Poplar: Extending the Java Programming Language for Evolvable Component Integration

Johan T. Nyström Persson

March 2012

Abstract

In the last few decades, software systems have become less and less atomic, and increasingly built according to the component-based software development paradigm: applications and libraries are increasingly created by combining existing libraries, components and modules. Object-oriented programming languages have been especially important in enabling this development through their essential feature of encapsulation: separation of interface and implementation. Another enabling technology has been the explosive spread of the Internet, which facilitates simple and rapid acquisition of software components. As a consequence, now, more than ever, different parts of software systems are maintained and developed by different people and organisations, making integration and reintegration of software components a very challenging problem in practice.

One of the most popular and widespread object-oriented programming languages today is the Java language, which through features such as platform independence, dynamic class loading, interfaces, absence of pointer arithmetic, and bytecode verification, has simplified component-based development greatly. However, we argue that Java encapsulation, in the form supported by its interfaces, has several shortcomings with respect to the need for integration. API clients depend on the concrete forms of interfaces, which are collections of fields and methods that are identified by names and type signatures. But these interfaces do not capture essential information about how classes are to be used, such as usage protocols (sequential constraints), the meaning and results of invoking a method, or useful ways for different classes to be used together. Such constraints must be communicated as human-readable documentation, which means that the compiler cannot by itself perform tasks such as integrating components and checking the validity of an integration following an upgrade. In addition, many trivial interface changes, such as the ones that may be caused by common refactorings, do not lead to complex semantic changes, but they may still lead to compilation errors, necessitating a tedious manual upgrade process. These problems stem from the fact that client components depend on exact syntactic forms of interfaces they are making use of. In short, Java interfaces and integration dependencies are too rigid and capture both insufficient and excessive information with respect to the integration concern.

We propose a Java extension, Poplar, which enriches interfaces with a semantic label system, which describes functional properties of variables, as well as an effect system. This additional information enables us to describe integration requests declaratively using integration queries. Queries are satisfied by integration solutions, which are fragments of Java code. Such solutions can be found by a variety of search algorithms; we evaluate the use of the well-known partial order planning algorithm with certain heuristics for this purpose. A solution is guaranteed to have at least the useful effects requested by the programmer, and no destructive effects that are not permitted. In this way, we generate integration links (solutions) from descriptions of intent, instead of making programmers write integration code manually. When components are upgraded, the integration links can be verified and accepted as still valid, or regenerated to conform to the new components, if possible. The design of Poplar is such that verification and reintegration can be carried out in a modular fashion. Poplar aims to provide a sound must-analysis for the establishment of labels, and a sound may-analysis for the deletion of labels. We describe the semantics of Poplar informally using examples, and provide a formal specification of Poplar, which is based on Middleweight Java (MJ). We describe an implementation of a Poplar integration checker and generator,

called Jardine, which compiles Poplar code to pure Java. We evaluate the practical applicability of Jardine through a case study, which is carried out by refactoring the JFreeChart library. We also discuss the applicability of Poplar to Martin Fowler's well known collection of refactorings. Our results show that Poplar is highly applicable to a wide range of refactorings and that the evolution of integrated components becomes considerably simpler.

Acknowledgements

I would like to thank professor Shinichi Honiden for his supervision of my Ph.D studies. His academic and financial support has been very valuable to me. He gave me the freedom to pursue my research in my own way, while at the same time offering essential advice and guidance during difficult times. I learned a great deal while I was studying in his research lab.

I am also grateful to Masami Hagiya and the rest of the thesis committee: Shigeru Chiba, Ichiro Hasuo, Zhenjiang Hu and Hidehiko Masuhara for valuable feedback and discussions and for the time and effort that went into their thorough judgment of my thesis.

I would like to thank the fellow students, researchers and friends who helped me by reading paper drafts, discussing research, commenting on my presentations, collaborating on side projects, and so on, even though I did not always return the favour as well as I should have: Rey Abe, Yukino Baba, Valentina Baljak, Lodewijk Bergmans, Christoph Bockish, Daisuke Fukuchi, Levent Gürgen, Richard Hayden, Katsushige Hino, Liyang Hu, Atsushi Igarashi, Taku Inoue, Fuyuki Ishikawa, Fan Jiang, Adrian Klein, Benjamin Klöpper, Tsutomu Kobayashi, Yuta Maezawa, Mohammad Reza Motallebi, Hiroyuki Nakagawa, Eric Platon, Christian Sommer, Toriumi Susumu, Yoshinori Tanabe, Kenji Tei and Florian Wagner, and all the members of the Honiden laboratory. To Yoshinori Tanabe and Fan Jiang I am also grateful for their translation of my thesis abstract into Japanese.

Alexandre Pichot was an intern at the National Institute of Informatics for six months during the final year of my studies. His efforts were very valuable in developing Jardine, the main implementation of the language described in this work.

During my Ph.D studies I had the good fortune to be able to collaborate with Levent Gürgen, Gabriel Keeble-Gagnère and Christian Sommer on significant side projects. I acquired a lot of knowledge and skills in the course of these collaborations.

I would also like to acknowledge the secretaries at NII and at the University of Tokyo, who assisted with administrative tasks and often helped me resolve language barrier related problems: Saki Narimatsu, Kyoko Oda, Kaori Sato and Akiko Shimazu.

While I studied for my bachelors degree at Imperial College, London, I benefited greatly from being supervised by Andrew Cheadle and Anthony Field, an experience that introduced me to computer science research for the first time. I was also fortunate to be working as a software developer under Sam Jervis and under Peer Vonna-Michell before I started my PhD studies; I learned much from working with them and with other colleagues.

I would also like to thank my other friends, in Japan and abroad, who have supported me in various ways during the Ph.D. process: Jacob Ehnmark, Sebastian Kelm, Johan Lörne, William Marjerison Goto, Johan Paulsson, Hans Ryding, Amanda Weiss, and especially Yoriko Yamamura.

Finally, I would like to thank my parents Maivi Nyström and Tommy Persson, who have always supported and encouraged me.

Contents

List of Figures

List of Algorithms

List of Terms

Client component A component that contains a query.

Component In this work, we consider a component to simply be a set of classes.

Composite property A conjunction of properties. Provides a simpler way of specifying multiple properties simultaneously.

External resource A resource with concrete state in one class that provides properties for another class.

Integration link A query and an associated solution.

Jardine Our implementation of a Poplar compiler. See Chapter 5.

Label The most basic unit of Poplar's specification language. Expressions are described in terms of a lower bound on the set of labels currently associated with them.

Label signature A triplet of three sets: additions, invariants and subtractions, which describe, either for a given variable or for a set of variables, the assumptions and effects of an argument in terms of its labels. This is a lower bound and it is not necessary to report all labels.

Method contract In Poplar, a pair of a label signature and a mutation summary. Together, these two specify a method in terms of its assumptions and effects about labels.

Mutation (of a resource) A change to the concrete state associated with a resource. This results in the loss of all properties associated with that resource and the specified object/objects, unless otherwise specified.

Mutation summary A list of all the resources that may be mutated as the result of executing a method or constructor. This is described in terms of an upper bound.

Poplar$_0$, Poplar$_1$ Two formalisations of Poplar, based on MJ, a calculus for a fragment of Java. See Chapter 4.

Poplar signature See *method contract*.

Produce query A query that requests the production of a new variable of a given type and a given label set.

Property A label that corresponds to a predicate on an object's concrete state. Associated with a resource.

Query An integration request. In Poplar, dependencies from one component on another may be expressed declaratively using queries. Concrete code (a solution) replaces the query at compile time. For a given query, solutions can change over time in response to component evolution.

Resource A group of properties and concrete state (fields). The erasing of properties is described in terms of mutation of resources.

Service component A component that provides declarations (methods and fields) that can be used as building blocks for satisfying a query.

Solution A code fragment that satisfies a query. Jardine replaces queries by their solutions, found using a search algorithm, at compile time.

Tag An immutable (impossible to erase) label that does not necessarily correspond to a predicate on an object's concrete state. For instance, it may be used to identify constants.

Transform query A query that requests that a new label set should be associated with an existing variable.

Uniqueness kind Uniqueness kinds classify references according to whether they may be aliased or not, and according to whether new aliases may be created.

1 Introduction

All flows, nothing stays.

Heraclitus (Lives of the Philosophers
by Diogenes Laërtius)

Change is universal. The sun rises and sets, tides rise and fall. Cars accelerate and decelerate. Elements freeze, melt, evaporate and condense. Circuits are switched on and off, new laws are made and old ones are repealed. Species evolve. The world consists of systems of changing, interconnected entities.

I, too, underwent change. Before embarking on the work presented here I worked as a software developer for two years. One of the tasks I busied myself with during that time is sometimes called maintenance programming: ensuring that a software system is up to date and remains functional when one of its components has changed. Even though one eventually becomes somewhat proficient at it, maintenance programming is not a glorious task. Component interfaces typically consist of APIs, which, in many current programming languages, are groups of classes, fields and methods. These can be thought of the components' vocabularies. A typical situation would be that an argument had been added to a function or a family of functions, or a function had been moved to a different module, or a similarly small change resulting from some refactoring need. Propagating or compensating for these changes in a large code base is not easy, nor is it enjoyable. Could this problem be automated, and what else would one gain if one automated it?

Technological innovations are often justified by the desire to automate a repetitive task. But in the course of finding a solution to a simple problem, one often discovers more subtle fundamental problems.

Computing, and digital technology in general, derives its power from its precision. But complex systems are usually not constructed atomically, but rather built piecemeal. Circuit builders have integrated circuits. Plumbers have standard pipe and joint sizes. Screws, nuts and bolts have standard shapes where the appropriate tools will fit precisely. Cars are designed to fit roads, and shoes are built to fit people's feet. In all these cases, systems are put together from different parts that are designed by different people at different times, yet they are able to interoperate well.

In some cases this interoperability derives from an intentional imprecision, or slack, in the specification. For instance, integrated circuits often specify acceptable upper and lower ranges on their operating temperature and supply voltages. A chip may be specified as expecting an input voltage of 4-6 V instead of precisely 5 Volts. Cars for normal roads are built according to a legislated maximum width, length and weight, and if they weigh less than the maximum tolerable weight, there will be no problem

driving them on the roads. The world is full of tolerance ranges that act as buffers for change. When a small change is applied somewhere in the system, if the change is within the acceptable range, the system will not break, but instead absorb the change or propagate it to other components and keep functioning. But when we consider the realm of software, or logic, the same is usually not true.

A small mutation in the DNA (genotype) of a cell often expresses itself as a small change in the cell's observable characteristics (phenotype), if at all. Such mutations often do not constitute a threat to the cell's survival or basic functioning. Gojobori et al. found, when studying virus mutations, that the vast majority of mutations were synonymous substitutions, which are redundant with respect to the production of amino acids [44]. On the other hand, when the first bacterial cell with completely synthetic DNA was made to self-reproduce, researchers found that even very minor errors in the genome would, under the circumstances of that experiment, prevent cell replication from taking place [42]. So it seems that in biological systems too, interconnections with a wide variety of different tolerances to parametric changes appear.

Unlike cars, shoes, and circuits, many software systems are never delivered to the customer in a final, irreversible way. Software is often provided and updated on an ongoing basis, even when it has been installed at customer sites far away. This trend has intensified in recent years, thanks to the ability to deliver software through the internet. Companies release continuous updates to their products and operating systems after the products have been sold. Customers are expected to install the latest updates on a regular basis in order to retain qualities such as compatibility and security. In a complex product built from software as well as hardware components, the former might never stop changing.

Computer systems are not always fragile with respect to changes. The Internet is a remarkable example of a resilient architecture, where hosts are continually added, removed, reconfigured or moved without threatening the system's basic operation. Error-correcting codes such as the Hamming code [81] are capable of detecting signal distortion within a prescribed range. However, interconnections between software components remain fragile. Since McIlroy's NATO address [78] in 1968, the software industry has dreamed of reusable components. But the concrete connections have always been made by referencing explicit procedure, message, function or method names. Even though some layers of indirection have sometimes been added, in various forms, fundamentally it is impossible to impose tolerance ranges on the inputs and outputs of these components. Variability is minimal, and changes cannot be absorbed or propagated. Incompatible changes are fatal poisoning.

The maintenance programming task was often complex at first, but later mechanical, once one had decided precisely how to adapt something to a particular change. It seemed as if this task could be automated somehow, if the way programming languages wire components and modules together was changed slightly.

1.1 Procedure Calls as an Assembly Language

Object-oriented programming has been an important step towards making software more modular and towards easing the modelling of external entities in software. Object-oriented languages emphasise grouping related data and procedures (*encapsulation*), hiding implementation details in the process. As implementation details are hidden from client objects, dependencies can only contain a limited amount of knowledge about objects and components. The information that may be known publically is usu-

ally called an *interface*. The constraint that dependencies may depend only on interfaces, and not on implementation, enables *polymorphism*: different concrete objects with different implementations, but with compatible interfaces, become substitutable for each other.

The Java programming language [10] is one of the most widespread object-oriented programming languages today. One of the reasons for it becoming widespread is that programs execute on a *virtual machine*: rather than being compiled directly to CPU instructions, they are compiled to instructions for an abstract machine [72]. In effect, the abstract machine encapsulates the physical hardware. It is then a separate problem for virtual machine implementors to implement the abstract machine, either through interpretation or through just-in-time compilation (immediately before the program is run) to the concrete instructions of the hardware CPU. In addition to this virtual machine, Java has several features that directly support encapsulation and modularity. There are several explicit visibility levels in classes, which can be used to establish a more refined visibility hierarchy than a binary distinction between exposed and not exposed. Unlike in C-family languages, direct memory access and pointer arithmetic is not possible, so there will never be any invalid pointers, eliminating a large class of program errors. These and other innovations help make Java one of the most practical programming languages for component-based software development to date.

However, Java (and almost all other programming languages in widespread use today) still has an essential construct that limits the modularity that can be granted even by encapsulation and polymorphism. This problem was identified by Shaw in 1993 [103]: *procedure calls are the assembly language of software interconnections*. A module depending on another invokes methods (Java procedures) with specific names, argument types and argument orderings, and the methods themselves are invoked in a specific order. A surprising amount of potentially harmful detail is encoded in sequences of these seemingly innocent invocations: the semantics of a single method, of each possible ordering of methods, requirements on each argument, any possible exceptions that might be thrown, and so on. Integration links consisting of sequences of method calls are, we believe, one of the fundamental sources of compositional fragility in Java programming. This is both because they can render a system impossible to compile, by failing to respond to a purely syntactic change, such as a renaming of a method, or because they may fail to catch a potentially poisonous change, such as a change in method semantics that has been introduced without any corresponding syntactic change. The work presented here is an attempt to create and maintain connections between software components in a novel way, so as to introduce more variability into the connections. Our goal is both to detect poisonous changes when they occur and to help propagate harmless changes whenever possible.

1.2 Inspirations for Our Work

In this work, we argue that sequences of method calls and field accesses that integrate components with each other should be constructed automatically rather than written manually. Furthermore, we argue that they should be constructed in such a way that their enduring validity can be checked automatically against new versions of components.

We are inspired by several existing lines of research. Prospector [75] is an interactive tool that generates code fragments as suggestions to the programmer. These fragments are found by mining a code base assumed to be valid in advance and potentially

applying simple transformations to the result. Especially interesting is Prospector's user interface, where the user requests a fragment *by asking for a way to produce a variable of a specific type in a specific context*. However, this approach is dependent on being able to mine a code base, and it is fundamentally designed for interactive use; we are interested in approaches that are fully automatic and integrated with the programming language itself, since the problem fundamentally originates at this level.

Typestate and protocol specifications [110, 27] provide a way to encode temporal specifications. A typical use of typestate for an object-oriented language is to be describe each object as a finite state machine and annotate methods with state transitions or invariants. This is a significant improvement on raw sequences of method invocations, since invalid sequences can be ruled out. Clients can be checked to confirm that their usage of a given interface conforms to a given protocol specification.

AI planning [40, 77] is the problem of identifying a sequence of actions that transform a starting state into a goal state, given a set of available actions in some domain. Typically, actions may interfere with each other and undo previously achieved results. We may observe that the problem of constructing a valid sequence of method calls to achieve some integration purpose very much resembles AI planning: some starting condition is available in the client component's context (the available variables and the state that they are known to be in), the integration has some goal, and there is a finite amount of actions available (method invocations and field accesses). In Prospector, goals were expressed as the production of a given type. We believe that the goal of a component integration can always be described in terms of either the computation of a value with some properties, or the production of a side effect. We are not aware of any existing work that uses a combination of typestate/protocol specifications and AI planning; however, Alfonso has investigated ways to interactively provide conformance recommendations from protocol specifications [6].

We are also inspired by labelled argument selection, which has previously been applied to labelled lambda calculus [1, 39, 38]. In this calculus, arguments are selected automatically based on a matching of their labels with function signatures. This is attractive since it removes the need to identify procedure arguments by their ordering. We are not aware of any application of labelled argument selection to imperative object-oriented languages. While languages such as Lisp [106] and ADA [68] allow the programmer to reorder parameters or omit optional parameters based on labels, they do not use this facility to automatically select arguments from a set of available expressions. We believe that combining labels with a temporal formalism, such as typestate, provides a necessary foundation for automated argument selection in imperative object-oriented languages.

1.3 Hypothesis and Contributions

Considering the inspirational work we have just mentioned, we would like to create a Java extension that combines these ideas. The hypothesis we seek to confirm is that *a combination of AI planning, labelled variables and temporal specifications, when augmenting the Java programming language, can yield a fully automatic component integration technique that is robust to evolution*. The main contribution of this thesis is an extended version of Java, Poplar, that confirms our hypothesis. We provide the design of Poplar, a formalisation, an implementation and a case study.

In this work, we are not aiming to provide a highly precise analysis, but strive mainly for simplicity and soundness. We also do not aim to provide a highly efficient

compiler and checker, if doing so would require significant work. These and other aspects of our work can potentially be enhanced in the future, independently of each other. Our main goal is to investigate the basic viability of our approach.

The design of Poplar is described in Chapter 2. The key elements of Poplar are labels, integration queries and resources. Poplar reasons about variables based on the set of labels that they are currently considered to possess. Integration queries can express requests for variables of specific types and with specific labels. Labels may, like typestate, correspond to concrete predicates on the state of objects; in this case, we call them *properties*. A high level overview of Poplar's design has previously been published in [87].

In Chapter 3, we describe Poplar's language elements and semantics in detail through the use of examples. This chapter aims to give a detailed, but readable, understanding of how the language works.

In Chapter 4, we specify Poplar as an extension of Middleweight Java (MJ [18]), a core calculus for an imperative fragment of Java. We formalise two systems, Poplar_0 and Poplar_1. Poplar uses guarantees about the creation and deletion (for properties) of labels. We argue that Poplar provides a sound must-analysis for labels, and a may-analysis for resource mutations (which represent potential label deletion). The analysis depends on reasoning about the aliasing of references; we use a simple scheme that classifies references according to whether they are unique and whether new aliases may be created. The analysis is modular. This chapter also discusses related work in the context of the formalisation.

In Chapter 5, we describe a proof of concept implementation of a Poplar compiler, called Jardine. Jardine performs component integrations, checks method contracts, and compiles Poplar code to ordinary Java classes. There is no need for special runtime support. Jardine was developed jointly with Alexandre Pichot, of the University of Pierre and Marie Curie in Paris, France. Jardine extends an existing Java compiler, JKit [92], with several new compilation stages. The most important new stages are the *Poplar checking stage*, which checks method contracts using the system formalised in Chapter 4 and assigns Poplar types, and the *Query solving stage*, which performs the actual integration process. We discuss the key algorithms used in these stages. The planning algorithm used for query solving is partial order planning [83, 77], which gradually strengthens a partial ordering of actions, although our design does not fundamentally restrict the choice of planning or search algorithm.

In Chapter 6, we perform a case study by applying Poplar to an existing Java library, JFreeChart. We show how the JFreeChart API can be evolved and the changes compensated for automatically by Jardine in a wide range of refactorings. We then discuss the theoretical application of Poplar to Fowler's well known collection of refactorings [34].

Chapter 7 concludes the thesis. We discuss general related work, confirm our hypothesis and discuss future work.

2 The Design of Poplar

In this chapter, we introduce Poplar informally. We begin by introducing some general background - object-oriented programming and the Java language - in Section 2.1. We then make some observations in Section 2.2, which illustrate the difficulties of evolving Java components and help motivate our design. In Section 2.3, we then discuss some existing solutions that have relevance to our problem and that inspired our design. In Section 2.4, we gradually introduce the design of Poplar - labels, properties, queries and resources - and their rationale.

2.1 Background

2.1.1 Object-oriented programming

Object-oriented programming (OOP) has now been a major paradigm for decades. Perhaps the most crucial features of OOP are encapsulation, dynamic message passing, and polymorphism [94, p. 225] [99, p. 249] [85]. The principle of encapsulation states that the implementation of an object should be separated from its interface. In other words, as client objects, we should not need to know how an object performs its task, only how to interact with it. This in turn enables polymorphism: once we are dependent not on the full details of an object, but only on partial and observable details, that object becomes replaceable by other objects with compatible interfaces but different implementations. Finally, message passing, often implemented as virtual method calls, emphasises object communication through messages. The receiver of a message is often not known at compile time but must be found at runtime through some dynamic process.

All of these features were present in some form in Simula-67, one of the earliest object-oriented languages. Smalltalk emphasised message passing and defined the notion of *protocols*, collections of messages that could be received and sent, which inspired Java's interfaces.

Modern object-oriented languages such as Java and C# have the notions of classes, which serve as blueprints, and objects, which are *instances* of classes. Classes often also act as the only mechanism for structuring the program in the large. Languages such as OCaml have separate module and class systems, but in the absence of this, classes typically also play the role of modules. Modern object-oriented languages typically also allow for the creation of new classes through the *subclassing* operation. Through subclassing, a new class is defined as a set of changes to an old class. In Java, new definitions may be added and old ones may be redefined. This is not only a means to incrementally create a hierarchy of increasingly precise definitions, but also

a way of explicitly describing the type hierarchy. In subclassing, the supertype is identified explicitly, leading to *nominal subtyping*, which is another common feature of modern object-oriented languages. On the other hand, *structural subtyping*, in which definitions are inspected for compatibility to determine whether types have a subtype relationship or not, has not been as successful outside of academia as nominal type systems [94, p. 251].

2.1.2 The Java programming language

The Java programming language was introduced by Sun Microsystems in 1995. In its early research stages it was called Oak [45]. Originally a niche language with somewhat controversial features, by 2011 it was the most popular programming language in widespread use, according to the TIOBE programming community index [117]. In that year it received a rating of 17.874%, a number that indicates the share of programming related discussions and pages on the Internet linked to Java. For most of the previous ten years it also held the top spot, surpassing languages such as C, C++, Lisp and SQL. Although the search engine based methodology of this index is not a perfect reflection of programmers' day to day activities, these numbers do give us an indication of Java's popularity today.

In the late 1990's, Java was marketed and known as a programming language for the world wide web. Java Applets were a new technology for the web browser, which promised to deliver interactive content in a way that the prevailing web standards of the day could not do [122]. During the 2000's, the situation changed and Java became dominant in various market segments - embedded systems, middleware, web service backends, and mobile devices [114, p. 262] - while becoming less dominant on the web browser and on the desktop. However, throughout these changes, or perhaps enabling them was a consistent focus on platform independence and network computing.

Java borrows syntax from C++ and concepts from languages such as Smalltalk and Simula-67.

Platform independence through a virtual machine. Java is often described not as a programming language but as a *programming platform* consisting of three parts: the language itself, a virtual machine, and a standardised set of class libraries. The virtual machine (JVM) is a runtime environment that is capable of executing Java bytecode [72]. The design of the bytecode, the class file format, in which bytecode instructions are stored, and the language itself means that it is relatively easy to produce efficient implementations of the JVM on many different kinds of underlying hardware platforms.

Bytecode. The Java compiler compiles the language not to assembly code for a hardware CPU, as compilers typically do, but to the bytecode format which is capable of running on the JVM[72, p. 171]. The bytecode format resembles the instruction set of a physical CPU in some ways, while differing from it in others. For instance, it has the concepts of registers and a stack, but there is no possibility of addressing a heap location explicitly by pointing to its offset from some starting position. Indeed, pointer arithmetic is completely absent both from the Java language and from the Java bytecode. This removes one of the most frequent sources of crashes due to memory access errors in languages such as C. Recently, there has also been an influx of new languages for the Java platform that compile to Java bytecode, taking advantage of the wide availability of libraries

and components. Some examples are Scala, Clojure (which is an implementation of Lisp), and JPython (which is an implementation of Python).

Dynamic classloading. Class files containing bytecode can be obtained from almost any source at run time, and then incorporated into the running program [72, p. 155]. Indeed, many features of the language were designed specifically to enable downloading of classes from a network, such as the Internet. When classes are loaded into the JVM, they are verified with respect to a number of predefined safety properties, such as the validity of stack accesses, before being compiled and run[72, p. 140]. This enables many basic security invariants to hold up even when some classes are compiled by a potentially untrusted compiler. An important consequence of the dynamic classloading mechanism is that the full set of classes that will be used by a program is, in general, impossible to know at compile time. Thus, when developing analyses and transformations for Java, modular analyses, which can operate on one class at a time without necessarily having access to the entirety of the program, are much more attractive than non-modular analyses.

A strong type system. It is impossible to cast a variable to a type it is not a member of in Java. Attempting to do so will result in a runtime error.

Absence of multiple inheritance. Although Java permits multiple inheritance of interfaces, which specify names and type signatures of class members, in contrast with the C++ language, it does not permit multiple inheritance of implementations [45, p.75]. Prohibiting this prevents the ”diamond problem” whereby different implementations for the same method are inherited with equal priority, resulting in compile time ambiguity.

Native methods Java is able to invoke methods written in C (or compatible languages) by using the *Java Native Interface*, JNI. In fact, almost all Java I/O, OS access and other system operations are ultimately done through JNI [45, p. 218].

Garbage collection. In contrast with C++, Java is garbage collected: there is no explicit deallocation operation cite[p. 447]Arnold:2005uq. This helps software developers avoid errors such as memory leaks, where heap memory grows irreversibly if pointers are lost before being deallocated.

Checked exceptions Each method that may potentially throw an exception must either declare that it does so, or catch the exception in a `try ... catch` statement [45, p. 297]. This annotation-based analysis of potential side effects is similar to *resource mutation* in Poplar, which we will introduce later to manage properties.

Package structure Java classes are grouped in packages, and the visibility modifiers provide a way of hiding classes and class members to the outside of a package [45, p. 153]. The package structure must be reflected in the source code and class file (compiled code) directory layout. This provides a natural mechanism for programmers to structure a program. It is common to prefix package names with an inversion of an internet domain name to prevent name clashes with other organisations; for instance, the Poplar implementation described in Chapter 5 resides in the package `jp.ac.nii.jardine`. Classes are imported using their fully *qualified class name*; `com.alpha.System` is a different class from `com.beta.System` but either may be referenced as `System` once imported.

Reflection. Java programs may reason about themselves using the reflection API, a set of classes designed to represent Java programs themselves [10, p. 397]. This paves the way for interesting metaprogramming techniques that might not otherwise be possible. For instance, it is possible to inspect a class and look for a method with a particular name or type signature.

Strong concurrency support. Java was designed from the ground up with concurrency in mind [45, p. 553]. In particular, each object can act as a monitor, and the synchronized monitor indicates that a method needs to acquire and release a lock on an object when it executes. Any object can act as a lock. In this work we will not consider concurrency aspects; we leave these for future work.

2.1.3 Component-based programming

There are many different perspectives on what a software component should be and what level of granularity it should be viewed at. Szyperski takes the view that software components are *units of software deployment, of third party composition, with no externally observable state*[114, p.36]. In this work, we take a more general view; we simply assume that a component is a set of classes, which may or may not be supplied by the same developer who is using it. We discuss existing component-based frameworks more generally in Section 7.1.7

2.1.4 Refactorings

Refactorings are systematic rewritings of source code with the aim of improving architecture, quality or performance. They usually differ from other changes to source code since they involve structural changes, not merely addition, change or removal of detail. For this reason, changes that involve multiple classes are often refactorings, and refactorings often involve multiple classes. Fowler et al [34]. give the following definitions of refactoring as a noun and as a verb:

> *Refactoring* (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour. [34, p. 53]

> *Refactoring* (verb): to restructure software by applying a series of refactorings without changing its observable behaviour. [34, p. 54]

They also give a list of more than 60 different Java refactorings, including items such as *extract interface*, *remove parameter*, *self encapsulate field*, *pull up field*, and so on. They recognise that refactoring may cause problems when it leads to interface changes:

> One of the important things about objects is that they allow you to change the implementation of a software module separately from changing the interface. You can safely change the internals of an object without anyone else's worrying about it, but the interface is important - change that and anything can happen. [34, p. 64]

Dinstinguishing between *published interfaces* (where the programmer cannot change all the clients of the interface alone) and *unpublished interfaces*, they simply recommend that programmers do not publish interfaces too early, and that once interfaces have been published, they should not be removed. Beyond this point, the old APIs

```
1  class Account {
2    void deposit(int amount) { ... }
3  }
4  class Customer {
5    void deposit(Account a, int amount) {
6      a.deposit(amount);
7    }
8  }
```

Figure 2.1: Two small components, one dependent on the other

should always be supported, and refactorings should be reflected in new methods and fields only. Java does in fact provide a deprecated annotation that allows programmers to indicate that API elements are not to be used in new code, but the value of this keyword is dependent on the willingness and ability of API clients to change their code when they get deprecation warnings.

