. The evolution code goes over all the host nodes, which are the top-level objects in the system, changing their views, and creating new caching managers.

The amount of code to implement evolution is relatively small (less than 40 lines of code, compared to 8300 for the whole system, excluding comments and empty lines). Very little code is needed because we only need to change the views of host nodes; all the other referenced objects will have their views changed implicitly when they are accessed. In the host node class, fields storing the caching managers are not shared, and masked types ensure that they are initialized in the evolved system. With a slightly different configuration, we can actually run the two variants of the system at the same time, using the same set of host node objects. View-dependent types ensure that network messages of correct versions are created and accepted for each of the systems.

On the other hand, if we were to use other mechanisms, e.g., the adapter design pattern, dynamic software updating would be much harder, if not impossible, to implement. The reason is that the adapter and the adaptee classes in general are not related through subtyping, and therefore a lot more code needs to be updated to switch to using the adapter type.

3.5 Related work

Adaptation. The Adapter design pattern [30] is a protocol for implementing adaptation. However, this and other related patterns are tedious and error-prone to implement, rely on statically unsafe type casts, and do not preserve object identity or provide bidirectional adaptation as J&$_s$ does.

Expanders [85] are a mechanism for adaptation. New fields, methods, and superinterfaces can be added into existing classes. Expanders are more expressive than *open classes* [13], which can only add new methods. Method dispatch is statically scoped, so expanders do not change the behavior of existing clients. New state is added by wrapper classes; a map from objects to wrappers ensures uniqueness of wrapper instances.

CaesarJ [52, 1] is an aspect-oriented language that supports adaptation with *wrappers* called *aspect binders*. Wrappers and expanders are similar. They both can extend wrapped classes with new states, operations, and superinterfaces; no duplicate wrappers are created for objects; and dynamic wrapper selection is similar to expander overriding. Wrappers in CaesarJ are less transparent: a wrapper constructor must be called to get a wrapper instance, whereas expander operations can be applied directly to objects. Multiple inheritance in CaesarJ makes wrapper selection ambiguous; J&$_s$ disambiguates via views.

Both expanders and CaesarJ wrappers share limitations: it is impossible to override methods in the original family, and therefore there is no dynamic dispatching across families; object identity is not preserved; the use of expanders and wrappers is limited to adaptation, since the adapter family cannot be used independently of its original family. Class sharing in J&$_s$ provides more flexibility.

The Fickle$_{\text{III}}$ [16] language has a *re-classification* operation for objects to change their classes. Re-classifications are similar to view changes in J&$_s$, but directly change the behavior of all existing references to the object; therefore, effects are needed to track the change. A re-classification might leave fields uninitialized, while masked types in J&$_s$ ensure that after a view change, fields are initialized before use. Fickle$_{\text{III}}$ does not support class families.

Chai$_3$ [72] allows traits to be dynamically substituted to change object behaviors, similar to view changes in J&$_s$. However, *Chai*$_3$ does not support families, and the fact that traits do not have fields makes it harder to support manipulation of data structures.

Some work on adaptation, including pluggable composite adapters [53], object teams [32], and delegation layers [62], also has some notion of families of classes. However, family extensibility does not apply to the relationship between the family of adapter classes and the family of adaptee classes. Therefore, these mechanisms either do not support method overriding and dynamic dispatch between the adapter and adaptee families [53, 32], or have a weaker notion of families in which programmers have to manually "wire" inheritance relationships between the base family and the delegation family [62]. These mechanisms all use lifting and lowering, introduced in [53], to convert between adapter and adaptee classes. Lifting and lowering are similar to view changes in J&$_s$, but are not symmetric and do not support late binding.

Family inheritance. Several different mechanisms have been proposed to support family inheritance, including virtual classes, nested inheritance, variant path types, mixin layers, etc. In all these family inheritance mechanisms, families of classes are disjoint, whereas with class sharing, different families can share classes and their instances.

Virtual classes [50, 51, 21, 24, 12] are inner classes that can be overridden just like methods. Path-dependent types are used to ensure type safety. The soundness of virtual classes has been formally proved by Ernst et al. [24], and by Clarke et al. [12].

Nested inheritance [56] supports overriding of nested classes, which are similar to virtual classes. Nested intersection [58] adds and generalizes intersection types [68, 14] in the context of nested inheritance to provide the ability to compose extensions.

Both virtual classes and nested inheritance support *higher-order hierarchies* [23].

Virtual classes support *family polymorphism* [22], where families are identified by the enclosing instance. Nested inheritance supports what Clarke et al. [12] called *class-based family polymorphism*, where each dependent class defines a family of classes nested within and also enclosing it. With prefix types, any instance of a class in the family can be used to name the family.

Although virtual classes differ from nested inheritance, class sharing should be applicable to any family inheritance mechanism.

Variant path types [38] support family inheritance without dependent types, using exact types and relative path types (similar to `this.class`) for type safety. These exact types are very different from those in J&$_s$: in a J&$_s$ exact type $A.B!.C$, exactness applies to the whole prefix before !, that is, $A.B$; in an exact type $A@B.C$ in [38], exactness applies to the simple type name right after @, that is, B.

Mixin layers [71] generalizes *mixins* [5]. Mixins are classes that can be instantiated with different superclasses, and mixin layers are mixins that encapsulate other mixins. Mixin layers support family inheritance: when a mixin layer is instantiated, all the inner mixins are instantiated correspondingly.

Virtual types [81, 8, 82, 35] are type declarations that can be overridden. Virtual types are more limited than virtual classes: they provide family polymorphism but not family inheritance. Scala [60, 61] supports family polymorphism and composition through virtual types, path-dependent types, and mixin composition. It also supports parametric polymorphism. Scala does not have virtual classes and does not support family inheritance. Scala has *views* that are implicitly-called conversion functions. Scala views do not provide adaptation: they create new instances in the target types.

Sharing in functional languages. Wadler's views [83] are an isomorphism between a new data type and an existing one, which is similar to a sharing declaration. Of course, there are obvious differences: Wadler's views are for a functional setting, and primarily relate abstract data types and types inductively defined with pattern matching, whereas J&$_s$ creates sharing relationships only between overridden and overriding classes.

The SML module system [49] has *sharing constraints*, which require functor module parameters to agree on type components. SML sharing constraints dictate applicability of functors; J&$_s$ sharing constraints dictate applicability of method bodies.

More recently, Dreyer and Rossberg [**?**] propose MixML that generalizes ML modules and mixin modules into a unified framework. MixML also supports merging of recursive modules, which has some similarity with nested intersection, although there is no overriding and therefore the merged modules have to be statically conflict-free.

Safe dynamic software updating. There is prior work on the problem of safely updating software without downtime.

Barr and Eisenbach [2] propose a framework to support dynamic update of Java components that satisfy the binary compatibility requirement [78], which also includes a custom classloader. The goal is to provide a tool that keeps Java libraries up to date, rather than to improve the extensibility of the language.

Duggan [19] describes a calculus that combines Wadler's views and SML sharing constraints to support hot-swapping modules. J&$_s$ differs because it does not copy values across different versions; instead, it generates different views on the same object.

Proteus [75] finds proper timing for a given global update with static analysis. Abstract and concrete types in Proteus bear some resemblance to inexact and exact types in J&$_s$: abstractly typed variables allow values of different concrete types, and inexactly typed variables may store objects with different exact views.

CHAPTER 4

HOMOGENEOUS FAMILY SHARING

Homogeneous family sharing lifts the sharing mechanism from class-level sharing to true family-level sharing. When a derived family is declared as sharing with the base family, every pair of corresponding classes from the two families are shared, without explicit sharing declarations for individual nested classes. For each new class introduced in the shared derived family, an implicit *shadow class* is generated in the base family, which ensures that all nested classes are still homogeneously shared, and provides a new kind of extensibility. Compared to heterogeneous sharing shown in Chapter 3, homogeneous family sharing presents a cleaner, more scalable solution to the problem of integrating in-place extensibility and family extensibility. For this reason, we believe that homogeneous sharing is the right way to support sharing, and include it in the latest version of the J& language.

4.1 Class sharing without families

4.1.1 From inheritance to sharing

In an object-oriented language like Java, inheritance offers a way to reuse and extend existing code. For example, as shown in Figure 4.1, a graphical user interface (GUI) library may contain a class for describing a button, with a method `draw` for drawing the button. Suppose the programmer would like to have buttons that are rendered with a shadow underneath. The programmer could declare a `PrettyButton` class that extends the original `Button` class through

```
class Button {          | class PrettyButton extends Button {
  void draw(...) {      |   void draw(...) {
    ...                 |     ... draw the shadow ...
  }                     |   }
}                       | }
```

Figure 4.1: A GUI button class and its extension

inheritance. In Figure 4.1, the subclass `PrettyButton` *overrides* the `draw` method, originally introduced in the superclass `Button`, so when the `draw` method is called with an object of `PrettyButton`, the overriding version in the subclass is executed.

However, inheritance has the limitation that the new functionality of drawing a shadow is not available to any object of the existing `Button` class. An application that creates instances of the `Button` class cannot enjoy the prettier GUI element without code changes, nor can a running application with `Button` objects be easily upgraded to the new look.

The ability to augment existing objects with extended functionality is called *adaptation*. The adapter design pattern [30] may be used to implement adaptation, but it requires the programmer to write much code that is error-prone, and relies on statically unsafe type casts. The programmer also has to manually manage the relationship between the adapter object and the adapted object.

Class sharing, as proposed in this paper, provides a language-based solution to the problem of adaptation. In this approach, the new `PrettyButton` class may be declared to *share* with the `Button` class, rather than to inherit it, as shown below.

```
  class PrettyButton shares Button { ... }
```

The meaning of the declaration above is twofold: first, `PrettyButton` is a subtype of `Button` as in the case of inheritance; second, every `Button` object is also an object of `PrettyButton`, and vice versa. The functionality implemented in `PrettyButton` is available to every object of `Button`, even if the object were created before `PrettyButton` was loaded. An object created as a `PrettyButton` may also access the original version of the `draw` method. Therefore, sharing can be seen as symmetric subclassing: an object of either of the two classes inherits functionality from the other class, and each class may also override the other. It also provides symmetric adaptation that preserves object identity.

The original form of class sharing in J&$_s$ [65] does not support the sharing declaration shown above, which is sharing between two top-level classes that are not nested in another class or package. J&$_s$ only allows sharing between corresponding classes—classes of the same name—from different families. The homogeneous sharing proposed in this paper generalizes to sharing between individual classes that are top-level, and to sharing between those contained in the same class or package.

4.1.2 Views and view changes

When two classes are declared as being shared, a single object might be treated as an instance of either one. Each class is a distinct *view* of that object. At run time, an object reference may be modeled as a pair $< \ell, C! >$ of a heap location ℓ and an *exact type* [7] $C!$ as the view. The exact type $C!$ here represents the fact that at run time, the view on that object through the reference is known to be exactly class C, not any subclass of it.

The view defines the behavior of the object when it is accessed through the reference. For example, an object of class `Button` uses the overriding version of the `draw` method if accessed through a reference with the view `PrettyButton!`, but not through a `Button!` view.

In order for an object to obtain a new reference with a new view, the *view change* operation—written `(view` T`)`e, where T is the target type, and e is the source expression to apply the view change on—should be used. For example, a `Button` object may obtain a new reference, through which the new `draw` method can be called, although the receiver object is still the same.

```
Button! b1 = new Button(...);
PrettyButton! b2 = (view PrettyButton!)b1;
b1.draw(...); // the old draw method in Button
b2.draw(...); // the new draw method in PrettyButton
```

4.1.3 Sharing relationships

A sharing declaration establishes a *sharing relationship* between the two classes. For example, there is a sharing relationship between `Button` and `PrettyButton`, represented as `Button!` $\leftrightarrow$ `PrettyButton!`.

All the sharing declarations together induce an equivalence relation on classes that is the reflexive, symmetric, and transitive closure of the declared sharing relationships. The J& type system maintains the sharing relation, and uses it to type-check view change operations.

```
package GUI;                 package xlucentGUI extends GUI;
class Button {...}           class Button {
class RadioButton              void draw(...) {...}
  extends Button {...}       }
class Window {               class Window {
  Button close; ...            void draw(...) {...}
}                            }
...                          ...
```

Figure 4.2: A base GUI family and an extension

4.2 Homogeneous family sharing

4.2.1 Family inheritance

The J& language supports nested inheritance [56] and nested intersection [58]. Nested inheritance is inheritance at the granularity of a *namespace* (a package or a class), which defines a family in which related classes are grouped. When a namespace inherits from another (base) namespace, all the namespaces nested in the base namespace are inherited, and the derived namespace can *override*, or *further bind* [50] inherited namespaces, changing nested class declarations, similarly to virtual classes [50, 51, 21, 24, 12]. In addition, nested intersection supports *composing* families with generalized *intersection types* [68, 14], which is essentially multiple inheritance at the family level.

Family inheritance is useful, because in real software systems, the functionality that needs to be extended often spans multiple classes that are related to each other, through inheritance or mutual references. For example, as shown in Figure 4.2, the GUI library mentioned in Section 4.1.1 should also include classes for various GUI elements, including the `Button` class in Figure 4.1. Suppose we would like to extend the entire GUI library to support translucent widgets. With the J& language, the extension can be declared at the family level, as shown on the right of Figure 4.2. Every class in the base package `GUI` has a corresponding subclass with the same class name in the derived package `xlucentGUI`.

J& supports *scalable extensibility* [56], where there is no need to declare cross-family inheritance for individual classes, and the code that needs to be written in the derived family is proportional to the added functionality.

The relationships between classes in `GUI` are preserved in `xlucentGUI` with *late binding of type names*. For example, the `Window` class has a field for the "close" button with the type `Button`. In `GUI.Window`, it refers to the `Button` class in `GUI`, and in `xlucentGUI.Window`, it refers to that class of `xlucentGUI`. The type safety of late-bound type names is ensured with *prefix types* and *dependent classes*: the unqualified field type `Button` is actually sugar for the prefix type **GUI**`[`**this**.**class**`]`.**Button**, which means that the container package of `Button` is a subtype of `GUI`, and encloses the run-time class of the special variable `this`. Therefore, the `Window` object stored in `this` and the `Button` object stored in the field `close` will always be in the same family.

Family inheritance in J& preserves all three kinds relationships in the derived family: referencing, subclassing, and sharing. On the contrary, J&$_s$ does not preserved sharing relationships.

4.2.2 Family sharing

This paper introduces a new version of J& that scales sharing to the granularity of a family, analogously to the way that J& scales inheritance to the granularity of a family. For example, in order to adapt existing objects from the `GUI` family with the new drawing methods declared in `xlucentGUI`, the following *family sharing* declaration may be used, with the rest of the code shown in Figure 4.2 remaining the same.

```
package xlucentGUI shares GUI;
```

With the sharing declaration, the `xlucentGUI` is a derived family of `GUI`, just as with the inheritance declaration in Figure 4.2. Therefore, the shared derived family `xlucentGUI` inherits all the nested classes from the base family `GUI`, and preserves all the relationships among them, including sharing relationships. If the `GUI` package contains the `PrettyButton` class that is declared to share with `Button`, in the `xlucentGUI` package, `Button` and `PrettyButton` are still shared.

The difference from nested inheritance is that corresponding classes from the two families (e.g., `GUI.Button` and `xlucentGUI.Button`) also become shared, and view changes can be used to move an object from one family to another. For example, a `Button` object from the `GUI` family may acquire the ability to be translucent:

```
  GUI!.Button b1 = ...;
  xlucentGUI!.Button b2 = (view xlucentGUI!.Button)b1;
```

In J&, a sharing relationship between two namespaces recursively applies to all their corresponding nested namespaces. Therefore, family sharing is *homogeneous*. This compares to the original *heterogeneous* class sharing [65], where sharing is declared for individual pairs of corresponding classes between two families, but not all nested classes need be shared.

Homogeneous family sharing has several advantages over heterogeneous sharing:

- *It is a more scalable extensibility mechanism.*

  There is no need to declare sharing for individual pairs of classes from two different families, which might be tedious and error-prone. This also makes the sharing mechanism more scalable, because if a class only needs to be shared but not overridden, its declaration may be omitted from the source code of the derived family.

- *It provides modular type checking with reduced annotation burden on the programmer.*

  In particular, homogeneous sharing does not require *sharing constraints* [65], because it is easier to prove that two types are shared. If two types T_1 and T_2 are declared to be shared, all corresponding nested types are also shared, that is, $T_1!.C \leftrightarrow T_2!.C$. (See Section 4.3 for a formal treatment.)

  For example, a variable `b1` of type `GUI!.Button` might point to an object of the `GUI.RadioButton` class or to one of the `GUI.PrettyButton` class. In either case, the view change `(view` **`xlucentGUI!.Button`**`)`**`b1`** is type-safe, because sharing is declared between `GUI` and `xlucentGUI`—therefore, all classes in `GUI` that are subtypes of `GUI!.Button` have shared counterparts in `xlucentGUI`.

  View changes support late binding, where the actual run-time view of the result of `(view` T`)`e is a subtype of the target type T that is shared with the run-time view of the value of e. With heterogeneous sharing, in order to prove that the view change is valid, the type system has to inspect all the subclasses of both the source type (the type of e) and the target type T, and it has to recheck it every time the view change code is inherited by a different family. Therefore, heterogeneous sharing in J&$_s$ uses sharing constraints to make type-checking modular. With homogeneous sharing, these constraints are superfluous.