Bloch echoes these recommendations [19]. His recommended principles for effective use of the Java language include items such as *minise the accessibility of classes and members*, *favor immutability* and *favor composition over inheritance* (since inheritance breaks encapsulation). In other words, conventional wisdom regarding published interfaces in the Java community is to minimise them as much as possible, since it is recognised that they will eventually be a drag on software evolvability and reusability. It seems clear that the evolvability and reuse friction associated with published interfaces is a major scalability constraint on component-based software. Component-based software cannot exist unless components are able to communicate with each other, but this can only occur through interfaces.

Dig et al studied breaking changes in several large software systems [28]. In each system, more than 80% of breaking changes were due to refactorings. It is clear that there is a significant tension between developers' desire to keep components up to date and the risk of syntactic and semantic breaking changes.

2.2 Observations on Java Component Evolution

In order to establish a basic intuition about the nature of the problem we are addressing, let us begin by considering a component based software development scenario in Java, and some of the issues that may emerge as components evolve. We will define a component to be a set of classes with a common purpose or application domain. We consider a situation containing two components: a component that provides bank accounts (A), and one that contains customer classes that interact with these accounts (C). B provides the Account class, and C provides the Customer class. Figure 2.1 illustrates this. In this example, we have omitted declarations that are irrelevant for the point we would like to highlight, and we will do so throughout this chapter.

The main dependency that can be seen in the example is that the Customer class explicitly invokes a method on an instance of the Account class. In doing so, the designer of the Customer class is creating dependencies on several aspects of the Account class:

- The name of the method deposit
- The number of arguments of the method

- The types of the arguments of the method
- The ordering of the arguments of the method
- Assumptions about the arguments of the method, their purpose, and states prior to and after the method invocation
- The purpose of the method: those side effects and return values (if any in this case, there is none), that the Customer class is explicitly interested in
- The expected state of the Account class prior to and after the invocation of the method, including state that the Customer class is not interested in for its own sake. This includes assumptions about methods that should or should not have been invoked prior to or after the invocation of this method.
- Any exceptions that might be thrown by the method (in this case, it is assumed that there are none)

Each of these dependencies may give rise to what is sometimes called a *breaking change* - a change in the depended-on component that gives rise to bugs in its client components.

Many of the potential breaking changes may be compensated for by applying a simple transformation, which often is immediate from the nature of the change itself. For instance, when an argument was added, instructions were supplied as to how this additional value was to be obtained. On the other hand, when the purpose of the method changed subtly, there was no obvious way for clients to compensate, and client component developers needed to consider carefully how to respond to the change. We may call the former case, when upgrading is a matter of mechanically applying some kind of transformation, a *syntactic breaking change*. The latter case, where no obvious transformation exists, may be called a *semantic breaking change*.

The Java compiler responds very differently to these various changes. For instance, many syntactic changes will cause the program to no longer compile, since a method invocation with the wrong number of arguments, or with the wrong argument types, is not valid Java code. On the other hand, most of the semantic changes, in as far as they have no syntactic counterpart, will be invisible to the compiler, potentially allowing bugs to be introduced into the program quietly. These are the essential problems we wish to address: syntactic changes, which can be mitigated by a simple transformation, should be handled automatically by the compiler and not require tedious manual upgrade work by programmers. Semantic changes should be detected and trapped by the compiler, alerting programmers to the possibility of an error, since in this case careful developer consideration is essential.

The issue of semantic breaking changes is highly problematic. In a perfect world, component publishers would publish complete formal specifications of the semantics of each declaration in all of their interfaces - design by contract. However, because of the additional effort required, because of the conflict between the expressiveness of such specifications and the feasibility of verifying them, this is not always done in practice. In addition, publishers stand to gain by providing less detail in their specification, since this makes their component open-ended and amenable to future evolution, to some degree. But clients stand to gain from assuming more details than what has been specified, since this simplifies their development task. If client component developers can make something work by trial and error, they often assume that the condition they

depend on will remain true for the foreseeable future, even if it was not specified. This kind of scenario is a breaking change waiting to happen.

We argue that it is possible to improve the design of the Java language to better deal with both syntactic and semantic breaking changes. Syntactic breaking changes occur when an interface undergoes structural or naming changes, for example due to refactoring. When such changes have occurred, but the underlying capabilities of the component have not been reduced or diminished, ideally the compiler should discover by itself how to acquire equivalent functionality using the new interface. With Poplar, this is possible, since we will declaratively express the intent behind an integration of two components in a fine-grained way, rather than specifying each individual step (Java statement) in the integration. Following a syntactic breaking change, equivalent functionality can be constructed from the same integration request, assuming the component still provides the functionality. Semantic breaking changes occur when the contract of a method has changed in unexpected ways. Poplar provides a flexible and fine grained form of design by contract, in the sense that the purpose of each variable in an interface declaration, for instance each argument as well as the receiver and the return value in a method invocation, can be specified precisely. The problem of maintaining the exact meaning of each method or field in Java is reduced to the problem of maintaining the exact meaning of individual labels in Poplar. Adding a label as a post- or a precondition can strengthen and weaken the specification, respectively. Removing a label as a pre- or a postcondition is similar. Because of the finer granularity of this problem, semantic changes can be expressed naturally and are easily captured when integration links are checked or regenerated.

2.2.1 The structure of integrating code

Consider the JDBC client in Figure 2.2. The goal of this client is to read a specific value (target) from the database. In order to obtain it, four inputs are needed: p, query, url and column. Three statements, beginning from DriverManager.getConnection can be informally identified as the bridge between the first inputs and the ultimate target value. Although we have not conducted a formal study, we argue that this three part division into inputs, bridge and goal is a typical scenario for a client that integrates with an API. Furthermore, many refactorings, especially purely syntactic ones, often result in necessary changes to the bridge part, but not to the other parts. One perspective on the solution that we will present in this work is that we strive to generate, regenerate and verify this ”bridge section” of integrations automatically.

2.3 Inspirations

We have seen that there is a fundamental tension between refactorings and evolvable component integration. We observed that integrating code can often be considered to consists of inputs, a bridge, and a goal, and we believe that greater evolvability can be achieved by focussing on generation, verification and regeneration of the bridge. We now discuss some techniques, fields and existing systems that served as inspirations for our solution.

```
1
2  void m(Properties p) { /* Inputs: p, url, query, column */
3    String url = "jdbc:mysql://localhost:3306/...";
4    String query = "select * from data where item.value > 5;";
5    int column = 1;
6
7    /* The following 3 statements bridge the goal and the inputs */
8    Connection c = DriverManager.getConnection(url, p);
9    Statement s = c.createStatement();
10   ResultSet rs = s.executeQuery(query);
11
12   while (rs.next()) {
13     int target = rs.getInt(column); //Target value (goal)
14   //...
15   }
16 }
```

Figure 2.2: Client code that uses the JDBC database API. It is possible to distinguish between inputs, a goal and a bridge between the two.

2.3.1 Typestate and protocols

For many classes in object-oriented languages, it is not always valid to invoke every possible method. There are often temporal constraints on valid and invalid sequences of method invocations. One formalism that can be used to model such temporal constraints is typestate, which was introduced by Yellin and Strom [110], and later adapted to object-oriented languages by Deline and Fähndrich [27]. Typestates in object-oriented languages correspond to predicates on the objects' concrete state. By knowing an object's typestate, if we know what state is assumed by each method and what state is established after the invocation of each method, we can know what methods may be invoked without error. Typical applications include classes that in some way deal with I/O, for instance streams and socket that must be opened or closed, or classes that may be initialised in some way before certain messages can be accepted. Clearly, many exceptions and runtime errors in languages like Java could be avoided completely if typestate checking could always be applied easily.

Figure 2.4 shows the source code for a Socket implementation in a hypothetical Java-like programming language with typestates. Figure 2.3 shows the typestate diagram induced by the specification. All the methods specify a transition, except the constructor, which "transitions" from no state into the initial state @raw, send and receive, which only describe an invariant, and bind, which describes a disjunction between two possible transitions.

An example of a valid method sequence in this model is new Socket, bind, connect, send, receive, receive, close. An example of an invalid sequence is new Socket, bind, send. Note that the meaning of each state - the concrete predicate on the class' fields and other resources it holds - is not specified. It is possible to specify and verify such predicates using other checking techniques, but doing this is not necessary to benefit from the typestate formalism.

We survey typestate and protocol checking techniques in greater detail in Section 4.5.1.

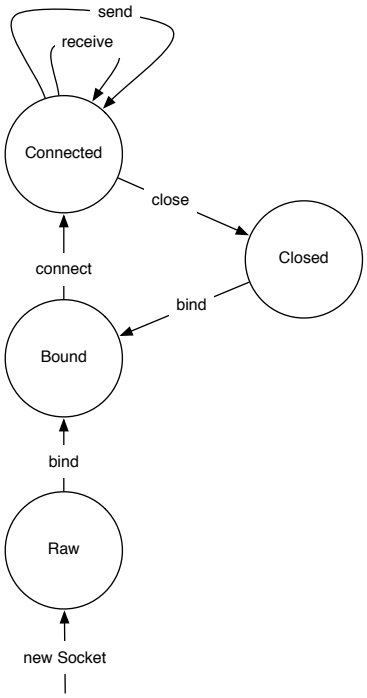


Figure 2.3: Typestate diagram for a socket implementation

```
1  class Socket {
2    Socket() this:->@raw { ... }
3    bind() this: @raw->@bound, this: @closed->@bound { ... }
4    connect() this: @bound->@connected { ... }
5    close() this: @connected->@closed { ... }
6    send() this: @connected { ... }
7    receive() this: @connected { ... }
8  }
```

Figure 2.4: A socket implementation in a hypothetical object-oriented language

2.3.2 AI planning

The AI planning problem is the problem of identifying a sequence of actions that establish a given goal state from a given starting state. Typically, a set of actions is available, and their effects, as well as the goal and starting conditions, are specified in some kind of domain logic. Normally actions may conflict with each other by undoing each other's effects. If not, then the order in which actions are executed is irrelevant, and solving the planning problem would then be very simple. The presence of conflicts means that the search space becomes considerably larger, and heuristics are often needed to speed up the search.

AI planning is widely applied in domains such as artificial intelligence and robots, industrial systems, including industrial process planning, and web service composition [21]. It is often a natural way of modelling problems that occur in the physical world. A classic example is a "blocks world" problem, in which a robot moves around in a landscape where blocks have been stacked in different ways. The robot can move blocks around, stack them on top of each other and so on, and the problem to be solved is often specified as a combination of blocks that are to be stacked in a certain way [100, p. 370] Broadly speaking, AI planning algorithms can be divided in two classes: plan-space algorithms, which search the space of all possible plans, and state-space algorithms, which search the state of all possible states that may be achieved.

We discuss related work in the field of AI planning in Section 7.1.3.

2.3.3 Prospector: fragment mining and assembly based on types

In [75], Mandelin et al develop the notion of *jungloid synthesis*. Observing that users of APIs often have difficulties in learning how to convert from one type to another (or how to construct a given type in a particular context), they have built an interactive system, Prospector, for the Eclipse platform that attempts to synthesize type conversions. For synthesis, a query is needed. A *jungloid query* is a pair (τ_{in}, τ_{out}), where τ_{in} and τ_{out} are class types.

Such a query asks the question "given that type τ_{in} is visible in our con, how can we construct type τ_{out} "? They then solve these queries by composing *elementary jungloids*. Examples of elementary jungloids are field access, static method or constructor invocation, instance method invocation, widening reference conversions (implicit upcasts) and downcasts. By composing a sequence of these, we obtain a series of Java statements.

When using this approach to synthesis, there is no guarantee that the resulting code will have the desired effects when executed. Jungloid synthesis is ultimately constrained only by the type system. There is also potentially a very large set of solutions for a given query. In order to ensure the usefulness of the results, the authors employ a heuristic which, for instance, favors short solutions over long solutions. It also mines existing code in order to discover which downcasts are safe to make. Finally, since it is an interactive tool, users are able to choose the result they want from a list of candidates.

Despite the simplicity of the approach, jungloid synthesis appears to be a useful technique: in 18 out of 20 test cases, the desired code fragment was among the top 4 results generated by Prospector, and the mean time spent finding results was 0.23s per query.

2.4 The Elements of Poplar

In Section 2.3 we listed several existing fields and systems that inspired the way we approach the problem of Java component integration. Our working hypothesis is that it is possible to construct a Java language extension that uses stateful labels and planning algorithms to construct evolvable component integration links. In this section we show, step by step, the design of Poplar, the language we will use to test our hypothesis.

2.4.1 Labelled variables

The fundamental building blocks of Java interfaces are method and field declarations. Field declarations can be seen as relating two values, owner and field, to each other. In contrast, a method having a void return type with n parameters relates $n + 1$ different values to each other, and $n + 2$ values if it has a non-void return type. This interrelatedness of values encoded by interfaces leads to many of the problems mentioned above. As a first attempt at breaking them down, we may attempt to select field owners (message receivers) and method arguments automatically in some fashion.

New users of a Java library, who are not familiar with its correct usage, often use type information to guess how to produce the results they desire. This tendency was exploited in Prospector [75]. For very specialised types, the valid combinations of interface fragments are so few that a correct solution can be found relatively quickly by simply looking at the permitted type of each argument and receiver. This is not the case for very general types, such as String and int, however. Clearly, a String can represent a wide range of data, such as a person's name to a file name or a URL, and an integer could represent an account number, the height of a building, or the current time of the day. Type information alone is not enough for correct selection. Based on this observation, we introduce *labelled variables* as a fundamental element of Poplar. Labels refine the constraints imposed by the type system such that correct candidates for method receivers and arguments can be selected unambiguously. The technique of argument selection by labels has previously been applied in the context of labelled lambda calculus [1, 38, 39] and OCaml. We are not aware of any existing application of this idea in Java.

As an example of how this might work, consider the standard library API for getting the time and date in Java. It changed substantially between version 1.4 and version 1.5 of the language. In version 1.4, the following code was used to obtain the current hour of the day:

```
1  Date now = new Date();
2  int hour = now.getHour();
```

In Java 1.5 and later versions, the following code is used:

```
1  Calendar now = Calendar.getCalendar();
2  int hour = now.get(Calendar.HOUR_OF_DAY);
```

We could capture this information in labels with Java 1.4 as shown in Figure 2.5. Here, *tags* is our term for labels that are immutable, that cannot be erased. (In Section 2.4.4 we will introduce *properties*, which are labels that can be established and erased while the program executes.) The tag nowHour identifies an integer that represents the current hour of the day (at the time when the value was produced). By declaring this tag in the interface TimeAndDate, and requiring all users of the label to use this interface, we can disambiguate between different labels with the same name, a standard procedure for Java declarations. The declaration for the Date constructor

```
1  interface TimeAndDate {
2    tags(int) nowHour, nowMinute, nowSecond;
3  }
4
5  class Date implements TimeAndDate {
6    tags currentTime;
7    Date()
8      result: +currentTime. { ... }
9    int getHour()
10     this: currentTime,
11     result: + nowHour. { ... }
12   /* Similar annotations for getMinute(), getSecond(), etc. */
13 }
```

Figure 2.5: TimeAndDate annotations for Java 1.4

```
1  Class Calendar implements TimeAndDate {
2    tags(int) hourMarker, minuteMarker, secondMarker;
3    tags defaultTimeZone;
4
5    final int HOUR_OF_DAY(hourMarker) = 11;
6    /* etc. */
7
8    Calendar()
9      result: +defaultTimeZone. {     }
10   int get(int selector)
11     this: defaultTimeZone,
12     selector: hourMarker,
13     result: nowHour). {...} /* Similar, alternative annotations for
           minute, second etc */
14 }
```

Figure 2.6: TimeAndDate annotations for Java 1.5

says that the returned value will have the tag currentTime. Finally, the declaration for the getHour method says that assuming that the owning object has the currentTime label, the return value will be an int variable with the label nowHour.

And in Java 1.5, as shown in Figure 2.6. This code uses the same labels as before, but the API is structured differently. However, it is still possible to request an int variable with the label nowHour in order to obtain the current hour.

2.4.2 Queries

The fundamental unit of composition in Poplar is the declarative *integration query*. A query requests either the production of a variable with a label or a set of labels, or the transformation of an existing variable so that it acquires a new label. Given a query, it is possible to use a search algorithm to find a solution that satisfies it, within any constraints specified by the surrounding method's signature, in terms of resource mutations and property changes.

In Figure 2.5 and Figure 2.6 we showed Poplar-annotated time and date components for Java 1.4 and Java 1.5, respectively. In order to integrate either of these service components with a client component, the following query may be used.

```
1  class TimeUtils implements TimeAndDate {
```

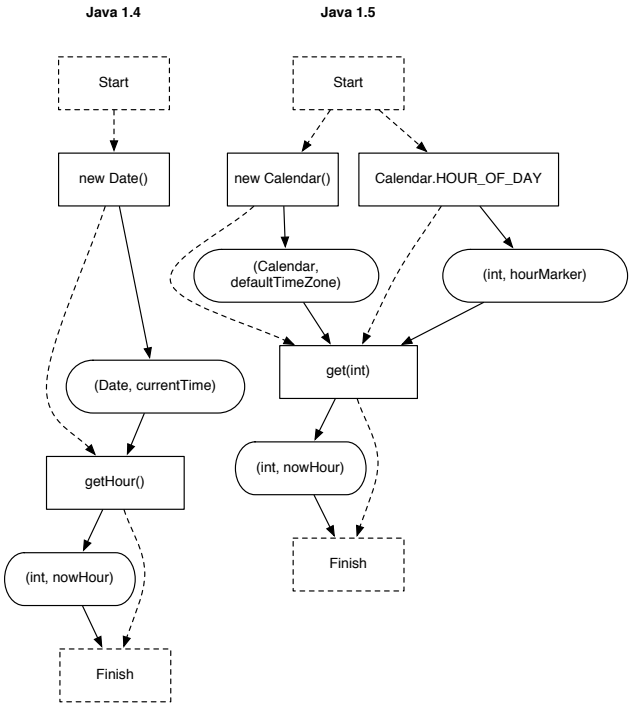


Figure 2.7: Visual representation of solutions to an integration query requesting the production of the current hour, for Java 1.4 and 1.5, respectively.

```
2     void printHour() {
3       int hour = #produce(int, nowHour);
4       System.out.println("The current hour is: " + hour);
5     }
6   }
```

After planning has been applied, assuming the solution search succeeds, the solutions might resemble the diagrams shown in Figure 2.7.

2.4.3 Partial order planning

The design of Poplar does not restrict the choice of planning or search algorithm that is to be used for the code generation, but in early experiments on a prototype, we have found Partial Order Planning (POP) to be a useful algorithm.

POP gradually refines a partial ordering of some set of actions. In principle, it searches the space of all possible plans, instead of searching the space of all possible states, as many planners do. POP is a good fit for the problem addressed here, because 1) Java statements are already in some sense partially ordered, through dataflow dependencies for instance, and 2) POP is relatively easy to understand and influence, and it should be a good fit for situations where some amount of human interaction may be needed, for instance in tweaking annotations, disambiguating between values and so on. The basic idea of the POP algorithm is that it gradually strengthens an ordering of actions, inserting causal links (connecting post- and preconditions of related

actions) and new actions as necessary while maintaining a set of open preconditions. A backward search from the goal towards the initial condition is performed as follows.

- Step 1. Initialise the plan to have two pseudo-actions *start* and *finish*. The effects of the start action are identical to the assumed environment of the plan, i.e. the starting conditions. The preconditions of the finish action are identical to the goals of the plan.
- Step 2. If there are no open preconditions, stop. A solution has been found.
- Step 3. Select an open precondition in the current plan.
- Step 4. For all available actions that achieve the precondition and are either already in the plan or not in the plan, create a successor plan with this action added. Also add ordering constraints and causality links (which pair preconditions with postconditions) for the new action.
- Step 5. For all the successors, resolve any conflicts among the causality links that might have arisen by strengthening the ordering constraints. If this is not possible, discard the successor.
- Step 6. Recurse on each successor plan. Go to step 2.

2.4.4 Properties

Unlike the labelled lambda calculus, Java is an imperative language, and expressions may have side effects. Primitive variables and composite objects may change their values and states over time. This makes it necessary to consider the temporal dimension of labels: if labels are to represent the functional properties of a variable, they may need to be introduced and removed as a result of program execution.

We have already discussed typestate checking (Section 2.3.1). Typestates correspond to concrete configurations of the state contained in a class, but do not encode their meaning in any description language. It is left to the programmer to ensure that methods establish the conditions represented by typestates if they claim to do so, and that methods do not expect any conditions beyond the declared expectations. Because they expose temporal constraints without exposing any implementation details of classes, typestates are relatively easy to combine with the substitution-based behavioural subtyping principle expected by languages like Java. In [27], subclasses may refine what a particular typestate means, for instance.

In Poplar, instead of using finite state machines, we introduce a more general construct, which we term *properties*. A property is a label that may be established as well as erased. Like a typestate, it implicitly corresponds to a predicate on the concrete state of a class. However, since they are not expected to form a protocol, an object may possess several properties simultaneously. A class with n properties will have 2^n possible configurations. Accordingly, methods may express invariants, preconditions and postconditions as sets of properties instead of as single states. They are not entirely unconstrained, however: we will describe below how we combine them with an effects system to simplify the task of keeping track of which properties of an object are intact at any given time.

We also introduce *composite properties*, which is a higher level property that, for a given class, combines a set of lower level properties. This is reminiscent of Yellin and Stroms original typestate concept [110] in the sense that it allows us to describe trees of

properties inside classes, where higher level, more composite properties clearly denote a greater degree of initialisation, in a sense, and lower level properties a lesser one.

We thus have two kinds of labels in Poplar: tags, which are immutable and do not change over time, and properties, which may change over time, according to principles that we describe below. These labels help refine the type system to a point where we will find it practical to select methods, fields and method arguments automatically. This forms the basis of Poplars automated integration technique.

2.4.5 Resources and effects

We mentioned above that properties may be established as well as erased, and that an object may have any number of properties at a given time. In addition, subclasses may define new properties not possessed by their superclasses. In order to integrate this concept elegantly with the subtyping constraints expected in an imperative object-oriented language like Java, it is necessary to encapsulate the properties somehow. Clearly, when we view an object at the level of one of its superclasses, we might invoke a method that ends up erasing or establishing many properties at the level of the subclass, through dynamic dispatch. Subclassing and method overriding would be impossible if we were forced to document this in full detail at every level in the class hierarchy. In order to address this problem, we adapt Boyland and Greenhouses side effect tracking system, which is based on abstract regions [46].

Boyland and Greenhouse's system is motivated by a desire to track reads and writes of variables. In order to be able to do this without exposing implementation details, they define abstract regions, which are sets of concrete fields. Instead of annotating each method with information about which fields it may potentially read or write, they annotate it with information about which regions it may read or write. This permits subclasses to add more state to a region without necessarily changing the effect summaries of methods, making method overriding natural. In Poplar, however, we are not interested in tracking reads and writes of fields - rather, we are interested in the addition and removal of properties. One way of adapting the Boyland-Greenhouse system might be to describe, for each property, which concrete fields of a class it may establish its predicate in. But this is imprecise in the case where a field may be used to implement multiple different properties in subtle ways; for instance, if separate properties are established in separate parts of an array, or in separate bits of an integer. It seems undesirable and inflexible to develop a specification language that describes precisely exactly what parts of what variables may be used for a particular property. In addition, concrete state changes cannot be tracked through field accesses if they are implemented in native code. For example, all I/O in Java, including opening, closing, reading and writing files and sockets, is implemented in native libraries using the Java Native Interface (JNI). Concrete implementations, which may be written in a wide variety of programming languages, back the corresponding Java methods. Therefore, instead of grouping fields into abstract regions, we group methods into *abstract resources*. We also associate properties with resources. Any invocation of a method in a resource is considered to nullify all of the properties associated with that resource, except for those properties that the method declares that it preserves or establishes.

Throughout our design we have used behavioural subtyping and the *substitution principle* as a guideline: subtype objects must always be able to take the place of any of their supertypes [73]. We integrate resources with properties by using the following principles.

Well-defined mapping from properties to resources. For each property that we declare, we must know which property or which properties it may be established in, and hence whose mutation it would be sensitive to. We establish this mapping in code by declaring properties inside declarations of resources. If a property is declared in more than one resource, it is considered to be sensitive to mutations of both.

Well-defined mutation summaries of methods. As with regions in the Boyland-Greenhouse system, each method must declare the set of resources that it may potentially mutate. Together with the mapping mentioned above, we can use this information to ascertain whether a given method may potentially destroy a given property.

Substitution principle. Again, analogous to the Boyland-Greenhouse effect system, we allow subclasses to redefine resources and properties as long as they do not contradict superclass definitions. For instance, subclasses may add more state to a given resource. Properties may be defined in terms of different state or a different predicate on the same state. The only thing that is not permitted is moving a property or a field into a different resource.

An example of properties and resources may be seen in Figure 2.8. The class Door has four properties in two resources. In theory there are 16 possible configurations. However, the designer has intended @closed and @open to be mutually exclusive properties, as well as the @unlocked and @locked pair. We do not provide a way to express this; the class designer must ensure that impossible combinations cannot be established. In the example we show it is indeed impossible: for example. there is no method that establishes @open without also erasing @closed. The method open implicitly mutates the resource this.main, so any property in that resource that has not been specified as an invariant or a postcondition of the method is assumed to be lost.

The constructor initially sets up the class to be @unlocked and closed. The default state for a new object with no specification associated with its constructor is to have no properties at all.

The methods lock, unlock, open and close are considered to implicitly mutate the resources that they have been declared in. However, the method closeAndLock has been declared outside of these resources and must declare the resources that it mutates.

Notice that some methods have specifications like ++@open while others have +@open. The former is called a *basic establisher* or a *direct establisher*. Only such methods are allowed to directly mutate the state that corresponds to the property. The latter kind of method is called an *indirect establisher*. For such methods we know that a direct establisher will eventually be called, directly or indirectly, and the Poplar compiler verifies that this indeed happens.

Label signatures and mutation of resources

In order to conveniently reason about the effects of expressions and statements more correctly, we use the concepts of *label signatures* and *mutation summaries*. We formalise these concepts and their usage in Section 4.2.1; here we informally give the intuition behind them.

Label signatures are triplets of three sets: $\mathsf{LS} = (\mathsf{LS}_+, \mathsf{LS}_=, \mathsf{LS}_-)$, where for a given Java fragment, the sets contain all the known additions, invariants and subtractions of labels, respectively. Additions correspond to ++ and + prefixes, subtractions correspond to a - prefix, and invariants correspond to no syntactic prefix (indicated by =

```
1  class Door {
2
3    resource lock {
4      properties @unlocked, @locked;
5      private boolean isLocked;
6
7      //these two methods implicitly mutate this.lock
8      void lock() this: @closed, ++@locked. { isLocked = true; }
9      void unlock() this: @closed, ++@unlocked. {isLocked = false; }
10     }
11   }
12   resource main {
13     properties @open, @closed;
14     private boolean isOpen;
15
16     //these two methods implicitly mutate this.main
17     void open() this:@unlocked, ++@open. { isOpen = true; }
18     void close() this:@unlocked, ++@closed. { isOpen = false; }
19   }
20
21   void closeAndLock() mutates this.main, this.lock:
22     this: +@locked, +@closed. {
23     close(); lock();
24   }
25
26   Door() result: ++@unlocked, ++@closed. { isLocked = false; isOpen =
         false; }
27 }
```

Figure 2.8: Example of properties and resources.

in $\mathsf{LS}_{=}$). Thus, methods with Poplar specifications, like the ones we saw in Figure 2.3, each have a corresponding label signature.

Mutation summaries describe what resources are mutated by the invocation of a method. Resource mutation corresponds to a change in the underlying data that properties in that resource describe. We reason about such a change conservatively: if there is no label signature at all, all properties are considered to be lost when a mutation occurs. If the label signature describes additions and invariants, these properties will become or remain established after the mutation. Thus, it is not necessary for the label signature to be fully precise, as long as mutation summaries definitely capture all possible resource mutations, given that we accept underestimating the label sets of variables. A lack of precision may translate to the inability to automatically construct some integrations, but not into unsound code.

A label signature and a mutation summary (LS, ρ) together describe a fragment's effects in a composable way. Consider the following examples, based on the Socket class in Figure 2.3.

```
1 Address a, a2;
2 //...
3 Socket s = new Socket(); //s: @closed
4 s.bind(a); //s: @closed, @bound; mutates s.state
5 s.connect(a2); //s: @open; mutates s.state
```

Here we have shown the properties that are established after each statement has executed. Note that @bound is lost implicitly due to the resource mutation of this.state at s.connect. There is no explicit -@bound annotation.

```
1 Socket s = new Socket(); //({s: @open}, {a:connectionAddress, a2:
      connectionAddress}), {s.state}
2 s.bind(a); //({s: @open}, {a: connectionAddress, a2:connectionAddress},
       {s: @closed}), {s.state}
3 s.connect(a2); //({s: @open}, {a:connectionAddress, a2:
      connectionAddress}, {}), {s.state}
```

Now we have annotated each statement with the (LS, ρ) pair obtained *by considering that statement and the following statements together*. We call this operation *chaining*. Thus the (LS, ρ) pair at s.connect describes that statement only, while the (LS, ρ) pair at new Socket describes the effects of all of the three statements. This can be read as: "the fragment adds the property @open to the variable s, given that a and a2 have the tag connectionAddress. The resource s.state will be mutated." This is enough information to understand the effects of the statement sequence and to combine it with effects of other statements.

2.4.6 Fragment specifications

A label signature and a mutation summary together fully describe a statement or a code fragment, both in terms of its useful effects and its potentially destructive side effects. As we have seen, they can be sequentially composed to express the effect of executing fragments in series. We obtain *a lower bound on the labels that will be established and an upper bound on the labels that will be erased by a fragment*. We may refer to a pair of a label signature and a mutation summary as a method contract (if associated with a method) or a fragment specification.

Consider a fragment of Java code that contains only method invocations, constructor invocations, field reads and writes and assignments - in particular, one that does not contain flow control statements such as branches (if, for, while, switch, etc.) and exception throws. If each statement has an associated Poplar signature, which describes what properties are assumed and altered for each operand and for any return value, as well as which resources are mutated for these variables, we are able to infer the Poplar signature of the entire fragment by inspecting the signature of each statement contained in it. For resource mutations, we simply take the union of the mutation set of each statement. For property changes to variables, we may track each variable throughout the fragment and remember the set of properties it has after each statement has been executed. Note that new variables may be created in the course of executing the fragment. In Chapter 4 we will in fact develop support for inferring specifications of if-statements, but not for the other control flow constructs.

A consequence of being able to infer the specification of a fragment is that we will be able to inspect a method and confirm that the method body is valid with respect to a given method contract.

2.4.7 Benefits of the resource and effect model

We have two main reasons for using a nonstandard model instead of a standard typestate formalism. First, properties and resources are a particularly good fit for planning. It is very easy to map Java methods to actions in a planning domain, since we can know, by inspecting the mutation summary of a method, whether a given property might be destroyed by it or not. This means that by extension, we can know whether a causal link in a plan (a link between some action's effect and another action's precondition) might potentially have a conflict with a given action.

Second, properties are very well suited to the problem of software evolution in that they allow us to *specify methods as bounded ranges rather than precise specifications*. As we have seen, a label signature and a resource mutation together form a lower and an upper bound. The relation between a stronger and a weaker specification is immediate and very easy to compute: it suffices to check if the labels being established are a superset of a specified set, while resource mutations are a subset. This has the additional benefit that *client queries do not need to match states precisely*. In a classical typestate formulation, each state is an atomic name. If we had used this approach, we would have been less able to evolve client and service components independently. When clients request a set of labels, any method that happens to provide that set, or a supserset of it, as a postcondition is potentially able to satisfy it. In Figure 2.9 we show two separate socket implementations, one that is general and one that supports IPV6. The class Client contains two queries. The query in m can be satisfied by either of the two classes, but the query in n can only be satisfied by the class IPV6Socket. This class provides stronger guarantees about the effects established by its methods, in the form of additional established labels. In this case, properties like @connected and @ipv6Connected have a refinement relationship: the latter is understood as a refinement of the former. This kind of relationship is not necessary, in fact no particular relationships among labels are assumed, although this is one of many possible relationships. It is the responsibility of programmers to manage and communicate the precise meaning of labels, although one could imagine some kind of tool support for this in the future.

It should also be noted that it is easy to encode classical typestate protocols in our formalism. In the Socket class we encode the Socket protocol from Figure 2.4 in our specification language. Essentially, each transition @a $\rightarrow$ @b now takes the form -@a, ++@b, or -@a, +@b for an indirect transition, assuming that @a and @b are in the same resource. In a sense, properties in a given resource may be thought of as being very simple orthogonal state machines with only two possible states, and by providing method specifications, these are composed to form more complex transition systems.

2.4.8 Uniqueness

Imperative object-oriented languages such as Java have freely assignable pointers, giving rise to the problem known as aliasing: at any given time, an object may be pointed to by any number of references, and it is usually nontrivial to determine whether two pointers may point to the same object or not at a given time. In order to reason about the state of objects in a language like Java with an acceptable level of precision, it is necessary to address this problem in some way. A wide range of different techniques have been attempted, including islands [51], balloons, ownership [24] and fractional permissions [15, 20].

Since we would like to focus on evaluating our approach without unnecessary complications, at this stage, we take a simplistic approach, based on annotating references with *uniqueness kinds*. This simply classifies references into three basic classes: unique, maintain and normal. Unique references are logically unshared and cannot be accessed through any other reference (with certain exceptions, which we discuss below). In addition, once a reference is considered to be unique, it cannot become non-unique. Maintain references may be aliased, but they will not acquire any further aliases in the future as a result of program execution. Maintain is strictly a weaker condition than unique. Fresh references are new, and can be reassigned to another uniqueness kind once. Finally, normal references, which have no annotation, are con-

```
1  //Standard implementation
2  class Socket implements ISocket {
3    resource state {
4      properties @raw, @bound, @connected, @closed;
5      Socket() this:++@raw. { ... }
6      bind(Address a) this: -@raw, ++@bound. { ... }
7      connect() this: -@bound, ++@connected. { ... }
8      close() this: -@connected, ++@closed. { ... }
9      send() this: @connected. { ... }
10     receive() this: @connected { ... }
11   }
12 }
13
14 //IPV6 implementation
15 class IPV6Socket implements ISocket {
16   resource state {
17     properties @raw, @bound, @connected, @closed,
18       @ipv6Connected, @ipv6Bound;
19     Socket() this:++@raw. { ... }
20     bind(IPV6Address a) this: -@raw, ++@ipv6bound, ++@bound. { ... }
21     connect() this: -@bound, -@ipv6bound, ++@ipv6connected, ++
           @connected. { ... }
22     close() this: -@connected, -@ipv6connected, ++@closed, ++
           @ipv6closed. { ... }
23     send() this: @ipv6connected. { ... }
24     receive() this: @ipv6connected. { ... }
25   }
26 }
27
28 class Client {
29   //Can be satisfied by Socket and IPV6Socket
30   void m() { ISocket s = #produce(ISocket, @connected); }
31   //Can be satisfied by IPV6Socket only
32   void m2() { ISocket s = #produce(ISocket, @ipv6Connected); }
33 }
```

Figure 2.9: Encoding the socket protocol with resources and properties

Kind	Assumption	Guarantee
Normal	None	None
Unique	No aliases	No future aliases
Maintain	None	No future aliases
Fresh	No aliases	None

Figure 2.10: Uniqueness kinds used in Poplar

servatively assumed to be aliased. Figure 2.10 summarises the assumptions and guarantees for these uniqueness kinds.

Annotations such as unique and maintain apply to heap (static) references only. Temporary (dynamic) aliases may be created, even from restricted references, as a result of method invocation and argument passing, for instance. We will restrict such dynamic aliases to prevent accidental simultaneous access to two ”unique” references to the same object.

It should be noted that our implementation supports destructive reads, which adds two more uniqueness kinds. We discuss this in Section 5.6.

We discuss existing work on alias confinement in Section 4.5.3.

2.4.9 Workflow

The basic workflow of Poplar is illustrated in Figure 2.11. Initially, a new system is designed. At this point, the system may contain client components, which contain integration queries, and service components, which provide building blocks that can satisfy queries. The same component may be both a service component and a client component. The Poplar compiler attempts to satisfy the integration queries. If this fails, there are no solutions, and the system must be redesigned accordingly. If this succeeds, the system has been integrated and is ready for use.

At some point in the future, either the client components or the service components may change. A semantic or syntactic change should be accompanied by a change in the corresponding Poplar annotations. Following such a change, the Poplar compiler is run to verify that integration links are still valid. If they are, the new interface is compatible with the old one. Semantic breaking changes, if any, are trapped at this stage, and in this case the verification fails. The verification also fails if the new interface is syntactically incompatible with the old one. In both cases, a reintegration can be attempted. The reintegration attempts to generate new solutions to the queries using the updated components. If this succeeds, the system is again ready for use. If this fails, the changes are too substantial for Poplar to overcome, and the system must be redesigned again.

From this figure, we can identify three functions that a Poplar compiler should perform: method contract verification, generation of integration links and integration link verification. Our implementation performs the former two (see Chapter 5). In addition, we describe how integration link verification could be implemented as a future extension.

2.4.10 Modularity of analyses and transformations

Java supports local typechecking and compilation by design. This feature is essential to its success as a language for scalable software development, which potentially involves

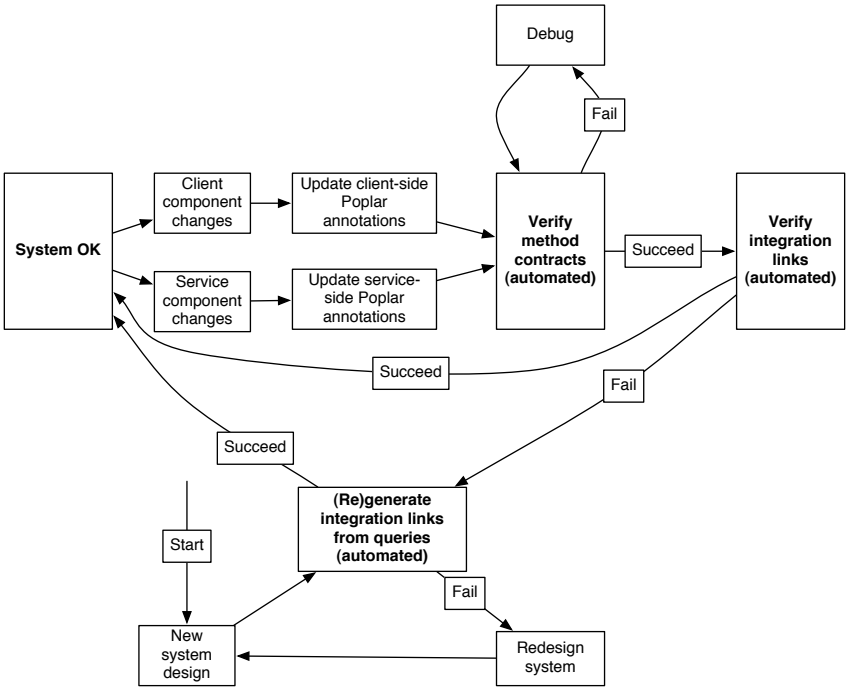


Figure 2.11: The Poplar software development workflow

large teams and many different libraries in a given software system. When a source file changes, it is sufficient to inspect the source code of only that file (typically a single class or interface), and to inspect compiled type signature information of any type directly referenced by that file, in order to recompile it. Unlike in languages like C and C++, there is no mechanism for directly including a source file in another.

Since we would like Poplar to act as a preprocessor for Java, we require that all of our analyses and transformations should work, in this regard, in the same way as Java compilation. That is, we would like to be able to compute analysis results for a source file based on the source code in that file only, as well as precompiled summary information for any files directly referenced by that file. Poplar makes use of the following analyses.

Generation of integration links. The principal Poplar integration method is replacement of declarative queries by solutions to them, which are found by a search algorithm.

Checking of integration links. When classes that were used as part of a particular integration link have changed, and when these changes are visible in their API, including in the Poplar annotations of said API, it is necessary to check that the integration link is still valid with respect to the new version of the API. If it is not, then it needs to be regenerated.

Mutation checking. Every Poplar method is annotated with a mutation summary, which lists the abstract resources that may potentially be mutated by that method.

In this analysis, we must ensure that the mutation summaries do not omit any mutation. That is, if method m1 calls method m2, and method m2 mutates resource r, then we must ensure that m1 also reports the mutation of r.

Label checking. For every invocation of a Poplar method, we must make sure that the necessary labels of the receiver and the arguments have been set up prior to the invocation, if the method declares preconditions.

We will see in Chapter 4 that all of these analyses can be performed locally for each method by using summary information about methods from other classes. The summary information that is needed is the Poplar signature of each method and field being referenced in other classes. This is analogous to Java compilation, where the type signature for each method and field that is being referenced from the current source file is needed. In theory, a Poplar compiler can easily store Poplar signatures in Java class files without breaking compatibility with existing tools, since class files support any number of nonstandard attributes.

2.4.11 Summary

Poplar adds several features to the Java language. The most fundamental idea is a labelling of variables according to their valid uses with various APIs. Some labels, called tags, are immutable, while properties are typestate-like in that they may be established and erased during program execution. This labelling provides the basis of the flexible approach to component interdependencies that we introduce: as API:s change structurally, labels stay the same if there has been no functional change to a particular value, and if there has been a functional change, the labels may also change, correctly giving rise to a compilation error. An effect system inspired by Boyland and Greenhouse's abstract regions describes how properties may potentially be erased, allowing us to conservatively track, for each variable, the set of properties that are live at each point in the program. Methods have mutation summaries that describe which regions they may mutate on which variables, from which we are directly able to infer the set of properties that may be erased. A simple uniqueness system that tracks whether references are unique and whether they will remain unique is introduced to help increase the precision of the effect system. These features give rise to some subtle complications when formalised, which we will discuss in the following chapter.

3

Language Reference

In this chapter, we discuss the various language elements of Poplar in detail. We seek to provide a detailed understanding of Poplar semantics through natural language descriptions. The following chapter use this chapter as a basis in order to describe in detail, through a formalisation, how the Poplar checking and compilation mechanisms work.

In general, we only specify those language elements that are novel in Poplar. The other language elements already exist in Java, and we discuss them in so far as is necessary to understand Poplar. The Java language specification may be consulted for those details that we cannot provide here.

Along with the concepts in this chapter, we will also introduce the associated syntax. Sometimes we use horizontal lines, such as $\overline{x}$. In general, this means a repetition in the form $x_1, \ldots x_n$. We will treat $\overline{x}$ as a set when necessary, so we may write, for instance, $x_j \in \overline{x}$.

The specification described here and in the next chapter is based on the Java core calculus known as Middleweight Java, or MJ. MJ is constrained in several respects, among them that it has no interfaces, abstract classes, generics, primitive types, arrays or exceptions. Accordingly we will not specify Poplar with respect to such language features.

3.1 Syntax

The version of Poplar that we discuss here, and that we also will formalise in the following chapter, is called Poplar_0. We first give its syntax in full and then discuss individual parts in detail. Note that Poplar_0 assumes that each method and constructor has exactly one argument, but sometimes we will give example that have several. It is straightforward to generalize Poplar_0 to any number of arguments, at the cost of some additional complexity. Also note that even though Poplar_0lacks primitive types, we will discuss how Poplar should treat primitives.

Program

P ::= $cd_1 \ldots cd_n : \overline{s}$

Class definition

cd ::= class C extends C
$\{fd_1 \ldots fd_k \; cnd \; rd_1 \ldots rd_j$
$md_1 \ldots md_n\}$

Field definition

fd ::= $C \; f[: (\overline{[@p_1, \; \ldots \; @p_n \rightarrow] \; l_1, \; \ldots \; l_k})]$;

Resource definition

rd ::= resource r{properties $@p_1, \; \ldots @p_n$;
$fd_1 \ldots fd_k\}$

Constructor definition, method definition

cnd ::= $C \; (C_a \; x_a)[\rho :][ls]\{\text{super}(e_a); s_1 \ldots s_n\}$
md ::= $\tau \; m(C_a \; x_a)[\rho :][ls]\{s_1 \ldots s_k\}$

Mutation summary, qualified resource

ρ ::= mutates $qr_1, \; \ldots qr_n$
qr ::= $r \mid \text{any}(C).r \mid x.r$

Label signature, label condition, label

ls ::= $\overline{x : \; lc_1, \; \ldots \; lc_n};$
lc ::= $++ @p \mid +l \mid l \mid -@p \mid U$
l ::= $t \mid @p$ — Tag, property

Uniqueness

U ::= fresh | unique | maintain

Return type

τ ::= $C \mid$ void

Expression

e ::= $x \mid$ null $\mid e.f$ — Variable, null, field access
| $(C)e$ — Cast
| pe — Promotable expression

Promotable expression

pe ::= $e.m(e_a) \mid$ new $C(e_a)$ — Method invocation, object creation

Statement

s ::= ; — No-op
| pe; — Promoted expression
| if $(e == e)\{s_1 \ldots s_k\}$ else
$\{s_{k+1} \ldots s_n\}$ — Conditional
| $e.f = e$; — Field assignment
| $C \; x[: U]$ — Local variable declaration
| $x = e$; — Variable assignment
| return e; — Return
| $\{s_1 \ldots s_n\}$ — Block
| $x =$ #produce$(C, U, l_1, \; \ldots \; l_n)$; — Produce query
| #transform$(x, l_1, \; \ldots \; l_n)$; — Transform query
| drop $l_1, \; \ldots \; l_n$; — Drop labels

3.2 Labels

Label			
l	::=	$t \mid @p$	Tag, property

Labels are either *tags* or *properties*. In both cases, they are identified by a unique name declared inside a particular class: the label $C.l$ is different from $C'.l$. Resolution of labels functions in principle in the same way as resolution of other Java class members, so package names are also respected: $pa_1.pa_2.C.l$ is different from $pa_1.pa_3.C.l$, where pa_i are packages.

Poplar tracks the label sets of values (primitives or objects). Each value has a set of labels at any given time. Poplar is allowed to underestimate the set of labels associated with a value, but overestimating it is not allowed.

All labels can be used to model protocol states. For a type that can potentially have n different labels, a value can be considered to be in one of 2^n states, where each state is a set of labels. It is possible to constrain the state space so that not all of these combinations are permitted for a given type. We note here a difference with typestate systems: typestates are usually atomic labels, but Poplar states are sets of labels. However, some typestate systems model explicit substates, which are not included in Poplar. It is possible to simulate sub-labels without including them as a first class concept.

All labels have a *temporal contract*, which refers to the way that they are used to model temporal states as sets of labels, and an *external semantic contract*, which refers to what the user expects from a label. Users who write queries are expected to have some idea about the meaning of labels that they request. This meaning can be communicated formally or informally, just as Java API semantics can be communicated to programmers formally or informally today. We believe that the notion of a label with an external semantic contract, which users are expected to understand, is something that has not previously been studied in typestate research.

3.2.1 Tags

Tags are the simplest form of labels. They are not associated with any mutable state on the value they are associated with. They are intended for describing irreversible side effects and for identifying constants. Tags have a well defined point of establishment but no point of destruction.

3.2.2 Properties

Properties are, in a sense, a generalisation of typestates. In addition to having a temporal contract and an external semantic contract, they also have an *internal semantic contract*. This is a concrete predicate on the internal state of some object. It makes no sense for primitive values to have properties, since their state cannot change. The predicate that a property corresponds to is not defined explicitly, and it is up to the programmer who specifies it to establish it correctly in methods that claim to do so. Properties have both well defined points of establishment and well defined points of destruction. The point of establishment is always a basic establisher method, and the point of destruction is always a resource mutation. We will define these concepts in the following sections.

3.2.3 Example

In the following example, the class Socket has the properties @raw, @bound and @open.

```
1  class Socket {
2    tags(byte[]) received;
3
4    resource state {
5      properties @raw, @bound, @open;
6      SocketAddress boundTo;
7
8      Socket() this: ++@raw. { ... }
9      void bind(SocketAddress bindPoint) this: -@raw, ++@bound. {
10       this.boundTo = bindPoint;
11       //...
12     }
13     void connect() this: -@bound, ++@open. { ... }
14     void send(byte[] data) this: @open. { ... }
15     byte[] receive() this: @open; result: ++received. { ... }
16   }
17 }
```

The external semantic contract of @open is that the socket is ready to send and receive data. The external semantic contract of @raw and @bound is that the socket is not ready for this. These notions are communicated informally to the user.

The temporal contracts of these properties are simply the set of valid method sequences. For example, it is valid to invoke connect after bind, but it is not valid to invoke send unless connect has been invoked first. Just like in typestate checking, these sets form a state machine. The temporal contracts help enforce the external semantic contract in this case, though in general they cannot capture it fully. For example, the tag received for byte[] has the external semantic contract that the data has been received on a network socket. This fact cannot be enforced automatically.

The property @bound has the internal semantic contract that boundTo will initialised and correspond to the currently bound address. This contract is established and maintained manually by the class designer.

3.3 Resources

Resource definition

rd ::= resource r{properties $@p_1, \ldots @p_n$;
$fd_1 \ldots fd_k\ md_1 \ldots md_k\}$

Label condition, label

lc ::= $++@p \mid +l \mid l \mid -@p$

l ::= $t \mid @p$ Tag, property

fd: field definition, md: method definition, r: resource name

Abstract resources group related state and properties. The concrete data members of a class that implement the property predicates of some set of properties may be declared together with the properties themselves inside a resource. A property may be thought of as a configuration of a resource in some class, rather than a configuration of the class itself (however, properties are also allowed to extend their internal predicate to state that is not part of any resource).

For a given class, each property must be declared inside exactly one resource. This establishes an unambiguous mapping from properties to resources.

Resources express label destruction: methods may be annotated with lists of mutated resources, which gives an upper bound on the properties that may be lost as a result of executing a given method. This is essentially a side effect summary. Resources are also used to partially restrict mutation of an object's state. Fields declared inside a resource are part of that resource's concrete state. These fields may generally not be changed by methods that have not declared that they do so.

3.3.1 Resource access levels

Any given method may have one of three different access levels to a given resource.

No access. This is the case if the method does not declare that it mutates the resource. The method may not change the resource's concrete state directly, and it may not invoke any other method that may be deemed to mutate this resource. The method can be considered to have no side effects at all with respect to the given resource.

Simple access. The method declares that it mutates the resource, but it has no annotations of the form $+ + p@$ in its contract (where $@p$ is a property). The method is allowed to invoke other methods that mutate the resource, but it cannot mutate the concrete state directly.

Raw access. The method declares that it mutates the resource, and it has at least one annotation of the form $++@p$ in its contract. This method must directly establish the internal predicate of such properties, so it is allowed and expected to directly mutate the concrete state. It is also allowed to invoke other methods that mutate the resource. This is the strongest form of access. The method designer has full responsibility for the method's body corresponding to its contract.

3.3.2 Resource mutations

A resource is considered to be mutated when one of the following things occur:

- An unconstrained resource field (to be defined) in that resource is written to
- A method that claims to mutate the resource is invoked
- Labels of a constrained resource field (to be defined) are lost to such a degree that the owning object also loses labels

If a method invocation may lead to any of the above cases occurring, it must report a corresponding mutation.

3.3.3 Implicit mutations

If a method is declared inside a resource, it implicitly has simple access to the resource. We call this an implicit mutation. Methods declared outside the resource must declare any mutation of it.

3.3.4 Example

Please see the following section for an example.

3.4 Fields

Field definition

fd ::= $C\ f[:\ (\overline{[@p_1,\ \ldots\ @p_n \rightarrow]\ l_1,\ \ldots\ l_k})];$

f: field name

Poplar supports three kinds of fields: *plain fields*, *unconstrained resource fields* and *constrained resource fields*. Plain fields are not associated with any resource, and mutations or writes on them are untracked (unless there are aliased mutations). Resource fields are associated with some resource.

3.4.1 Plain fields

Plain fields are declared outside of resources and have no checked relation to them, although they may still be used explicitly by programmers to help implement properties.

3.4.2 Unconstrained resource fields

These are fields that are associated with a resource but without any constraints. Writes and mutations are tracked and interpreted as mutations of the corresponding resource.

3.4.3 Constrained resource fields

Fields can be constrained to indicate that their state depends on the state of the owning object. Sets of labels of the owning object may be associated with sets of labels on the field. Because this constraint may apply recursively, it is possible to indirectly constrain a deep hierarchy of fields through the top level object.

The labels of constrained fields can only be changed by methods that belong to their owning object. This is necessary in order to enable encapsulation and modular compilation; otherwise all methods of all classes would need to know about the full constraints of all fields of all other classes. This restriction is enforced through the mechanism of *acceptable mutations* (to be defined).

Constrained fields are implicitly unique. If they could be aliased, the mutation of any object of some compatible type could potentially violate the constraints of many different objects in remote heap locations.

The labels of a constrained field can never be violated with respect to the owning object's state. When a value is first assigned to the field, it must have the correct initial label set. When constrained labels are lost, the corresponding state of the owning object are lost simultaneously.

A resource designer may want to choose to use a constrained resource field, rather than an unconstrained one, because they want to make the field available for use in Poplar solutions, but they want to link the field's labels to the owning object's state. On the other hand they are also less free to use such a field as they wish because of the constraints. An unconstrained resource field gives more flexibility in its usage, and have the benefit that mutations will be tracked properly, but cannot as easily be integrated into solutions. Plain fields give full freedom but cannot be used in solutions at all.

3.4.4 Example

In the following example, the class Message is using sockets to receive messages.

```
1
2  class Message {
3    int receiveCount = 0;
4    resource data {
5      properties @tsSet, @msgSet, @reset;
6
7      int timestamp: ((@tsSet) -> (msgTimestamp));
8      String message: ((@msgSet) -> (msgValue));
9      int msgId;
10
11     public Message(int timestamp, String message)
12     timestamp:msgTimestamp; message:msgValue;
13     result: ++@tsSet, ++@msgSet. {
14       this.timestamp = timestamp;
15       this.message = message;
16     }
17
18     void receiveFrom(Socket s) s: @open; this: ++@tsSet, ++@msgSet. {
19         timestamp = readTimestamp(s);
20       message = readMessage(s);
21       msgId = readMessageId(s);
22       receiveCount += 1;
23       }
24
25     void reset() this: ++@reset. {
26       msgId = -1;
27       timestamp = -1;
28       message = null;
29     }
30   } //End of resource data
31
32   //Explicit mutation, since declared outside resource
33   void indirectReset() mutates this.data: {
34     reset();
35   }
36
37   //Read data from the socket
38   int readTimestamp(Socket s) result: ++msgTimestamp. { ... }
39   String readMessage(Socket s) result: ++msgValue. { ... }
40   int readMessageId(Socket s) { ... }
41 }
```

The fields timestamp and message are constrained resource fields. If the owning object, of type Message, has at least the property @tsSet, then these are guaranteed to have at least the labels msgTimestamp and msgValue, respectively. When these fields are assigned, in the constructor and in the receiveFrom method, they have sufficient labels to uphold this constraint. Note that the constructor, the receiveFrom method, and the reset method are all declared inside the resource data. This means that they mutate it implicitly. Thus, the properties @tsSet and @msgSet are lost by the reset method. For this reason, reset is allowed to write values freely to msgId and timestamp. since they have no constraints when these properties have been lost. The fields' state by the end of this method will be consistent with our expectation.

The field msgId is an unconstrained resource field, so methods that write to it must have raw access to the resource data. reset has raw access, because of its ++@reset annotation, so the write is permitted. However msgId is never constrained in terms of what labels it should have.

The method indirectReset must report that it mutates this.data since it is declared outside the resource data, but invokes reset, which has raw access.

The field receiveCount is a plain field, which is entirely unconstrained, and any method in the class may write to it.

3.5 Expression contracts

A *label signature* and a *mutation summary* together fully describe the preconditions and effects of a statement or expression. We first describe these two concepts independently, and then we describe operations that apply to them.

3.5.1 Label signatures

Label signature, label condition

$$
\begin{array}{lll}
ls & ::= & x_1 : lc_{1,1},\ \ldots\ lc_{1,i};\ \ldots \\
 & & x_n : lc_{n,1} \ldots\ lc_{n,j}; \\
lc & ::= & ++@p \mid +l \mid l \mid -@p \mid U
\end{array}
$$

Label signatures are declared together with the methods that they apply to, and also computed for each expression as part of method checking. They describe those label set changes that occur as a result of invoking the method or evaluating an expression, and that are not simple destructions. Simple destructions are captured by resource mutations (described in the following section). Label signatures may contain the following entries.