- *A data structure consisting of multiple interconnected objects may safely change its view from one family to another.*

  For example, a non-translucent GUI would typically be represented as a tree data structure with a `Window` object as the root, and various other objects for buttons, menus, etc.. When a view change to `xlucentGUI!.Window` is applied to the `Window` object, any reachable object in the `GUI` family (e.g., the one stored in the field `close` in the `Window` class) would definitely have a view in `xlucentGUI` to change to. In fact, field accesses in J& automatically and lazily trigger these view changes.

  In heterogeneous sharing, a shared class may contain a field with an unshared type, in which case the field actually has several duplicates, one for each class that is shared with the class that declares the field. Masked types are used to ensure unshared objects stored in these fields do not leak into incompatible families. This adds to the annotation burden, and complicates reasoning about the code. Homogeneous family sharing does not need masked types for this purpose.

```
class A1 {
  class B {
    void m() {...}
  }
}
class A2 shares A1 {
  class C extends B { // introduces a shadow class in A1
    shadow void m()   // a shadow method
    { ... }
  }
}
```

Figure 4.3: Shadow classes and shadow methods

4.2.3 Shadow classes and shadow methods

Declaring shadow classes and methods. In the J& language, a derived family may introduce new nested classes not present in a base family it shares with. In order to keep sharing homogeneous and type-safe, these new nested classes in general need to be shared as well. Shadow classes are introduced in the base family to make this possible.

Figure 4.3 shows a simple example that illustrates the situation. The derived family enclosed in `A2` is shared with the base family in `A1`. `A2` inherits the nested class `B` from `A1`, and introduces a new class `C` that is a subclass of `A2.B`. Because sharing is homogeneous, the two types `A1!.B` and `A2!.B` must be shared, and therefore a view change from `A2!.B` to `A1!.B` must be allowed. But as shown in following code, the source expression `b2` of type `A2!.B` may actually refer to an object of class `A2.C`. For the view change to work, `A1` needs to contain a class that is shared with `A2.C`.

```
A2!.B b2 = new A2.C();
A1!.B b1 = (view A1!.B)b2;
b1.m(); // invokes the shadow method
```

J& introduces shadow classes to ensure sharing is still homogeneous and to enable the above view change. When a shared derived family (e.g., `A2`) introduces a new nested class (e.g., `A2.C`), a shadow class (`A1.C` in the example) with the same name as the new nested class is created in the base family. The derived family that introduces the new nested class is called the *originating family* for the shadow class, and the nested class is called the *originating class*. The shadow class is shared with its originating class, and inherits all the relationships from the originating family. Thus, the shadow class is a class that the base family inherits from the derived family.

The shadow class may need to behave differently than the originating class in order to be consistent with other classes in the base family. J& allows the originating class to declare *shadow methods*, with the method modifier `shadow`. Though shadow methods are declared in the derived (originating) family, the semantics of shadow methods is as if they were defined within an explicit declaration of the shadow class in the base family.

Modular type-checking of shadow classes Shadow classes are generated from their originating classes. Therefore, the type system needs to know the originating class before a shadow class is mentioned. On the other hand, modularity of the type system requires that it should be possible to type-check a base family without knowing about its derived families that are not used in the source code of the base family. For example, inside class `A1` in Figure 4.3, the type name C does not have a meaning before knowing the existence of A2.C.

J& ensures the modularity of the type system by disallowing direct naming of a shadow class in the source code. For example, given the declarations shown in Figure 4.3, one cannot mention A1.C, the fully qualified name of the shadow class, anywhere in the source code, nor can she mention just C in the context of A1. Instead, the language overloads the disambiguation usage of prefix types [58] to provide an indirect way of naming a shadow class that embeds the name of the originating class: in the J& source code, a shadow class is referred to as $P[T].C$, where P is the originating family, T is a type in the same family as the shadow class, and C is the simple name of the originating class. Therefore, the shadow class in Figure 4.3 may be mentioned as A2[A1.B].C.

This naming scheme also solves the problem of name conflicts between shadow classes. Suppose in addition to the declarations in Figure 4.3, there is another shared derived family A3, which introduces a new nested class that happens to have the name C as well:

```
class A3 shares A1 {
  class C extends B {...}
}
```

Then there will be two shadow classes in A1, both having the same name C, and there is no modular way to detect this situation. However, indirect naming in J& prevents the potential name conflict, by naming the two shadow classes differently, as A2[A1.B].C and A3[A1.B].C respectively.

The syntax for naming shadow classes looks somewhat heavy, but it is unlikely to be used frequently. We expect shadow classes to be normally used as a way to introduce new subclasses of some known, normal class in the base family. The base family would generally use objects of the shadow class through the known superclass, without explicitly mentioning the shadow class. The syntax is also not necessary in the originating family, e.g., inside the declaration of a shadow method, where the syntax coincides with the automatic desugaring of unqualified type names.

4.2.4 Open families

Shadow classes and shadow methods provide a new kind of extensibility. Families are now *open*, analogously to open classes [13]. An existing family can be extended in a modular way with new functionality, including new classes, without modifying the code of the family. This contrasts both with heterogeneous sharing and the original version of J&, where families are *locally closed worlds* that do not allow adding new nested classes without modifying existing code. In that prior work, it is possible to statically enumerate all the nested classes of a family.

This kind of extensibility is also different from that provided by sharing itself. Family sharing modularly adds new functionality to objects that belong to an existing family, but the new behavior is only accessible by viewing the objects in the shared derived family. On the other hand, shadow classes are available in the base family. For example, the shadow method `m()` declared by `A2.C` in Figure 4.3 may be invoked through a receiver object of static type `A1!.B`—definitely in the base family—if the receiver object is an instance of the shadow class `A2[A1.B].C`.

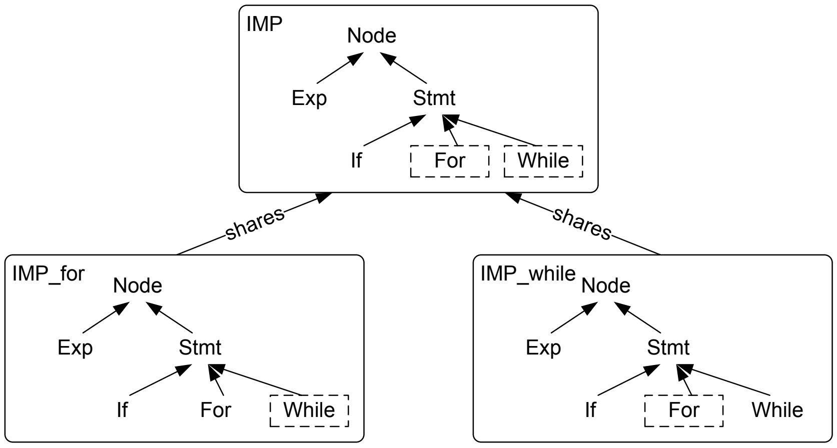

Figure 4.4: Sketch of the IMP compiler structure. Shadow classes in dashed boxes.

To show the extensibility provided by homogeneous family sharing and open families, let us consider an example of in-place translation [65] in the context of a simple compiler.

J& supports type-safe translation of data structures between shared families, in a mostly in-place fashion. When a data structure (in this case, an AST) is translated from the source family to the target family, it is possible that only a fraction of the objects in the structure needs to be explicitly translated, and the rest of them remain structurally the same. If the source family and the target family are shared, one can simply apply view changes to objects that do not need translation, avoiding the generation of many new objects, while still ensuring the entire data structure behaves consistently in the target family.

Figure 4.4 shows a sketch of the class hierarchy of a compiler for a small imperative language and its extensions. The base family `IMP` describes a simple core language without any loop construct. The two derived families `IMP_for` and `IMP_while` respectively extend the base family with for-loops and while-loops. The compiler translates a program from `IMP_for` to `IMP_while`, rewrit-

```
package IMP;
abstract class Node { ... }
...
class If extends Stmt {
  Exp cond; Stmt body;
  If(cond, body) { this.cond = cond; this.body = body; }
}

package IMP_for shares IMP;
abstract class Node {
  Node translate() {
    return this; // by default, no translation
  }
}
class If extends Stmt {
  Stmt translate() { // covariant return type
    Exp cond = this.cond.translate();
    Stmt body = this.body.translate();
    if (cond != this.cond || body != this.body)
      return new If(cond, body);
    else
      return this;
  }
}
class For extends Stmt {
  Exp init, cond, inc;
  Stmt body;
  Stmt translate() {
    ... new IMP_while[this.class].While(...); ...
  }
}
```

Figure 4.5: In-place translation for loop statements

ing for-loops to while-loops.

Figure 4.5 illustrates J& code that does in-place translation from `IMP_for` to `IMP_while`. The two derived families are both shared with the base family `IMP`, and by transitivity, they are also shared. The class `For` in `IMP_for` and the class `While` in `IMP_while` both introduce shadow classes in `IMP`, which are then inherited by the other derived family. Since there is a shadow class `While` in `IMP_for`, the translation code implemented in Figure 4.5 actually generates objects still in the `IMP_for` family. Objects of the shadow class are generated on line 29. Translation of `If` (lines 17–22) lazily generates a new `If` node only if its children have changed. After the recursive translation is done on an AST, a single view change operation applied to the root moves the entire AST to the `IMP_while` family:

```
  IMP_for!.Node source = ...;
  IMP_for!.Node temp = source.translate();
  IMP_while!.Node target = (view IMP_while!.Node)temp;
```

In this example, the use of shadow classes facilitates the translation, which is between two "sibling" families without going through the base family. In fact, it is not easy, if not impossible, to first translate to `IMP`, and then to `IMP_while`. This example is different from the in-place translation example considered before in [65], which only supports translation from a derived family to its base family.

4.3 Formal semantics and soundness

This section formalizes homogeneous family sharing in a core language called JHS. Not all the features of the J& language are modeled in JHS, in order to focus on sharing. For example, virtual types, explicit multiple inheritance and intersection types are omitted.

programs	Pr	$::= < \overline{L}, e >$
class declarations	L	$::=$ `class` $C\ ES\ \{\overline{L}\ \overline{F}\ \overline{M}\}$
superclass declarations	ES	$::=$ `extends` T $\mid$ `shares` T
field declarations	F	$::=$ `[final]` $T\ f = e$
method declarations	M	$::=$ `[shadow]` $T\ m(\overline{T}\ \overline{x})\ \{e\}$
types	T	$::= \top \mid \circ \mid T.C \mid p.$`class` $\mid P[T] \mid T!$
classes	P	$::= \top \mid \circ \mid P.C$
values	v	$::= < \ell, P! >$
access paths	p	$::= v \mid x \mid p.f$
expressions	e	$::= v \mid x \mid e.f \mid e_0.f = e_1 \mid e_0.m(\overline{e}) \mid e_1;\ e_2 \mid$ `new` $T(\overline{f} = \overline{e})$ $\mid$ `(view` $T)e$
typing environments	Γ	$::= \emptyset \mid \Gamma, x{:}T \mid \Gamma, p_1 = p_2$

Figure 4.6: Syntax of JHS

4.3.1 Syntax

Figure 4.6 shows the syntax of the JHS core language. The notation $\overline{a}$ is used for both the list $a_1, \ldots, a_n$ and the set $\{a_1, \ldots, a_n\}$, for $n \geq 0$.

A program Pr is a pair $< \overline{L}, e >$ of a set of class declarations $\overline{L}$ and an expression e (the `main` method). Each class C has a superclass declaration, which is either a normal superclass declaration `extends` T, or a shared superclass declaration `shares` T. There are two special classes: $\top$ is the superclass of all the other classes, similar to `Object` in Java; $\circ$ is the single top-level class that all other classes are nested within.

Method declarations may have an extra modifier `shadow` for declaring shadow methods.

JHS supports explicit exact types, and exactness applies to the entire type preceding "!". $T!.\overline{C}!$ is considered equivalent to $T.\overline{C}!$, and $T!$ is equivalent to T if T is already exact.

A value v, also called a reference, is a pair of a heap location ℓ and the associated view $P!$, which is a class in its exact form. Expressions are mostly standard, with the addition of a view change operation (`view` T)e.

For simplicity, we omit the `null` value from JHS, and require that all field declarations come with default initializations.

The typing environment contains aliasing information about access paths. An entry $p_1 = p_2$ means p_1 and p_2 are aliases that also have the same run-time view. As in [12], this kind of information is not needed in the static semantics, but just in the soundness proof.

4.3.2 Lookup functions

The class table CT, defined in Figure 4.7, contains the declaration for any explicit or implicit class that is *not* a shadow class. The extended class table CT' contains synthesized declarations for all the shadow classes, in addition to the declarations in CT. Both CT and CT' are assumed to be global information.

CT′-SHADOW states that if P' introduces a nested class C, that is, no superclass of P' contains a *normal* nested class named C, it also introduces a shadow class in any of its shared superclasses P—$P'.C$ is the originating class of these shadow classes. Shadow methods in $P'.C$ are treated as if they were normal methods declared in shadow classes.

Figure 4.7 also shows auxiliary functions for looking up various class members like fields and methods: $\mathsf{super}(P)$ gives the superclass declaration of P, either a normal superclass or a shared one; $\mathsf{shadowMethods}(P)$ collects all the

$\boxed{CT(P)}$

$$\frac{Pr = <\overline{L}, e>}{CT(\circ) = \texttt{class}\ \circ\ \texttt{extends}\ \top\ \{\overline{L}\}}\ (\text{CT-OUT}) \qquad \frac{CT(P) = \texttt{class}\ C'\ ES\ \{\overline{L}\ \overline{F}\ \overline{M}\} \qquad L_i = \texttt{class}\ C\ \ldots}{CT(P.C) = L_i}\ (\text{CT-EXP})$$

$$\frac{CT(P) = \texttt{class}\ C'\ \ldots\ \{\overline{L}\ \overline{F}\ \overline{M}\} \qquad \texttt{class}\ C\ \ldots \notin \overline{L} \qquad \vdash P \sqsubset^* P' \qquad CT(P'.C) = \texttt{class}\ C\ ES\ \{\ldots\}}{CT(P.C) = \texttt{class}\ C\ ES\ \{\}}\ (\text{CT-IMP})$$

$\boxed{CT'(P)}$

$$\frac{CT(P) \neq \bot}{CT'(P) = CT(P)}\ (\text{CT}'\text{-NORM}) \qquad \frac{\begin{array}{c} CT'(P) \neq \bot \qquad CT(P.C) = \bot \qquad \vdash P' \underset{\leftrightarrow}{\sqsubset}^* P \\ CT(P'.C) = \texttt{class}\ C\ ES\ \{\overline{L}\ \overline{F}\ \overline{M}\} \\ \nexists P''\ .\ CT(P''.C) \neq \bot \wedge \vdash P' \sqsubset P'' \end{array}}{CT'(P.C) = \texttt{class}\ C\ ES\ \{\mathsf{shadowMethods}(P'.C)\}}\ (\text{CT}'\text{-SHADOW})$$

$\boxed{\text{Member lookup}}$

$$\frac{CT'(P) = \texttt{class}\ C\ ES\ \{\overline{L}\ \overline{F}\ \overline{M}\}}{\begin{array}{c} \mathsf{super}(P) = ES \\ \mathsf{ownFields}(P) = \overline{F} \\ \mathsf{ownMethods}(P) = \overline{M} - \mathsf{shadowMethods}(\overline{M}) \\ \mathsf{shadowMethods}(P) = \{M' \mid \texttt{shadow}\ M' \in \overline{M}\} \end{array}} \qquad \frac{\begin{array}{c} \Gamma \vdash T \trianglelefteq S \qquad \mathsf{fields}(S) = \overline{F} \\ F_i = [\texttt{final}]\ T_f\ f = e \end{array}}{\mathsf{ftype}_{decl}(\Gamma, T, f) = T_f}$$

$$\frac{T_f^{decl} = \mathsf{ftype}_{decl}(\Gamma, T, f)}{\mathsf{ftype}(\Gamma, T, f) = T_f^{decl}\{\!\{\Gamma;\ T/\texttt{this}\}\!\}}$$

$$\mathsf{fields}(P) = \bigcup_{\vdash P \sqsubset^* P_i} \mathsf{ownFields}(P_i) \qquad \frac{\begin{array}{c} \Gamma \vdash T \trianglelefteq P \qquad \mathsf{methods}(P) = \overline{M} \\ M_i = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\} \end{array}}{\mathsf{mtype}(\Gamma, T, m) = (\overline{x}{:}\overline{T}) \to T_{n+1}}$$

$$\mathsf{methods}(P) = \bigcup_{\vdash P \sqsubset^* P_i} \mathsf{ownMethods}(P_i)$$

$$\frac{\overline{F} = [\texttt{final}]\ \overline{T}\ \overline{f} = \overline{e}}{\mathsf{fnames}(\overline{F}) = \overline{f}} \qquad \frac{\mathsf{methods}(P) = \overline{M} \qquad M_i = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\}}{\mathsf{mbody}(P, m) = M_i}$$

Figure 4.7: Lookup functions

shadow method declarations from P, with the `shadow` modifier removed; the functions $\mathsf{fields}(P)$ and $\mathsf{methods}(P)$ collect all the field and method declarations from P and its superclasses, excluding shadow methods, which are treated as declared in shadow classes; $\mathsf{fnames}(\overline{F})$ is the set of all field names in field declarations $\overline{F}$; $\mathsf{ftype}_{decl}(\Gamma, T, f)$ returns the declared type of field f, which might be a type dependent on `this`; $\mathsf{ftype}(\Gamma, T, f)$ substitutes the receiver type T for `this.class`; $\mathsf{mtype}(\Gamma, T, m)$ and $\mathsf{mbody}(P, m)$ look up the type and the declaration of a method m.