Direct addition, basic establisher A direct addition is denoted by $++l$ for some label l. It indicates that the method is directly establishing a property by changing the associated resource state. The programmer guarantees that this happens and takes responsibility for it being done correctly. A method annotated with $++l$ for some label l is called a *basic establisher* for that label.

Addition An addition is denoted by $+l$. It indicates that the method is indirectly establishing a property by invoking another method that is annotated with either $+l$ or $++l$. Thus, $+l$ ultimately corresponds to the invocation of some method annotated with $++l$, save for the case of infinite recursion (which we cannot detect). It is not mandatory to report all the labels established by a method.

Invariant An invariant is denoted simply by l. It indicates that the method needs the label as a precondition and that it will remain intact following the method's invocation. If a label is erased and established again, the invariant is not satisfied.

Subtraction A subtraction is denoted by $-l$. It indicates that the method needs the label as a precondition and that the label will be lost following the method's invocation. Precondition labels that are lost and established again are also expressed as $-l$: we do not distinguish between these two cases.

Direct additions are only included in label signatures that are part of method declarations. Label signatures for general expressions and statements, which are computed by the compiler rather than written manually, only have additions, invariants and subtractions.

Label signatures may be thought of as providing a lower bound on labels that an expression or statement establishes. Invariants must be reported exactly, and subtractions may be over-reported (although there is no benefit in doing so).

3.5.2 Mutation summaries

Mutation summary, qualified resource

ms ::= mutates $qr_1, \ldots qr_n$

qr ::= $r \mid \mathsf{any}(C).r \mid x.r$

r: resource name

Mutation summaries are sets of mutated resources, together with a mutation subject. Mutation subjects can be of the form any(C), which signifies any value of type C, or x, which is a single variable. Like label signatures, mutation summaries are declared together with the methods that they apply to, and also computed for each expression as part of method checking. A mutation of any(C).r is interpreted as removing all properties in resource r from values of type C or a subtype. A mutation of $x.r$ is interpreted as removing those properties from the variable x. In both cases, properties that belong to the addition, invariant or direct addition sets of any associated label signature are exceptions from this removal.

Mutation summaries may be thought of as providing an upper bound on labels that are lost as a result of evaluating an expression or statement.

3.5.3 The chaining operation

The chaining operation is used to compose the contracts of two expressions when one is executed before another, yielding an overall contract that describes the execution of both. This can be used in statement sequences such as s1; s2 to describe s1 executing before s2, but also within expressions such as a method invocation e0.m(e1, e2), where e0 is evaluated first, then e1, and finally e2, before the method m itself can be invoked.

The formal definition of the chaining operation is given in Section 4.2.1. Here we would simply like to give an intuition for how the operation works and what it is supposed to do. We use the symbol LS to refer to a label signature, and ρ to refer to a mutation summary. Given a first contract (LS_1, ρ_1) and a second contract (LS_2, ρ_2), we would like to compute the resulting overall contract, (LS, ρ). We use the symbol $\oplus$ to denote the chaining operation: $(\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}_2, \rho_2) = (\mathsf{LS}, \rho)$.

In order to compute the resulting contract, we must compute LS and ρ. ρ is the simplest: it is simply the union of ρ_1 and ρ_2. There are some special cases involving renaming of values (such as writing to fields and to variables), which make this slightly more complicated, and we will return to this later. In order to compute LS, we need to compute additions, invariants, and subtractions. We may note the following.

- If the postconditions of the first contract satisfy some preconditions of the second contract, those preconditions can be removed from the overall result, since they do not need to be provided externally. Otherwise they must be included as preconditions (invariants or subtractions) in the overall result.

- If the second contract needs a precondition that the first contract does not provide as a postcondition, and the first contract also mutates a resource that this label

is in, then there is no possibility of passing the label from outside, through the first contract and into the second contract. The label would always be lost prior to its intended use. In this case the second contract's precondition can never be satisfied, and it makes no sense to chain the two contracts in this situation. The expressions that these two contracts correspond to should not be evaluated in sequence.

- Additions from the first contract that are not erased by the second contract may be viewed as additions of the overall fragment.
- Preconditions of the first contract are always preconditions of the entire fragment.

We refer to Section 4.2.1 for the full details of this operation.

3.5.4 The alternation operation

Alternation is used to compose contracts when either one of two possibilities will be executed, but we do not know which one. The only application of this operation is in if-statements, where we cannot know statically which branch will be taken. Clearly, if the contracts of two alternate branches differ greatly, the precision of the overall analysis will be poor. The definition is given at the end of Section 4.2.1.

3.5.5 Subsumption of contracts

Given two contracts (LS_1, ρ_1) and (LS_2, ρ_2) we define the subsumption relation:

$$(\mathsf{LS}_1, \rho_1) \prec (\mathsf{LS}_2, \rho_2)$$

This relation is true whenever the first contract can take the place of the second: in method overriding, in query solving, in method body checking, and so on. The definition is given in 4.2.5. This can be seen as a formalisation of the *substitution principle* in subtyping. For example, if $(\mathsf{LS}_1, \rho_1) \prec (\mathsf{LS}_2, \rho_2)$, then the former contract must have the same or fewer mutations, it must establish the same or more labels, and it must assume the same or fewer labels.

3.6 Uniqueness of references

Uniqueness

U ::= fresh | unique | maintain

In order to reason about aliasing, we constrain references according to their *uniqueness kinds*. We distinguish between *dynamic aliases*, which are created on the stack as a result of method invocation (for instance, as return values or method arguments) and *static aliases*, which are stored on the heap, in fields. For Unique references, at most one static alias exists at any one time. For Normal references, any number of static aliases may exist. Dynamic aliases may always be created, but they are constrained to avoid exposing two Unique references that actually point to the same object.

Uniqueness kinds can be used to gain additional precision in the analysis and solution finding done by Poplar. In particular, mutations are defined with respect to the

uniqueness kinds of method receivers and arguments. Unique mutations need not be reported globally, but non-unique mutations potentially have a global effect across the entire program. At compile time, Poplar checks the correct use of uniqueness kinds, and incorrect creation or use of aliases and references will be trapped as an error. The user may want to change the uniqueness kinds of some variables when such errors are caught, or when no solution can be found for a query due to excessive mutations.

The uniqueness kinds are as follows.

Fresh References to new objects are always Fresh . We enforce this when reasoning about constructor invocations. Accordingly, constructor bodies may not create any static aliases of the object that is being initialised. When fresh references are assigned to a variable or field for the first time, they must be converted into some other uniqueness kind. Any of the other three kinds are valid.

Unique Unique references have at most one static alias. A unique reference cannot be assigned to a field or a variable, although it can be used as an argument or receiver in method invocations and constructor invocations. Unique fields and variables must initially be populated by Fresh references.

Maintain Maintain references may be either Normal or Unique . That is, static aliases may exist. To preserve the uniqueness guarantee for unique, no new aliases are created. When a Maintain value is used, it is assumed to potentially have aliases. When method arguments or receivers are of the kind Maintain , they are polymorphic and may receive as input either Normal , Unique or Maintain references.

Normal Normal references have no special syntax and are assumed to potentially be aliased. They are entirely unconstrained in that we may always create new aliases of them.

When a method or constructor is invoked, the receiver (if any) and the method arguments are called the *input variables*. When we invoke a method, we check that any Unique expressions that are passed in are passed in only once. This guarantees that the receiving method will never see two Unique variables that point to the same object. This is called *non-repetition of unique inputs*.

When a Unique expression is obtained from an object x that is Normal or Maintain , either as the return value of a method invocation or as a field access, we conservatively assume that the obtained expression is of the kind Maintain . This is because the caller might be holding two different references to the same object x and access the same Unique value through it on two different paths. Conservatively treating such expressions as Maintain again prevents the danger of having two Unique expressions that point to the same object. Thus, the only way to access a Unique expression as Unique through another object x is if x itself is Unique . This is called *returned uniqueness rewriting*.

Mutation summaries of methods that take input variables of the Maintain kind are different for each call site, depending on the input variables. If a method has an input variable x of type C and an associated mutation $x.r$ for a resource r, then the mutation is reported as any(C).r when the variable passed in is Normal or Maintain , and simply as x.r when the variable passed in is Unique or Fresh . This conversion is performed at the call site when the caller's method body is checked.

3.6.1 Example

The following example shows basic permitted and forbidden use of uniqueness kinds.

```
1  class C {
2    Object field;
3    void start(Object u, Object m, Object n)
4      u: unique; m: maintain. {
5      toUnique(u);
6      toMaintain(u);
7      toNormal(u); //Forbidden
8
9      toUnique(m); //Forbidden
10     toMaintain(m);
11     toNormal(m); //Forbidden
12
13     toUnique(n); //Forbidden
14     toMaintain(n);
15     toNormal(n);
16   }
17
18   Object makeFresh() result: unique. {
19     Object f = new C();
20     return f; //Fresh becomes unique
21   }
22   void toUnique(Object u) u: unique. {
23     this.field = u; //Forbidden
24   }
25   void toMaintain(Object m) m: maintain. {
26     this.field = u; //Forbidden
27   }
28   void toNormal(Object n) {
29     this.field = n;
30   }
31 }
```

3.6.2 Example 2

Suppose that the socket class from earlier in this chapter is augmented as follows.

```
1  class Socket {
2    tags(byte[]) received;
3
4    resource state {
5      properties @raw, @bound, @open;
6      SocketAddress boundTo;
7
8      Socket() this: ++@raw. { ... }
9      void bind(SocketAddress bindPoint) this: maintain, -@raw, ++@bound.
           {
10       this.boundTo = bindPoint;
11       //...
12     }
13     void connect() this: maintain, -@bound, ++@open. { ... }
14     void send(byte[] data) this: maintain, @open. { ... }
15     byte[] receive() this: maintain, @open; result: ++received. { ... }
16   }
17 }
```

Each method now indicates, using the keyword maintain, that it does not create any aliases of the receiver. Now consider the following client class.

```
1  class SocketUser {
2    void close(Socket s) mutates any(Socket).state: { s.close(); }
3    void closeUnique(Socket s) mutates s.state: s: unique. { s.close(); }
4    void closeMaintain(Socket s) mutates s.state: s: maintain. { s.close
         (); }
5
6    Socket s;
7    void setSocket(Socket s) s: maintain. { this.s = s; //Forbidden }
8    void setSocket2(Socket s) { this.s = s; //OK }
9    String getAddress(Socket s) result: unique. {
10     return s.remoteHost; //Forbidden, result must be maintain
11   }
12   void sendBoth(byte[] data, Socket s1, Socket s2) s1: unique, s2:
         unique. {
13     s1.send(data);
14     s2.send(data);
15   }
16   void doSendBoth(byte[] data, Socket s) s: unique. {
17     sendBoth(data, s, s); //Forbidden
18   }
19 }
```

The method close must report the mutation of s.state as any(Socket).state, because s is taken to be Normal . The methods closeUnique and closeMaintain need only report a mutation of s.state. The method setSocket is trying to create an alias of s by assigning it to a field, but this is forbidden, given that it has the kind maintain. On the other hand, setSocket2 is allowed to perform this assignment.

The method getAddress is reporting a unique return value, but the value returned actually cannot be unique, since it is obtained through a non-unique intermediate reference s. It must be reported as maintain.

The method sendBoth is invoking send on both of its arguments, which are declared to be unique. When doSendBoth invokes this method, it is passing the argument s twice for unique arguments to the same method. This is forbidden, as it creates the illusion of two unique references which are actually references to the same object. All input variables that are unique and maintain must have no repetition among themselves when a method or constructor is invoked.

3.7 Overriding and subclassing

3.7.1 Overriding of properties and resources

We previously mentioned that properties have external semantic contracts, internal semantic contracts, and temporal contracts. If external contracts are overridden in unexpected ways, this must be communicated to the user manually, as is normal. In general, these contracts should be the same or strengthened in subclasses. The overriding of temporal contracts is constrained by our definition of valid method overriding. Subclass methods must in general be more permissive than superclass methods with respect to any given label transition, so the contract can only be weakened (more permissive) in the subclass. When viewed as a state machine, this means that it would have more or identical transitions. Internal semantic contracts may be overridden in subclasses by using additional state, or by using stronger predicates on the same state. It is up to implementers to ensure that these predicates are provided and used correctly. Resources in subclasses may also be associated with additional state. This state can then be used to implement properties in the resource. It is not possible to remove some

field from a superclass resource or associate it with a different resource. Subclass resources may also have additional properties that were not part of the same resource in the superclass. It is not possible to move a property to a different resource when overriding.

Constraints on basic establishers

A common initialisation pattern for methods that establish some state in a class is what Fähndrich et al. called *sliding methods* [27]: methods that first call the corresponding superclass method, which is supposed to establish the superclass condition, and then establish additional state for the current *class frame*. Such methods are virtual, so an invocation always goes to the bottom frame at first, and then initialisation proceeds gradually from the top frame down. This pattern is a natural fit when the condition to be established is incrementally defined in subclasses. However, when using this pattern, the problem of *partially established conditions* occur. When only some class frames have been initialised and the method call has not yet proceeded all the way to the bottom frame, the condition is in an intermediate degree of establishment between not being present and being fully present. Fähndrich et al. solve this problem by reasoning explicitly about each class frame, and declaring for each method which class frames of some typestate it initialises. In Poplar, this problem applies to properties, but we wish to avoid the complication of reasoning about each class frame independently. Our approach is instead to view the property as not initialised until every class frame has been initialised.

We enforce this constraint by imposing several constraints on *basic establishers*, which are the methods that are annotated with $++\,@p$ for some property $@p$.

- Each basic establisher must invoke the superclass' corresponding method as its first statement, if it exists.
- The property $@p$ is considered not to be established by the invocation of the superclass' basic establisher, so it is not available when the subclass basic establisher is executing.
- Subclasses must override all basic establisher methods from superclasses.

We now show what this restriction means in practice using an example.

```
1  class Base {
2    resource r {
3      properties @p;
4      int i;
5      void makeP() this: ++@p. {
6        i = 0;
7      } }
8    void useP() this:@p { ... }
9  }
10 class E1 extends Base {
11   resource r {
12     int j;
13     void makeP() this: ++@p. {
14       super.makeP(); //mandatory!
15       j = 0; //stronger def.
16       useP(); //forbidden
17   } }
18 }
19 class E2 extends Base {
20   resource r {
21     String x;
22     void makeP() this: ++@p. {
23       super.makeP(); //mandatory!
24       x = ""; //different def.
25       useP(); //forbidden
26     } }
27 }
```

In this example, the base class defines a property @p. The classes E1 and E2 extend and redefine this property by adding more state to the resource r. Because makeP is a basic establisher, it is forced to call the superclass establisher as its first statement in the classes E1 and E2. It is forbidden to make use of @p within this basic establisher, as we do here in E1 and E2, since there might be other superclasses that we do not yet know about. *Basic establishers are not allowed to use the property they establish.*

3.7.2 Overriding of methods

For an overridden method to be valid, in general, it must have a compatible method contract. Preconditions must be weakened or identical, and postconditions must be strengthened or identical. Mutations must be the same or weakened. We express this in terms of the subsumption relation $\prec$: if method m_1 has the contract (LS_1, ρ_1) and method m_2 has the contract (LS_2, ρ_2), and m_2 overrides m_1, then for the overriding to be valid, we need to have that $\mathsf{LS}_2 \prec \mathsf{LS}_1$ and $\rho_2 \prec \rho_1$. See Section 3.5.5.

In terms of labels, this has the natural meaning: methods must expect the same labels or fewer, and they must establish the same labels or more, compared to the method they override. In terms of mutations it is slightly more subtle: $\mathsf{any}(C).r$ is a valid overriding of $\mathsf{any}(C').r$ if we have that $C \prec C'$, for instance.

3.8 Method Checking

Method checking is the process of checking that each method and constructor body is valid with respect to its contract. This checking is modular and is carried out by inspecting local source code and the contracts of other methods that are invoked and fields that are accessed. In principle this has two parts: *uniqueness checking*, which checks

that references and aliases are handled correctly with respect to uniquenesses, and *label checking*, which checks that labels and mutations are handled correctly. These two aspects can be checked separately or together, although uniqueness information must be present when label checking is carried out. In our implementation uniquenesses are checked separately beforehand (see Chapter 5).

Uniqueness checking is simply checking that field and variable writes are carried out in accordance with the rules specified in Section 3.6. In the remainder of this section we will discuss label checking. Label checking is carried out by computing the expression and statement contract for each expression and statement (see Section 3.5. With the exception of the if-statement, the expressions are combined together *in the order that they are executed*, by using the chaining operation (Section 3.5.2). For if-statements, the alternation operation is used instead. In this way, an overall contract for the method body is obtained. In principle, the method body is a single statement, since in our formalisation, a sequence of statements is also a statement. When this single contract has been obtained, it is checked against the *prior and posterior expanded signatures* of the method to verify that labels are handled correctly.

3.8.1 Method body checking

Method and constructor bodies are checked to verify that they are Poplar type safe. Poplar type safety is intended to guarantee that each time a value is assumed to have some set of labels, it has at least those labels. That is, we wish to guarantee that those labels have been established at some point before being used, and they have not been destroyed between the point of establishment and the point of use. By extension, this guarantees that the internal semantic contract (predicate) of each property is upheld, that temporal contracts (protocol rules) are not being violated, and that external semantic contracts are being upheld. This final point also relies on the meaning of external semantic contracts being communicated properly from developers to users, something that we cannot check mechanically.

A method body is said to be valid if there is at least one label assignment that is provided at the start at the method and that leads to all label uses throughout the method body being satisfied in the manner described above, and that also leads to the method satisfying its postcondition with no resource mutations beyond what has been reported.

Poplar must track values as they flow between variables, fields and methods. There are several ways that a value (primitive or object) can be copied or renamed: through a method or constructor invocation (as a receiver or parameter), as a return value, and through a field write or a variable write. When we check an expression that performs such a renaming or overwriting, we first compute the contract of the inner expression (which is being assigned to something), and then rewrite this contract so that the relevant labels and mutations are re-bound to the expression's new name. Information about the labels that were previously associated with the left hand side of a write, if any, is lost, just as the previous value of a variable is lost when it is overwritten.

Method invocation

Constructor definition, method definition

cnd ::= $C\ (C_a\ x_a)[\rho :][ls]\{\mathsf{super}(e_a); s_1 \ldots s_n\}$

md ::= $\tau\ m(C_a\ x_a)[\rho :][ls]\{s_1 \ldots s_k\}$

Promotable expression

pe ::= $e.m(e_a) \mid \mathsf{new}\ C(e_a)$ Method invocation, object creation

When a method invocation is checked, the associated method type is looked up. The method type describes a prototypical contract of the method (label signature and mutation summary). The concrete receiver and parameters are substituted into this contract to obtain the resulting contract of the invocation. Uniqueness checking (Section 3.6) is also performed.

Acceptable mutations

Throughout the checking of a method body, the states of local fields and variables only are being tracked. Fields in other objects are not being tracked, for instance. We only permit mutations where we can track the effects. For example, statements are not permitted to mutate fields in other objects, or return values from methods that have been invoked (unless they are Fresh , which means that the caller may take over control of them), since these cannot be tracked. Permitting these mutations might lead to object configurations becoming inconsistent. We call this limitation *acceptable mutation checking*.

If-statements

For branches such as if-statements, we cannot know statically which branch will be taken. Thus, we conservatively compute the effects of executing either one of these statements. For example, we consider the union of mutation summaries, but the intersection of added labels, the union of subtracted labels, and so on. The precision of this analysis is clearly very low if the two branches are very different.

3.9 Queries and query solving

Statement

s ::= ;

$x = \#\mathsf{produce}(C, U, l_1, \ \ldots\ l_n);$	Produce query
$\#\mathsf{transform}(x, l_1, \ \ldots\ l_n);$	Transform query

Queries have two forms: the produce-query and the transform-query. Both are treated as statements, so it is not possible to assign a query to a variable, for instance. The assignment in x = #produce... is part of the query itself.

The produce-query is requesting a variable of a given type, with (possibly) a given uniqueness, and with a given set of labels. The resulting variable may be a new object, or it may be an existing value that was obtained from something that happened to be available in the query context. There is no way to force the production of a new object.

The transform-query is requesting that additional labels should be associated with an existing variable.

Solutions to queries are sequences of statements that are found by the compiler at compile time. These solutions are substituted for the queries. Each statement in a solution is either a field access, a method invocation, or a constructor invocation. Notably, no branches or loops are generated. Consider a poplar method m, with method body mb that contains queries $q_1, \ldots q_n$, and suppose that solutions $s_1, \ldots s_n$ have been found. These solutions are valid with respect to the method m if and only if the method body remains valid with respect to its contract when the solutions have been inserted in place of the queries. (Section 3.8).

When method body checking is carried out, queries are assumed to already have minimal solutions that satisfy them. That is, it is assumed that they do what they request. When a concrete solution is substituted later it will have been found in such a way that it does not violate the contract that was assumed when the method body was checked.

Poplar's semantics and formalisation are mostly independent from the precise algorithm that is used to find solutions for queries. We describe how we use Partial Order Planning to find solutions in Section 5.9.

3.10 External resources

Often it is convenient to think about an object as having a property, even though the concrete state of that property is in a different object. For instance, it is convenient to think of a list item as having the property of "being in a list" even though the item itself does not have the state that controls this - the list object does. In order to model this situation, we introduce *external resources*. Properties in such resources we call *external properties*. We previously described labels in terms of three contracts: the external semantic contract, the internal semantic contract, and the temporal contract. Properties in external contracts can be thought of as having *an internal semantic contract for a different value* than the value that the other two contracts apply to.

We call the object having the internal contract the *host object*, and the object having the other two contracts the *hosted object*. These two may be of the same type, or they may be of different types. This allows us to models constructs such as lists and containers. An example is given in Figure 3.1. The class Cell is the host class, and the class Item is the hosted class. An Item object can obtain the property @inCell by being inserted into the cell using set(Item).

There is a fundamental source of inaccuracy in external resources. Consider the example in Figure 3.1. The class Cell provides a property @inCell for the external class Item. When we write to fields of the resource, they are identified as raw(ext(Item)) by our analysis. But there is no way to identify the precise external variable of type Item that this mutation "belongs" to, and we leave it to the programmer to identify explicitly, in the mutation summary, which variable (or possibly any(Item)) has had an external resource mutated. This is because there is not necessarily an obvious way to link such a mutation to an operation involving a specific reference, unlike for normal resources, where the mutation is always on the this object. Thus, in this example, remove and remove2 have different, but valid, mutation summaries, even though they contain the same code.

It should be noted that even though we cannot identify the object that has its external resource mutated, we can insist that the programmer identifies *some* object (re-

```
1  class Cell {
2    resource[Item] data {
3      properties @inCell;
4      Item i;
5    }
6    void set(Item in) mutates any(Item).ext(Cell).data:
7      in: ++@inCell. {
8      i = in;
9    }
10   void remove(Item out) mutates out.ext(Cell).data: {
11     if (i == out) {
12       i = null; //identified here as raw(ext(Item))
13     }
14   }
15   void remove2(Item out) mutates any(Item).ext(Cell).data: {
16     if (i == out) {
17       i = null; //identified here as raw(ext(Item))
18     }
19   }
20 }
```

Figure 3.1: Identifying mutations of external resources

source subject in our terminology above). The mutation cannot pass completely unnoticed. Future extensions, such as ownership systems, may lead to a way of increasing our precision here.

We specify external resources formally in an extension of Poplar_0, which we call Poplar_1. This is described in Section 4.3.

3.11 Concluding remarks

This chapter has discussed the semantics of Poplar informally and by example. The following chapter will formalise the intuition given here and define precisely how Poplar checking is performed.

4
Formalising Poplar

So far we have informally shown the concepts that make up the Poplar Java language extension. However, without precise definitions there can be no possibility of implementing or fully understanding the subtleties of a programming language. In this chapter we formalise Poplar in detail. The semantics of Poplar are described informally in Chapter 3. It is recommended to read this description, at least in part, before reading the present chapter.

Often, a full specification of a statically typed programming language involves the following items.

Syntax. The syntax defines all the possible syntactic forms of the language.

Typing judgments. Syntactic forms are often built as composites of other syntactic forms; for instance, in MJ, a field write `x.f = e` relates the variable x, the field f and the expression e to each other. Typing judgments define, for each syntactic form, what relationships the types of the subexpressions must have to each other in order for the overall expression to be well-typed.

Type safety proof. A type safety proof shows, usually by induction, that during the course of its evaluation, a well-typed program cannot in some way go wrong. The two parts of a classical type safety proof are the *progress lemma* and the *preservation lemma*, where the former shows that a well-typed term can always be rewritten into another term, and the latter shows that during such evaluation, which rewrites terms into new forms, type safety is preserved in the new form. One consequence of a type safety proof is that a well typed program should have no unexpected consequences at run time, for instance by ending up in a state where evaluation has not finished but there is no evaluation rule that may be applied. More generally, a type safety proof guarantees that the constraints placed by the type system on the various terms will always be upheld.

Formal semantics. The semantics of a language define its evaluation rules in some way. One may further distinguish between static and dynamic semantics. Classical approaches to the specification of static semantics include *denotational semantics*, in which terms are defined in terms of mappings to mathematical entities, *operational semantics*, in which terms are specified in terms of operations on an underlying abstract machine, and *axiomatic semantics*, in which the language is primarily specified as axioms and the meaning of each term is what can be proven about it. [94, p. 33]. Instead of a formal semantics, we describe our semantics informally in Chapter 3.

We cannot introduce the theory of type systems and operational semantics in detail here; for an introduction to the study of statically typed programming languages we recommend Pierce's book [94].

In formalising a new Java extension, it is naturally to build on one of several core calculi that exist, since the full Java type system is somewhat unwieldly. Several well-studied such calculi exist.

Featherweight Java (FJ). FJ [53] is a truly minimal core calculus that omits a large amount of the standard Java features. It has no primitive types, no interfaces, no static classes or members and no exceptions. It also does not have any mutable state, making it a functional fragment of Java. Nevertheless it is Turing complete, and any valid FJ program is also a valid Java program, since the syntax is compatible. FJ has been very influential, and many studies of Java extensions are directly based on FJ, for instance alias annotations [2], ownership and immutability coupled with generics [134] and prototype-based component evolution [128] and others, such as [15], are inspired by FJ's design.

Middleweight Java (MJ). Like FJ, MJ [18] lacks primitive types, interfaces, static classes and members and exceptions. However, it does have mutable state, permitting variables to be re-assigned after initialisation. MJ also models the scope structure of Java faithfully. These changes make the calculus larger than FJ, but it is a good choice for our work, since we are interested in reasoning about state changes, which would be impossible in FJ. MJ has the additional advantage that a fragment of the Boyland-Greenhouse effect system [46], which Poplar is inspired by, was formalised in MJ and proved correct by the MJ authors [18]. This, and the fact that MJ has the concept of object identity (which FJ lacks) may make a similar correctness proof for Poplar simpler to carry out in the future.

Classic Java. Classic Java [33] also models mutable state, but it is not a valid subset of Java, and it was primarily designed with the intent of modelling mixins.

We use MJ as a basis for our formalism. We will proceed by introducing it and describing its essential features. Next, we formalise a core of Poplar, called Poplar_0, in Section 4.2. Finally, we formalise a larger version, called Poplar_1, in Section 4.3.

In this chapter, we provide the syntax and typing judgments for Poplar_0 and Poplar_1. By specifying the Poplar typing judgments, we will be formalising the notion of a valid, well-typed Poplar code fragment; our integration mechanism functions by searching for such well-typed Poplar code fragments. We do not provide any special semantics for Poplar. The reason is that the additional Poplar concepts serve as a compile time aid only; they have no purpose at runtime and therefore need no runtime semantics. When a Poplar program is compiled to a MJ program, all syntactic forms that are unique to Poplar are removed and only MJ forms remain. Thus, Poplar has the same operational semantics as MJ. When it is necessary, in order to shed light on some subtleties of our design, we will discuss selected parts of MJ semantics explicitly.

Although it is desirable to provide a type safety proof, due to time constraints, such a proof is not included in this work. However, we will present a lemma, which we believe will be useful in proving Poplar's type safety. We will see that those Poplar judgments that correspond to MJ judgments have either been strengthened, in the sense that more conditions must be satisfied for the judgment to be used, or unchanged. This means that a well-typed Poplar fragment is at least a well-typed MJ fragment when it has been compiled to MJ.

4.1 Middleweight Java (MJ)

We introduce the fundamentals of MJ briefly in this section in order to make the Poplar_0 and Poplar_1 specifications more accessible. For the full details of MJ, the reader should consult the MJ technical report [18].

In MJ, class types are denoted by C. Statement types, which include class types but also void, are denoted by τ.