For simplicity, we assume there are no name conflicts in JHS, that is, all the

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

$$\frac{\begin{array}{c}\mathsf{super}(P.C) = \texttt{shares}\ T \\ \vdash P_1 \sqsubset^{*} P \qquad T\{\!\!\{\emptyset;\ P_1/\texttt{this}\}\!\!\} = P_2\end{array}}{\vdash P_1.C \underset{\leftrightarrow}{\sqsubset} P_2}\ (\text{SHARE-DECL}) \qquad \frac{\vdash P_1 \underset{\leftrightarrow}{\sqsubset} P_2}{\vdash P_1.C \underset{\leftrightarrow}{\sqsubset} P_2.C}\ (\text{SHARE-FB})$$

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

$$\frac{\begin{array}{c}\mathsf{super}(P.C) = \texttt{extends}\ T \\ \vdash P_1 \sqsubset^{*} P \qquad T\{\!\!\{\emptyset;\ P_1/\texttt{this}\}\!\!\} = P_2\end{array}}{\vdash P_1.C \sqsubset P_2}\ (\text{SC-DECL}) \qquad \frac{\vdash P_1 \sqsubset P_2}{\vdash P_1.C \sqsubset P_2.C}\ (\text{SC-FB}) \qquad \frac{\vdash P_1 \underset{\leftrightarrow}{\sqsubset} P_2}{\vdash P_1 \sqsubset P_2}\ (\text{SC-SHARE})$$

Figure 4.8: Sharing and subclassing

field declarations use different field names, unrelated methods have different names, and classes that do not override each other also have different names.

4.3.3 Sharing and subclassing

Subclassing relationships among classes are defined in Figure 4.8. The judgment $\vdash P_1 \underset{\leftrightarrow}{\sqsubset} P_2$ states that P_1 is a shared subclass of P_2. With homogeneous family sharing, when two classes are declared to be shared, all the nested classes are also automatically shared, according to SHARE-FB. On the other hand, $\vdash P_1 \sqsubset P_2$ states that P_1 is a subclass of P_2, either shared or not.

4.3.4 Prefix types

The meaning of a non-dependent prefix type $P[P']$ is either a subclass of P that nests P', or a shared superclass of P that nests P', as captured in the auxiliary function $\mathsf{prefix}(P, P')$, shown in Figure 4.9. JHS generalizes prefix types to include the second case, for naming shadow classes.

As in [59], we only consider prefix types $P[T]$ where the index T is exactly one level deeper in the nesting hierarchy than the bound P. However, more general prefix types can be encoded.

4.3.5 Type substitution

The rules for type substitution are shown in Figure 4.9. Type substitution $T\{\!\{\Gamma;\ T_x/x\}\!\}$ substitutes T_x for `x.class` in T in the context of Γ. The typing context Γ is used to look up field types when substituting a non-dependent class into a field-path dependent class.

For type safety, type substitutions on the right-hand side of a field assignment or on the parameters of method calls must preserve the exactness of the declared type. Therefore, only values from the family that is compatible with the receiver are assigned to fields or passed to method code. Exactness-preserving type substitution $T\{\!\{\Gamma;\ T_x/x!\}\!\}$ is also shown in Figure 4.9.

4.3.6 Static semantics

The static semantics of JHS is summarized in Figure 4.10, Figure 4.11, Figure 4.12, and Figure 4.13. Figure 4.10 shows rules for type well-formedness, type bounds, final path typing, and final path equalities. Figure 4.11 shows type sharing and subtyping rules. Figure 4.12 shows expression typing rules. Figure 4.13 defines program well-formedness.

Sharing relationships between types. The sharing judgment $\Gamma \vdash T_1 \leftrightarrow T_2$, shown in Figure 4.11, states that a value of type T_1 may become a value of T_2

$$\begin{aligned}
\mathsf{paths}(\top) &= \emptyset \\
\mathsf{paths}(\circ) &= \emptyset \\
\mathsf{paths}(T.C) &= \mathsf{paths}(T) \\
\mathsf{paths}(p.\texttt{class}) &= \{p\} \\
\mathsf{paths}(P\,\texttt{[}\,T\,\texttt{]}) &= \mathsf{paths}(T) \\
\mathsf{paths}(T!) &= \mathsf{paths}(T)
\end{aligned}$$

$$\begin{aligned}
\mathsf{prefixExact}_k(\top) &= \mathsf{false} \\
\mathsf{prefixExact}_k(\circ) &= \mathsf{true} \\
\mathsf{prefixExact}_k(T.C) &= \begin{cases} \mathsf{false} & \text{if } k = 0 \\ \mathsf{prefixExact}_{k-1}(T) & \text{otherwise} \end{cases} \\
\mathsf{prefixExact}_k(p.\texttt{class}) &= \mathsf{true} \\
\mathsf{prefixExact}_k(P\,\texttt{[}\,T\,\texttt{]}) &= \mathsf{prefixExact}_{k+1}(T) \\
\mathsf{prefixExact}_k(T!) &= \mathsf{true}
\end{aligned}$$

$$\mathsf{prefix}(P, P') = \begin{cases} P'' & \text{if } P' = P''.C \wedge \vdash P'' \sqsubseteq^* P \\ P'' & \text{if } P' = P''.C \wedge \vdash P \underset{\leftrightarrow}{\sqsubseteq}^* P'' \\ \bot & \text{otherwise} \end{cases}$$

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

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

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

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

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

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

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

$$\frac{p.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = T_p \qquad T_p \neq p'.\texttt{class} \qquad \mathsf{ftype}(\Gamma, T_p, f) = T_f}{p.f.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = T_f}$$

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

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

$$\frac{T\{\!\{\Gamma;\ T_x/x\}\!\} = T' \qquad \forall k.\ \mathsf{prefixExact}_k(T) \Rightarrow \mathsf{prefixExact}_k(T')}{T\{\!\{\Gamma;\ T_x/x!\}\!\} = T'}$$

Figure 4.9: Auxiliary definitions

through a view change, and vice versa. SH-NEST states that sharing is between families: when two classes are shared, all the corresponding nested types are also shared. SH-DECL collects sharing relationships from class declarations.

Shadow classes ensure that the two shared families are symmetric in the sharing relation, and therefore JHS does not need directional sharing relationships as in the J&$_s$ calculus, simplifying the semantics.

Subtyping. The subtyping judgment $\Gamma \vdash T \leq T'$ states that T is a subtype of T' in context Γ, and type equivalence $\Gamma \vdash T \approx T'$ is sugar for a pair of subtyping judgments.

$\boxed{\Gamma \vdash T}$

$$\frac{CT'(P) \neq \bot}{\Gamma \vdash P}\ (\text{WF-SIMP}) \qquad \frac{\Gamma \vdash_{\mathsf{final}} p : T}{\Gamma \vdash p.\texttt{class}}\ (\text{WF-FIN}) \qquad \frac{\Gamma \vdash T}{\Gamma \vdash T!}\ (\text{WF-EXACT})$$

$$\frac{\Gamma \vdash T \quad \Gamma \vdash T \trianglelefteq P \quad CT'(P.C) \neq \bot}{\Gamma \vdash T.C}\ (\text{WF-NEST}) \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash T \quad \Gamma \vdash T \trianglelefteq P' \quad \mathsf{prefix}(P, P') \neq \bot}{\Gamma \vdash P[T]}\ (\text{WF-PRE})$$

$\boxed{\Gamma \vdash T \trianglelefteq P}$

$$\Gamma \vdash P \trianglelefteq P\ (\text{BD-REFL}) \qquad \frac{\Gamma \vdash T \trianglelefteq P}{\Gamma \vdash T.C \trianglelefteq P.C}\ (\text{BD-NEST}) \qquad \frac{\Gamma \vdash P[T] \quad \Gamma \vdash T \trianglelefteq P'}{\Gamma \vdash P[T] \trianglelefteq \mathsf{prefix}(P, P')}\ (\text{BD-PRE})$$

$$\frac{\Gamma \vdash T \trianglelefteq P}{\Gamma \vdash T! \trianglelefteq P}\ (\text{BD-EXACT}) \qquad \frac{\Gamma \vdash_{\mathsf{final}} p : T \quad \Gamma \vdash T \trianglelefteq P}{\Gamma \vdash p.\texttt{class} \trianglelefteq P}\ (\text{BD-FIN})$$

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

$$\Gamma \vdash_{\mathsf{final}} < \ell, P! > \ : P!\ (\text{F-REF}) \qquad \frac{x : T \in \Gamma}{\Gamma \vdash_{\mathsf{final}} x : T}\ (\text{F-VAR}) \qquad \frac{\Gamma \vdash_{\mathsf{final}} p : T \quad T_f = \mathsf{ftype}(\Gamma, T, f)}{\Gamma \vdash_{\mathsf{final}} p.f : T_f}\ (\text{F-GET})$$

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

$$\frac{p_1 = p_2 \in \Gamma}{\Gamma \vdash p_1 = p_2}\ (\text{A-ENV}) \qquad \frac{\Gamma \vdash_{\mathsf{final}} p : T}{\Gamma \vdash p = p}\ (\text{A-REFL}) \qquad \frac{\Gamma \vdash p_2 = p_1}{\Gamma \vdash p_1 = p_2}\ (\text{A-SYM})$$

$$\frac{\Gamma \vdash p_1 = p_2 \quad \Gamma \vdash p_2 = p_3}{\Gamma \vdash p_1 = p_3}\ (\text{A-TRANS}) \qquad \frac{\Gamma \vdash p_1 = p_2 \quad \Gamma \vdash_{\mathsf{final}} p_1.f : T_f \quad \Gamma \vdash_{\mathsf{final}} p_2.f : T_f}{\Gamma \vdash p_1.f = p_2.f}\ (\text{A-FIELD})$$

Figure 4.10: Static semantics: auxiliary judgments

Most subtyping rules are similar to those in Chapter 3, but without any rule about masked types or intersection types. S-SHARE states that the subtyping relationships are preserved by a shared family, and implies that shadow classes in the base family inherit subtyping relationships from the originating family.

Expression typing. The rules for expression typing $\Gamma \vdash e : T$ (Figure 4.12) are mostly standard, with the addition of T-VIEW that states a view change expression is valid when the source and the target types are shared.

Program typing. Program typing rules are in Figure 4.13. P-OK states the rule for a program to be well-formed; L-OK, F-OK, and M-OK are the well-formedness rules for declarations of classes, fields, and methods. EXT-OK and SH-OK are the well-formedness rules for inheritance and sharing declarations.

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

$$\Gamma \vdash T \leftrightarrow T \;(\text{SH-REFL}) \qquad \frac{\Gamma \vdash T_1 \leftrightarrow T_2}{\Gamma \vdash T_2 \leftrightarrow T_1}\;(\text{SH-SYM}) \qquad \frac{\Gamma \vdash T_1 \leftrightarrow T_2 \qquad \Gamma \vdash T_2 \leftrightarrow T_3}{\Gamma \vdash T_1 \leftrightarrow T_3}\;(\text{SH-TRANS})$$

$$\frac{\Gamma \vdash T_1 \leftrightarrow T_2}{\Gamma \vdash T_1! \leftrightarrow T_2!}\;(\text{SH-EXACT}) \qquad \frac{\begin{array}{c}\Gamma \vdash T_1 \leftrightarrow T_2 \\ \Gamma \vdash T_2.C\end{array}}{\Gamma \vdash T_1.C \leftrightarrow T_2.C}\;(\text{SH-NEST}) \qquad \frac{\begin{array}{c}\Gamma \vdash T \trianglelefteq P \\ \mathsf{super}(P.C) = \mathsf{shares}\ T' \\ T'\{\!\{\Gamma;\ T/\mathtt{this}\}\!\} = T''\end{array}}{\Gamma \vdash T.C! \leftrightarrow T''!}\;(\text{SH-DECL})$$

$$\boxed{\Gamma \vdash T \leq T'}$$

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

$$\Gamma \vdash T.C! \leq T!.C \;(\text{S-EXACT}) \qquad \frac{\Gamma \vdash_{\mathsf{final}} p : T}{\Gamma \vdash p.\mathtt{class} \leq T}\quad(\text{S-FIN}) \qquad \frac{\Gamma \vdash T \qquad \Gamma \vdash T \trianglelefteq P}{\Gamma \vdash T \leq P}\;(\text{S-BOUND})$$

$$\frac{\Gamma \vdash_{\mathsf{final}} p : T!}{\Gamma \vdash p.\mathtt{class} \approx T!}\;(\text{S-FIN-EXACT}) \qquad \frac{\Gamma \vdash p_1 = p_2}{\Gamma \vdash p_1.\mathtt{class} \approx p_2.\mathtt{class}}\;(\text{S-ALIAS}) \qquad \frac{\Gamma \vdash T_1 \leq T_2 \qquad \Gamma \vdash T_2.C}{\Gamma \vdash T_1.C \leq T_2.C}\;(\text{S-NEST})$$

$$\frac{\Gamma \vdash P[T.C]}{\Gamma \vdash P[T.C] \approx T}\;(\text{S-PRE-E1}) \qquad \frac{\Gamma \vdash P[T] = T'}{\Gamma \vdash P[T!] \approx T'!}\;(\text{S-PRE-E2}) \qquad \frac{\Gamma \vdash T_1 \leftrightarrow T_2 \qquad \Gamma \vdash T_1.\overline{C} \leq T_1.\overline{C'}}{\Gamma \vdash T_2.\overline{C} \leq T_2.\overline{C'}}\;(\text{S-SHARE})$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 \leq T_2 \\ \Gamma \vdash P[T_1] \qquad \Gamma \vdash P[T_2]\end{array}}{\Gamma \vdash P[T_1] \leq P[T_2]}\;(\text{S-PRE-I}) \qquad \frac{\begin{array}{c}\vdash P_1 \sqsubset P_2 \\ \Gamma \vdash P_1[T] \qquad \Gamma \vdash P_2[T]\end{array}}{\Gamma \vdash P_1[T] \approx P_2[T]}\;(\text{S-PRE-B}) \qquad \frac{\begin{array}{c}\Gamma \vdash T \trianglelefteq P \\ \mathsf{super}(P.C) = \ldots\ T' \\ T'\{\!\{\Gamma;\ T/\mathtt{this}\}\!\} = T''\end{array}}{\Gamma \vdash T.C \leq T''}\;(\text{S-DECL})$$

Figure 4.11: Static semantics: sharing and subtyping

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

$$\frac{\Gamma \vdash_{\mathsf{final}} p : T}{\Gamma \vdash p : p.\mathtt{class}}\quad(\text{T-FIN}) \qquad \frac{\Gamma \vdash e : T \qquad \Gamma \vdash T \leq T'}{\Gamma \vdash e : T'}\;(\text{T-SUB}) \qquad \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 ; e_2 : T_2}\quad(\text{T-SEQ})$$

$$\frac{\begin{array}{c}\Gamma \vdash e : T \\ \Gamma \vdash T \leftrightarrow T'\end{array}}{\Gamma \vdash (\mathtt{view}\ T')e : T'}\;(\text{T-VIEW}) \qquad \frac{\begin{array}{c}\Gamma \vdash e : T \\ \mathsf{ftype}(\Gamma, T, f) = T_f\end{array}}{\Gamma \vdash e.f : T_f}\;(\text{T-GET}) \qquad \frac{\begin{array}{c}\Gamma \vdash T \qquad \Gamma \vdash \overline{e} : \overline{T} \\ \overline{T} = \mathsf{ftype}_{decl}(\Gamma, T, \overline{f})\{\!\{\Gamma;\ T!/\mathtt{this!}\}\!\}\end{array}}{\Gamma \vdash \mathtt{new}\ T(\overline{f} = \overline{e}) : T!}\;(\text{T-NEW})$$

$$\frac{\begin{array}{c}\Gamma \vdash e : T_f \qquad \Gamma \vdash x : T \\ \mathsf{ftype}_{decl}(\Gamma, T, f) = T_f^{decl} \\ T_f^{decl}\{\!\{\Gamma;\ T/\mathtt{this!}\}\!\} = T_f\end{array}}{\Gamma \vdash x.f = e : T_f}\quad(\text{T-SET}) \qquad \frac{\begin{array}{c}n = \#(\overline{e}) = \#(\overline{x}) \qquad \Gamma \vdash e_0 : T_0 \qquad \Gamma \vdash \overline{e} : \overline{T'} \\ \mathsf{mtype}(\Gamma, T_0, m) = (\overline{x} : \overline{T}) \to T_{n+1} \\ \forall i \in 1..n+1.\ T_i\{\!\{\Gamma;\ T_0/\mathtt{this!}\}\!\} = T'_i\end{array}}{\Gamma \vdash e_0.m(\overline{e}) : T'_{n+1}}\quad(\text{T-CALL})$$