MJ programs are typed with respect to a *class table*, denoted by Δ. The class table is in itself divided into a triplet of field, constructor and method tables:

$$\Delta \stackrel{\mathsf{def}}{=} (\Delta_f, \Delta_c, \Delta_m)$$

These three are constructed directly from the syntax using well defined mappings. We retain this notation in our formalisation, but we extend the field, constructor and method tables with more information.

4.1.1 Judgment forms

MJ typing judgments have the form $\Delta; \Gamma \vdash e : C$ and $\Delta; \Gamma \vdash s : \tau$ for expressions and statements, respectively. We use the extended judgment forms

$$\Delta; \Gamma \vdash e : C : U : \bar{l} : \mathsf{LS!}\rho$$

and

$$\Delta; \Gamma \vdash s : \tau : \mathsf{LS!}\rho$$

respectively. These forms will be introduced below.

MJ statements are denoted by s, expressions by e, and variables by x. We have kept these metavariables; see the syntax section below.

Subclassing relation

In our system, the subclassing relation has been unchanged from MJ. These judgments are applied with respect to a well formed program p (to be defined).

$$\text{TR-IMMEDIATE}\ \frac{\mathsf{class\ C_1\ extends\ C_2}\{\ldots\} \in p}{C_1 \prec_1 C_2} \qquad \text{TR-TRANSITIVE}\ \frac{C_1 \prec C_2 \qquad C_2 \prec C_3}{C_1 \prec C_3} \qquad \text{TR-EXTENDS}\ \frac{C_1 \prec_1 C_2}{C_1 \prec C_2}$$

$$\text{TR-REFLEXIVE}\ \frac{}{C \prec C}$$

4.2 Poplar_0 : A Minimal Poplar

Poplar_0 is the initial version of Poplar that we will formalise. Its main difference from its larger sibling, Poplar_1, is that it lacks external resources and composite properties.

We begin by formalising the fundamental concepts of Poplar in Section 4.2.1. We then give the syntax in Section 4.2.2. Several mappings are needed to translate from the syntax to the concepts used by our typing judgments; we give these gradually as needed.

MJ terms can be broadly divided into three categories: expressions, promotable expressions and statements. Statements are the basic building blocks of method bodies, which are lists of statements separated by semicolons. They implicitly have the void type: they do not evaluate to any value, but they may have side effects. Expressions, on the other hand, do evaluate to some value. Promotable expressions are expressions that may also be used as statements if they are followed by a semicolon. This is a feature that Java inherited from C-like languages. For instance, supposed that the variable x has type C and that the method C.m returns type D. Then the expression x.m() has type D, but if it is used as a statement, using the syntax x.m();, then the result is discarded.

We follow the MJ specification [18] in presenting Poplar typing judgments for terms one category at a time. Expression judgments are presented in Section 4.2.7, promotable expression judgments in Section 4.2.8, and statement judgments are presented in Section 4.2.9. Poplar queries behave like statements in many ways, but we nevertheless present their corresponding judgments separately in Section 4.2.11.

We use a horizontal line, as in $\overline{x}$, to denote (if syntax) a repetition such as $x_1, \ldots x_n$. If the context is not syntax, we will treat $\overline{x}$ as a set whose members are of the form x_i.

We have simplified Poplar_0 and Poplar_1 further by assuming that each method or constructor has exactly one argument. It is straightforward to generalise the systems presented here to n-argument methods.

4.2.1 Label signatures and chaining

Labels, properties and resources

We denote labels by l and resources by r. For a given label, $\mathsf{res}(l)$ gives the resource that it is sensitive to, so that it would be erased if that resource was mutated. Variables are denoted by x. ρ is a *mutation summary*, a set of mutated resources. ρ is a set whose members have the form $x.r$, $\mathsf{raw}(r)$ or $\mathsf{any}(C).r$, where C is a type. We write $\rho(x)$ to denote $\{r \mid x.r \in \rho\}$.

For a resource, $\mathsf{sens}(C)(r)$ gives the set of sensitive labels that would be lost if it was mutated. We also write

$$\mathsf{sens}(\Gamma, \rho) \stackrel{\mathsf{def}}{=} \{x.l \mid x.\mathsf{res}(l) \in \rho \vee (\Gamma \vdash x : C \wedge C \prec C' \wedge \mathsf{any}(C').\mathsf{res}(l) \in \rho\}$$

$$\mathsf{sens}(\Gamma, \rho, \overline{x.l}) \stackrel{\mathsf{def}}{=} \mathsf{sens}(\Gamma, \rho) \cap \{\overline{x.l}\}.$$

The inverse function $\mathsf{rem}(\Gamma, \rho, \overline{l})$ gives the set of remaining labels after all the resources have been mutated: $\mathsf{rem}(\Gamma, \rho, \overline{x.l}) \stackrel{\mathsf{def}}{=} \{x_i.l_i \mid x_i.l_i \notin \mathsf{sens}(\Gamma, \rho, \overline{x.l})\}$.

Labels that may be erased are called *properties*, and those that are not associated with any resource are called *tags*. A resource declaration in class C of the form

```
1 resource r { properties @a, @b, @c; }
```

induces the following res and sens mappings: $\mathsf{res}(C)(@a) = \{r\}, \mathsf{res}(C)(@b) = \{r\}, \mathsf{res}(C)(@c) = \{r\}, \mathsf{sens}(C)(r) = \{@a, @b, @c\}$.

In Poplar_0, for simplicity, we assume that resource and label names are globally unique.

Label signatures and chaining

LS is a *label signature*, written as $(\mathsf{LS} \stackrel{\mathsf{def}}{=} (\mathsf{LS}_+, \mathsf{LS}_=, \mathsf{LS}_-))$. Each of $\mathsf{LS}_+, \mathsf{LS}_=, \mathsf{LS}_-$ is a map that maps variables to sets of labels. However, when convenient we will also

treat them as sets and use operations such as $\cup$ and $\cap$. Members of these sets have the form $x : l$, where x is a variable, and l is a label. When using them as maps, we write, for example, $\mathsf{LS}_+(x)$ to denote $\{l|(x : l) \in \mathsf{LS}_+\}$. We will write $\mathsf{LS}(x)$ to denote $(\mathsf{LS}_+(x), \mathsf{LS}_=(x), \mathsf{LS}_-(x))$. We will write $\mathsf{LS}[x \mapsto y]$ to denote the operation of renaming x to y, i.e.

$$\mathsf{LS}[x \mapsto y] \stackrel{\mathsf{def}}{=} (\mathsf{LS}_+\backslash\{(x : l)\}, \mathsf{LS}_=\backslash\{(x : l)\}, \mathsf{LS}_-\backslash\{(x : l)\}) \cup (\{y : \mathsf{LS}_+(x)\}, \{y : \mathsf{LS}_=(x)\}, \{y : \mathsf{LS}_-(x)\})$$

Label signatures and mutation summaries describe *fragments*, which are sequences of statements. In a label signature, LS_+ describes labels added by a fragment, $\mathsf{LS}_=$ describes labels that are invariant for the fragment (both pre- and postconditions, and not lost temporarily), and LS_- describes preconditions that will be lost due to the fragment. We write $\mathsf{precond}(\mathsf{LS})$ to indicate $\mathsf{LS}_= \cup \mathsf{LS}_-$ and $\mathsf{postcond}(\mathsf{LS})$ to indicate $\mathsf{LS}_= \cup \mathsf{LS}_+$.

Chaining, or sequential composition, is one of the central operations on label signatures. The $(\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}_2, \rho_2)$ operation chains label signatures, for the case where a fragment described by (LS_1, ρ_1) is evaluted immediately before a fragment described by (LS_2, ρ_2). Note that chaining of label signatures is defined with respect to corresponding mutation summaries and also produces a tuple of a label signature and a mutation summary. First we will define a binary predicate $(\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}_2, \rho_2)\ \mathsf{ok}$ which indicates whether it is valid to chain the two signatures or not.

$$\Gamma \vdash (\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}_2, \rho 2)\ \mathsf{ok} \iff \forall l \in (\mathsf{LS}_2^= \cup \mathsf{LS}_2^-).l \in (\mathsf{LS}_1^= \cup \mathsf{LS}_1^+) \vee l \notin (\mathsf{sens}(\Gamma, \rho_1) \cup \mathsf{LS}_1^-)$$

This predicate ensures that for each fragment, its preconditions are either immediately satisfied by the preceding fragment that we wish to adjoin, or its preconditions are not erased by the preceding fragment (and may then be satisfied by some other, even earlier fragment).

Given that $(\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}_2, \rho_2)\ \mathsf{ok}$, we define $(\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}_2, \rho_2)$ as

$$\Gamma \vdash (\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}_2, \rho_2) \stackrel{\mathsf{def}}{=} ((\mathsf{LS}_+, \mathsf{LS}_=, \mathsf{LS}_-), \rho) \quad \text{where}$$

$$\mathsf{LS}_+ \stackrel{\mathsf{def}}{=} (\mathsf{rem}(\Gamma, \rho_2, \mathsf{LS}_1^+) \cup \mathsf{LS}_2^+)) \qquad \backslash(\mathsf{LS}_2^- \cup \mathsf{LS}_1^= \cup \mathsf{LS}_1^-)$$

$$its \stackrel{\mathsf{def}}{=} \mathsf{sens}(\Gamma, \rho_1, \mathsf{LS}_2^= \setminus \mathsf{LS}_1^=) \cup \mathsf{sens}(\Gamma, \rho_2, \mathsf{LS}_1^= \setminus \mathsf{LS}_2^=)$$

$$\mathsf{LS}_= \stackrel{\mathsf{def}}{=} (\mathsf{LS}_1^= \cup \mathsf{LS}_2^=) \setminus its \qquad \backslash(\mathsf{LS}_1^+ \cup \mathsf{LS}_2^-)$$

$$\mathsf{LS}_- \stackrel{\mathsf{def}}{=} (\mathsf{LS}_2^- \cup \mathsf{LS}_1^- \cup its) \qquad \backslash\mathsf{LS}_1^+$$

$$\rho \stackrel{\mathsf{def}}{=} \rho_1 \cup \rho_2$$

The intuition behind this operation is as follows. For LS_+ we want to capture added labels that remain "added", for $\mathsf{LS}_=$ invariant labels that remain invariant, and for LS_- any subtracted preconditions. For LS_+, what are the new labels that will be added if the second fragment is executed before the first? The labels added by the first fragment will only remain "added" if they are not lost to the mutations in the second fragment. The rem function computes this set. The labels added by the second fragment will clearly

remain added since we are not executing anything after this addition, that we know of. So we take the union of these two sets, but we must also remove the subtractions of the second fragment and the invariants of the first, since it's possible for the same label to be in, for instance, the invariants of the first fragment and the additions of the second. In this case it has clearly not been added, being required to exist before the first fragment executes. Hence the subtraction of sets at the end of the LS_+ expression. $\mathsf{LS}_=$ and LS_- follow a similar pattern. its tracks those labels that have transitioned from invariants to subtractions. The sens function in the its definition computes those labels that would have been invariants from the first fragment, but that are lost to mutations in the second fragment.

We say that a label signature LS is *well-formed* if $\mathsf{LS}_+, \mathsf{LS}_-$ and $\mathsf{LS}_=$ are disjoint sets. A pair LS, ρ is well-formed if, for any $e.l \in \mathsf{LS}_-$, $e.\mathsf{res}(l) \in \rho$. In this case, we write $\Delta \vdash (\mathsf{LS}, \rho)$ ok.

Lemma 1 (Well-formed chaining)**.** *Suppose that LS_1 is well-formed and LS_2 is well-formed. If $\Gamma \vdash (\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}_2, \rho_2)$ ok and $\Gamma \vdash (\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}_2, \rho_2) = (\mathsf{LS}, \rho)$, then LS is well-formed.*

Proof. We prove this by case analysis of the origin of any non-empty intersection of the three sets.

First, assume that $\mathsf{LS}_= \cap \mathsf{LS}_+ \neq \emptyset$. Then, from definitions, either $\mathsf{LS}_1^= \cap \mathsf{LS}_2^+ \neq \emptyset$ or $\mathsf{LS}_1^+ \cap \mathsf{LS}_2^= \neq \emptyset$. But $\mathsf{LS}_1^=$ is excluded from LS_+, and LS_1^+ is excluded from $\mathsf{LS}_=$, so none of these sets can be in $\mathsf{LS}_= \cap \mathsf{LS}_+$. Hence, $\mathsf{LS}_+ \cap \mathsf{LS}_= = \emptyset$.

Second, assume that $\mathsf{LS}_= \cap \mathsf{LS}_- \neq \emptyset$. The intersection cannot come from LS_2^- since this is excluded from $\mathsf{LS}_=$. So it must come from $\mathsf{LS}_1^- \cap \mathsf{LS}_2^=$. But if $\Gamma \vdash (\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}_2, \rho_2)$ ok, then $\mathsf{LS}_2^= \cap \mathsf{LS}_1^- = \emptyset$. Thus, $\mathsf{LS}_= \cap \mathsf{LS}_- = \emptyset$.

Third, assume that $\mathsf{LS}_- \cap \mathsf{LS}_+ \neq \emptyset$. LS_1^+ is excluded from LS_-, so this is only possible if LS_2^+ has a non-empty intersection with LS_1^- or with its. LS_1^- is excluded from LS_+, so the former is impossible. Any intersection $\mathsf{LS}_2^+ \cap its$ must come from $\mathsf{LS}_1^=$. But $\mathsf{LS}_1^=$ is also excluded from LS_+. Therefore, $\mathsf{LS}_+ \cap \mathsf{LS}_- = \emptyset$.

□

Note also that trivially, $\Delta \vdash (\mathsf{LS}_1, \rho_1)$ ok $\wedge\ \Delta \vdash (\mathsf{LS}_2, \rho_2)$ ok $\implies \Delta \vdash (\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}_2, \rho_2)$ ok.

Remark. Our definition of well-formedness insists that each label is *either* an addition, an invariant, or a subtraction, and never, for a given variable, a member of more than one of the three sets in a label signature. This is not the only conceivable notion of well-formedness. Consider the following class.

```
1  class C {
2    resource r {
3      properties @a;
4    }
5    void m() this: @a. { ... }
6    void n() mutates this.r: { ... }
7    void o() this: +@a. { ... }
8    void sequence() mutates this.r: this: -@a. {
9      m(); n(); o();
10   }
11 }
```

The sequence m(); n(); o(); needs @a as an initial precondition for m. The property is lost by n but re-established by o. Our formalisation considers the overall effect of the method sequence to be a subtraction of @a even though it is re-established in

the end.. An alternative way of reasoning about it would be to permit a non-empty intersection between the addition and subtraction sets of label signatures. Then one could express that a label is lost and re-established. We leave an investigation of this alternative design for future work.

In addition to chaining of label signatures, we will need *disjunctive composition* for the case where either one of two fragments might execute. We use this to type if ... else -statements.

$$(\mathsf{LS}_1, \rho_1) \otimes (\mathsf{LS}_2, \rho_2) \stackrel{\mathsf{def}}{=} ((\mathsf{LS}_1^+ \cap \mathsf{LS}_2^+,$$
$$(\mathsf{LS}_1^= \cup \mathsf{LS}_2^=) \setminus (its \cup \mathsf{LS}_1^+ \cup \mathsf{LS}_2^+),$$
$$(\mathsf{LS}_1^- \cup \mathsf{LS}_2^- \cup its) \setminus (\mathsf{LS}_1^+ \cup \mathsf{LS}_2^+))$$

Where $its \stackrel{\mathsf{def}}{=} \mathsf{sens}(\Gamma, \rho_1, \mathsf{LS}_2^= \setminus \mathsf{LS}_1^=) \cup \mathsf{sens}(\Gamma, \rho_2, \mathsf{LS}_1^= \setminus \mathsf{LS}_2^=)$ (same as previously).

4.2.2 Syntax and symbols

We now repeat the syntax of Poplar$_0$. In this syntax, horizontal lines indicate comma-separated repetition. For instance, $\overline{x}$ would be a shorthand for $x_1, \ ... \ x_n$.

In addition, the following metavariables and symbols are used.

	Symbol	Expanded form
Method type	μ	$(U)C : U \to U : \mathsf{LS} : \rho$
Constructor type		$C : U : \mathsf{LS} : \rho$
Field type		$C : U : \bar{l} : r$
Label signature	LS	$(\mathsf{LS}_+, \mathsf{LS}_=, \mathsf{LS}_-)$
Class table	Δ	$(\Delta_c, \Delta_m, \Delta_f)$
Constructor lookup	Δ_c	
Method lookup	Δ_m	
Field lookup	Δ_f	
Typing context	Γ	
Method receiver	rec	
Argument	arg	
Return value	ret	

Program

P ::= $cd_1 \ldots cd_n : \overline{s}$

Class definition

cd ::= class C extends C
$\{fd_1 \ldots fd_k \; cnd \; rd_1 \ldots rd_j$
$md_1 \ldots md_n\}$

Field definition

fd ::= $C \; f[: (\overline{[@p_1, \; \ldots \; @p_n \rightarrow] \; l_1, \; \ldots \; l_k})]$;

Resource definition

rd ::= resource r{properties $@p_1, \; \ldots @p_n$;
$fd_1 \ldots fd_k\}$

Constructor definition, method definition

cnd ::= $C \; (C_a \; x_a)[\rho :][ls]\{\text{super}(e_a); s_1 \ldots s_n\}$
md ::= $\tau \; m(C_a \; x_a)[\rho :][ls]\{s_1 \ldots s_k\}$

Mutation summary, qualified resource

ρ ::= mutates $qr_1, \; \ldots qr_n$
qr ::= $r \mid \text{any}(C).r \mid x.r$

Label signature, label condition, label

ls ::= $\overline{x : \; lc_1, \; \ldots \; lc_n;}$
lc ::= $++ @p \mid +l \mid l \mid -@p \mid U$
l ::= $t \mid @p$ — Tag, property

Uniqueness

U ::= fresh | unique | maintain

Return type

τ ::= $C \mid$ void

Expression

e	::=	$x \mid$ null $\mid e.f$	Variable, null, field access
	$\mid$	$(C)e$	Cast
	$\mid$	pe	Promotable expression

Promotable expression

pe	::=	$e.m(e_a) \mid$ new $C(e_a)$	Method invocation, object creation

Statement

s	::=	;	No-op
	$\mid$	pe;	Promoted expression
	$\mid$	if $(e == e)\{s_1 \ldots s_k\}$ else $\{s_{k+1} \ldots s_n\}$	Conditional
	$\mid$	$e.f = e$;	Field assignment
	$\mid$	$C \; x[: U]$	Local variable declaration
	$\mid$	$x = e$;	Variable assignment
	$\mid$	return e;	Return
	$\mid$	$\{s_1 \ldots s_n\}$	Block
	$\mid$	$x =$ #produce$(C, U, l_1, \; \ldots \; l_n)$;	Produce query
	$\mid$	#transform$(x, l_1, \; \ldots \; l_n)$;	Transform query
	$\mid$	drop $l_1, \; \ldots \; l_n$;	Drop labels

4.2.3 Uniqueness kinds

We introduced uniqueness kinds in Section 2.4.8. The kinds supported by Poplar$_0$are Unique , Maintain , Fresh and normal. We enforce the semantics of these kinds fundamentally by tracking assignments. For example, assignment of unique references to any variable or field is forbidden.

We use the metavariable U to denote uniqueness kinds. The relation $U \rightsquigarrow U'$ to indicates that a value of kind U may flow to the kind U'. It is defined as follows.

$$U \rightsquigarrow U \qquad \mathsf{Normal} \rightsquigarrow \mathsf{Maintain} \qquad \mathsf{Unique} \rightsquigarrow \mathsf{Maintain} \qquad \mathsf{Fresh} \rightsquigarrow U$$

If a value has been passed in as an argument to a method or constructor, we will tag its uniqueness kind, writing U^{arg}, enabling us to track its origin as an argument later. This is to prevent methods from returning a unique argument as a unique return value, for instance, which would appear to the caller as two unique values when they actually are aliases for each other.

4.2.4 Method, constructor and field typing

Over this and the next few sections we give the typing judgments for Poplar$_0$. We write $C \prec C'$ to indicate subtyping and $C \prec_1 C'$ to indicate immediate subtyping. We denote method, constructor and field types by Δ_m, Δ_c and Δ_f. However, ours contain more information than those of MJ.

Method types

Method types are defined with respect to the md syntactic form (method definition). The lookup function $\Delta_m(C)(m)$ gives the type of method m in class C.

$$\Delta_m(C)(m) \stackrel{\mathsf{def}}{=}$$

$$\begin{cases} (U_{\mathsf{rec}})C_a : U_a \to \tau : U_{\mathsf{ret}} : rewrite(\mathsf{LS}, \rho) & \text{where } md_i = \tau\ m(C_a\ x_a)\rho\ \mathsf{LS}\{\ldots\} \\ & \text{and } U_a = uniq(\mathsf{LS}_=(\mathsf{arg}_1)), U_{\mathsf{ret}} = uniq(\mathsf{LS}_=(\mathsf{ret})), \\ & U_{\mathsf{rec}} = uniq(\mathsf{LS}_=(\mathsf{rec})) \\ \Delta_m(C')(m) & \text{where } m \notin md_1 \ldots md_n \end{cases}$$

$$\text{where } \mathsf{class}\ C\ \mathsf{extends}\ C'\{\ldots md_1 \ldots md_n\} \in p$$

$$\text{and } uniq(\overline{l}) \stackrel{\mathsf{def}}{=} U \text{ if } U \in \overline{l}, \mathsf{Normal}\ \ otherwise$$

$$\text{and } rewrite(\mathsf{LS}, \rho) \stackrel{\mathsf{def}}{=} (\mathsf{LS}_{++} \cup \mathsf{LS}_+, \mathsf{LS}_=, \mathsf{LS}_-)!(\rho \cup \bigcup_{r \in \mathsf{res}(\mathsf{LS}_{++})} \mathsf{raw}(r))$$

The LS belonging to the method type is extracted directly from the syntax. Labels of the forms $++l, +l, l, -l$ are used to form the $\mathsf{LS}_{++}, \mathsf{LS}_+, \mathsf{LS}_=$ and LS_- sets, respectively. However, we use the $rewrite$ operation to merge LS_{++} and LS_+ immediately when we compute method types. We will write $direct(\mathsf{LS})$ to identify these labels later. In other words, $direct(\mathsf{LS}') \stackrel{\mathsf{def}}{=} \mathsf{LS}_{++}$ if $(\mathsf{LS}', \rho') = rewrite(\mathsf{LS}, \rho)$. The main purpose of tracking such "direct" access is to disallow direct writes to fields in resources, except for when it is needed to directly establish a property.

A method or constructor declaration is only well-formed if $\mathsf{LS}_{++} \cap \mathsf{LS}_{+} = \emptyset$ prior to the rewrite. In addition, the intersections of $\mathsf{LS}_{+}, \mathsf{LS}_{=}$ and LS_{-} must be empty, as usual. $\mathsf{LS}(x)$ is only well formed if it has at most one member of the form U.

Constructor types

Constructor types are similar to method types. Their corresponding lookup function is Δ_c.

$$\Delta_c(C) \stackrel{\text{def}}{=} C_a : U_a : rewrite(LS, \rho) \text{ where class } C \text{ extends } C'\{\ldots cnd \ldots\}$$

$$\text{and } cnd = C(C_a\ x_a)\ \rho\ \mathsf{LS}\{\ldots\} \text{ and } U_a = uniq(\mathsf{LS}(\mathsf{arg}_1))$$

A constructor declaration is only well-formed if $\mathsf{precond}(\mathsf{LS}(\mathsf{ret})) = \emptyset$.

Field types

In Poplar_0, $\Delta_f(f)(\bar{l})$ denotes the specific type of the field, including its guaranteed labels, while the owning object has the labels $\bar{l}$. The general type of a field is a function, $\Delta_f(f) : L \to C \times U \times L^\star \times R$, where L is the set of all labels and R is the set of all resources.

$$\Delta_f(C)(f)(\bar{l}) \stackrel{\text{def}}{=}$$

$$\begin{cases} C'' : \mathsf{Normal}\ : \emptyset : fr(C)(fd_i) \text{ where } fd_i = C''\ f, \text{ for some } i, 1 \le i \le k\ and \Delta_f(C')(f) \uparrow \\ C'' : \mathsf{Unique}\ : \bar{l}' : fr(C)(fd_i) \text{ where } fd_i = C''\ f : (\bar{l}_1 \to \bar{l}_1^f \ldots \bar{l}_n \to \bar{l}_n^f), \text{ and } \bar{l}' = \bigcup_{\bar{l}_j \subseteq \bar{l}} \bar{l}_j^f, \\ \qquad\qquad \text{for some } i, 1 \le i \le k \\ \Delta_f(C')(fd)\ otherwise \end{cases}$$

$$fr(C)(fd) \stackrel{\text{def}}{=} \begin{cases} r\ where\ \mathsf{class}\ C\ \mathsf{extends}\ C'\{\overline{fd'}\ \overline{cnd}\ r\{fd, fd_2, \ldots fd_n\}\ rd_2, \ldots rd_m\ \overline{md}\} \\ \emptyset\ otherwise \end{cases}$$

Example. Consider the following class.

```
1  class C {
2    resource r {
3      properties @a, @b, @c;
4      C f:(@a->@b, @b->c);
5    }
6    D m(C x) this: @b, -@c; x: unique, ++@a. {
7      //...
8    }
9  }
```

The method type of D.m is as follows:

$$(\mathsf{Normal})C : \mathsf{Unique} \to D : \mathsf{Normal} + (\{x : @a\}, \{\mathsf{this} : @b\}, \{\mathsf{this} : @c\})!\{\mathsf{raw}(\mathsf{this}.r)\}$$

The field C.f will have the type:

$$\mathsf{C} : \mathsf{Unique}\ : \{@b, @c\} : \mathsf{r}$$

if the owning object has at least properties $\{@a, @b\}$. If the owning object only has the property $@a$, for instance, then the field f is considered to have type

$$\mathsf{C} : \mathsf{Unique} \ : \{@b\} : \mathsf{r}.$$

4.2.5 Subsumption of label signatures, resources and mutation summaries

We define orderings on label signatures (LS), resources (r) and mutation summaries (ρ). The intuition behind this subsumption is, generally, that $\mathsf{LS} \prec \mathsf{LS}'$ if LS can take the place of (override) LS' in a subclass in a sound manner, $\rho \prec \rho'$ if ρ can take the place of ρ' in a subclass in a sound manner, and $r \prec r'$ if r is a more specific (smaller) resource than r'.

$$\mathsf{LS} \prec \mathsf{LS} \qquad (\mathsf{LS}_+ \cup \bar{l}, \mathsf{LS}_=, \mathsf{LS}_-) \prec (\mathsf{LS}_+, \mathsf{LS}_=, \mathsf{LS}_-)$$

$$(\mathsf{LS}_+, \mathsf{LS}_=, \mathsf{LS}_-) \prec (\mathsf{LS}_+, \mathsf{LS}_=, \mathsf{LS}_- \cup \bar{l}) \qquad r \prec r$$

$$\frac{\Delta;\Gamma \vdash r \prec r' \qquad \Delta;\Gamma \vdash C \prec C'}{\Delta;\Gamma \vdash \mathsf{any}(C).r \prec \mathsf{any}(C').r'} \qquad \frac{\Delta;\Gamma \vdash r \prec r'}{\Delta;\Gamma \vdash x.r \prec x.r'}$$

$$\frac{\Delta;\Gamma \vdash r \prec r' \qquad \Delta;\Gamma \vdash x : C' \qquad \Delta;\Gamma \vdash C' \prec C}{\Delta;\Gamma \vdash x.r \prec any(C).r} \qquad \Delta;\Gamma \vdash \rho \prec \rho$$

$$\frac{\Delta;\Gamma \vdash \rho \prec \rho'}{\Delta;\Gamma \vdash \rho \prec \rho' \cup \{r\}} \qquad \frac{\Delta;\Gamma \vdash \rho \prec \rho' \qquad \Delta;\Gamma \vdash r \prec r'}{\Delta;\Gamma \vdash \rho \cup \{r\} \prec \rho' \cup \{r'\}}$$

$$\frac{\Delta;\Gamma \vdash \rho \prec \rho' \qquad \Delta;\Gamma \vdash \mathsf{LS} \prec \mathsf{LS}'}{\Delta;\Gamma \vdash (\mathsf{LS}, \rho) \prec (\mathsf{LS}', \rho')}$$

Note that given these rules, $\rho \prec \rho'$ if and only if any elements of the form $\mathsf{raw}(r)$ are the same in both ρ and ρ'. $\mathsf{raw}(r)$ corresponds to the ability to write a resource's concrete data directly and is inserted in method types where the syntax contains a direct addition (++) for some property in that resource.

4.2.6 Well-formed class definitions, part 1

In this section we give the judgments that identify well-formed definitions and well-formed overriding.