Figure 4.12: Static semantics: expressiong typing

$$\frac{\circ \vdash \overline{L}\ \textsf{ok} \qquad \emptyset \vdash e : T \qquad \emptyset \vdash T \qquad \sqsubset^{+}\ \text{acyclic}}{\vdash < \overline{L}, e >\ \textsf{ok}} \tag{P-OK}$$

$$\frac{\begin{array}{c}\forall C'.\ CT'(P.C.C') \neq \bot \Rightarrow P.C \vdash CT'(P.C.C')\ \textsf{ok} \\ P.C \vdash \overline{F}\ \textsf{ok} \qquad P.C \vdash \textsf{ownMethods}(P.C)\ \textsf{ok} \qquad P \vdash ES\ \textsf{ok} \\ \forall P_i. \vdash P.C \sqsubset^{+} P_i \Rightarrow \vdash P.C \text{ conforms to } P_i\end{array}}{P \vdash \texttt{class}\ C\ ES\ \{\overline{L}\ \overline{F}\ \overline{M}\}\ \textsf{ok}} \tag{L-OK}$$

$$\frac{T \neq \circ \qquad \texttt{this}: P \vdash T \qquad \textsf{paths}(T) \subseteq \{\texttt{this}\} \qquad \neg\textsf{prefixExact}_0(T)}{P \vdash \texttt{extends}\ T\ \textsf{ok}} \tag{EXT-OK}$$

$$\frac{\texttt{this}: P \vdash T \qquad T\{\!\{\emptyset;\ P/\texttt{this}\}\!\} = P.C}{P \vdash \texttt{shares}\ T\ \textsf{ok}} \tag{SH-OK}$$

$$\frac{\begin{array}{c}CT'(P) = \texttt{class}\ C\ ES\ \{\overline{L}\ \overline{F}\ \overline{M}\} \\ CT'(P') = \texttt{class}\ C'\ ES'\ \{\overline{L'}\ \overline{F'}\ \overline{M'}\} \\ \forall i, j.\ \left(L_i = \texttt{class}\ D\ ES_i\ \ldots \wedge L'_j = \texttt{class}\ D\ ES'_j\ \ldots\right) \Rightarrow ES_i = ES'_j \\ \forall i, j.\ \left(M_i = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\} \wedge M'_j = T'_{n+1}\ m(\overline{T'}\ \overline{x'})\ \{e'\}\right) \Rightarrow P \vdash M_i \text{ overrides } M'_j\end{array}}{\vdash P \text{ conforms to } P'}$$

$$\frac{\begin{array}{c}M = T_{n+1}\ m(\overline{T}\ \overline{x})\ \ldots\ \{e\} \\ M' = T'_{n+1}\ m(\overline{T'}\ \overline{x'})\ \ldots\ \{e'\} \\ \#(\overline{x}) = \#(\overline{x'}) = \#(\overline{y}) \qquad \overline{y} \cap (\overline{x} \cup \overline{x'}) = \emptyset \\ \Gamma = \texttt{this}: P, \overline{y} : \overline{T}\{\overline{y}/\overline{x}\} \qquad \vdash \Gamma\ \textsf{ok} \\ \Gamma \vdash \overline{T}\{\overline{y}/\overline{x}\} \approx \overline{T'}\{\overline{y}/\overline{x'}\} \qquad \Gamma \vdash T_{n+1}\{\overline{y}/\overline{x}\} \approx T'_{n+1}\{\overline{y}/\overline{x'}\}\end{array}}{P \vdash M \text{ overrides } M'}$$

$$\frac{\texttt{this}: P \vdash T \qquad \textsf{paths}(T) \subseteq \{\texttt{this}\} \qquad \neg\textsf{prefixExact}_0(T) \qquad \texttt{this}: P \vdash e : T}{P \vdash [\texttt{final}]\ T\ f = e\ \textsf{ok}} \tag{F-OK}$$

$$\frac{\begin{array}{c}\Gamma = \texttt{this}: P, \overline{x} : \overline{T} \qquad \vdash \Gamma\ \textsf{ok} \qquad n = \#(\overline{x}) \qquad x_0 = \texttt{this} \\ \Gamma \vdash T_{n+1} \qquad \Gamma \vdash e : T_{n+1} \qquad \textsf{FV}(e) \subseteq \{x_0, \overline{x}\} \qquad \textsf{paths}(\overline{T}, T_{n+1}) \subseteq \{\texttt{this}\}\end{array}}{P \vdash T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\}\ \textsf{ok}} \tag{M-OK}$$

Figure 4.13: Program typing

4.3.7 Operational semantics

A small-step operational semantics for JHS is shown in Figures 4.14 and 4.15. A heap H is a function mapping pairs $< \ell, f >$ of memory locations and field names to values v. Heap updates are represented as $H[< \ell, f >:= v]$.

A reference set R, which contains all the references v that have been generated during evaluation, no matter whether they are reachable from e, is also part of the evaluation configuration e, H, R. The set R is only for the proof of soundness: it prevents us from losing path equalities needed in the proof.

$$\begin{array}{lll}
\text{heaps} & H & ::= \emptyset \mid H, <\ell, f> \mapsto v \\
\text{reference sets} & R & ::= \emptyset \mid R, v \\
\text{evaluation contexts} & E & ::= [\cdot] \mid E.f \mid E.f = v \\
& & \quad \mid E;\ e \mid (\texttt{view}\ TE)e \mid (\texttt{view}\ P!.\overline{C})E \\
& & \quad \mid E.m(\overline{e}) \mid v.m(\overline{v}, E, \overline{e}) \mid \texttt{new}\ TE(\overline{f} = \overline{e}) \\
& & \quad \mid \texttt{new}\ P!.\overline{C}(\overline{f} = \overline{v}, f = E, \overline{f'} = \overline{e}) \\
\text{type evaluation contexts} & TE & ::= TE.C \mid E.\texttt{class} \mid P\texttt{[}TE\texttt{]} \mid TE!
\end{array}$$

Figure 4.14: Definitions for operational semantics

$\boxed{e, H, R \longrightarrow e', H', R'}$

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

$$\frac{H(\ell, f) = v \qquad \mathsf{ftype}(\emptyset, P!, f) = P'!.\overline{C} \qquad R' = R, \mathsf{view}(v, P'!.\overline{C})}{<\ell, P!>.f, H, R \longrightarrow \mathsf{view}(v, P'!.\overline{C}), H, R'} \quad \text{(R-GET)}$$

$$\frac{H' = H[<\ell, f> := v]}{<\ell, P!>.f = v, H, R \longrightarrow v, H', R} \quad \text{(R-SET)}$$

$$\frac{\mathsf{mbody}(P, m) = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\} \qquad n = \#(\overline{v}) = \#(\overline{x})}{<\ell, P!>.m(\overline{v}), H, R \longrightarrow e\{<\ell, P!>/\texttt{this}, \overline{v}/\overline{x}\}, H, R} \quad \text{(R-CALL)}$$

$$\frac{\ell\ \texttt{fresh} \qquad H' = H[(\ell, \overline{f}) := \overline{v}] \qquad R' = R, <\ell, P.\overline{C}!>}{\texttt{new}\ P!.\overline{C}(\overline{f} = \overline{v}), H, R \longrightarrow <\ell, P.\overline{C}!>, H', R'} \quad \text{(R-ALLOC)}$$

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

$$\frac{R' = R, \mathsf{view}(v, P!.\overline{C})}{(\texttt{view}\ P!.\overline{C})v, H, R \longrightarrow \mathsf{view}(v, P!.\overline{C}), H, R'} \quad \text{(R-VIEW)}$$

Figure 4.15: Small-step operational semantics

The evaluation rules (Figure 4.15) take the form $e, H, R \longrightarrow e', H', R'$. Most of them are standard. R-GET shows that field accesses implicitly trigger view changes, which ensures that objects that reference each other always behave consistently.

Type evaluation. Types in `new` expressions and view change expressions may be dependent, and therefore need to be evaluated according to the type evalu-

$$\mathsf{view}(<\ell, P!>, P'!.\overline{C}) = <\ell, P'.\overline{C'}!>$$
$$\text{where } P = P''.\overline{C'} \text{ and } \vdash P'! \leftrightarrow P''!$$

Figure 4.16: The view function

ation contexts *TE* (Figure 4.14) and the type equivalence rules (Figure 4.11). A fully evaluated type has the form $P!.\overline{C}$, which is a simple class that has an exact prefix and is not dependent on any access path. There is always an exact prefix, because $\circ$ is exact. Dependent class $<\ell, P!>$ `.class` evaluates to $P!$. Prefix types are evaluated according to rules S-PRE-E1 and S-PRE-E2, which can be seen as normalization rules, reducing the types on the left-hand side of $\approx$ to those on the right-hand side.

View changes. As shown in Figure 4.16, the auxiliary function view defines the operational semantics for view changes. Because sharing is homogeneous, the generated run-time view $P'.\overline{C'}!$ is well defined, shares with the original view $P''.\overline{C'}!$, and is a subtype of the target type $P'!.\overline{C}$. Therefore, as long as the source type and target type are shared, a view change expression is well-formed.

4.3.8 Soundness

We prove the following soundness theorem of the JHS core language, using the standard technique of proving progress and subject reduction [87].

Theorem 4.3.1 *(Soundness) If* $\vdash <\overline{L}, e>$ ok*, and* $\vdash e : T$*, and* $e, \emptyset, \emptyset \rightarrow^* e', H, R$*, then either* $e' = v$ *and* $\lfloor H, R \rfloor \vdash v : T$*, or* $\exists e'', H', R'$*, such that* $e', H, R \longrightarrow e'', H', R'$*.*

$$\frac{\begin{array}{c}<\ell, P!>\in R \qquad \vdash \mathsf{ftype}(\emptyset, P, f) \trianglelefteq P_f \\ <\ell', P'!> = \mathsf{view}(H(\ell, f), P_f) \qquad \texttt{final}\ T_f\ f = e \in \mathsf{fields}(P)\end{array}}{<\ell, P!>.f = <\ell', P'!>\in \lfloor H, R \rfloor}$$

Figure 4.17: Runtime typing environments

$$\frac{\begin{array}{c}\mathsf{FV}(e) = \emptyset \qquad \mathsf{refs}(e) \subseteq R \\ <l, P!>, <l, P'!>\in R \Rightarrow \vdash P! \leftrightarrow P'! \\ H(\ell, f) = v \Rightarrow \exists P\ .\ <\ell, P!>\in R \wedge \vdash v : \mathsf{ftype}(\emptyset, P!, f)\end{array}}{\vdash e, H, R}$$

Figure 4.18: Runtime configuration well-formedness

The proof requires several preliminary definitions and lemmas, especially for proving subject reduction.

Runtime typing. In the soundness proof, expressions are typed using a typing environment $\lfloor H, R \rfloor$ constructed from the heap H and the reference set R, which contains aliasing information for fields. Figure 4.17 shows the definition of $\lfloor H, R \rfloor$.

A run-time configuration is well-formed, represented as $\vdash e, H, R$, shown in Figure 4.18, if e has no free variable, all the references in e are included in R, references with the same location in R have shared views, and the type of the value stored in a field is consistent with at least one view of the containing object.

Extensions of typing environments A typing environment Γ' is an *extension* of Γ, if $\Gamma \subseteq \Gamma'$. Extensions preserve almost all the typing judgments, as described in Lemma 4.3.2.

Lemma 4.3.2 *If Γ_2 is an extension of Γ_1, then the following statements are true:*

- *If $\Gamma_1 \vdash T$, then $\Gamma_2 \vdash T$.*

- *If* $\Gamma_1 \vdash T \trianglelefteq P$, *then* $\Gamma_2 \vdash T \trianglelefteq P$.
- *If* $\mathsf{ftype}(\Gamma_1, T, f) = T_f$, *then* $\mathsf{ftype}(\Gamma_2, T, f) = T_f$.
- *If* $\mathsf{mtype}(\Gamma_1, T, m) = (\overline{x}\!:\!\overline{T}) \to T_{n+1}$, *then* $\mathsf{mtype}(\Gamma_2, T, m) = (\overline{x}\!:\!\overline{T}) \to T_{n+1}$.
- *If* $\Gamma_1 \vdash p = p'$, *then* $\Gamma_2 \vdash p = p'$.
- *If* $T\{\!\{\Gamma_1;\ T_x/\mathtt{x}\}\!\} = T'$, *then* $T\{\!\{\Gamma_2;\ T_x/\mathtt{x}\}\!\} = T'$.
- *If* $\Gamma_1 \vdash T_1 \leq T_2$, *then* $\Gamma_2 \vdash T_1 \leq T_2$.
- *If* $\Gamma_1 \vdash v\!:\!T$, *then* $\Gamma_2 \vdash v\!:\!T$.

PROOF: The proof is by induction on the derivation of these judgments. □

Substitutions. This section describes several preliminary lemmas about both type and value substitutions.

The typing of field accesses and that of method calls both depend on the typing of their receivers, although the static type of the receiver might not be as precise of the dynamic type of the receiver object. Therefore, we need Lemma 4.3.3, which states that type substitution is covariant.

Lemma 4.3.3 *If* $\Gamma \vdash T_1 \leq T_2$, *then for any type* T, *we have* $\Gamma \vdash T\{\!\{\Gamma;\ T_1/x\}\!\} \leq T\{\!\{\Gamma;\ T_2/x\}\!\}$.

PROOF: The proof is by structural induction on T. Note that there is no arrow type in JHS. □

As a corollary of Lemma 4.3.3, we can prove Lemma 4.3.4, since the ftype function basically just involves a type substitution.

Lemma 4.3.4 *If* $\Gamma \vdash T_1 \leq T_2$*, and* $\textsf{ftype}(\Gamma, T_1, f) = T_f$*, and* $\textsf{ftype}(\Gamma, T_2, f) = T'_f$*, then* $\Gamma \vdash T_f \leq T'_f$*.*

JHS supports dependent types, and therefore value substitution may also apply to types. The following lemma states that value substitution preserves subtyping.

Lemma 4.3.5 *If* $\Gamma, x\!:\!T_x \vdash T_1 \leq T_2$*, and* $\Gamma \vdash v\!:\!T_x$*, then* $\Gamma\{v/x\} \vdash T_1\{v/x\}\!:\!T_2\{v/x\}$*.*

PROOF: The proof is by induction on the derivation of $\Gamma, x\!:\!T_x \vdash T_1 \leq T_2$. □

Lemma 4.3.6 states that sharing is preserved by value substitutions.

Lemma 4.3.6 *If* $\Gamma, x\!:\!T_x \vdash T_1 \leftrightarrow T_2$*, and* $\Gamma \vdash v\!:\!T_x$*, then* $\Gamma\{v/x\} \vdash T_1\{v/x\} \leftrightarrow T_2\{v/x\}$*.*

PROOF: The proof is by induction on the derivation of $\Gamma, x\!:\!T_x \vdash T_1 \leftrightarrow T_2$. □

Now we can proof the value substitution lemma:

Lemma 4.3.7 *(Value substitution)* *If* $\Gamma, x : T_x \vdash e : T$*, and* $\Gamma \vdash v : T_x$*, then* $\Gamma\{v/x\} \vdash e\{v/x\}\!:\!T\{v/x\}$*.*

PROOF: The proof is by induction on the derivation of $\Gamma, x : T_x \vdash e : T$, using Lemma 4.3.5 and Lemma 4.3.6. □

The following lemma helps make the value substitution lemma applicable to the case of proving preservation for method calls, since T-CALL uses exactness-preserving substitution.

Lemma 4.3.8 *If* $\Gamma, x\!:\!T_x \vdash T$*, and* $\Gamma \vdash v\!:\!T_x$*, then* $\Gamma\{v/x\} \vdash T\{v/x\} \leq T\{\!\{\Gamma;\ T_x/x!\}\!\}$.

PROOF: Note that $T\{v/x\}$ is actually $T\{\!\{\Gamma;\ v.\texttt{class}/x!\}\!\}$, and $\Gamma \vdash v\!:\!T_x$ implies that $\Gamma \vdash v.\texttt{class} \leq T_x$. Then we can apply Lemma 4.3.3 to get the proof. □

Sharing of field types. Unlike the J&$_s$ calculus, the homogeneous sharing JHS core language ensure that field types are always shared for all the shared views of the containing object. Therefore, there is no need to duplicate fields or to use masked types. This property is proved in Lemma 4.3.9.

Lemma 4.3.9 *If* $\vdash P! \leftrightarrow P'!$*, and* $\mathsf{ftype}(\emptyset, P!, f) = T_f$*, and* $\mathsf{ftype}(\emptyset, P'!, f) = T'_f$*, then* $\vdash T_f \leftrightarrow T'_f$.

PROOF: It is easy to see that both $\mathsf{ftype}(\emptyset, P!, f)$ and $\mathsf{ftype}(\emptyset, P'!, f)$ are obtained from the same declared field type $T_f^{decl} = \mathsf{ftype}_{decl}(\emptyset, P!, f)$.

If T_f^{decl} is not dependent, then the proof is trivial. If T_f^{decl} is a dependent type without any prefix types in it, that is, $\texttt{this}.\texttt{class}.\overline{C}$, the proof is also not hard, by SH-NEST. The interesting case is when T_f^{decl} is a prefix type, which can be proved using Lemma 4.3.10. □

Lemma 4.3.10 *If* $\vdash P_1!.\overline{C_1} \leftrightarrow P_2!.\overline{C_2}$*, and* $\vdash P\,[\,P_1!.\overline{C_1}\,]$*, and* $\vdash P\,[\,P_2!.\overline{C_2}\,]$*, then* $\vdash P\,[\,P_1!.\overline{C_1}\,] \leftrightarrow P\,[\,P_2!.\overline{C_2}\,]$*.*

PROOF: By induction on the derivation of $\vdash P_1!.\overline{C_1} \leftrightarrow P_2!.\overline{C_2}$. There are several cases:

- SH-REFL

  Trivial.

- SH-SYM

  Proved by directly using the induction hypothesis.

- SH-TRANS

  Proved by directly using the induction hypothesis.

- SH-EXACT

  Then both prefix types are evaluated using S-PRE-E2, $P_1!.\overline{C_1} = P_1!$, and $P_2!.\overline{C_2} = P_2!$. By SH-EXACT, we have $\vdash T_1 \leftrightarrow T_2$, and $T_1! = P_1!$ and $T_2! = P_2!$, i.e., T_1 and T_2 are just P_1 and P_2 with exactness inserted somewhere. Then by the induction hypothesis, $\vdash P\,[\,T_1\,] \leftrightarrow P\,[\,T_2\,]$. By S-PRE-E2, we know that $P\,[\,P_1!\,]$ evaluates to $P\,[\,T_1\,]\,!$, and $P\,[\,P_2!\,]$ evaluates to $P\,[\,T_2\,]\,!$. Therefore, by SH-EXACT, the proof follows.

- SH-NEST

  Then both $P\,[\,P_1!.\overline{C_1}\,]$ and $P\,[\,P_2!.\overline{C_2}\,]$ can be evaluated according to S-PRE-E1, since they are not exact. By SH-NEST, $P_1!.\overline{C_1} = T_1.C$, $P_2!.\overline{C_2} = T_2.C$, and $\vdash T_1 \leftrightarrow T_2$. By S-PRE-E1, $P\,[\,P_1!.\overline{C_1}\,]$ evaluates to T_1, and $P\,[\,P_2!.\overline{C_2}\,]$ evaluates to T_2. Then the proof follows.

- SH-DECL

  Then there exists P' and C, such that $P_1!.\overline{C_1} = P_1! = P'.C!$, and $P_2!.\overline{C_2} = P_2!$, and $CT'(P'.C) = \ldots$ `shares` $T' \ldots$, and $T'\{\!\{\emptyset;\ P'/\mathtt{this}\}\!\} = P_2$. By SH-OK in Figure 4.13, $T'\{\!\{\emptyset;\ P'/\mathtt{this}\}\!\} = P'.C'$. Now we can see that $P_2 = P'.C'$. Therefore $P[P_1!] = P'! = P[P_2!]$. Then the proof follows by SH-REFL.

□

Progress. We first prove the progress lemma (Lemma 4.3.11), which is the easier than proving subject reduction.

Lemma 4.3.11 *(Progress) If $\vdash e, H, R$, and $\lfloor H, R \rfloor \vdash e : T$, then either $e = v$ or $\exists e', H', R'$, such that $e, H, R \longrightarrow e', H', R'$.*

PROOF: The proof is by structural induction on e. There are the following cases:

- $e = v$

  Trivial.

- $e = x$

  Vacuously true, since e has no free variable.

- $e = e_0.f$

  There are two cases:

  - $e_0 \neq v$

    Then by the induction hypothesis, $\exists e_0', H', R'$. $e_0, H, R \longrightarrow e_0', H', R'$. Then R-CONG applies.

- $e_0 =< \ell, P! >$

  Then by R-GET, $e' = \mathsf{view}(H(\ell, f), \mathsf{ftype}(\emptyset, P!, f))$, and $H' = H$, and $R' = R, \mathsf{view}(H(\ell, f), \mathsf{ftype}(\emptyset, P!, f))$.

- $e = (e_0.f = e_1)$

  There are again two cases:

  - $e_0 \neq v$ or $e_1 \neq v$

    Then the induction hypothesis and R-CONG applies.

  - $e_0 = v_0$ and $e_1 = v_1$

    The proof is similar to that for the second case with $e = e_0.f$, using R-SET instead of R-GET.

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

  There are two cases:

  - $e_i \neq v$ for some i $(0 \leq i \leq n)$

    The proof follows from the induction hypothesis and R-CONG.

  - $e_i = v_i$ $(0 \leq i \leq n)$

    Suppose $v_0 =< \ell, P! >$. By T-CALL, we know that $\mathsf{mtype}(P, m)$ is well-defined. Then $\mathsf{mbody}(P, m)$ is also well-defined according to their definitions. Therefore R-CALL applies.

- $e = e_1;\ e_2$

  There are two cases:

  - $e_1 \neq v$

    The proof follows from the induction hypothesis and R-CONG.

- $e_1 = v$

  Then R-SEQ applies.

- $e = \texttt{new}\ T(\overline{f} = \overline{e})$

  There are three cases:

  - $T \neq P!.\overline{C}$

    Then we should continue evaluating the type T, as described in Section 4.3.7.

  - $T = P!.\overline{C}$ and $e_i \neq v$ for some i $(1 \leq i \leq n)$

    The proof follows from the induction hypothesis and R-CONG.

  - $T = P!.\overline{C}$ and $e_i = v_i$ $(1 \leq i \leq n)$

    Then R-ALLOW applies.

- $e = (\texttt{view}\ T)e_0$

  There are again three cases:

  - $T \neq P!.\overline{C}$

    Then we need to continue evaluating the type T.

  - $T = P!.\overline{C}$ and $e_0 \neq v$

    The proof follows from the induction hypothesis and R-CONG.

  - $T = P!.\overline{C}$ and $e_0 = v$

    Then R-VIEW applies.

□

Subject reduction. Before proving that typing is preserved by evaluating an expression, we first prove that evaluation of a final access path preserves the typing.

Lemma 4.3.12 *If* $\vdash e, H, R$, *and* $\lfloor H, R \rfloor \vdash_{\mathsf{final}} p : T$, *and* $p, H, R \longrightarrow p', H', R'$, *then* $\vdash p', H', R'$, *and* $\lfloor H', R' \rfloor \vdash_{\mathsf{final}} p' : T'$, *and* $\lfloor H', R' \rfloor \vdash T' \leq T$.

PROOF: The proof is by induction on the derivation of $\lfloor H, R \rfloor \vdash_{\mathsf{final}} p : T$. Since the final access path p can take a step, it must be of the form $p_0.f$. There are then two cases:

- $p_0 \neq v$

  Then R-CONG applies, and $p_0, H, R \longrightarrow p'_0, H', R'$. By F-GET, $\lfloor H, R \rfloor \vdash_{\mathsf{final}} p_0 : T_0$, and $T = \mathsf{ftype}(\lfloor H, R \rfloor, T_0, f)$. Then by the induction hypothesis, $\lfloor H', R' \rfloor \vdash_{\mathsf{final}} p'_0 : T'_0$, and $\lfloor H', R' \rfloor \vdash T'_0 \leq T_0$. Then by F-GET, $\lfloor H', R' \rfloor \vdash_{\mathsf{final}} p'_0.f : \mathsf{ftype}(\lfloor H', R' \rfloor, T'_0, f)$, that is, $T' = \mathsf{ftype}(\lfloor H', R' \rfloor, T'_0, f)$. Then by Lemma 4.3.4, $\lfloor H', R' \rfloor \vdash T' \leq T$.

- $p_0 =< \ell, P! >$

  Then R-GET applies. Let $H(\ell, f) = v$, and $\mathsf{ftype}(\emptyset, P!, f) = P'!.\overline{C}$, and $< \ell', P''! >= \mathsf{view}(v, P'!.\overline{C})$. Then $p' =< \ell', P''! >$. By F-REF, $\lfloor H, R \rfloor \vdash_{\mathsf{final}} p_0 : P!$. By F-GET, $T = \mathsf{ftype}(\lfloor H, R \rfloor, P!, f)$. By S-FIN, $\lfloor H, R \rfloor \vdash p.\texttt{class} \leq T$, and by the definition of run-time typing environment, $\lfloor H, R \rfloor \vdash p = p'$. Thus by S-ALIAS, $\lfloor H, R \rfloor \vdash p'.\texttt{class} \leq T$. Since $H' = H$ and $R' = R, p'$, $\lfloor H', R' \rfloor$ is an extension of $\lfloor H, R \rfloor$. By Lemma 4.3.2, $\lfloor H', R' \rfloor \vdash p'.\texttt{class} \leq T$. By F-REF, $\lfloor H', R' \rfloor \vdash_{\mathsf{final}} p' : P''!$, that is, $T' = P''!$. By S-FIN-EXACT, $\lfloor H', R' \rfloor \vdash p'.\texttt{class} \approx P''!$. Therefore $\lfloor H', R' \rfloor \vdash P''! \leq T$ by S-TRANS, that is, $\lfloor H', R' \rfloor \vdash T' \leq T$.

□

Now let us prove subject reduction (Lemma 4.3.13).

Lemma 4.3.13 *(Subject reduction) If* $\vdash e, H, R$*, and* $\lfloor H, R \rfloor \vdash e : T$*, and* $e, H, R \longrightarrow e', H', R'$*, then* $\vdash e', H', R'$ *and* $\lfloor H', R' \rfloor \vdash e' : T$*.*

PROOF: It is not hard to prove that the resulting run-time configuration e', H', R' is still well-formed, by inspecting the definition of configuration well-formedness, and how the configuration can change for each of the operational semantic rule.

Now let us prove $\lfloor H', R' \rfloor \vdash e' : T$, by induction on the derivation of $\lfloor H, R \rfloor \vdash e : T$.

First, if the derivation ends with T-SUB, we can use the induction hypothesis and the extension lemma to do the proof. Therefore, let us now only consider the case without T-SUB as the last rule in the derivation. We also consider different cases for e. (Note that most of the congruence cases are very easy to prove, just using the induction hypothesis and the appropriate typing rules).

- $e = v$

  Vacuously true, because there is no step to take.

- $e = x$

  Vacuously true, because $\lfloor H, R \rfloor$ does not have variable bindings, and therefore e contains no free variables.

- $e = e_0.f$

  There are several cases for e_0:

  - $e_0 =< \ell, P! >$

    There are again two cases for the rule used for typing e, depending on whether f is a final field or not.

    * T-FIN

      Then by R-GET, $e, H, R \longrightarrow v, H, R'$, where $H(\ell, f) = v'$, and $v = \mathsf{view}(v', \mathsf{ftype}(\emptyset, P!, f))$, and $R' = R, v$.

      We can still use T-FIN to type v. According to Lemma 4.3.12, the proof follows.

    * T-GET

      Then operationally, it is still R-GET, the same as in the previous case. We still have $e, H, R \longrightarrow v, H, R'$, where $H(\ell, f) = v'$, and $v = \mathsf{view}(v', \mathsf{ftype}(\emptyset, P!, f))$, and $R' = R, v$.

      Although the typing of $< \ell, P! >$ might not be as precise as $P!$, by Lemma 4.3.4, we only need to consider $P!$ as the receiver type.

      Let $\mathsf{ftype}(\emptyset, P!, f) = T_f$. We need to prove $\lfloor H, R' \rfloor \vdash v : T_f$.

      By the definition of $\vdash e, H, R$, we have $H(\ell, f) = v' \Rightarrow \exists < \ell, P''! > \in R \wedge \vdash v : \mathsf{ftype}(\emptyset, P''!, f)$. Since $< \ell, P! >, < \ell, P''! > \in R$, we have $\vdash P! \leftrightarrow P''!$. Then by Lemma 4.3.9, $\vdash \mathsf{ftype}(\emptyset, P!, f) \leftrightarrow \mathsf{ftype}(\emptyset, P''!, f)$. Then by the definition of the auxiliary function view, $\vdash \mathsf{view}(v', T_f) : T_f$, and therefore $\lfloor H, R' \rfloor \vdash v : T_f$.

  - $e_0 \neq v$

    Then R-CONG applies. There are two cases for deriving $\lfloor H, R \rfloor \vdash e_0.f : T$.

  - T-FIN

    The proof goes similarly to the corresponding case above, using Lemma 4.3.12.

  - T-GET

    Then $\lfloor H, R \rfloor \vdash e_0 : T_0$, and $\mathsf{ftype}(\lfloor H, R \rfloor, T_0, f) = T$. By the induction hypothesis, $\lfloor H', R' \rfloor \vdash e'_0 : T_0$. Also, $\lfloor H', R' \rfloor$ is an extension of $\lfloor H, R \rfloor$. Therefore, $\mathsf{ftype}(\lfloor H', R' \rfloor, T_0, f) = T$. Then $\lfloor H', R' \rfloor \vdash e'_0.f : T$.

- $e = (e_0.f = e_1)$

  There are several cases for e_0.

  - $e_0 =< \ell, P! >$

    Again, there are two cases for e_1:

    - $e_1 = v$

      Then R-SET applies. By T-SET, $\lfloor H, R \rfloor \vdash v : T$, where $T = \mathsf{ftype}(\lfloor H, R \rfloor, P!, f)$. Here, we still only consider $P!$ with the same reason mentioned before for the case of $e =< \ell, P! > .f$. Since $\lfloor H', R \rfloor$ is an extension of $\lfloor H, R \rfloor$, we still have $\lfloor H', R \rfloor \vdash v : T$.

    - $e_1 \neq v$

      This is the congruence case.

  - $e_0 \neq v$

    This is the congruence case.

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

  There are two cases:

  - $e_i \neq v$ for any i $(0 \leq i \leq n)$

    This is the congruence case.

- $e_0 =< \ell, P! >$, and $\overline{e} = \overline{v}$

  Then R-CALL applies, and $H' = H$, $R' = R$, and $e' = e_m\{< \ell, P! > /\texttt{this}, \overline{v}/\overline{x}\}$, where $\mathsf{mbody}(P, m) = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e_m\}$. We also have $\lfloor H, R \rfloor \vdash e : T'_{n+1}$, where $T'_i = T_i\{\!\{\lfloor H, R \rfloor;\ P!/\texttt{this}\}\!\}$ $(i = 1, \ldots, n+1)$. According to M-OK, $\texttt{this} : P', \overline{x} : \overline{T} \vdash e_m : T_{n+1}$, where $\vdash P! \leq P'$. Then the proof follows from Lemma 4.3.7 and Lemma 4.3.8.

- $e = e_1;\ e_2$

  There are two cases:

  - $e_1 \neq v$

    This is the congruence case.

  - $e_1 = v$

    Then R-SEQ applies. We also have $e' = e_2$, $H' = H$, and $R' = R$. The proof follows easily.

- $e = newP!.\overline{C}(\overline{f} = \overline{e})$

  There are two cases:

  - $e_i \neq v$ for any i $(1 \leq i \leq n)$

    This is the congruence case.

  - $\overline{e} = \overline{v}$

    Then R-ALLOC applies, and $e' =< \ell, P.\overline{C}! >$. By T-NEW, $T = P.\overline{C}!$. Then by T-FIN, $\lfloor H', R' \rfloor \vdash < \ell, P.\overline{C}! >: P.\overline{C}!$.

- $e = (\texttt{view}\ P!.\overline{C})e_0$

  By T-VIEW, $T = P!.\overline{C}$. There are two cases for e_0:

- $e_0 \neq v$

  This is the congruence case.

- $e_0 = < \ell, P_0! >$

  Then R-VIEW applies. We have $e' = \mathsf{view}(< \ell, P_0! >, P!.\overline{C})$. By T-VIEW, there exists T_ℓ such that $\lfloor H, R \rfloor \vdash < \ell, P_0! > : T_\ell$ and $\lfloor H, R \rfloor \vdash T_\ell \leftrightarrow P!.\overline{C}$. Now it is easy to see that T_ℓ has the form of $P'!.\overline{C}$, otherwise it cannot be shared with $P!.\overline{C}$. Therefore $\lfloor H, R \rfloor \vdash P_0! \leq P'!.\overline{C}$, and $\vdash P! \leftrightarrow P'!$. Since $\lfloor H', R' \rfloor$ is an extension of $\lfloor H, R \rfloor$, $\lfloor H', R' \rfloor \vdash P_0! \leq P'!.\overline{C}$. By the definition of the view function, $e' = < \ell, P.\overline{C'}! >$, where $P_0 = P'.\overline{C'}$. By S-SHARE, $\lfloor H', R' \rfloor \vdash P!.\overline{C'} \leq P!.\overline{C}$. Therefore $\lfloor H', R' \rfloor \vdash P.\overline{C'}! \leq T$, which implies that $\lfloor H', R' \rfloor \vdash e' : T$.

□

Soundness. Finally, the soundness theorem (Theorem 4.3.1) can be proved. It follows directly from Lemma 4.3.11 and Lemma 4.3.13.

4.4 Experience

This section presents our experience with implementing in-place translation for a compiler for the λ-calculus. This is not a large example, but uses the language features in a sophisticated way—combining multiple levels of sharing and family-level intersection. The example compiler is inspired by the Polyglot framework, and it encapsulates most of the interesting issues that arise in making Polyglot extensible, while demonstrating the advantages of homogeneous

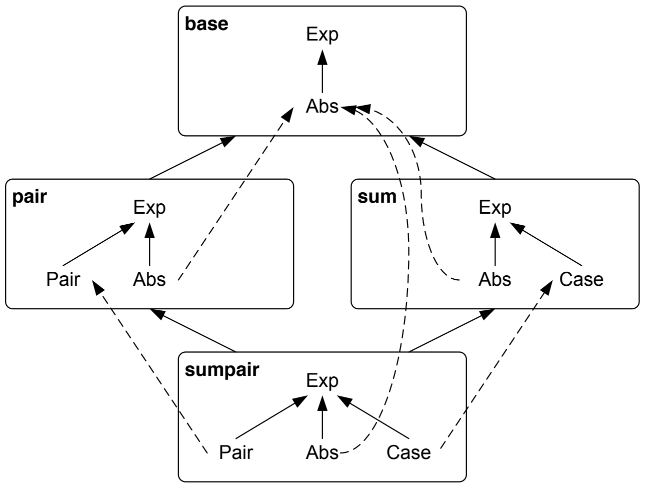

Figure 4.19: Compiler structure with heterogeneous sharing. Translator and some AST nodes not shown.