There is a subtlety with direct additions of properties. When a subclass redefines the concrete meaning of a property, it is possible for a class to be in an intermediate state where a superclass invariant has been established but the subclass invariant has not. Deline and Fähndrich solved this problem by using frame typestates [27], which encode a separate typestate for each class frame, instead of having a single typestate for the entire class. Here we wish to avoid this complication, so instead we insist that each property should be established in all class frames atomically. We enforce this by requiring that all direct addition methods are overridden by their subclasses, by requiring that methods have at least the direct additions of the methods that they override, by requiring direct addition methods to call their corresponding superclass method, and by not considering a property to be established until the bottom frame has been reached. In this way direct addition methods come to resemble constructors.

We leave a more elegant solution (possibly including frame properties as a natural equivalent of frame typestates) for future work.

$$\text{T-CType}\ \frac{C \in dom(\Delta)}{\Delta \vdash C\ \mathsf{ok}} \qquad \text{T-VType}\ \frac{}{\Delta \vdash \mathsf{void}\ \mathsf{ok}} \qquad \text{T-PropOK}\ \frac{\mathsf{res}(C)(@p) \subseteq \mathsf{res}(C')(@p) \quad C \prec_1 C'}{\Delta \vdash @p\ \mathsf{ok}}$$

$$\text{T-MType}\ \frac{\Delta \vdash C_a\ \mathsf{ok} \quad \Delta \vdash \tau\ \mathsf{ok} \quad \Delta \vdash (\mathsf{LS}, \rho)\ \mathsf{ok} \quad \mathsf{precond}(\mathsf{LS})(\mathsf{ret}) = \emptyset \quad (\mathsf{postcond}(\mathsf{LS})(\mathsf{ret}) = \emptyset \vee \tau \neq \mathsf{void}) \quad U_{\mathsf{rec}} \neq \mathsf{Fresh} \quad U_a \neq \mathsf{Fresh} \quad U_r et \neq \mathsf{Fresh}}{\Delta \vdash (U_{\mathsf{rec}})C_a : U_a \to \tau : U_{\mathsf{ret}} : \mathsf{LS}!\rho\ \mathsf{ok}}$$

$$\text{T-ConsOK}\ \frac{\Delta \vdash C_a\ \mathsf{ok} \quad \mathsf{precond}(LS)(\mathsf{rec}) = \emptyset \quad \Delta_c(C) = C_a : U_a : \mathsf{LS}!\rho}{\Delta \vdash \Delta_c(C)\ \mathsf{ok}} \qquad \text{T-FieldOK1}\ \frac{C \prec_1 C' \quad \Delta \vdash C_f\ \mathsf{ok} \quad \bar{l}'_f \subseteq \bar{l}_f \quad fr(C)(f) \subseteq fr(C')(f) \quad \Delta_f(C)(f) = C_f : U_f : \bar{l}_f \quad \Delta_f(C')(f) = C'_f : U_f : \bar{l}'_f}{\Delta \vdash C.f\ \mathsf{ok}}$$

$$\text{T-FieldOK2}\ \frac{\Delta \vdash C_f\ \mathsf{ok} \quad C \prec_1 C' \quad \Delta_f(C)(f) = C_f : U_f : \bar{l}_f \quad f \notin dom(\Delta_f)(C')}{\Delta \vdash C.f\ \mathsf{ok}} \qquad \text{T-FieldsOK}\ \frac{\Delta \vdash C.f_1\ \mathsf{ok} \ldots \Delta \vdash C.f_n\ \mathsf{ok} \quad dom(\Delta_f(C)) = \{f_1 \ldots f_n\}}{\Delta \vdash \Delta_f(C)\ \mathsf{ok}}$$

$$\text{T-MethOK1}\ \frac{\Delta_m(C)(m) = \mu \quad \mathsf{LS} \prec \mathsf{LS}' \quad \rho \prec \rho' \quad C \prec_1 C' \quad U'_a \rightsquigarrow U_a \quad U'_{\mathsf{rec}} \rightsquigarrow U_{\mathsf{rec}} \quad U_{\mathsf{ret}} \rightsquigarrow U'_{\mathsf{ret}} \quad \mu = (U_{\mathsf{rec}})C_a : U_a \to \tau : U_{\mathsf{ret}} : \mathsf{LS}!\rho \quad \Delta_m(C')(m) = (U'_{\mathsf{rec}})C_a : U'_a \to \tau : U'_{\mathsf{ret}} : \mathsf{LS}'!\rho' \quad \Delta \vdash \mu\ \mathsf{ok} \quad direct(\mathsf{LS}') \subseteq direct(\mathsf{LS})}{\Delta \vdash C.m\ \mathsf{ok}}$$

$$\text{T-MethOK2}\ \frac{\Delta \vdash \mu\ \mathsf{ok} \quad m \notin dom(\Delta_m(C')) \quad \Delta_m(C)(m) = \mu \quad C \prec_1 C'}{\Delta \vdash C.m\ \mathsf{ok}} \qquad \text{T-MethsOK}\ \frac{\Delta \vdash C.m_1\ \mathsf{ok} \ldots \Delta \vdash C.m_n\ \mathsf{ok} \quad dom(\Delta_m(C)) = \{m_1 \ldots m_n\}}{\Delta \vdash \Delta_m(C)\ \mathsf{ok}}$$

$$\text{T-PropsOK}\ \frac{\Delta \vdash @p_1 \ldots \Delta \vdash @p_n\ \mathsf{ok} \quad dom(\mathsf{res}(C)) = \{@p_1 \ldots @p_n\}}{\Delta \vdash \mathsf{res}(C)\ \mathsf{ok}} \qquad \text{T-ClassOK}\ \frac{\Delta \vdash \Delta_f\ \mathsf{ok} \quad \Delta \vdash \Delta_m(C)\ \mathsf{ok} \quad \Delta \vdash \Delta_c(C)\ \mathsf{ok} \quad \Delta \vdash \mathsf{res}(C)\ \mathsf{ok}}{\vdash \Delta\ \mathsf{ok}} \forall C \in dom(\Delta)$$

For valid overriding, we require that superclasses are overridden in a way that does not invalidate the contracts that they expose through Poplar annotations. In particular, a property defined in a certain resource in a superclass cannot be moved to new or different resources in subclasses (however, additional properties can be introduced). Mutations and label signatures abide by the substitution principle: overriding methods will, if they are valid, have stronger or equal postconditions and weaker or equal preconditions.

Note that a Fresh variable cannot be written directly to a field or a variable, so the variable this will remain Fresh until returned from a constructor.

For full well-formedness definitions, we also need to define well-formed classes and method bodies. However, we need several preliminaries before we can do this, so we present the remainder of the well-formedness judgments in Section 4.2.10.

Acceptable mutations. The judgment $\Delta;\Gamma \vdash e!\overline{r}$ ok indicates whether it is acceptable to mutate the resources $\overline{r}$ of the expression e. It is defined as follows.

$$\Delta;\Gamma \vdash e!\emptyset \text{ ok} \qquad \Delta;\Gamma \vdash x!\overline{r} \text{ ok} \qquad \Delta;\Gamma \vdash \text{this}.f!\overline{r} \text{ ok} \qquad \frac{\Delta;\Gamma \vdash e!\overline{r} \text{ ok}}{\Delta;\Gamma \vdash (C)e!\overline{r} \text{ ok}}$$

$$\frac{\Delta;\Gamma \vdash e : C : \text{Fresh}}{\Delta;\Gamma \vdash e!\overline{r} \text{ ok}}$$

Invocation substitutions are used to bind the label signatures and mutation summaries of methods and constructors to concrete expressions when they are invoked. Method types contain the placeholders ret, rec, arg to indicate return value, receiver and argument, respectively. After binding them with invsub, we obtain method signatures that correspond to the caller side view of a method invocation. Note that the mutation summary is adjusted depending on whether the inputs may be aliased or not.

$$\text{invsub}(\text{LS}, \rho, e_r, C_r, U_r, e_a, C_a, U_a) \stackrel{\text{def}}{=}$$

$$(\text{invsub}(\text{LS}, e_r, e_a), \text{invsub}(\rho, e_r, C_r, U_r, e_a, C_a, U_a))$$

$$\text{invsub}(\text{LS}, e_r, e_a) \stackrel{\text{def}}{=} lflow(lflow(\text{LS}, \text{rec}, e_r), \text{arg}, e_a))$$

$$\text{invsub}(\rho, e_r, C_r, U_r, e_a, C_a, U_a) \stackrel{\text{def}}{=} rflow(rflow(\rho, \text{rec}, e_r, C_r, U_r), \text{arg}, e_a, C_a, U_a))$$

$$\text{cnsub}(\text{LS}, \rho, e_a, C_a, U_a) \stackrel{\text{def}}{=} (rflow(\rho, \text{arg}, e_a, C_a, U_a), lflow(\text{LS}, \text{arg}, e_a))$$

$$lflow(\text{LS}, e, e') \stackrel{\text{def}}{=} \begin{cases} \text{LS}[e \mapsto e'] & \text{if } e = \text{this}.f \text{ or } e = x \\ \text{LS} & \text{otherwise} \end{cases}$$

$$rflow(\rho, e, e', C, U) \stackrel{\text{def}}{=} \begin{cases} \rho[e \mapsto \text{any}(C)] & \text{if } U = \text{Maintain or } U = \text{Normal} \\ \rho[e \mapsto \text{this}.f] & \text{if } e = \text{this}.f \text{ and } U \notin \{\text{Maintain}, \text{Normal}\} \\ \rho[e \mapsto x] & \text{if } e = x \text{ and } U \notin \{\text{Maintain}, \text{Normal}\} \\ \rho & \text{otherwise} \end{cases}$$

4.2.7 Typing judgments for expressions

Generally, typing judgments take the following form: $\Delta;\Gamma \vdash e : C : U : \overline{l} : \text{LS}!\tau$, where C is the type of the expression, U is a uniqueness kind, $\overline{l}$ is the set of labels associated with it (at its point of declaration), LS is the label signature associated with evaluating the expression, and τ is the mutation summary associated with evaluating the expression. Sometimes we will abbreviate this rather long judgment for the sake of brevity, when not all information is needed. For instance, we may write $\Delta;\Gamma \vdash e : U$ or $\Delta;\Gamma \vdash e : C$.

Many of the typing judgments use the chaining operation $(\text{LS}_1, \rho_1) \oplus (\text{LS}_2, \rho_2)$ (see Section 4.2.1. We remind the reader that each time we make use of this operation, we also implicitly require that $(\text{LS}_1, \rho_1) \oplus (\text{LS}_2, \rho_2)$ ok, since otherwise the former operation is undefined. For the sake of brevity we do not spell this out explicitly.

$$\text{TE-VAR}\quad \frac{\Delta \vdash \Gamma \ \mathsf{ok} \qquad \vdash \Delta \ \mathsf{ok}}{\Delta; \Gamma, x : C : U \vdash x : C : U : \overline{l'} : \mathsf{LS}} \quad \text{where } \mathsf{LS} = \{\emptyset, \{x_= : \bar{l}'\}, \emptyset\} \text{ and}$$

$$\text{TE-SUBLABELS}\quad \frac{\Delta; \Gamma \vdash e : C : U : \bar{l} : \mathsf{LS}!\rho}{\Delta; \Gamma \vdash e : C : U : \bar{l}' : \mathsf{LS}!\rho} \quad \text{where } \bar{l}' \subseteq \bar{l}$$

$$\text{TE-FIELDACCESSTHIS}\quad \frac{\Delta; \Gamma \vdash \mathsf{this} : C : U : \bar{l} : \mathsf{LS}}{\Delta; \Gamma \vdash \mathsf{this}.f : C_1 : U_1' : \bar{l}_1 : \mathsf{LS}_2} \quad \begin{array}{l}\text{where } \mathsf{LS}_1 = \{\emptyset, \{\mathsf{this}.f_= : \bar{l}_1\}, \emptyset\} \\ \text{and } U_1' = uniqueReturn(U_2, U_1), (\mathsf{LS}_2, \rho_2) = (\mathsf{LS}, \emptyset) \oplus (\mathsf{LS}_1, \emptyset)\end{array}$$

$$\text{TE-FIELDACCESS}\quad \frac{\begin{array}{c}\Delta; \Gamma \vdash e : C : U : \bar{l} : \mathsf{LS} \\ \Delta_f(C)(f)(\bar{l}) = C_1 : U_1 : \bar{l}_1 \qquad e \neq x\end{array}}{\Delta; \Gamma \vdash e.f : C_1 : U_1' : \bar{l}_1 : \mathsf{LS}} \quad \text{where } U_1' = uniqueReturn(U_2, U_1)$$

$$\text{TE-NULL}\quad \frac{\Delta \vdash C \ \mathsf{ok} \qquad \Delta \vdash \Gamma \ \mathsf{ok} \qquad \vdash \Delta \ \mathsf{ok}}{\Delta; \Gamma \vdash \mathsf{null} : C}$$

$$\text{TE-UPCAST}\quad \frac{\Delta; \Gamma \vdash e : C_2 \qquad C_2 \prec C_1 \qquad \Delta \vdash C_1}{\Delta; \Gamma \vdash (C_1)e : C_1}$$

$$\text{TE-DOWNCAST}\quad \frac{\Delta; \Gamma \vdash e : C_2 \qquad C_1 \prec C_2 \qquad \Delta \vdash C_1}{\Delta; \Gamma \vdash (C_1)e : C_1}$$

$$\text{TE-STUPIDCAST}\quad \frac{\begin{array}{c}\Delta; \Gamma \vdash e : C_2 \qquad C_2 \not\prec C_1 \\ C_1 \not\prec C_2 \qquad \Delta \vdash C_1\end{array}}{\Delta; \Gamma \vdash (C_1)e : C_1}$$

4.2.8 Typing judgments for promotable expressions

These judgments check method invocations and constructor invocations. The following items must be checked:

- Are the argument and the receiver of compatible types?
- Are the uniqueness kinds of the argument and the receiver acceptable with respect to the method signature?
- Are the mutations that the method's invocation would cause acceptable?

The helper function $invocationOk$ is used to check non-repetition of unique inputs. The helper function $uniqueReturn$ is used to rewrite the uniqueness of return values in the case of a receiver that is not definitely unique. The need for this procedure is explained in Section 3.6. $cnsub$ and $invsub$ are used to rewrite the method type by substituting the concrete expressions that are passed in for placeholders such as arg, rec and ret.

$$\textsc{TE-Method}\quad \frac{\begin{array}{c}\Delta;\Gamma \vdash e' : C' : U' : \mathsf{precond(LS)(rec)} : \mathsf{LS'}!\rho' \qquad C_1 \prec C_a \\ \Delta;\Gamma \vdash e_1 : C_1 : U_1 : \mathsf{precond(LS)(arg_1)} : \mathsf{LS}_1!\rho_1 \\ \Delta_m(C')(m) = (U_{\mathsf{rec}})C_a : U_a \to \tau : U_{\mathsf{ret}} : \mathsf{LS}!\rho \\ \Delta \vdash e!\rho(\mathsf{this})\ \mathsf{ok} \qquad \Delta \vdash e_1!\rho(\mathsf{arg}_1)\ \mathsf{ok} \qquad invocationOk(\Gamma, \{e', e_1\}) \\ U' \rightsquigarrow U_{\mathsf{rec}} \qquad U_1 \rightsquigarrow U_a \qquad U'_{\mathsf{ret}} = uniqueReturn(U', U_{\mathsf{ret}}) \\ (\mathsf{LS''}, \rho'') = (\mathsf{LS'}, \rho') \oplus (\mathsf{LS}_1, \rho_1) \oplus \mathsf{invsub(LS}, \rho, e', C', U', e_1, C_1, U_1)\end{array}}{\Delta;\Gamma \vdash e'.m(e_1) : \tau : U'_{\mathsf{ret}} : \mathsf{postcond(LS)(ret)} : \mathsf{LS''} \setminus direct(C'.m)!\rho''}$$

$$\textsc{TE-New}\quad \frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : C'_1 : \mathsf{precond(LS)(arg_1)} : U'_1 : \mathsf{LS'}!\rho' \\ \Delta_c(C) = C_1 : U_1 : \mathsf{LS}!\rho \qquad C'_1 \prec C_1 \\ \Delta \vdash e_1!\rho(\mathsf{arg}_1)\ \mathsf{ok} \qquad invocationOk(\Gamma, \{e_1\}) \qquad U'_1 \rightsquigarrow U_1 \\ (\mathsf{LS''}, \rho'') = ((\mathsf{LS'}, \rho') \oplus \mathsf{cnsub(LS}, \rho, e_1, C_1, U_1))\end{array}}{\Delta;\Gamma \vdash \mathsf{new}\ C(e_1) : C : \mathsf{postcond(LS)(ret)} : \mathsf{Fresh}\ : \mathsf{LS''}!\rho''}$$

$$invocationOk(\Gamma, \overline{e}) \stackrel{\mathsf{def}}{=} \forall e_i((\Gamma \vdash e_i : \mathsf{Unique}\ \vee \Gamma \vdash e_i : \mathsf{Fresh}\ ) \implies e_i \notin (\overline{e} \setminus e_i) \wedge \Gamma \vdash e_i : \mathsf{Fresh} \implies e_i \neq x)$$

$$uniqueReturn(U_{\mathsf{rec}}, U_{\mathsf{ret}}) \stackrel{\mathsf{def}}{=} \begin{cases} \mathsf{Maintain} & \text{if } U_{\mathsf{rec}} \in \{\mathsf{Maintain}, \mathsf{Normal}\} \text{ and } U_{\mathsf{ret}} = \mathsf{Unique} \\ U_{\mathsf{ret}} & \text{otherwise} \end{cases}$$

4.2.9 Typing judgments for statements

Typing judgments (statements). These take the form $\Delta;\Gamma \vdash s : \tau : \mathsf{LS}!\rho$, where s is a statement, τ is its type, and LS and ρ specify the effects of the statement.

For TS-VARWRITE and TS-FIELDWRITE we make use of a replacement operation for label signatures: $LS_1 \odot LS_2 \stackrel{\mathsf{def}}{=} LS_1 \setminus \{LS_1(x) \mid x \in dom(LS_2)\} \cup LS_2$. It is clear that if LS_1 is well-formed and LS_2 is well-formed, then $LS_1 \odot LS_2$ is well-formed.

TS-DROPTHIS

$$\frac{\Delta;\Gamma \vdash \mathsf{this} : C : U : \bar{l}_1 : \mathsf{LS}_1!\rho_1 \quad \bar{l}_1 = \bar{l} \uplus \bar{l}_2 \text{ and } \overline{r} = \bigcup \mathsf{res}(\bar{l})}{\Delta;\Gamma \vdash \mathsf{drop}\ \bar{l}; : \mathsf{void} : (\mathsf{LS}_1, \rho_1) \oplus (\{\mathsf{this}_- : \bar{l}\}, \overline{r})}$$

TS-RETURN

$$\frac{\Delta;\Gamma \vdash e : C : U : \bar{l} : \mathsf{LS}!\rho \quad \mathsf{LS}' = \mathsf{LS} \odot \{\{\mathsf{ret}_+ : \bar{l}\}, \emptyset, \emptyset\} \quad U \neq \mathsf{Unique}^{\ \mathsf{arg}} \wedge U \neq \mathsf{Maintain}^{\ \mathsf{arg}}}{\Delta;\Gamma \vdash \mathsf{return}\ e; : C : \mathsf{LS}'!\rho}$$

T-CSUPER

$$\frac{\begin{array}{c}\Delta;\Gamma' \vdash e_1 : C_1' : U_1 : \bar{l}_1 : LS_1!\rho_1 \quad \Delta_c(C') = C_1 : U_1 : \mathsf{LS}'!\rho' \quad C_1' \prec C_1 \\ C \prec_1 C' \quad \Gamma(\mathsf{this}) = C, \Gamma = \Gamma' \uplus \{\mathsf{this} : C\} \\ (\mathsf{LS}'', \rho'') = (\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}', \rho') \\ (\mathsf{LS}, \rho) = (lflow(\mathsf{LS}'', \mathsf{arg}, e_1), rflow(\rho'', \mathsf{arg}, e_1, C_1', U_1))\end{array}}{\Delta;\Gamma \vdash \mathsf{super}(e_1); : \mathsf{void} : \mathsf{LS} \setminus direct(\mathsf{new\ C'})!\rho}$$

TS-VARWRITE

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash x : C : U : \bar{l} : \mathsf{LS}_1 \quad x \neq \mathsf{this} \\ \Delta;\Gamma \vdash e : C' : U' : \bar{l}' : \mathsf{LS}_2!\rho_2 \quad C' \prec C \\ U' = \mathsf{Normal} \vee (U' = \mathsf{Fresh} \wedge e \neq x') \\ U' \rightsquigarrow U \quad \mathsf{LS} = \mathsf{LS}_2 \odot \{\{x_+ : \bar{l}'\}, \emptyset, \emptyset\}\end{array}}{\Delta;\Gamma \vdash x = e; : \mathsf{void} : \mathsf{LS}!\rho_2}$$

TS-PLAINFIELDWRITE

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e : C_1 : U_1 : \bar{l}_1 : \mathsf{LS}_1!\rho_1 \\ \Delta;\Gamma \vdash e_2 : C_2 : U_2 : \bar{l} : \mathsf{LS}_2!\rho_2 \\ \Delta_f(C_1)(f)(\bar{l}_1) = C_3 : U_3 : \bar{l} : \emptyset \\ (\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}_2, \rho_2) = (\mathsf{LS}, \rho) \\ U_2 \rightsquigarrow U_3 \quad C_2 \prec C_3 \\ U_f = \mathsf{Normal} \vee (U_2 = \mathsf{Fresh} \wedge e_2 \neq x)\end{array}}{\Delta;\Gamma \vdash e.f = e_2; : \mathsf{void} : \mathsf{LS}!\rho}$$

TS-RESFIELDWRITE

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathsf{this} : C_1 : \bar{l}_1 : \mathsf{LS}_1!\rho_1 \quad C_2 \prec C_3 \\ \Delta;\Gamma \vdash e_2 : C_2 : \mathsf{Fresh} : \bar{l} : \mathsf{LS}_2!\rho_2 \\ \Delta_f(C_1)(f)(\bar{l}_1) = C_3 : \mathsf{Unique} : \bar{l} : r \\ (\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}_2, \rho_2) \odot (\{\mathsf{this.f}_+ : \bar{l}\}, \emptyset, \emptyset) = (\mathsf{LS}, \rho) \\ e_2 \neq x\end{array}}{\Delta;\Gamma \vdash \mathsf{this}.f = e_2; : \mathsf{void} : \mathsf{LS}!\rho \cup \{\mathsf{raw(this}.r)\}}$$

TS-IF

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \overline{s_1} : \mathsf{void} : \mathsf{LS}_1!\rho_1 \quad \Delta;\Gamma \vdash \overline{s_2} : \mathsf{void} : \mathsf{LS}_2!\rho_2 \\ \Delta;\Gamma \vdash e_1 : C' : \mathsf{LS}_1'!\rho_1' \quad \Delta;\Gamma \vdash e_2 : C;' : \mathsf{LS}_2'!\rho_2' \\ (\mathsf{LS}, \rho) = ((\mathsf{LS}_1', \rho_1') \oplus (\mathsf{LS}_2', \rho_2')) \oplus ((\mathsf{LS}_1, \rho_1) \otimes (\mathsf{LS}_2, \rho_2)) \quad C' \prec C'' \vee C'' \prec C'\end{array}}{\Delta;\Gamma \vdash \mathsf{if\ (e1 == e2)}\ \{\overline{s_1}\}\ \mathsf{else}\ \{\overline{s_2}\} : \mathsf{void} : \mathsf{LS}!\rho}$$

TS-BLOCK

$$\frac{\Delta;\Gamma \vdash s_1 \ldots s_n : \mathsf{void} : \mathsf{LS}!\rho}{\Delta;\Gamma \vdash \{s_1 \ldots s_n\} : \mathsf{void} : \mathsf{LS}!\rho}$$

TS-PE

$$\frac{\Delta;\Gamma \vdash pe : \tau : \bar{l} : \mathsf{LS}!\rho}{\Delta;\Gamma \vdash pe; : \mathsf{void} : \mathsf{LS}!\rho}$$

TS-NOOP

$$\frac{\vdash \Delta\ \mathsf{ok} \quad \Delta \vdash \Gamma\ \mathsf{ok}}{\Delta;\Gamma \vdash ; : \mathsf{void}}$$

T-BONE

$$\frac{C \prec_1 \mathsf{Object}}{\Delta;\Gamma, \mathsf{this} : C \vdash \mathsf{super}() : \mathsf{void} : \emptyset!\emptyset}$$

TS-STUPIDIF

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \overline{s_1} : \mathsf{void} : \mathsf{LS}_1!\rho_1 \quad \Delta;\Gamma \vdash \overline{s_2} : \mathsf{void} : \mathsf{LS}_2!\rho_2 \\ \Delta;\Gamma \vdash e_1 : C' : \mathsf{LS}_1'!\rho_1' \quad \Delta;\Gamma \vdash e_2 : C;' : \mathsf{LS}_2'!\rho_2' \\ (\mathsf{LS}, \rho) = ((\mathsf{LS}_1', \rho_1') \oplus (\mathsf{LS}_2', \rho_2')) \oplus ((\mathsf{LS}_1, \rho_1) \otimes (\mathsf{LS}_2, \rho_2)) \quad C' \not\prec C'' \wedge C'' \not\prec C'\end{array}}{\Delta;\Gamma \vdash \mathsf{if\ (e1 == e2)}\ \{\overline{s_1}\}\ \mathsf{else}\ \{\overline{s_2}\} : \mathsf{void} : \mathsf{LS}!\rho}$$

The need for the rule TS-DROPTHIS may be non-obvious. The rule allows us to explicitly abandon properties of this and register the corresponding resource as being mutated. This is needed in some cases when writing to fields that belong to resources, and when the surrounding method is annotated with this:-@p for some property. Once the property has been dropped at a specific point, constraints on what values can be

written to some fields may be less restrictive. It is also used when direct addition methods override other direct addition methods (see rule T-MBODY). Note that the user is not expected to write drop statements manually. The checker will insert them as required.

Statement sequences

MJ has two rules for statement sequences, TS-INTRO for sequencing where the first statement is a local variable declaration, and TS-SEQ for all other sequences. This approach changes the typing context appropriately to include the new variable in TS-INTRO. We have retained this approach, adjusted it to handle our effects appropriately, and added two new rules for sequencing of statements, TS-SEQVARWRITE for variable writes and TS-SEQFIELDWRITE for field writes. Field writes and variable writes are thus constrained twice: first in the judgments that type the write statements themselves, and then in the sequencing with other statements. The reason for this is that when a write such as $x = e$ is carried out, the expression e may already have its own label signature and mutation summary associated with it by the type system. As the expression is assigned to x, we must now associate the existing LS and ρ with the variable x instead of with the expression e. This is done by the helper functions $lflow$ and $rflow$ in these sequencing judgments. The exact same principle applies to field writes.