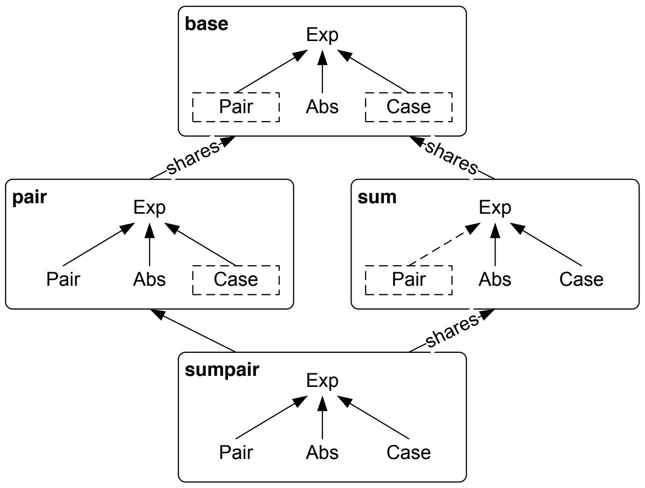

Figure 4.20: Compiler structure with homogeneous sharing. Translator and some AST nodes not shown.

family sharing. Therefore, lessons learned from the experience with it will prove useful for dealing with larger software systems such as Polyglot.

Implementing translations for sums and pairs. The compiler example translates the λ-calculus extended with sums and pairs to the simple lambda calculus. A way to accomplish this goal as an in-place translation was first demonstrated for heterogeneous sharing [65], so this example offers a way to compare the expressive power of homogeneous and heterogeneous sharing.

The compiler is implemented as four families of classes. The `base` family describes simple λ-calculus as the target language. Two derived families, `sum` and `pair`, share AST classes with the base family but extend it with sums and pairs respectively. They also implement in-place translation to the simple λ-calculus. The last derived family, `sumpair`, composes the `sum` and `pair` families, leading to a compiler that supports both sums and pairs.

Comparing the two kinds of sharing. Figure 4.19 and Figure 4.20 show the structures of the compiler when implemented with heterogeneous sharing and with homogeneous sharing.

The J&$_s$ version has about 250 lines. Sharing is declared between individual pairs of classes, represented with dashed arrows that go across family boundaries. The class hierarchy is more complex with these interfamily relationships, and several class declarations have to be written just to declare sharing, making the code less scalable. Among the 23 class declarations, 12 of them are just for sharing declarations. Moreover, the existence of new subclasses (`Pair` and `Case`) in the `pair` and `sum` families means the `Exp` type cannot be shared. Therefore, all subexpression fields with type `Exp` must be masked in sharing relationships. There are in total 17 masked types in the J&$_s$ code, and 3 sharing constraints. The extra syntax increases the annotation burden for the programmer.

The J& version has about 200 lines. There are only three sharing declarations, shown in Figure 4.20, and they are all between families. None of the 12 class declarations in the J&$_s$ version that were needed for declaring sharing are necessary, making the code simpler and more scalable. For example, the declaration of `sumpair` is reduced to just one line:

```
  class sumpair extends pair shares sum { }
```

All the masked types and sharing constraints from the J&$_s$ version are also removed.

Although it is only declared explicitly to share with `sum`, `sumpair` also shares with `pair` and `base`, because of transitivity. It is equivalent to declare `sumpair` to share with `pair`, or even with `base` in a more “symmetric” declaration:

```
  class sumpair extends pair & sum shares base { }
```

The comparison shows that homogeneous sharing is simpler and more scalable, and has a lower annotation burden. The annotation burden is low even in an absolute sense given what is achieved. Therefore, homogeneous sharing seems more likely to be adopted by ordinary programmers.

CHAPTER 5
EFFICIENT IMPLEMENTATION OF FAMILY EXTENSIBILITY AND CLASS SHARING

Besides improved expressiveness, provable type-safety, and accessible syntax, a good implementation is also important for the success of a new language design. First of all, we would like the target code generated by the compiler to have a good performance. Therefore, programmers are not paying too much run-time overhead for the benefits of the new language. Another less obvious, but also important issue is the performance of the compiler itself. There is a generally higher tolerance for the running time of a compiler, but nevertheless, it should have an acceptable performance. Moreover, in order to compile large-scale software systems, the compiler should be *scalable*: the compilation time is proportional to the size of the source code.

All the implementations presented in this chapter are source-to-source translation to Java, and are implemented in Java using the Polyglot compiler framework [57]. When the target language is Java, the primary challenge for improving run-time performance is the simulation of multiple inheritance—either implicit as the result of family inheritance, or explicit as introduced by the J& language—with single inheritance in Java. The compiler and the runtime system have to explicitly route method calls to their correct implementations, and field accesses to correct containing objects. For example, with declarations in the following J& code, the class `A2.B2` inherits from both `A1.B2` and `A2.B1`, either of which could be the target of a method dispatch: calling `m1` on an instance of `A2.B2` would dispatch to the implementation in `A1.B2`, and calling `m2` on the same object would dispatch to the implementation in `A2.B1`. The dispatch cannot directly use mechanisms in Java; either compiler generated code or the runtime system has to figure out where to route a call.

```
class A1 {
  class B1 { }
  class B2 extends B1 {
    void m1() { ... }
  }
}
class A2 extends A1 {
  class B1 {
    void m2() { ... }
  }
}
```

Operations like method calls and field accesses are fairly common in object-oriented languages, and highly optimized in most modern compilers and runtime systems. Therefore, our implementations need to do a good job on them as well in order to achieve a good performance.

On the other hand, the main challenge for making the J& compiler scalable is how to deal with implicit classes, which are inherited by a derived family, and which do not have any source code. Extensible software systems written in the J& language usually have a large number of implicit classes. Therefore, the compiler, especially the code generation phase, should do little work on any implicit class, in order to be scalable.

Previous work [56, 58] presents two implementations for family inheritance, which are summarized in Section 5.1. The first implementation in [56] produces target code that has acceptable performance, but the compilation is not scalable—for an extensible software system like the Polyglot compiler framework, which has many implicit classes in the extensions, the time and space required by the compiler becomes unacceptable. The second implementation in [58] supports scalable compilation, but uses complicated run-time data structures for method dispatching, field accesses, etc., and therefore has poor run-time performance.

This chapter describes a new kind of implementation for family inheritance, which is based on the Java classloader. A custom classloader is included in the run-time system to synthesize code for implicit classes. This implementation achieves the best run-time performance and the best compiler scalability among all the existing J& implementations. Extensions for supporting heterogeneous and homogeneous sharing are also presented in this chapter.

One of the benefits of a classloader-based implementation, compared to building a new virtual machine from scratch (e.g., as in [**?**]), is that the classloader can be easily used with most existing Java VMs, which are ubiquitously deployed nowadays. The performance results in Section 5.4 show that the classloader-based implementation has low run-time overhead.

5.1 Background on implementing family inheritance

This section reviews the two implementations presented in previous work [56, 58]. The first compiler appears in [56], and translates Jx to Java. This section presents the extended version that translates the original J& language (without family sharing) to Java, the same as the second compiler [58]. The two compilers share parsing and type-checking code. They only differ in their translation strategies, especially for implicit classes.

The first translation is *static*, generating code for all the J& classes, either explicit or implicit. For each J& class, the compiler collects all the fields, finds the target for dispatching each method, and includes all the information in the target Java code. The run-time performance is acceptable, but an excessive amount of code is produced for software with many implicit classes, making the compiler slow and unscalable. Although the static translation is not scalable, it does not duplicate code; each method implementation in the source code is type-checked only once and translated into one copy of target code.

The second translation generates no code for implicit classes; the runtime system uses data structures like hashtables to store object fields and to record how methods should be dispatched. Finding the right target to dispatch a method takes some time at run time, and more importantly, method calls and field accesses involve extra operations like hashtable lookups that are expensive. Thus the run-time overhead of this dynamic strategy is very high. The main advantage is that the translation is scalable, generating code that is proportional to the size of the source code, and the compiler handles large software systems more gracefully, especially when there are a lot of implicit classes.

5.1.1 Static translation

The compiler of the original J& language (without sharing) is a 27-kLOC (lines of code, excluding blank lines, comments, and automatically generated parser code) extension of the Polyglot base compiler. As most of the code for dispatch-

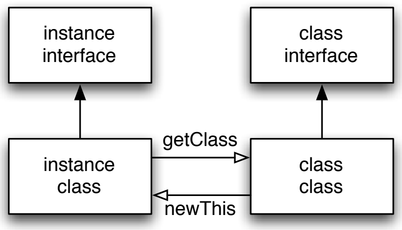

Figure 5.1: Static class translation

ing methods, accessing fields, and checking type information at run time is generated statically, the runtime support is minimal, and has only 250 LOC.

Translating classes. As shown in Figure 5.1, each J& class is represented by four classes: an *instance class*, an *instance interface*, a *class class*, and a *class interface*. The instance interface is traditionally called the *method interface*, because it contains the translated signatures of all the instance methods and field getters and setters. The class interface is also called the *static interface*, which contains code for run-time type inspection, as well as the translated signatures for static methods.

All these classes and interfaces are generated for every J& class, including implicit classes. This makes the translation purely static, and not scalable.

References to a J& class or interface T are translated to references to T's instance interface. Inheritance between instance interfaces captures multiple inheritance relationships between their J& classes. Dependent types, 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, which implements the instance interface. The instance class contains all the instance fields and their accessors. The instance class contains no real implementation for any method, which is in the class class.

Each J& class T has a singleton class class object at run time. The instance class contains a reference to its class class singleton, and a `getClass` method for getting the singleton. Static methods of T are translated to instance methods of the class class. Instance methods of T are first converted to static methods by adding an additional `self` parameter that contains the receiver object, and then handled just as static methods. The class class also provides functions for accessing run-time type information for implementing `instanceof` and casts, for constructing instances of the class, for computing prefixes, and for accessing virtual types.

The class class of T implements the class interfaces of all the supertypes of T. It contains signatures for all static methods, and also a factory method, named `newThis`, for each constructor.

Methhod dispatching. In the J& language, as in Java, instance methods are dispatched on the receiver object. Since every class, including implicit ones, is translated statically, the compiler can find the right target to dispatch each method call at compile time.

For an instance method `m` of a J& class T, a static version `m_static` is generated in the class class of T. If T contains a declaration for `m`, the method `m_static` would contain the translated method body, and if T just inherits `m` without overriding it, `m_static` will be just a one-line method to dispatch the call to the class class that contains the last overriding implementation of `m`. The instance class of T also contains a one-line method for each instance method of

T, which simply calls the static version in the class class of T.

Static methods become instance methods of class classes in the target code. Therefore, we only need to find the class class singleton object corresponding to the receiver type of the static method call. However, the receiver type might be dependent, e.g., `T[this.class].C.m()`, in which case the runtime system would first get the class class singleton of the instance class object stored in `self`, then compute the prefix type, get the nested class class `C`, and finally call the translated static method `m`.

Translating packages. To support package inheritance and composition, the representation of a J& package includes a *package interface* and a *package class*, analogous to the class interface and the class class. Packages have no instance classes or instance interfaces.

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, and an overriding class in a derived family never changes the most specific Java superclass. Therefore, incompatible Java classes are never intersected.

The static translation is currently unable to correctly handle super constructor calls to non-default Java constructors. A correct implementation is included in the classloader-based translation in Section 5.2.

5.1.2 Dynamic representation with data structures

The compiler is a 7-kLOC extension of the one that implements static translation. The runtime system is now responsible for computing and maintaining information like the target for every method dispatch, and therefore becomes larger: it has 1300 LOC.

The basic scheme of class and package translation is the same as the static translation in Section 5.1.1. There are also instance interfaces, instance classes, class interfaces, and class classes for J& classes, and package interfaces and package classes for J& packages. These classes and interfaces also contain similar things as described in Section 5.1.1. However, none of these classes are generated for implicit or intersection classes and packages. Each explicit J& class also has a *subobject class* in the target code, for storing instance fields, and instance classes no longer contain fields. An instance of an explicit class is represented as an object of its own instance class, and an implicit or intersection class is represented as the instance class of one of its explicit superclasses. Statically, every J& object is referenced through a special interface `JetInstance`, implemented by every instance class. The dynamic implementation also provides the same level of Java compatibility as the static translation.

Subobject classes and field accesses. An instance class of a J& class T in this translation no longer contains any instance fields; it becomes a container of subobjects, each for an explicit superclass of T, including T itself if it is explicit. The subobject class of a superclass T' contains all instance fields declared in T'; it does not contain fields inherited into T'. 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.

To get or set a field of an object, the `view` method is used to lookup the subobject for the superclass that declares the field. The field can then be accessed directly from the subobject. There are no field accessors in the generated code. For example, a field access `x.f` in J& source code is translated to `C$ext.view(x).f`, where `C$ext` is the subobject class of `C` that declares the field `f`.

Class classes and method dispatch. A class class declaration is created for each explicit J& class. For an implicit class T, the class class 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 similar functionality to that described in Section 5.1.1.

All methods, including static methods, are translated to instance methods of the class class. Each method has a *single-method interface* nested in the instance interface of the J& class that first introduces the method. The class class implements the corresponding single-method interfaces for all methods that it declares or overrides. The class class of the J& class that introduces a method `m` also contains a method named `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 and the receiver's class class is used.

Single-method interfaces allow us to generate code only for those methods that appear in the source code of the corresponding J& class. Therefore, implicit classes do not need to be translated.

Each virtual method call is translated into a call to the dispatch method, which does a lookup to find the class class of the most specific implementation. The class class object is cast to the appropriate instance interface and then the method implementation is invoked.

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 super constructor call is translated into a call to the appropriate initialization method of the superclass' class class.

Run-time type checking. When J& compiler does not generate code for implicit J& classes, the generated code cannot mention instance classes or class classes of implicit classes either, in order to be legitimate Java code. However, the J& source code may use implicit classes just as their explicit counterparts, e.g., in casts and `instanceof` expressions. The run-time type checking code needs to be translated in an indirect way.

Although there are various kinds of run-time type checking expressions, and one can check against not only fully-qualified types, but also prefix types, the very basic operation underlying any run-time type checking is to check whether $P_1 \leq P_2$, where both P_1 and P_2 are fully-qualified classes. The algorithm for `isSubclass`(P_1, P_2) is:

1. If $P_1 = P_2$, return `true`;
2. If P_2 is a declared superclass of P_1, return `true`;
3. If $P_1 = P'_1.C$ and $P_2 = P'_2.C$, then return `isSubclass`(P'_1, P'_2);
4. For each declared superclass P of P_1, if `isSubclass`(P, P_2), return `true`;
5. Otherwise, return `false`.

Optimizations. The implementation described above has several performance issues:

- A method dispatch involves a hashtable lookup to find the most specific superclass of the receiver class that contains an implementation. This is much slower than a normal method call in Java.
- A field access needs to lookup the subobject that contains the field. Again, the hashtable lookup is much slower than a normal field access in Java.
- Every 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.

The implementation caches the result for each hashtable lookup, and therefore the overhead of hashtable lookup is avoided for a repeated method call or field access. However, in a multi-threading setting, the caching code may require proper locking or thread-local data structures in order to be safe, which might offset the performance benefit of caching.

To reduce the overhead of allocation and garbage collection, one simple optimization is to not create subobjects for J& classes that do not introduce instance fields. The instance class of explicit J& class T may also inline the subobjects. These optimizations are not going to be implemented, because the classloader-based translation is already available, solving most of these performance problems.

5.2 Classloader-based implementation of family inheritance

The classloader-based translation for nested inheritance and nested intersection consists of a 4400-LOC compiler, as an extension of compiler described in Section 5.1.2, and a 3400-LOC run-time system, most of which is a custom classloader, implemented using the ASM bytecode manipulation framework [9].

The key idea of the classloader-based translation is to cache the result of the run-time search for the target class of a method dispatch or a field access in bytecode synthesized by the classloader, rather than in hashtables as in Section 5.1.2. Therefore, there is no complicated data structure to maintain, and no costly hashtable lookup at run time.

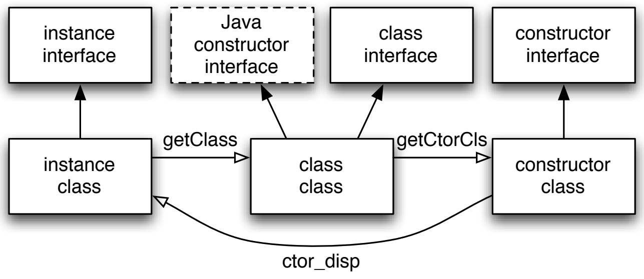

Figure 5.2: Updated class representation

5.2.1 Compiler translation

Translating classes. The compiler translation is scalable, that is, no target code is generated for implicit classes, as in Section 5.1.2. However, for each explicit J& class, the generated code is more similar to the static translation described in Section 5.1.1, without any single-method interface or any complicated run-time data structure management code. Thus the classloader-based implementation is not only scalable, but also generates smaller code than both implementations in Section 5.1.

Each explicit J& class is represented by the several classes, shown in Figure 5.2. The scheme is similar to that in Figure 5.1, with three additional classes—a *Java constructor interface*, a *constructor interface*, and a *constructor class*—for handling super constructor calls to Java constructors and the refined semantics of J& constructors.

Not every explicit class has a Java constructor interface; only those that directly inherit Java superclasses do.

Allocation. For a J& class T, the Java constructor interface contains translated signatures for all the constructors of the most specific Java superclass of T. The

```
class JavaClass {
  JavaClass(int i) {...}
}
```

```
class JetClass extends JavaClass {
  JetClass(int i) {
    super(i); // A super constructor call.
  }
}
```

Figure 5.3: An example with a Java constructor

instance class is a direct subclass of the most specific Java superclass—not a subclass of the instance class of any J& superclass of T—and therefore may call a Java super constructor during construction. The right Java super constructor call must be invoked at run time in order to correctly initialize those fields inherited from Java superclasses. For each constructor in the most specific Java superclass of T, the instance class has a constructor that issues the corresponding super constructor call, and the class class, which implements the Java constructor interface, has an instance creation method that calls that constructor in the instance class.

This kind of translation is necessary because when a J& object is ~~to be~~ created, we may only have a class class singleton, without knowing statically what exactly the J& class is—the type in the `new` expression may be dependent.

For example, Figure 5.3 shows the source code of a Java class `JavaClass`, and its J& subclass `JetClass`. The Java class has a non-default constructor that takes an integer parameter. The constructor of the J& class invokes that super constructor of the Java class. Figure 5.4 illustrates the translated code of `JetClass` that is related to object allocation: the Java constructor interface `JetClass_jctor`, the class class `JetClass_class`, and the instance

```
interface JetClass_jctor {
  JetInstance newInstance(int i);
}
class JetClass_class implements JetClass_jtor, ... {
  JetInstance newInstance(int i) {
    return new JetClass_inst(int i);
  }
}
class JetClass_inst extends JavaClass implements JetInstance, ... {
  JetClass_inst(int i) {
    super(i);
  }
}
```

Figure 5.4: Illustration of translation of code in Figure 5.3. Name mangling not shown for clarity.

class `JetClass_inst`. `JetInstance` is an interface that is part of the run-time system, and it is implemented by every instance class. As in the dynmaic translation in Section 5.1.2, every J& object is statically referenced through `JetInstance`.

In the latest implementation of the J& language, constructors may only be inherited and overridden by classes that are further bound in a derived family. Thus constructors are inherited and dispatched in a way different from normal methods, so additional constructor interfaces and constructor classes are generated to contain the translated signatures and code of the J& constructors.

Method dispatching. For each method declaration in a J& class T, which has to be explicit, a corresponding instance method is generated in the class class of T, with the translated method body. The class interface also contains the signature for the translated method. For every subclass T' of T, its class class would implement a dispatch method of the same signature, which calls the correct implementation from one of the superclasses of T'. If T' is explicit, the dispatch method is generated by the compiler; if T' is implicit, the dispatch method will be generated by the classloader. The compiler also makes use of inheritance between class classes to reduce the number of generated dispatch methods.

When a method `m` is invoked, the class class of the receiver might be an implicit class, but the generated code can only mention explicit classes. Therefore, the receiver class class needs to be cast to the class interface corresponding to the J& class T that first introduces the exact form of the invoked method `m`, that is, `m` in T must have the same signature, including the return type and exceptions, as the statically known invoked method.

5.2.2 Classloader-based run-time system

The J& run-time system primarily consists of three parts: superclasses and super interfaces of the classes and interfaces generated by the compiler, a classloader `JetClassLoader`, and several utility classes.

Base classes and interfaces. As shown in Figure 5.2, a J& class is translated to three classes: a class class, an instance class, and a constructor class. `JetClassInfo` is the common superclass of all the class classes, and `JetCtorClass` is the common superclass of all the constructor classes. The run-time system does not have a common superclass for instance classes, but only a common super interface `JetInstance`, because instance classes may inherit from their Java superclasses. There is also a superclass `JetPackageInfo` for all the package classes.

The superclasses implement several functionalities that are common to all

J& classes, e.g., run-time type inspection (casts and `instanceof`), evaluation of prefix types, and computation of fully qualified names for various kinds of classes as in Figure 5.2.

Classloader. The basic functionality that a classloader needs to implement is the `loadClass` method, which takes a fully qualified class name, and returns a `java.lang.Class` object that represents the class. The classloader also maintains a `HashMap` to store classes that have been loaded or synthesized, so that it does not generate one class twice. For a class name not already in the `HashMap`, `JetClassLoader` first checks whether it is a system class, or a class that belongs to the J& run-time system, and if so, it delegates to the system classloader to load the class. Then it checks whether the class can be found from the disk, i.e., whether it is generated from an explicit J& class, and load it if that is the case. If neither of the above two cases holds, it is probably an implicit J& class, and the classloader will try to synthesize it. There are various kinds of classes that the classloader synthesizes, and the most important and complicated ones and instance classes and class classes.

- Synthesis of instance classes.

  The synthesized instance class is similar to that described in Section 5.1.1: it collects field declarations from all the superclasses, and includes accessors for them. When the instance class of an implicit J& class T is synthesized, the classloader `JetClassLoader` iterates through instance classes of all the *explicit* superclasses of T, finds each field declaration that has not been collected, and generates the declaration, together with the accessor methods. Iteration of superclasses requires the class class, and therefore synthesis of an instance class always first triggers the synthesis of the corresponding class class.

- Synthesis of class classes.

  One of the most difficult parts to deal with in the synthesis of implicit classes is circular dependencies. As mentioned above, the synthesis of the instance class depends on that of the class class. On the other hand, the code of the class class mentions the instance class as well, e.g., in the `newInstance` method (Figure 5.4). To avoid circular dependencies, the J& classloader synthesizes the class class of an implict class T in two steps: first, it generates a class class for T that is basically a placeholder only containing information for iteration of the superclasses of T, which meets the requirements of synthesizing the instance class; second, the static initializer of the placeholder class class calls back the classloader to finish the rest of the synthesis (`newInstance` methods, dispatch methods, etc.), and to replace the class class singleton object for T with the updated one.

5.2.3 Load-time optimizations

ot only can the classloader synthesize implicit classes, ~~but~~ it can also rewrite explicit classes when they are loaded. The classloader-based implementation enables several load-time optimizations:

- Dependent types in J& cannot be completely evaluated at compile time. However, most of the dependent types in a typical J& program are dependent on `this.class`, and therefore may be pre-computed when each class class object is loaded and initialized. This kind of *per-class dependent* *type pre-computing* is important for the performance of J& programs with a lot of dependent types.

- When the J& source code mentions an implicit non-dependent class, the compiler generates a special form of `ldc` (load constant) instruction, which is rewritten by the classloader to a direct reference of some appropriate generated class of that implicit J& class.

- Each class class of T is rewritten to implement the static interface of every superclass of T, either explicit or implicit. Therefore, the complicated `isSubclass` algorithm in Section 5.1.2 is reduced to just an `instanceof` operation in the loaded bytecode. ~~Therefore~~ run-time type checking in the classloader-based implementation has almost no overhead compared to Java.

- The compiler generates hints that make the classloader insert the `dup` instruction, when a method call has two consecutive parameters that are the same. This is often the case in translated method calls, where the `dup` instruction is very efficient and convenient.

- Certain interface casts on method receivers can be removed in the bytecode but not in the Java source code. The classloader removes these casts to make the loaded program run faster.

5.2.4 Java compatiblity

The classloader-based J& implementation is backward compatible with Java. It also supports several features that were not present in previous implementations:

- With Java constructor interfaces (Figure 5.2), J& classes may inherit any Java class, and invoke non-default super constructors correctly.
- Non-static inner classes are supported. They can also be inherited and overridden just as static nested classes.

On the other hand, inheritance of a Java family is not supported yet. The proposed more efficient implementation (Section 5.5) ~~will~~ support that. Integration with Java generics seems to be straightforward, but has not been done ~~yet.~~

5.3 Implementing sharing

We have implemented prototype compilers for heterogeneous and homogeneous sharing, as Polyglot extensions based on the classloader-based implementation, presented in Section 5.2. The heterogeneous sharing implementation has a 5400-LOC compiler and a 4200-LOC run-time system, and the homogeneous sharing implementation has a 3900-LOC compiler and a 4400-LOC run-time system.

The core idea of sharing is that one object may be an instance of several shared classes, and every reference to the object could have a different view, which is one of the shared classes. Therefore, both implementations have similar translation and run-time support for views and view changes.

Heterogeneous sharing requires masked types and sharing constraints for type safety, and therefore has a more complicated type system. Homogeneous sharing has a much simpler type system, but the run-time system needs to support shadow classes.

5.3.1 Translating classes

asses are still translated basically in the same way as described in Section 5.2 and shown in Figure 5.2. For each explicit J& class, the compiler generates an instance class, a class class, and a constructor class, together with several interfaces. For each implicit J& class, the classloader ~~will synthesiz~~ se classes at load time.

The main difference for sharing is that all the shared classes have the same set of object instances. Therefore, the classloader maintains a representative instance class for each set of shared classes. At run time, objects of all these shared classes are created as instances of the representative instance class. When a new shared class that contains more object fields is loaded, the classloader updates the representative instance class to include the new fields. All ~~the~~ existing objects of the old representative instance class will be lazily converted to the most up-to-date representative instance class. The conversion works because objects of instance classes are referenced indirectly, as described in Section 5.3.2.

5.3.2 Supporting views and view changes

Reference objects. The sharing implementation adds a level to indirection to enable different references to the same object to have different views. Each object is referenced indirectly through a *reference object*, containing two fields: one points to the instance class object; and the other points to a class class object, which is the view associated with the reference. Reference objects are implemented with the `JetRefObject` class in the run-time system.

The behavior of the object is determined by the view. Method calls are dispatched on the views, and run-time type checking is also based on the views. A view change operation (`view` *T*)*e* is translated to generating a new reference object with the same instance class object, and a view compatible with *T*. The implementation memoizes the result to the most recent view change operation on any reference object, to avoid repeatedly generating the same reference object.

An object might obtain a new view implicitly. Moving an object from one family to another would implicitly move all the other objects that are transitively reachable through shared fields (unmasked fields in heterogeneous sharing, or all the fields in homogeneous sharing). Implicit view changes are carried out lazily, only at the time when objects are accessed through fields.

Colocation of instances and views. A separate reference object adds some overhead for accessing members of an object, and slows down object creation, because there are two objects to create for each `new` expression, instead of just one. The compiler optimizes the case by making the instance class inherit `JetRefObject` when there is no Java superclass other than `java.lang.Object`. With the colocation, the instance object becomes its own reference object, and there is no need to create a separate object at instantiation.

Optimization of field accesses. As described in Section 3.1.3 and Section 4.2.2, field accesses in both heterogeneous sharing and homogeneous sharing may trigger implicit view changes. However, not every field has to have a view change, and some of them may be accessed directly without breaking type-safety.

There are a couple of cases where a field access does not need an implicit view change:

- The field is not declared with a dependent type. Then the type of the field does not depend on the view of the container object, and therefore implicit view changes are not necessary.
- The field is used at a place where the exact view is not important. For example, in an expression `x.f.g`, the field access `x.f` may not need an implicit view change, as long as the second field `g` is declared in a class that does not have a shared superclass. Another example is when an explicit view change is applied to a field access.

5.3.3 Implementing heterogeneous class sharing

Type checking. Because of masks, the J&$_s$ language has a flow-sensitive type system. Every method is checked in two phases. The first phae is flow-insensitive, ignoring masks, and generates typing information necessary for building the control-flow graph. The second phase is essentially an intraprocedural data-flow analysis, which computes a type binding, possibly with masks, for each local variable (including `this`) at every program point.

View changes in heterogeneous sharing need to be type-checked in scopes of corresponding sharing constraints. When a method is inherited into a new family, the compiler checks whether each of its declared sharing constraints still holds, by enumerating all the subclasses of the two types mentioned in each constaint. If any of the constraints is no longer valid, the method needs to be overridden.

Duplicate fields. In heterogeneous sharing, shared classes do not necessarily share all their fields. As described in Section 3.2, new fields introduced in derived shared classes and fields with unshared types are not shared, and are masked in the sharing relationships. An unshared field may have multiple copies in the object, each for a shared view. The compiler generates information on masked fields in the target code, and the run-time system includes all the copies of an unshared field in the representative instance class.

When an unshared field is accessed, the view of the receiver object decides which copy of the field is actually used. Therefore, field accessors are implemented in class classes, rather than instance classes. To avoid implementing another set of field accessors in instance classes, the classloader *linearizes* the instance classes of all the classes in each set of shared classes: each instance class, upon loading, is rewritten by the classloader to inherit the most recent representative instance class in the set of shared classes that are already loaded, and becomes the new representative instance class. Therefore, a field accessor in the class class of J&$_s$ class T can access the field in the representative instance class by first casting it to the instance class of T, and then directly access the field. The linearization works because each class may declare at most one shared base class, and therefore it is impossible to join two nontrivial sets of shared classes at run time.

Although linearization helps improve the performance of field accesses in the J&$_s$ implementation, there is still some overhead. Homogeneous sharing does not have the problem.

5.3.4 Implementing homogeneous family sharing

Type checking. Type checking of homogeneous sharing is similar to that in the original J& compiler, without the complication of flow-sensitivity or enumeration of subclasses as in the case of heterogeneous sharing.

The type system needs to prove sharing relationships between types, in order to type-check view change operations. It collects sharing relationships from sharing declarations, recursively establishes sharing relationships to all the known nested classes and packages, and takes the reflexive, symmetric, and transitive closure to form the sharing relation. The relation is then used to type-check view changes. Type checking is modular and sound, and also conservative, because a true sharing relationship might not be recognized by the type system if it requires knowing the declarations in some derived familiy that has not been checked yet. In that case, the programmer must break a complex view change operation into multiple view changes that take smaller stpes. This arguably has some documentation value.

Shadow classes. Translation in the compiler of the J& language with homogeneous sharing is still scalable, in the sense that the amount of code generated by the compiler is proportional to the size of the source code. Therefore, no Java target code is produced for shadow classes. Instead, this is done lazily at run time.

When a nested class is loaded at run time, the classloader in the run-time system checks whether it is an originating class. If so, code for a shadow class is synthesized in each base family that is shared with the originating family. The implementation does not copy shadow-method code into shadow classes, but only generates one-line dispatch methods that call corresponding shadow methods declared in the originating class.

Recall from Section 4.2.3 that shadow classes are named specially in the source code, with the name of the originating class embedded in the syntax. For each explicit occurrence of a shadow class, the compiler generates code that calls the run-time system to load the originating class, triggering run-time synthesis of the shadow class. This is necessary, because the shadow class is declared by a derived family, which might not have been loaded when the base family mentions the shadow class.

Field accesses. As described in Section 5.3.3, an unshared field in heterogeneous sharing has multiple copies, each for a view of the containing object, and field accesses depend on views.

With homogeneous sharing, every field is shared, and there is only one copy to access regardless of the view. Therefore, field accessors can be implemented in instance classes where corresponding fields are stored. This makes field accesses faster in homogeneous sharing.

	cast	view change	virtual call	static call	field read	field write	allo-cation
J&$_s$	2.81	17.6	10.3	1.09	5.92	5.31	26.1
J&	3.12	18.3	9.26	1.09	3.13	2.97	26.3

Table 5.1: Microbenchmarks: average time per operation, in ns.

5.4 Results

This section shows some performance results for the classloader-based implementations of heterogeneous class sharing and homogeneous family sharing.

5.4.1 Microbenchmarks for sharing

We first compare the performance of the current J& implementation against J&$_s$, using some microbenchmarks to measure individual object operations. Every microbenchmark runs one operation 10^8 times in a loop. Table 5.1 shows the results. The testing hardware is a Thinkpad X200 with Intel L9400 CPU and 2GB memory, and the software environment consists of Windows Vista, Cygwin, and JVM 1.6.0_13.

The results confirm that object operations in the implementations of J&$_s$ and J& have similar performance, but field accesses with homogeneous sharing are almost twice as fast as J&$_s$ field accesses. With homogeneous sharing, field accesses are no longer view-dependent, and therefore accessor methods can be implemented in the same class where fields are located.

The current implementations of both J&$_s$ and J& still introduce noticeable performance overhead compared to Java, because of more complex subtyping relationships, indirections through reference objects, and the translation of field

	bh	bisort	em3d	health	mst	perimeter	power	treeadd	tsp	voronoi
Java	1.74	0.53	0.17	0.41	0.88	0.22	1.00	0.14	0.12	0.31
J& [58]	13.91	1.77	0.48	8.45	4.43	2.82	2.43	2.20	0.37	7.19
J& with classloader	2.02	0.71	0.22	0.71	1.06	0.39	1.14	0.21	0.16	0.59
J&$_s$	2.61	0.88	0.23	1.61	1.54	0.47	1.27	0.45	0.17	0.83
J& (JHS)	2.42	0.80	0.23	1.54	1.51	0.44	1.16	0.44	0.16	0.69

Table 5.2: Results for the jolden benchmarks. Average time over ten runs, in seconds.

accesses into accessor method calls. The overhead for larger programs is less than these microbenchmarks might suggest, because object operations account for only part of the execution time. In prior work [65], we present ideas for performance improvements, which should apply to the J& implementation as well.