In these rules we also check that the mutations that will be carried out on the written field or variable *after* the write (in program execution order) are acceptable, by using the $\Delta; \Gamma \vdash e!\rho$ ok judgment.This prevents, for instance, the future mutation of a resource of a return value from some function call. Such mutations must be prohibited since they may place other objects in inconsistent states.

$$\text{TS-SEQ}\quad \frac{\Delta;\Gamma \vdash s_1 : void : \mathsf{LS}_1!\rho_1 \qquad s_1 \neq \mathsf{C\ x} \qquad s_1 \neq x = e \qquad \Delta;\Gamma \vdash s_2 \ldots s_n : \tau : \mathsf{LS}_2!\rho_2 \qquad \{r \mid \mathsf{raw}(r) \in \rho_1\} \cap \mathsf{res}(\mathsf{precond}(\mathsf{LS}_2)) = \emptyset}{\Delta;\Gamma \vdash s_1 s_2 \ldots s_n : \tau : (\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}_2, \rho_2)}$$

$$\text{TS-SEQVARWRITE}\quad \frac{\Delta;\Gamma \vdash x = e; : \mathsf{void} : \mathsf{LS}_1!\rho_1 \qquad \Delta;\Gamma \vdash s_2 \ldots s_n : \tau : \mathsf{LS}_2!\rho_2 \qquad \Delta;\Gamma \vdash e!\rho_2(x)\ \mathsf{ok} \qquad \Delta;\Gamma \vdash e : C : U \qquad (\mathsf{LS}_1, \rho_1) \oplus (lflow(\mathsf{LS}_2, x, e), rflow(\rho_2, x, e, C, U)) = (\mathsf{LS}, \rho)}{\Delta;\Gamma \vdash x = e; s_2 \ldots s_n : \tau : \mathsf{LS}!\rho}$$

$$\text{TS-SEQFIELDWRITE}\quad \frac{\Delta;\Gamma \vdash \mathsf{this}.f = e; : \mathsf{void} : \mathsf{LS}_1!\rho_1 \qquad \Delta;\Gamma \vdash s_2 \ldots s_n : \tau : \mathsf{LS}_2!\rho_2 \qquad \Delta;\Gamma \vdash e!\rho_2(\mathsf{this}.f)\ \mathsf{ok} \qquad \Delta;\Gamma \vdash e : C : U \qquad (\mathsf{LS}_1, \rho_1) \oplus (lflow(\mathsf{LS}_2, \mathsf{this}.f, e), rflow(\rho_2, \mathsf{this}.f, e, C, U)) = (\mathsf{LS}, \rho)}{\Delta;\Gamma \vdash x = e; s_2 \ldots s_n : \tau : \mathsf{LS}!\rho}$$

$$\text{TS-INTRO}\quad \frac{\Delta;\Gamma, x : C : U \vdash s_1 \ldots s_n : \tau : \mathsf{LS}!\rho}{\Delta;\Gamma \vdash \mathsf{C\ x{:}U}; s_1 \ldots s_n : \tau : (\mathsf{LS} \setminus \mathsf{LS}(x))!(\rho \setminus \rho(x))}$$

4.2.10 Well-formed class definitions, part 2

Each method has a *prior* and a *posterior expanded signature*, which describes the properties of the fields in the owning class, as well as of the receiver, argument and return value, before and after executing a method. Given a method type,

$$\Delta_m(C)(m) = (\mathsf{LS})(U_{\mathsf{rec}})C_a : U_a \to \tau : U_{\mathsf{ret}}!\rho$$

and given that the class C defines fields $f_1 \ldots f_n$, we define

$$\mathsf{LS}_{\mathsf{pre}}(C)(m) \stackrel{\mathsf{def}}{=} (\mathsf{precond}(\mathsf{LS}), \emptyset, \emptyset) \uplus \bigcup_i (\{f_i : \bar{l}_i^{\mathsf{pre}}\}, \emptyset, \emptyset) \qquad \text{where}$$

$$\Delta_f(C)(f_i)(\mathsf{precond}(\mathsf{LS})(\mathsf{rec})) = T_i : U_i : \bar{l}_i^{\mathsf{pre}} : r$$

$$\mathsf{LS}_{\mathsf{post}}(C)(m) \stackrel{\mathsf{def}}{=} (\mathsf{LS}_+ \setminus \mathit{direct}(\mathsf{LS}), \mathsf{LS}_=, \emptyset) \uplus \bigcup_i (\{f_i : \bar{l}_i^{\mathsf{post}}\}, \emptyset, \emptyset) \quad \text{where}$$

$$\Delta_f(C)(f_i)(\mathsf{postcond}(\mathsf{LS})(\mathsf{rec})) = T_i : U_i : \bar{l}_i^{\mathsf{post}} : r$$

A method body will only be well typed if it establishes $\mathsf{LS}_{\mathsf{post}}(C)(m)$ given $\mathsf{LS}_{\mathsf{pre}}(C)(m)$. The domain of non-expanded label signatures, which are given directly in the syntax, is the receiver this, the return value ret, and arguments of the method. The domain of expanded label signatures also includes those fields of the receiver that belong to resources.

Note that the expanded posterior signature does not include the method's direct addition labels: the programmer is promising that these labels will be established through direct manipulation of state.

One acn also think about the prior and posterior signatures as being "ghost statements" that are inserted before and after the corresponding method body, with given assumptions and guarantees. If the chaining of the method body between these ghost statements succeeds, then the method body is valid.

Example. Consider the following class.

```
1  class C {
2    resource r {
3      properties @a, @b, @c, @d;
4      C f:(@a->b, @c->@d);
5    }
6
7    void m(C x) x: @a, this: -@a, +@c. {
8      //...
9    }
10 }
```

The method C.m will have the prior expanded signature

$$\mathsf{LS}_{\mathsf{pre}}(C)(m) = (\{\{\mathsf{this} \mapsto \{@a\}, x \mapsto \{@a\}, f \mapsto \{@b\}\}, \emptyset, \emptyset\})$$

The posterior expanded signature is

$$\mathsf{LS}_{\mathsf{post}}(C)(m) = (\{\{\mathsf{this} \mapsto \{@c\}, x \mapsto \{@a\}, f \mapsto \{@d\}\}, \emptyset, \emptyset\})$$

As a rule, only mutations that can be tracked in expanded signatures are permitted. The only exception is mutation of fresh variables, which is always permitted from the point

of view of a given method, since nobody else might have a reference to them, and thus nobody else can make assumptions about their state.

We can now give the remaining well-formedness judgments. Unlike in MJ, in Poplar$_0$ method bodies are well-formed only if they satisfy their contract by establishing the posterior expanded signature given the prior expanded signature. Thus, well-formedness of Poplar$_0$ method bodies depend on the Poplar types of their statements, and by extension, well-formedness of classes also depends on this.

T-MBODY

$$\frac{\begin{array}{c}\Delta_m(C)(m) = (U_{\mathsf{rec}})C_a : U_a \to \tau : U_{\mathsf{ret}} : \mathsf{LS}!\rho \\ \Delta; \Gamma \vdash s' : \tau' : U'_{\mathsf{ret}} : \mathsf{LS}'!\rho' \\ \tau' \prec \tau \qquad U'_{\mathsf{ret}} \rightsquigarrow U_{\mathsf{ret}} \\ mbody(C, m) = (x, \overline{s}) \qquad C \prec_1 C' \\ (LS_{\mathsf{pre}}(C)(m), \emptyset) \oplus (\mathsf{LS}', \rho') \prec (\mathsf{LS}_{\mathsf{post}}(C)(m), \rho) \\ \Gamma = \{\mathsf{this} : C : U_{\mathsf{rec}}^{\mathsf{arg}}, x : C_a : U_a^{\mathsf{arg}}\} \\ (direct(\mathsf{LS}) \neq \emptyset \wedge m \in dom(\Delta_m(C'))) \implies \overline{s} = ((\mathsf{C'})\mathsf{this}).\mathsf{m}(\mathsf{e}); \mathsf{drop}\ direct(C')(m); \overline{s_r}\end{array}}{\Delta \vdash altmbody(C, m, \overline{s'})\ \mathsf{ok}}$$

T-CDEFN

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \mathsf{super(e);} : \mathsf{void} : \mathsf{LS}_1!\rho_1 \\ \Delta; \Gamma \vdash \overline{s} : \mathsf{void} : \mathsf{LS}_2!\rho_2 \\ cnbody(C) = (x, \mathsf{super(e);}\, \overline{s}) \\ \Delta_c(C) = C_1 : U_1 : LS!\rho \\ \Gamma = \{\mathsf{this} : C : \mathsf{Fresh}\ , x : C_1 : U_1^{\mathrm{arg}}\} \\ (\mathsf{LS}_{\mathsf{pre}}(C)(C), \emptyset) \oplus (\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}_2, \rho_2) \prec (\mathsf{LS}_{\mathsf{post}}(C)(C), \rho)\end{array}}{\Delta; \vdash C\ \mathsf{cok}}$$

T-MBODYS

$$\frac{\begin{array}{c}\Delta \vdash mbody(C, m_1)\ \mathsf{ok} \ldots \Delta \vdash mbody(C, m_n)\ \mathsf{ok} \\ dom(\Delta_m(C)) = \{m_1 \ldots m_n\} \qquad C \prec_1 C' \\ \{m \mid m \in dom(\Delta_m(C')) \wedge direct(m) \neq \emptyset\} \subseteq dom(\Delta_m(C))\end{array}}{\Delta \vdash C\ \mathsf{mok}}$$

T-MDEFN

$$\frac{\begin{array}{c}\Delta \vdash altmbody(C, m, \overline{s})\ \mathsf{ok} \\ mbody(C, m) = (x, \overline{s})\end{array}}{\Delta \vdash mbody(C, m)\ \mathsf{ok}}$$

T-PROGDEF

$$\frac{\begin{array}{c}\Delta \vdash C_1\ \mathsf{cok}\ \ldots \Delta \vdash C_n\ \mathsf{cok} \\ \Delta \vdash C_1\ \mathsf{mok}\ \ldots \Delta \vdash C_n\ \mathsf{mok} \\ dom(\Delta) = \{C_1 \ldots C_n\}\end{array}}{\Delta \vdash p\ \mathsf{ok}}$$

4.2.11 Queries and satisfaction of queries

In principle, queries are statements that make no assumptions about their environments' labels, but provide the labels that they promise to establish (the solutions to queries may make assumptions about labels, though). We also give two judgments, SAT-PRODUCE and SAT-TRANSFORM, that are not typing judgments. Instead, they describe the constraints on valid solutions to queries.

$$\text{TS-PRODUCE}\quad \frac{\Delta;\Gamma \vdash x : C' : U' : \overline{l} : \mathsf{LS}'!\rho' \qquad C' \prec C}{\Delta;\Gamma \vdash x = \#\mathsf{produce}(C,\overline{l},U);: C : U : \overline{l} : \emptyset!\emptyset} \quad \text{where } U \rightsquigarrow U' \text{ and } U \neq \mathsf{Unique}$$

$$\text{SAT-PRODUCE}\quad \frac{mbody(C)(m) = (x, (\overline{s_1}\ x = \#\mathsf{produce}(C,\overline{l},U);\overline{s_2})) \quad \Delta;\Gamma \vdash e' : C' : U' : \overline{l} \qquad C' \prec C \quad \Delta \vdash altmbody(C,m,\overline{s_1}\ \overline{s}; x = e;\ \overline{s_2})\ \mathsf{ok}}{\Delta;\Gamma \vdash \overline{s}; x = e\ \mathsf{satisfies}\ \#\mathsf{produce}(C,\overline{l},U)} \quad \text{where } U' \rightsquigarrow U$$

$$\text{TS-TRANSFORM}\quad \frac{\Delta;\Gamma \vdash x : C : U : \overline{l} : LS!\rho}{\Delta;\Gamma \vdash \#\mathsf{transform}(x,\overline{l}');: \mathsf{void} : \emptyset!\emptyset}$$

$$\text{SAT-TRANSFORM}\quad \frac{mbody(C)(m) = (x_a, (\overline{s_1}\ \#\mathsf{transform}(x,\overline{l})\ \overline{s_2})) \quad \Delta;\Gamma \vdash \overline{s_1}\ \overline{s} : \tau : U : \overline{l}' : LS!\rho \quad \Delta \vdash altmbody(C,m,\overline{s_1}\ \overline{s}\ \overline{s_2})\ \mathsf{ok}}{\Delta;\Gamma \vdash \overline{s}\ \mathsf{satisfies}\ \#\mathsf{transform}(x,\overline{l})} \quad \text{where } \overline{l} \subseteq \mathsf{postcond}(LS)(x)$$

4.3 Poplar$_1$: Adding External Resources and Composite Properties

Poplar$_1$ extends Poplar$_0$ by adding several features that allow us to describe usage constraints of realistic Java programs more practically. Most importantly, we add *external resources*, which allow the concrete state of one object to describe properties of another object. We also add *composite properties*, which allow us to conveniently refer to a group of properties by a single name.

New and changed syntactic forms

The syntax of Poplar$_1$ is the same as for Poplar$_0$, except for the following changed and newly introduced syntactic forms.

Class definition

cd ::= class C extends C
$\{fd_1 \ldots fd_j\ cnd\ cpd_1 \ldots cpd_k$
$rd_1 \ldots rd_m\ md_1 \ldots md_n\}$

Resource definition

rd	::=	resource $\{\ rb\ \}$	Resource
	$\mid$	resource[C] r $\{\ rb\ \}$	External resource
rb	::=	properties $@p_1, \ldots @p_n; fd_1 \ldots fd_l$	Resource body

Qualified resource

qr	::=	$rs.rn$	Qualified resource
rs	::=	$x \mid \mathsf{any}(C)$	Resource subject
rn	::=	r	Resource name
	$\mid$	$\mathsf{ext}(C).r$	Ext. resource name

Composite property

cpd	::=	composite $@p = (@p_1, \ldots @p_c)$	Composite property definition

New and overridden judgments in Poplar$_1$

Subsumption of resources

$$\frac{\Delta;\Gamma \vdash x : C \qquad \Delta;\Gamma \vdash C \prec C'}{\Delta;\Gamma \vdash x \prec_s \mathsf{any}(C')} \qquad \frac{\Delta;\Gamma \vdash C \prec C'}{\Delta;\Gamma \vdash \mathsf{any}(C) \prec_s \mathsf{any}(C')} \qquad \frac{\Delta;\Gamma \vdash rs \prec_s rs'}{\delta;\Gamma \vdash rs.r \prec rs'.r}$$

$$\frac{\Delta;\Gamma \vdash rs \prec_s rs' \qquad C \prec C'}{\Delta;\Gamma \vdash rs.\mathsf{ext}(C).r \prec rs'.\mathsf{ext}(C').r} \qquad \frac{\Delta;\Gamma \vdash C_E \prec C'_E}{\Delta;\Gamma \vdash \mathsf{raw}(\mathsf{ext}(C_E)).r \prec \mathsf{any}(C'_E).\mathsf{ext}(C).r}$$

$$\frac{\Delta;\Gamma \vdash x : C'_E \qquad C_E \prec C'_E}{\Delta;\Gamma \vdash \mathsf{raw}(\mathsf{ext}(C_E)).r \prec x.\mathsf{ext}(C).r}$$

Well-formedness of class definitions

T-FIELDOK1

$$\frac{\begin{array}{c} C \prec_1 C' \qquad \Delta \vdash C_f\ \mathsf{ok} \qquad \bar{l}'_f \subseteq \bar{l}_f \\ \forall r \in fr(C)(f) \exists r'.r' \in fr(C')(f) \wedge r \prec r' \\ \Delta_f(C)(f) = C_f : U_f : \bar{l}_f \\ \Delta_f(C')(f) = C'_f : U_f : \bar{l}'_f \end{array}}{\Delta \vdash C.f\ \mathsf{ok}}$$

T-PROPOK

$$\frac{\begin{array}{c} \forall r \in \mathsf{res}(C)(@p).\exists r'.r' \in \mathsf{res}(C')(@p) \wedge r \prec r' \\ C \prec_1 C' \end{array}}{\Delta \vdash @p\ \mathsf{ok}}$$

Resource field write

TS-RESFIELDWRITE

$$\frac{\begin{array}{c} \Delta;\Gamma \vdash \mathsf{this} : C_1 : \bar{l}_1 : \mathsf{LS}_1!\rho_1 \qquad C_2 \prec C_3 \\ \Delta;\Gamma \vdash e_2 : C_2 : \mathsf{Fresh}\ : \bar{l}_2 : \mathsf{LS}_2!\rho_2 \\ \Delta_f(C_1)(f)(\bar{l}_1) = C_3 : \mathsf{Unique}\ : \bar{l}' : rn \\ (\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}_2, \rho_2) \odot (\{\mathsf{this.f}_+ : \bar{l}_2\}, \emptyset, \emptyset) = (\mathsf{LS}, \rho) \\ \bar{l}' \subseteq \bar{l}_2 \text{ and } e_2 \neq x \end{array}}{\Delta;\Gamma \vdash \mathsf{this}.f = e_2; : \mathsf{void} : \mathsf{LS}!\rho \cup \{\mathsf{raw}(rn)\}}$$

A composite property $@p = (@p_1, \ldots @p_n)$ is well-defined if each individual property $@p_i$ is declared independently in some (possibly external) resource. We define $\mathsf{res}(@p) \stackrel{\mathsf{def}}{=} \bigcup_i \mathsf{res}(@p_i)$, which affects the sens and rem mappings accordingly. Composite properties may recursive contain other composite properties but are not well-defined if the definition is cyclical. They behave like macros and are rewritten to

sets of their constituent simple properties when method types, field types and queries are formed from the source code. We do not specify this rewriting formally here.

The field type has been changed to indicate either the resource or the external resource (the two alternatives for the rn syntax) that a field may belong to.

External resources allow one class to host concrete state for another, providing it with properties. This allows us to models constructs such as lists and containers. In the judgments that concern external resources, we have identified external (hosted) types by the metavariables C_E and C'_E. C_E is the class that receives its properties from another class that contains concrete state. An example can be seen in Figure 3.1.

4.4 Discussion

4.4.1 Soundness

Although we do not provide a full type safety proof for our system, we believe that it is fundamentally sound. In this section we discuss its soundness properties and how they bear on the overall design and purpose of Poplar.

First, as we have already argued, for each syntactic form that remains in the compiled program, Poplar either strengthens typing judgments from MJ or leaves them unchanged. So in a MJ/Java sense of the word, type soundness has not been violated. The set of well-typed Poplar_0 and Poplar_1 programs is a subset of the set of well-typed MJ programs.

In order to discuss ther properties that we want a well-typed Poplar program to have, let us first establish a notion of Poplar soundness. Fundamentally we are interested in reasoning about the label sets of expressions, in the form of a *may-analysis*. It is always acceptable to underestimate the set of labels that an expression currently has, since we never "do anything" (generate code) based on the absence of a label, but only on the presence of one. Poplar has no negative preconditions. In order to establish this soundness, we need to argue two points: first, that *observed (used) labels have always been established somewhere prior to their use*, and second, that *used labels have not been erased prior to their use*.

Establishing labels

Labels are established directly by methods with ++l annotations. For properties, the method is expected to explicitly change some mutable state that corresponds to the property's predicate. For tags, the label need not correspond to a predicate, so there is no such expectation. In Poplar_0/Poplar_1, new labels can be associated with an expression in the following ways:

- By being provided as preconditions for a parameter or a receiver of a method or constructor
- As postconditions of invocations of other methods or constructors
- As conditional labels of a resource field (corresponding to the current labels of the field's owner)

When a method is invoked with preconditions, the invocation is checked by Poplar, so the first case only provides labels indirectly. In the case of an invocation of another

method, it may provide labels directly with ++l annotations, or indirectly with +l annotations. In the latter case it is checked; in the former case the user is guaranteeing that the label has been established. Finally, in the case of resource fields, any assignments are checked by Poplar and verified to conform to the field's constraints. Thus, all new labels have a single common origin in ++l annotations of methods and constructors.

Erasing labels

Labels are either tags and properties. For tags, since they do not correspond to any predicate or mutable state of expressions, it is impossible to erase them. They are intended to correspond to immutable facts about expressions. The interesting case, then, is the removal of properties.

The predicates that properties correspond to are not encoded anywhere but instead explicitly set up by their basic establisher methods (which have an annotation of the form ++@p). Each property must have at least one such basic establisher, or it cannot come into existence. Just as properties can be established both directly and indirectly, they can be lost directly and indirectly. A direct loss of a property would be a change of the mutable state that corresponds to the property, which also invalidates its predicate. An indirect loss of a property would be if a method invokes another method that directly invalidates the property. We are thus concerned with two problems: taking note of "direct losses" in the first place, and propagating them once they have been noticed.

Properties can in fact correspond to any state at all, not just state in the resource that the property belongs to. This is a design choice that gives the programmer the freedom to encode complex state in many different ways. For instance, in Poplar mixed with full Java, a single field, such as an array, could conceivably be used to encode many different properties in different resources. There are also native methods, which handle resources such as I/O handles that have no direct representation in Java.

The direct loss of a property is easiest to track if the property's state consists only of fields that are both not shared with other resources and declared inside the property's resource. In this case, the Poplar system will insist that a resource mutation should be declared whenever those fields are written (in rule TS-RESFIELDWRITE). In this case it is hard for the programmer to make a mistake and erase properties without declaring it. In more complex cases, the property will consist of state that cannot be tracked by Poplar, such as native methods or fields that do not belong to resources. In these cases, just as it is the programmer's responsibility to establish properties fully in ++@p methods, it is the programmer's responsibility to annotate all methods that could potentially destroy the property with the corresponding mutates annotation for the resource. We believe that in well-designed classes, where related state is grouped together - this is one of the cornerstones of object-oriented design - this will not be a difficult task.

Once the mutation of a resource has been indicated, the Poplar effect system will insist that this information is propagated accordingly, much in the same way Java exceptions are propagated. *An invocation of a top level method will always be annotated with all the externally visible resource mutations that can potentially take place.* In particular, mutations of any aliased variables and parameters and receivers passed in will always be visible. The only mutations that may be hidden are those of fresh or unique references that are not reachable from the outside while a method invocation takes place.

The main source of potentially lost mutations in the tracking of resource mutations is that for external resources, we have no way of associating the mutation of the re-

source with a specific object or group of objects. Thus, we rely on the programmer to declare this information honestly in those methods that perform such direct mutations of external resources. We leave potential remedies of this problem for future work. The approach taken in FUSION [58] may be applicable to this problem.

Towards a type safety proof

We have not provided a type safety proof in this chapter. Parkinson et al. prove the correctness of their version of the Boyland-Greenhouse effect system for MJ by making reads and writes of regions explicit and tracking them in an extended semantics [18]. It is possible that in the future, the correctness of Poplar_0 and Poplar_1 might be proved using a similar approach. We believe that Lemma 1 would be helpful in a type safety proof, since many of the judgments use the chaining operation, and this lemma demonstrates that label signatures remain well-formed after chaining, so further chaining will always be possible.

Summary

We now summarise our soundness argument. In Poplar, programmers have two key responsibilities with regards to label tracking: ensuring that properties are indeed correctly established by ++@p methods, and that they report all the ways that such a property may be erased by annotating methods with the corresponding mutates annotations. As long as this has been done correctly, Poplar will conservatively propagate this information throughout the program. We believe that Poplar provides a sound must-analysis for labels of expressions, and a sound may-analysis for aliasing and for resource mutations.

4.5 Related Work

In this section we discuss related work in the areas of typestate checking, effect systems, and alias confinement, since these areas are closely related to the design of Poplar_0 and Poplar_1. We discuss a wider range of related work in Section 7.1.

4.5.1 Typestate and protocols

Typestate checking was originally introduced by Strom and Yemini in 1986 [110] as a means of checking the degree of initialisation of each variable statically. The language that they study is not object-oriented, and does not permit arbitrary pointer assignment. They capture errors such as access of unitialised variables. In their system, typestates form a lower semilattice that describe the degree of initialisation of each type. Transitions between typestates occur as a result of program statements. As a result, restrictions on the typestate transitions form a finite state machine that constrains valid statement sequences in the language, excluding many nonsensical program errors. Yellin and Strom also require that there should exist a unique *typestate coercion* from higher to lower typestates, so that de-initialisations can always be found.

Object protocols are closely related to but different from typestate in that they emphasise valid message/method sequences that objects may receive and send. As such, they were first defined by Yellin and Strom [125]. They define a notion of protocol compatibility where a protocol is compatible with another if there is a sublanguage

relation between the languages represented by the protocols' automata. They consider software component in a very general sense, considering both concurrent and non-concurrent implementations. They describe an algorithm that, given an *interface mapping*, either synthesizes a protocol adapter, or concludes that no such adapter exists. Such adapters can store temporary data in memory cells, and translate incoming message sequences to the appropriate outgoing sequences.

In a later object-oriented adaptation of typestate by DeLine and Fähndrich [27], the requirement that typestates should model degrees of initialisation was dropped. DeLine and Fähndrich, and most of the following research on object-oriented typestate analysis, instead focus on the finite state machine aspect. Their system, called Fugue, is designed for a subset of C#. They get around the aliasing problem by providing explicit annotations such as NotAliased and MayBeAliased, similar to our Unique and Normal . These are typestates in themselves.

A key contribution of Fugue that makes object-oriented typestates intuitive is the introduction of *frame typestates* and *sliding methods*. In general, a typestate corresponds to a predicate on the concrete state of an object, that is, its fields[1]. Subclasses of a class that implement logically discrete states often extend the definition by adding more fields, implementing the same state change in a slightly richer way. In other words, subclasses must be able to override and extend typestates in the same way that they can override methods. However, subclass methods often make use of superclass methods when they implement their extended version of a particular state. To handle this, a typestate in fugue is modelled as a series of frame typestates - one part of the full state belongs to each class frame, that is, for each superclass level. Overriding, sliding methods can express what frame they assume for a particular state to be realised at, and what level they ultimately extend it to, if any. Each sliding method is assumed to make a call to its superclass counterpart before implementing its own additions to the state.

In this chapter, for the sake of simplicity, we avoided this problem by requiring that properties are always required across all class frames simultaneously. We do not permit the use of a property until it has been fully initialised at all levels in its class hierarchy. This is a significant restriction for the programmer, and it would be worthwhile to investigate, as a future extension, an explicit modelling of different degrees of establishment of a property. We expect that this can be done in the same way as Fugue models typestates; one challenge would be keeping the syntax simple while doing so.

The concrete predicates that typestates correspond to are typically not specified. Fugue specifies whether references may or may not be null, but no other constraint on an object's data is encoded. External contract checkers such as ESC/2 may be applied to confirm that typestates indeed correspond to given predicates. However, this is typically not a problem of great concern, since typestate transitions tend to be declared together with mutations of the data that they concern, especially in object-oriented programs.

Bierhoff and Aldrich add several new capabilities [14] beyond what was supported by Fugue. They introduce the notions of state refinement, method refinement and state dimensions to allow typestates to be integrated in a more natural way with objects, and in particular with behavioural subtyping, in which overridden methods may, for instance, have weaker preconditions and stronger postconditions. In their system, subclasses can refine states by adding increasingly fine-grained substates and refine methods by making use of the new substates.

[1] In full Java objects may also have access to state outside of the language itself through the use of native methods

Aldrich et al have proposed typestates as first-class objects supported by the type system [3]. They also deal with the aliasing problem by introducing explicit aliasing annotations such as shared, immutable, conserved, etc. in the language, thereby encoding the programmers' intentions. They introduce the programming language Plaid [113] as an implementation of these ideas. Plaid targets the JVM but emphasises runtime state change, and states have explicit runtime support. Plaid does not use the typestate formalism as an integration aid, but more traditionally as a formalism that supports programmers in simplifying the development of correct software.

Naeem and Lhotak [82] have defined a formal lattice-based operational semantics for multiple interacting objects, expanding the range of what can be checked from constraints on a single object to multi-object interactions.

Fink et al have implemented a multi-stage typestate verifier [32] for a subset of the Java language, excluding concurrency and special features such as generics. For selected typestate properties across a range of moderately sized input programs, their system detects errors with 93% accuracy in 10 minutes.

On the theoretical side, Field et al have obtained results [31] about the complexity of verifying typestate correctness in shallow programs, that is, programs where heap-allocated objects do not contain pointers to other objects. They demonstrate that depending on the state being verified, some instances can run in polynomial time, while others are more complicated. They are able to show that verifying that a file is not read after it has been closed can be done in polynomial time; verifying that a file is not read before it has been opened is PSPACE-complete. The reason for this high complexity is that the programs are basically unconstrained except for the requirement that they should be shallow. We will see in the next chapter that Poplar label checking is a much less complex problem.

Alfaro and Henzinger consider the verification of interface automata in their 2001 paper [5]. Their approach is based on an optimistic game-theoretical model; they assume that two interfaces are compatible if some environment E that makes them compatible exists, rather than considering them incompatible if there is some environment E' that makes them incompatible.

Alur et al describe [7] how to synthesise interface specifications for Java classes from their source code. They start by extracting the state transition system using predicate abstraction, and then solve a partial-information two player game on the representation obtained. Their method yields a maximally safe interface, which can, in the context of Java, take actions such as throwing exceptions when it detects an invalid method sequence. They evaluate a tool, JIST, for the extraction of these interfaces, and show it to be reasonably fast for moderately large classes.

In addition to static specification and checking of typestate, there are systems that learn and enforce temporal specifications at runtime. Gabel and Su [36] have designed one such system and showed that it produced valuable results on real world systems.

Kim, Bierhoff et al showed how to encode typestates in the JML Java specification language [16]. This makes the typestate concept available to the wide range of tools that are founded on JML and avoids the need for a nonstandard syntax.

Kuncak, Lam and Rinard redefined typestates in a more general way [65]: instead of associating a state with each object, they track general sets that correspond to states, and objects in a given set are considered to have the state associated with it. This fits abstract data types (ADTs) naturally.

We discussed how Prospector provides interactive developer support and suggests code fragments to the user based on pre-mined protocols. A similar system is Alfonso's system [6], in which protocols are explicitly specified in advance. An interactive IDE

plugin gives the user concrete suggestions as to how to change the code they have written in order to conform with the protocol specifications. This is an example of a use of the protocol formalism that does not constrain programmers, but supports them in ways that they may freely choose to accept or reject.

4.5.2 Effect systems

Type and effect systems extend type systems by also tracking side effects. Such systems can ensure that there is no unwanted interference between two code fragments. For an accessible overview of effect systems, a good choice is the book by Nielson, Nielson and Hankin [84, Chapter 5], though this book does not focus on Java.