5.4.2 Jolden benchmarks

We tested the J&$_s$ and the J& implementations with the jolden benchmarks [**?**] to study the performance overhead for code that does not use the new extensibility features of class sharing. All ten benchmarks, with few changes, are tested with five language implementations: Java, J& as described in [58], J& without sharing as described in Section 5.2, J&$_s$ with heterogeneous sharing, and the new version of J& with homogeneous family sharing. Table 5.2 compares the results. The testing hardware is a Lenovo Thinkpad T60 with Intel T2600 CPU and 2GB memory, and the software environment consists of Microsoft Windows XP, Cygwin, and JVM 1.6.0_07.

Table 5.2 shows that the use of a custom classloader greatly improves the performance, comparing the two implementations of J& (without class sharing). Unsurprisingly, nested inheritance/intersection and class sharing do introduce overhead. With the custom classloader, the performance overhead of supporting family inheritance is 42% compared to the highly optimized Java HotSpot VM. The J&$_s$ times show a 37% slowdown versus the classloader-based J& (without sharing) implementation, and 94% versus Java. On the other hand, homogeneous sharing has a better performance compared to J&$_s$: the two overhead numbers drop to 28% and 82%. This is consistent with the microbenchmarking results shown in Section 5.4.1.

The overhead seems reasonable, especially considering that the current implementation works as a source-to-source translation to Java, precluding many optimizations and implementation techniques. We expect that a more sophisticated implementation could remove much of the overhead.

Running programs in J&$_s$ or J& have the latent capability to be extended in many ways, so it is not surprising that there is some performance cost. But it seems software designers are often willing to pay a cost for extensibility, because they often add indirections and use design patterns that promote extensibility but have run-time overhead. We believe that J&$_s$ and J& remove the need for many such explicit extensibility hooks, while making code simpler. For systems where extensibility is more important than high performance [79], the existing implementation may already be fast enough.

5.4.3 Tree traversal

The jolden benchmarks do not use the new features provided by J&$_s$. To study the performance of view changes, especially on large data structures, we wrote

Tree height	16	18	20
Tree creation	0.110	0.287	1.295
Traversal before view changes	0.008	0.027	0.105
View changes	0.125	0.367	1.303
Traversal after view changes	0.006	0.025	0.099
Explicit translation	0.145	0.622	1.669

Table 5.3: Comparing view changes with explicit translation. Average time over ten runs, in seconds.

a small benchmark in which two families share classes that implement binary trees. A complete binary tree of a given height is first created in the base family, and an explicit view change is applied to the root of the tree. A depth-first traversal is carried out to trigger all the lazy implicit view changes. The testing environment is the same as in Section 5.4.2.

Table 5.3 summarizes the results. In-place adaptation, even with a traversal that triggers all the implicit view changes, is faster than an explicit translation that creates new objects in the derived family, and the running time is also close to that of the initial creation of the tree. The fourth row shows that once the implicit view changes are complete and the new reference objects have been memoized, traversals execute as fast as a traversal before the view changes. This benchmark does a complete traversal. Since view changes are lazily triggered, the relative performance of adaptation would look even better if not all nodes needed to be visited after adaptation.

5.5 More efficient implementation

As described in this chapter, the performance of the J& language has improved significantly with various compile-time and run-time techniques, especially the custom classloader. However, Section 5.4 shows that there is still a non-negligible at best performance overhead compared to Java. Moreover, the performance cost is paid upfront, even for programs that do not use the advanced extensibility features provided by the J& language.

It is hard to implement efficiently, partly because the J& language is very expressive: one class may have multiple explicit superclasses, every namespace in a nesting hierarchy may define a family that can be inherited and further bound, and any class may be shared. In a lot of cases, the expressive power makes it impossible to direclty map constructs or operations of the J& language to their counterparts in Java, which is the key to achieve good performance.

A possible strategy is to restrict the expressiveness for better performance. We could have a "light" version of the J& language, with the following changes in the semantics:

- No explicit multiple inheritance between classes. This corresponds better to the Java semantics, and helps make field accesses and object constructions more efficient. Of course, implicit inheritance from classes declared in the base family and multiple inheritance between packages, which are stateless, are still supported.
- Families are only defined by packages, not classes. This should not have too much practical impact, as package inheritance is expected to be the common use of family inheritance [56], which is also confirmed by our experience with the J& language.
- Packages are exact by default. Therefore, a class, written in its normal way, is not a subtype of the corresponding class in the base package. This fits the Java semantics better. Additional syntax would be needed to indicate

that a package is not exact in a typename, which may then refer to objects from derived families.

- Prefix types are desugared on the package-class boundary in a typename. A J&-light type `p.A` is equivalent to `p[this.class]!.A` in the original J& language.

With these changes, the default case in the J&-light language coincides with Java. In fact, one can easily reinterpret existing Java code as J&-light code, solving a long-standing problem of inheriting from a Java "family".

The implementation is still going to be based on classloader. When a family nested in package `p1` is inherited by a derived package `p2`, the classloader will synthesize all the classes in `p2` by copying code from `p1`, while basically rewriting each occurrence of `p1` to `p2`. Therefore, at least for the default case, the J&-light language would have almost no performance overhead.

CHAPTER 6

CONCLUSIONS

The goal of this work is to help people build more reliable and extensible software with a language-based approach. Building good software systems has always been a hard problem, especially when the systems are large and complex. The complexity of a large software system mostly comes from that it consists of many related components that are interacting in non-trivial ways, and therefore an effective language mechanism needs to be able to handle relations between components.

Masked types provide a strong safety guarantee for object initialization: uninitialized fields are never read, and therefore eliminate a significant source for reliability problems of software systems. Masked types are expressive enough to support many useful initialization idioms that work with interacting classes and objects, either through inheritance or mutual references. Methods and constructors explicitly express their initialization contracts through effects, which enable modular type checking even with inheritance. Conditional masks track dependencies between initialization states of mutually referencing objects, and support safe construction of cyclic data structures. These features make masked types applicable to large software with complicated inter-class relationships. Moreover, masked types has a low annotation burden because of its effective default annotations, and little requirement on reasoning about aliasing, making it even more suitable for realistic software systems, as confirmed by our experiences with the J\mask language.

Class sharing is the first language mechanism that combines the advantages of both family inheritance and adaptation, addressing the two important limitations of normal inheritance for supporting code reuse in large software systems. With class sharing, a family of interacting classes may be inherited together, preserving the relationships, with some of the classes selectively being shared between the base and the derived families. Class sharing supports new capabilities such as family adaptation, dynamic object evolution, and in-place translation. These capabilities are supported by a variety of mechanisms: dynamic views typed with dependent types statically track the families of values; sharing constraints support modular type checking of view changes; masked types ensure shared and unshared classes can be mixed safely. The resulting language is proved sound, and the expressiveness is demonstrated with realistic software examples.

Family sharing develops from class sharing, aiming to solve a major pain point of class sharing, that is, the code complexity and annotation burden introduced by language mechanisms designed for type safety. Family sharing generalizes class sharing to operate at the level of families, in a way similar to how nested inheritance generalizes ordinary class inheritance. New mechanisms like shadow classes enable a derived family to safely and homogeneously share with a base family, while still extending the base family with new classes. Shadow classes also make families open, and provide new kinds of extensibility. Family sharing avoids complicated language mechanisms like masked types and sharing constraints, which are needed by class sharing. Therefore code written with family sharing may be simpler, easier to reason about, and more accessible to programmers, while still providing the expressive power required for extending and reusing components in large-scale software systems.

A language design is never complete without a real working implementation. The dissertation presents how to implement family inheritance and the two sharing mechanisms in an efficient and scalable way. The implementation is based on the Polyglot compiler framework, with Java as its target language. The run-time system includes a custom classloader, which synthesizes implicit classes on-demand at load time, therefore making the size of the target code proportional to that of the source code, without incurring a large performance overhead at run time. Sharing is implemented through one level of indirection.

This dissertation presents design and implementation of language-based mechanisms that would make large-scale software systems more reliable and extensible, while preserving type safety and practicality. I expect programming languages to evolve continuously to better help programmers build safer, more scalable, more extensible, and possibly larger software systems.

BIBLIOGRAPHY

[1] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In Awais Rashid and Mehmet Aksit, editors, *Lecture Notes in Computer Science: Transactions on Aspect-Oriented Software Development I*, pages 135–173. Springer-Verlag, 2006.

[2] Miles Barr and Susan Eisenbach. Safe upgrading without restarting. In *Proceedings of 19th International Conference on Software Maintenance (ICSM)*, pages 129–137, 2003.

[3] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proc. 20th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 177–189, San Diego, CA, USA, October 2005.

[4] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proc. 22nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 301–320, October 2007.

[5] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proc. 5th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[6] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1978.

[7] Kim B. Bruce. Safe static type checking with systems of mutually recursive classes and inheritance. Technical report, Pomona College, 1997. `http://www.cs.pomona.edu/~kim/ftp/RecJava.ps.gz`.

[8] 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. Springer-Verlag, July 1998.

[9] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems, 2002. `http://asm.objectweb.org/current/asm-eng.pdf`.

[10] Patrice Chalin and Perry James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, 2007.

[11] Sigmund Cherem and Radu Rugina. Maintaining doubly-linked list invariants in shape analysis with local reasoning. In *Verification, Model Checking, and Abstract Interpretation, 8th International Conference (VMCAI 2007)*, Nice, France, January 2007.

[12] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: A simple virtual class calculus. In *AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, pages 121–134, 2007.

[13] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proc. 15th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 130–145, 2000.

[14] Adriana B. Compagnoni and Benjamin C. Pierce. Higher order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, 1996.

[15] Bruno C. d. S. Oliveira. Modular visitor components: A practical solution to the expression families problem. In *Proc. 23nd European Conference on Object-Oriented Programming (ECOOP)*, July 2009.

[16] Ferruccio Damiani, Sophia Drossopoulou, and Paola Giannini. Refined effects for unanticipated object re-classification: Fickle$_{\mathrm{III}}$. In *ICTCS*, pages 97–110, 2003.

[17] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 59–69, June 2001.

[18] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *Proceedings of 18th European Conference on Object-Oriented Programming (ECOOP'04)*, 2004.

[19] Dominic Duggan. Type-based hot swapping of running modules. *Acta Inf.*, 41(4):181–220, 2005.

[20] Torbjörn Ekman and Görel Hedin. Pluggable checking and inferencing of non-null types for java. *Journal of Object Technology*, 6(9):455–475, October 2007.

[21] Erik Ernst. *gbeta—a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, rhus, Denmark, 1999.

[22] Erik Ernst. Family polymorphism. In *Proc. 15th European Conference on Object-Oriented Programming (ECOOP)*, LNCS 2072, pages 303–326, 2001.

[23] Erik Ernst. Higher-order hierarchies. In *Proc. 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.

[24] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Proc. 33rd ACM Symp. on Principles of Programming Languages (POPL)*, pages 270–282, Charleston, South Carolina, January 2006.

[25] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proc. 2003 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOSPLA)*, pages 302–312, October 2003.

[26] Manuel Fähndrich and K. Rustan M. Leino. Heap monotonic typestate. In *Proceedings of the first International Workshop on Alias Confinement and Ownership (IWACO)*, July 2003.

[27] Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In *Proc. 22nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, October 2007.

[28] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 133–144, 2006.

[29] Kathleen Fischer and John Reppy. The design of a class mechanism for Moby. In *Proc. SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 37–49, 1999.

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1994.

[31] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 3rd edition, 2005. ISBN 0321246780.

[32] Stephan Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *Proc. Net Object Days*, 2002.

[33] David Hovemeyer and William Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, 2004.

[34] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 13–19, 2005.

[35] Atsushi Igarashi and Benjamin Pierce. Foundations for virtual types. In *Proc. Thirteenth European Conference on Object-Oriented Programming (ECOOP'99)*, number 1628 in Lecture Notes in Computer Science, pages 161–185. Springer-Verlag, June 1999.

[36] 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.

[37] Atsushi Igarashi and Benjamin C. Pierce. On inner classes. In *Informal Proceedings of the Seventh International Workshop on Foundations of Object-Oriented Languages (FOOL 7)*, Boston, MA, January 2000.

[38] Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. In *Proc. 22nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 113–132, New York, NY, USA, 2007. ACM.

[39] ECMA International. Eiffel analysis, design and programming language. ECMA Standard 367, June 2005.

[40] Haskell 98: A non-strict, purely functional language, February 1999. Available at `http://www.haskell.org/onlinereport/`.

[41] Anita K. Jones and Barbara Liskov. A language extension for expressing constraints on data access. *Comm. of the ACM*, 21(5):358–367, May 1978.

[42] *JSR 308: Annotations on Java Types*. Available at `http://groups.csail.mit.edu/pag/jsr308/`.

[43] John Lamping. Typing the specialization interface. In *Proc. 8th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 201–214, October 1993.

[44] K. Rustan M. Leino. Data groups: specifying the modification of extended state. In *Proc. 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 144–153, 1998.

[45] B. Liskov et al. CLU reference manual. In Goos and Hartmanis, editors, *Lecture Notes in Computer Science*, volume 114. Springer-Verlag, Berlin, 1981.

[46] B. Liskov and J. Guttag. Data abstraction. In *Abstraction and Specification in Program Development*, chapter 4, pages 56–98. MIT Press and McGraw Hill, 1986.

[47] Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Andrew C. Myers. *Theta Reference Manual*. Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, Cambridge, MA, February 1994. Available at `http://www.pmg.lcs.mit.edu/papers/thetaref/`.

[48] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proc. 15th ACM Symp. on Principles of Programming Languages (POPL)*, pages 47–57, 1988.

[49] David MacQueen. Modules for Standard ML. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, pages 198–204, August 1984.

[50] O. Lehrmann Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993.

[51] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism for object-oriented programming. In *Proc. 4th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 397–406, October 1989.

[52] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proc. 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 90–100, Boston, Massachusetts, March 2003.

[53] Mira Mezini, Linda Seiter, and Karl Lieberherr. Component integration with pluggable composite adapters. *Software Architectures and Component Technology*, 2000.

[54] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[55] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[56] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proc. 19th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 99–115, October 2004.

[57] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. 12th International Compiler Construction Conference (CC'03)*, pages 138–152, April 2003. LNCS 2622.

[58] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software composition. In *Proc. 21st ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 21–36, October 2006.

[59] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. Nested intersection for scalable software composition. Technical report, Computer Science Dept., Cornell University, September 2006. `http://www.cs.cornell.edu/nystrom/papers/jet-tr.pdf`.

[60] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language, June 2004. `http://scala.epfl.ch/docu/files/ScalaOverview.pdf`.

[61] Martin Odersky and Matthias Zenger. Scalable component abstractions.

In *Proc. 20th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 41–57, October 2005.

[62] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *Proc. 16th European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *Lecture Notes in Computer Science*, pages 89–110, Málaga, Spain, 2002. Springer-Verlag.

[63] Xin Qi and Andrew C. Myers. Homogeneous family sharing. July 2009. In submission.

[64] Xin Qi and Andrew C. Myers. Masked types for sound object initialization. In *Proc. 36th ACM Symp. on Principles of Programming Languages (POPL)*, pages 53–65, January 2009.

[65] Xin Qi and Andrew C. Myers. Sharing classes between families. In *Proc. SIGPLAN 2009 Conference on Programming Language Design and Implementation*, pages 281–292, 2009.

[66] 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.

[67] 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.

[68] John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, June 1996.

[69] Joel Richardson, Peter Schwarz, and Luis-Felipe Cabrera. CACL: Efficient fine-grained protection for objects. In *Proc. 1992 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 154–165, Vancouver, BC, Canada, October 1992.

[70] 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.

[71] 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.

[72] Charles Smith and Sophia Drossopoulou. Chai: Traits for Java-like languages. In *Proceedings of 19th European Conference on Object-Oriented Programming (ECOOP'05)*, pages 453–478, 2005.

[73] Amie L. Souter and Lori L. Pollock. The construction of contextual def-use associations for object-oriented systems. *IEEE Trans. Softw. Eng.*, 29(11):1005–1018, 2003.

[74] Amie L. Souter, Lori L. Pollock, and Dixie Hisley. Inter-class def-use analysis with partial class representations. In *PASTE '99: Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 47–56, 1999.

[75] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In *Proc. 32nd ACM Symp. on Principles of Programming Languages (POPL)*, pages 183–194, 2005.

[76] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering (TSE)*, 12(1):157–171, January 1986.

[77] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1987.

[78] Sun Microsystems. *Java Language Specification*, version 1.0 beta edition, October 1995. Available at `ftp://ftp.javasoft.com/docs/javaspec.ps.zip`.

[79] Tim Sweeney. The next mainstream programming language: a game developer's perspective. In *Proc. 33rd ACM Symp. on Principles of Programming Languages (POPL)*, page 269, January 2006.

[80] Don Syme. Initializing mutually referential abstract objects: The value recursion challenge. *Electronic Notes in Theoretical Computer Science*, 148(2):3–25, 2006.

[81] Kresten Krab Thorup. Genericity in Java with virtual types. In *Proc. Euro-*

pean Conference on Object-Oriented Programming (ECOOP), number 1241 in Lecture Notes in Computer Science, pages 444–471. Springer-Verlag, 1997.

[82] Mads Torgersen. Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 1998.

[83] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proc. 14th ACM Symp. on Principles of Programming Languages (POPL)*, pages 307–312, January 1987.

[84] Philip Wadler et al. The expression problem, December 1998. Discussion on Java-Genericity mailing list.

[85] Alessandro Warth, Milan Stanojević, and Todd Millstein. Statically scoped object adaptation with expanders. In *Proc. 21st ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Portland, OR, October 2006.

[86] Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. Shape analysis. In *Proc. 9th International Compiler Construction Conference (CC'00)*, pages 1–17, 2000.

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