The Boyland-Greenhouse effect system [46] was a key inspiration for this work. Its key contribution is grouping Java fields into abstract regions, which may also be refined into subregions. Methods specify both what regions they read and what regions they write, yielding the ability to identify when two methods (or fragments) interfere with each other. However, unlike our system, interference cannot be characterised as a set of lost properties. Only the absence or presence of interference can be detected.

Data groups [71, 70] were a precursor to the Boyland-Greenhouse effect system. They name groups of fields in objects polymorphically, and then annotate methods with information saying what groups they may modify. The main difference with the B-G system is that data groups are not disjoint, which means that the system cannot be used to detect interference.

The first effect system to be studied was in FX-87 [43]. Talpin and Jouvelot developed systems that permitted effect inference [115, 61], which reduces the burden of having to annotate each method by hand.

One form of effect analysis that has received special interest in the context of Java is *purity analysis*. A method is pure if it has no observable side effects. Note that this still permits the method to mutate its own internal state. Sălcianu and Rinard contribute a purity and side effect analysis [98] that combines escape analysis and pointer analysis. They are able to classify parameters into read-only parameters, through which no transitively reachable references are mutated, and safe parameters, whose transitively reachable references the method does not create any new parameters to. However, this system, and the other ones discussed so far, are not modular. Pearce developed one of the first *modular* purity systems for Java [93], which uses the properties of freshness - similar to the notion of fresh references used in this chapter - and locality, which is a slightly modified form of ownership. The notion of modularity used in his system is similar to Poplar's annotations in the sense that it generates summary information (annotations) for each method, and that a method can be checked by inspecting only its local source code and the summaries of methods it references.

4.5.3 Alias confinement

The concept of unique pointers was first studied by Minsky et al [80].

Clarke formalised ownership analysis as an extension of Pizza [24]. His system does not relate objects directly to each other, but rather relates *contexts* to each objects. Ownership analysis is at the expense of a slightly more complicated syntax, since objects must be parameterised with owner references if they are to take advantage of ownership annotations.

Aldrich et al have developed AliasFJ, a Featherweight Java (FJ) extension with alias annotations and a notion of ownership that relates objects directly to each other.

They have also showed how to infer alias annotations.

Islands [51] are a notion of encapsulation especially designed for alias confinement. An island is a subgraph of the heap reference graph. Objects in an island may hold any number of references to each other, but there must only be a single object, the ”bridge”, through which the island can be reached. This constraint is only for static references, though. Dynamic (stack) references between objects in an island and objects outside the island are permitted in either direction. We are not aware of any implementation of this concept, although it can be seen as a precursor to alias ownership.

Zhao, Palsberg and Vitek have developed the notion of confined types [133], also for FJ. Assuming a notion of a module, they assume that each module has a set of types that is strictly intended for internal use and a set intended for public use. Aliases are constrained on a type basis: references of confined types in a module may not be leaked to objects that belong to types of other modules.

A more general version of the alias confinement problem is *points-to analysis*, which has been studied for a long time in C-family languages, including Java. Points-to analyses have a wide range of applications, not least in compiler construction (CITE). The question asked here is, for a given reference, which objects may it point to? Such analyses are typically classified according to whether they are context-sensitive, whether they are flow-sensitive, and whether they are interprocedural or intraprocedural. Michael Hind's 2001 survey [50] gives an overview of classical results and questions in the area, which include algorithms include Steensgaard's algorithm [107] and Andersen's algorithm [8].

For a language like Java, it is generally hard to obtain precise points-to-analysis results through a modular analysis, and as we have already mentioned, Java's incremental classloading makes non-modular analyses a poor match for the language. Salcianu developed a method that is able to perform a correct analysis by storing pre-computed parameterised summaries for each method, and then later instantiating this information [101].

4.5.4 Other related work

Summers and Müller have proposed a lightweight type system for object initialisation [111]. It uses a simple escape analysis to prevent objects from escaping before they have been fully initialised. This could potentially be combined with our basic establishers to give programmers more freedom in how properties are initialised.

Jaspan and Aldrich [58] have developed an analysis, FUSION, for checking *relationships* in frameworks. This allows objects to relate to each other by means of named relationships, such as the relationship of being an item in a container. Interfaces can specify invariants, preconditions and postconditions in terms of relationships and their changes. This formalism corresponds, on some level, to the notion of external resources in Poplar, and it is likely that a combination of relationship analysis with resources could increase Poplar's precision significantly, especially with regards to the problem of identifying the object that corresponds to an external resource mutation.

4.6 Conclusion

In this chapter, we formalised Poplar, first as a minimal subset Poplar_0, and then as the slightly extended version Poplar_1, which provides external resources. We gave the

syntax of both systems, as well as typing judgments, well-formedness judgments and helper functions needed to type Poplar programs.

Poplar fundamentally provides a must-analysis for labels, and a may-analysis for mutation of resources, which is the main way that labels may be destroyed. Although we have not provided proved these analyses correct, we argue that we are always able to detect mutation of resources and require that labels are established before being used, and that we are able to propagate this information accurately.

The analysis presented in this section is fully modular and only makes use of the local source code and summary information of other classes' methods when judgments are evaluated. This chapter did not discuss how to implement a Poplar type checker; we will describe our implementation of such a checker in Chapter 5.

We saw that a complication is introduced by subclasses strengthening the meaning of properties by overriding basic establisher methods. In such cases, a property can be in one of several degrees of initialisation: it might have been established for some higher level class frames but not for some lower level ones. We solve this provisionally by insisting that properties should always be established for all class frames simultaneously, and making the programmer responsible for not making use of properties that have only been partly established. We leave a better solution for future work.

The main source of inaccuracy in the analysis is the way that aliases are handled: mutations are always identified either as $x.r$ for some variable x or as $any(C).r$ for some type C, the latter case corresponding to any variables that might be aliased. This is sound but not precise. A source of possible lost mutations is the fact that external resource mutations cannot be definitely linked to the variable that they occur on. It is the programmer's responsibility to identify such variables correctly, which adds a potential risk when the system evolves. However, we can always correctly identify the fact that *some* variable's external resource has been mutated.

We emphasise that the design of Poplar is not tightly coupled to any particular alias confinement system. Our approach based on uniqueness kinds was chosen for its relative simplicity, but a more sophisticated approach, for instance one that contains a notion of ownership, such as in AliasFJ [2], may be used to increase the precision of our approach.

This chapter has not discussed how to implement a checker or query solver for the system we have presented; we will address this in the next chapter.

5

The Design and Implementation of a Poplar Compiler

In this chapter we describe the design and implementation of Jardine, a Poplar compiler. Alexandre Pichot, of the University of Pierre and Marie Curie in Paris, France, contributed significantly to its development.

5.1 Selecting a foundation for Jardine

Since Poplar is an extension of Java, rather than implement an extended Java compiler from scratch, we chose to extend an existing Java compiler. In order to reduce the complexity of the implementation effort, and given that we deal with the core features of Java rather than sophisticated ones, we decided to base our work on a research compiler rather than an industrial compiler such as Oracle's official OpenJDK. Several Java compiler frameworks were considered.

JaCo [131] was developed to support the Java extension Keris[129].

JastAddJ JastAdd [29] is built on a novel aspect-oriented architecture, which facilitates addition of features to Java by writing only a very small amount of code, avoiding the need to explicitly handle the new feature in many different places throughout the compiler.

Jikes [52] is a high performance compiler developed by IBM.

JKit JKit [92] is a straightforward, recent Java 5 compiler developed by David Pearce for research and teaching purposes. It has been used to develop a modular Java purity checker [93] and is currently being used to develop the Whiley language.

Polyglot Polyglot[86] is a straightforward Java compiler designed for research and teaching. It has been used to develop a variety of Java extensions.

Spoon [55] is a Java compiler and processing framework developed by INRIA. It is well integrated with Eclipse and provides a Spoonlet architecture that allows new analyses and transformations to be plugged in easily.

We decided to use JKit as a basis for our compiler for several reasons. Firstly, a recent release is available. At this time of writing, the most recent release of JKit is from 2010, making it more current than the other compiler frameworks. Second, JKit has relatively mature support for Java 5, which included major changes to the Java

language, such as generics. This means that tools based on JKit can be used to process large, modern Java programs, as long as they do not invalidate existing functionality. Third, JKit has a relatively simple design that follows classical compiler principles, which was helpful in reducing the overall complexity of the implementation task.

In the following, we first describe the structure of JKit and the overall strategy we used when implementing Jardine. We then describe how Jardine modifies JKit and adds new components. We describe the Jardine compilation pipeline and each compilation stage, including key algorithms. After this we give some concluding remarks. Jardine has been released as open source software and is available for download from `http://www.poplar-lang.org`.

5.2 The tasks of a Poplar compiler

What tasks should a Poplar compiler perform? We discussed a possible workflow for Poplar software development in Section 2.4.9. Three tasks that rely on automated tool support were identified.

Query solving in which concrete integration code is constructed and substituted for queries. Such code is correct by construction.

Poplar checking or *method contract checking*, in which method and constructor bodies are verified against their method signatures. This is the equivalent of performing typestate checking.

Integration link verification in which, following a component upgrade, the new versions are tested for compatibility with the rest of the components without recompilation or re-solving of queries.

Our implementation performs query solving (see Section 5.9 and Poplar checking (see Section 5.8). Due to time constraints we have not implemented integration link verification, but we discuss how it could be implemented in a straightforward manner in Section 5.10.

It is interesting to note that these three tasks are, to a degree, independent. It is possible to benefit from Poplar checking without using the automated integration scheme at all. In this case Poplar would act purely as a static checker. It is also possible to use the automated integration without checking methods for conformance with their contracts, although we believe that this would have little merit, since it would be easier to introduce errors by mistake, and it is natural to try to reap the maximum benefit once one has added Poplar annotations to source code.

Our aim in producing this implementation has been to create a proof-of-concept compiler that is sound. We have not attempted to optimise for performance or to use the most sophisticated implementation techniques available. Whenever possible, we tried instead to use straightforward algorithms and data structures.

5.3 Mixed Java and Poplar compilation

The formalisation presented in Chapter 4 only specifies Poplar for an imperative core of Java. But JKit, the basis for Jardine, implements a full compiler for Java 5, which has additional features not specified in Poplar_0 or Poplar_1, including arrays, exceptions,

interfaces, abstract classes and generics. This means that Jardine necessarily has a different scope from the formalisation. It is compiling a mix of Java 5 and Poplar_1.

Jardine implements all of the Poplar_0 and Poplar_1 specifications, with two notable exceptions: the (drop) TS-DROP statement (Section 4.2.9) and well-formed overriding. The drop statement was left out due to time constraints. However, omitting it can never lead to unsound results, only to the inability to find valid Poplar types or solutions in some cases. Well-formed overriding of methods and fields was also unimplemented due to time constraints. These functions should be straightforward to implement in the future; a checking algorithm follows from the well-formedness judgments in a straightforward way (Section 4.2.6). In the absence of such checking, Jardine simply assumes that any overridden methods and fields have acceptable Poplar signatures.

There are also some features that have been implemented but not formalised. The most important difference is that Jardine can compile all of Java 5, which means that many statements lack a clear Poplar semantics. We handle this by partitioning methods and constructors into *Poplar methods* and *plain methods*, according to whether they have a Poplar signature or not. For Poplar methods we require that all statements should have a well defined Poplar semantics, and we check the method in full. For plain methods we perform no special checking, and compile them as ordinary Java methods. This raises the question as to what to do when data flows between Poplar methods and plain methods. Such data flow can pose a risk in two ways.

Data flow from plain to Poplar methods. This can happen if, for instance, a plain method calls a Poplar method that assumes certain labels to be present on the incoming parameters. The caller may not necessarily provide these labels, and there is no way to check compliance. The lower bound guarantee on labels is threatened.

Data flow from Poplar to plain methods. This can happen if a Poplar method must call a plain method. From the point of view of the caller, variables being passed as arguments may be constrained in terms of the resources that may be mutated, but there is no way to guarantee this once a reference has passed to plain territory. The upper bound guarantee on resource mutations is threatened.

In Jardine, we do not handle these problems. We simply permit data flow between plain and Poplar methods with no restrictions. This causes a minimum of inconvenience when we incrementally annotate existing software components with Poplar specifications (as we will see in Chapter 6. We assume that plain methods mutate no resources, and that incoming variables have all the labels that they are supposed to have. This approach is clearly unsound and presumes a high level of trust in the programmers. It would be straightforward to implement a more restrictive approach, for instance by producing warnings or errors when data crosses the Poplar/plain boundary. One additional minor feature that has been implemented but not formalised is that Jardine permits static methods and fields to have Poplar signatures. The reason for not formalising this was that MJ lacks static members. Static methods and fields are treated just like instance methods and fields except that they do not specify any constraints on the receiver, since they have none. We believe that this is sound.

5.4 An Overview of of JKit

The JKit Java compiler is itself implemented in Java. It has a straightforward architecture based on a staged pipeline, where each stage performs a small part of the compila-

tion process. Java files pass through multiple representations on their way from source code to compiled class files. For our purposes, the most important representations are the *AST* representation, which is represented by a class hierarchy in the jkit.java.tree package, and the *JIL* intermediate format representation, which is represented by a parallel hierarchy in the jkit.jil.tree package. For more detailed information on JKit than we give here, the reader may refer to the JKit websites [92, 91].

The following is an overview of the various compilation stages, in the order that they occur in the pipeline. Figure 5.1 shows this graphically.

Java file reading Reads source files and prepares the AST representation. The parser is generated from an ANTLR [90] grammar.

Skeleton discovery Scans Java source file ASTs for references to other classes.

Type resolution Resolves type names by examining imported packages and the classpath.

Skeleton builder Inserts fully qualified type names and other information into the skeletons found during skeleton discovery.

Scope resolution Resolves variables by identifying the scope (method, class, etc) that they belong to.

Type propagation Propagates type information from declarations to other expressions and statements.

Constant propagation Replaces known constant expressions with the concrete value that they represent.

Type checking Performs Java type checking.

Anonymous class rewrite Rewrites anonymous inner classes into classes with proper generated names.

Inner class rewrite Rewrites inner classes and adds the implicit reference to the outer class.

Enum rewrite Rewrites enums to the underlying classes and constants that represent them.

JIL generation Generates the JIL intermediate representation that is used for late stage analysis and rewriting.

Dead code elimination Eliminates dead code.

Definite assignment Checks that each variable has definitely been initialised.

Bypass methods Inserts additional redirection methods into classes with generics to allow the JVM to find all methods using the old-style non-generic method signature.

Bytecode generation Generates Java bytecode.

Peephole optimisation Performs local optimisations.

Classfile generation Generates Java classfiles.

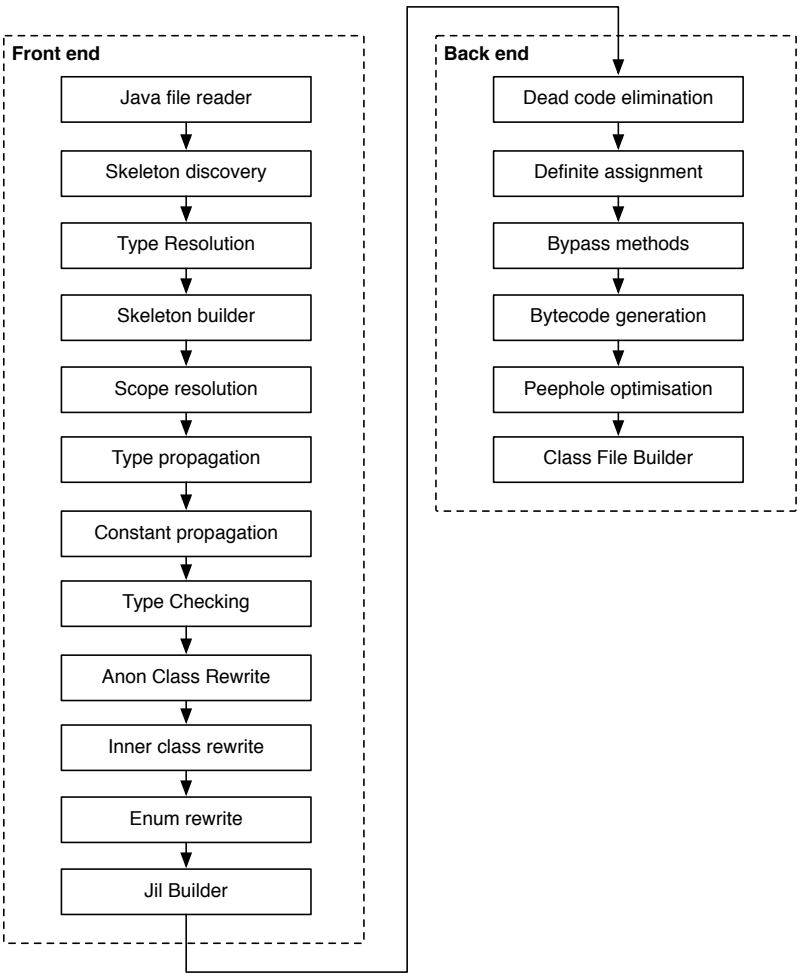


Figure 5.1: The JKit compiler pipeline

JKit package	Jardine package	Description
jkit	jp.ac.nii.jardine	Main package
jkit.bytecode	jp.ac.nii.jardine.bytecode	Bytecode representation and processing
jkit.compiler	jp.ac.nii.jardine.compiler	Main compiler framework
jkit.java.io	jp.ac.nii.jardine.io	AST construction
jkit.java.parser	jp.ac.nii.jardine.parser	Java/Poplar parsing
jkit.java.stages	jp.ac.nii.jardine.stages	Early compiler stages
jkit.java.tree	jp.ac.nii.jardine.tree	AST representation
jkit.jil	jp.ac.nii.jardine.jil	JIL representation
(none)	jp.ac.nii.jardine.util	Logging, collections and other utility classes
(none)	jp.ac.nii.jardine.poplar	Poplar-specific analysis and transformation
(none)	jp.ac.nii.jardine.planning	Partial order planning
(none)	jp.ac.nii.jardine.poplar.pop	Domain-specific planning for Poplar

Figure 5.2: Selected Java packages in JKit and Jardine, and their roles.

5.5 An Overview of Jardine

In extending JKit, we have treated it as a library and built Jardine as an application external to it to the greatest extent possible. In other words, we have tried to minimise the amount of changes made to JKit itself and instead opted to extend its classes and override methods selectively when changes had to be made. This strategy should allow us to upgrade Jardine to use future upstream versions of JKit with little effort. Packages in the jkit.* hierarchy have been given equivalents in the jp.ac.nii.jardine.* hierarchy. For instance, jkit.java.tree generally corresponds to jp.ac.nii.jardine.tree in our extended version. In the latter package, we have placed classes that override classes in the former. Figure 5.2 shows the relationships between packages in Jardine and JKit.

Jardine adds new stages to the JKit compiler pipeline and modifies some existing ones. The stages up to and including JIL generation are the compiler's frontend, and the subsequent stages are the backend. Our modifications took place in the frontend only. The JKit stages that have received nontrivial modifications are Java file reading, scope resolution, skeleton building, type resolution and type checking. The newly added stages in Jardine are label resolution, uniqueness resolution, Poplar checking and query solving. Figure 5.3 shows the Jardine compiler pipeline.

The nontrivial modifications to existing stages were as follows.

Java file reading We extended the existing ANTLR grammar to include the Poplar language extensions, such as resource and label declarations and Poplar signatures of methods. We generated a new parser and enhanced the file reading stage to check basic correctness of property and resource declarations, and construct additional representation objects where necessary.

Scope resolution The main task of this stage is to find the declaration site that corresponds to each variable use. We augmented it to take into account the fact that queries can reference and declare variables. For instance, produce-queries can take the form T x = #produce(x, T) which declares a variable of type T. This declaration is considered to be part of the query itself and not an assignment expression. Transform-queries can take the form #transform(x, l) which references the variable x. These variable declaration and use forms must be regarded

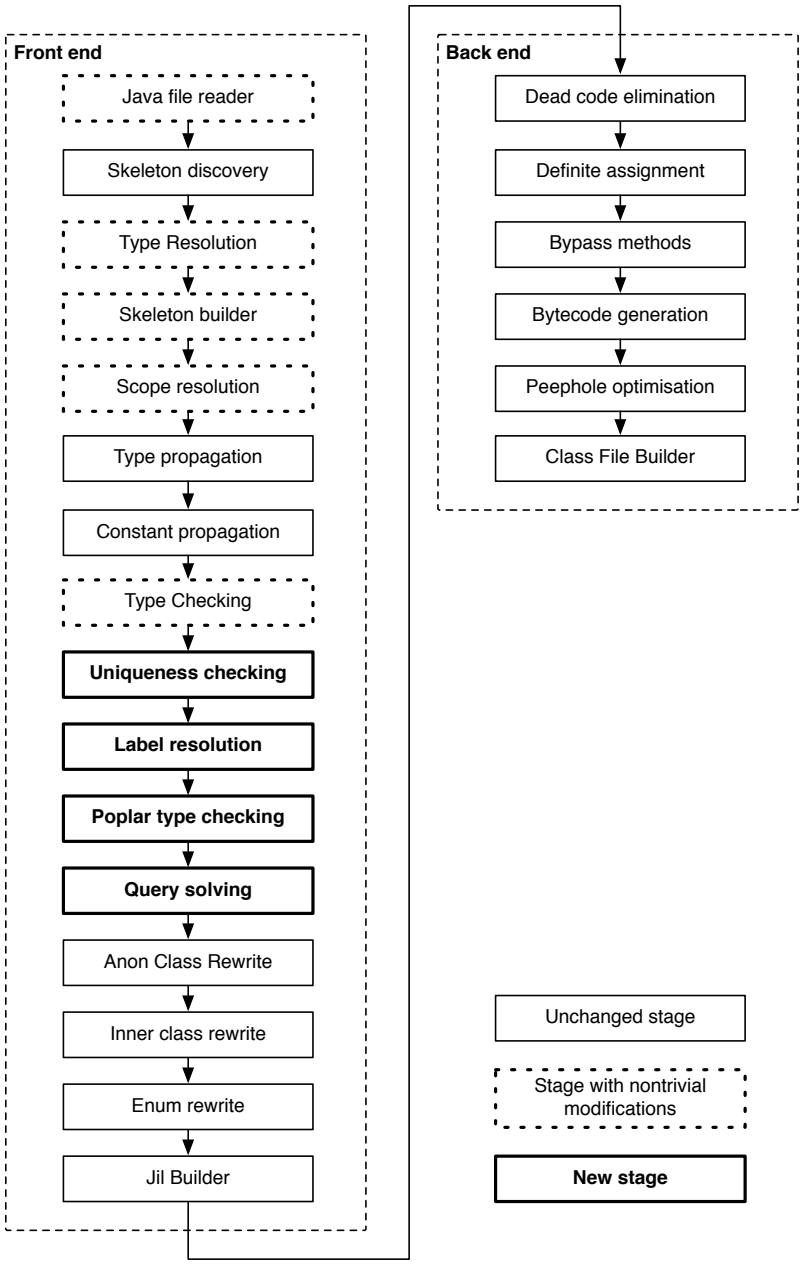


Figure 5.3: The Jardine compiler pipeline

Kind	Assumption	Guarantee
Normal	None	None
Unique	No aliases	No future aliases
Uniquer	No aliases	A single new alias is possible through destructive read
Maintain	None	No future aliases
Maintainr	None	A single new alias is possible through destructive read
Fresh	No aliases	None

Figure 5.4: Uniqueness kinds used in Jardine

as equal to preexisting definition and use forms.

Skeleton building This stage builds (by loading classes or carrying out additional compilation) compact representations of other classes encountered while compiling a given file. Such representations contain type signatures but no executable code. We extended the stage to also add Poplar declarations such as labels, resources and Poplar signatures to each class and class member in full.

Type resolution This stage attempts to resolve each unqualified type name mentioned in classes being compiled and transform them into fully qualified type names. Poplar introduces several new ways that a class can reference other types, including mentioning them in queries, as in the example above, and through declaring or mutating external resources. We extended this stage to also resolve such types.

Type checking We modified the type system itself to include uniqueness kinds as an attribute of every reference type, and enhanced the type checking stage to let queries pass without any further checking. The main checking of new Poplar typing rules was designed to take place not here, but in the uniqueness resolution stage and the Poplar checking stage.

5.6 Uniqueness Checking Stage

Poplar describes parameters, receivers and return values in terms of the assumptions made and invariants guaranteed about their uniqueness. Unique variables are definitely unique, and remain unique. Maintain variables may not be unique, but no new aliases are created. Normal variables, which are the default, have no associated assumptions. In addition, Maintainr and Uniquer (”maintain retains”, ”unique retains”) correspond to Maintain and Unique , except that they may create at most one new alias through a destructive read. These ”retains” kinds were not formalised in Chapter 4, although they are supported by Jardine. Figure summarises Jardine’s uniqueness kinds.

The uniqueness checking stage simply traces, for each variable, the data flow that it passes through and verifies that no incompatible flow takes place. This corresponds basically to the uniqueness restrictions imposed by the judgments in Chapter 4, with the difference that Maintainr and Uniquer also are checked. These two are treated exactly as Maintain and Unique with the exception that a destructive read is permitted for the transfer of a reference. In the example given in Figure 5.5, the first invocation in o, n(zz);, will be accepted. The second invocation, n(z); is invalid. In theory, the third invocation, n(zzz);, is valid, since a local variable goes out of scope, although this case has not yet been implemented. An explicit null assignment to a field or a variable is currently necessary for Jardine to accept the transfer.

```
1  class C {
2    C f:(unique);
3    void setF(C x) x: uniquer. { this.f = x; }
4    void n(C y) y: uniquer. { setF(y); }
5    void o(C z) z: unique. {
6      C zz = new C(); //unique
7      n(zz);
8      zz = null; //Destruction validates the previous method invocation
9      n(z); //Error: z is not (and cannot) be destroyed
10     C zzz = new C(); //unique
11     n(zzz); //Theoretically valid (goes out of scope) but not supported
12   }
13 }
```

Figure 5.5: Destructive read examples

5.7 Label Resolution Stage

The label resolution stage must, for each label that is referenced in a Poplar signature, composite property, or query, check that it has been declared exactly once in the imported classes, and annotate it with a mapping to the declaring class. This allows us to use properties from a given class in queries from another class, as we would expect. The resolution algorithm traverses all explicitly imported classes and packages when attempting to resolve labels.

5.8 Poplar Checking Stage

The Poplar checking stage contains the main procedure for checking that label assumptions are satisfied in user-written code. It also generates information that is used later by the query solving stage. Basically, this stage traverses each method and constructor body in each class and tries to find a valid Poplar type for it if it is a Poplar method, that is, if it has a Poplar signature. A Poplar type of a method or constructor body exists if and only if there is at least one way to satisfy all label constraints inside the method body and it satisfies its own contract. This means that at each point where some assumption about an expression's label set is being made, we must verify that the expression has at least the assumed labels.

The type checking performed by this stage corresponds closely to the theoretical framework described in Chapter 4. A complication arises from the fact that this system cannot be checked in a syntax-directed way, and for each method body, the checker must potentially explore a search space of partial typings until a full typing has been found.

5.8.1 Representation of Poplar types

Recall from Chapter 4 that a statement type has the form $\Delta; \Gamma \vdash s : \tau + \mathsf{LS!}\rho$, and that an expression type has the form $\Delta; \Gamma \vdash e : C : \bar{l} + \mathsf{LS!}\rho$.

The three most important classes that represent Poplar types are LabelSignature, StatementType and ExpressionType. ExpressionType is a subtype of StatementType since it needs to carry more information, such as the type and labels of the value computed. Statements do not need this, since in the Java evaluation model they are cast

```
1  class House {
2    resource access { properties @closed, @open, @secureClosed; }
3
4    Door d:(@closed -> @closed, @unlocked,
5      @open -> @open, @unlocked,
6      @secureClosed -> @closed, @locked);
7
8    Door getDoor() this: @open, result: +@unlocked. {
9      return d;
10   }
11 }
```

Figure 5.6: A field access with a disjunctive specification. For the field d to be @unlocked, the owning House has to be either @closed or @open.

to void type if they are not already of this type. These three types correspond directly to label signatures, statement types and expression types in our MJ-based formalisation. The label signature contains a mapping from *subjects* to *conditions*. A condition can be an addition, direct addition, subtraction or invariant coupled with a label, corresponding directly to what is written syntactically in Poplar method signatures. A subject can be bound or unbound: a bound subject is a local variable with a definite name, and an unbound subject is a receiver, a result or a parameter.

5.8.2 Principles behind the checking algorithm

As in the TS-SEQ rule and similar rules, checking of a method proceeds from bottom to top. In a statement sequence $s_1; s_2; \ldots s_{n-1}; s_n$, first s_{n-1} and s_n are typed, and then the chaining operation is applied to combine the types of these two statements. A complication arises from the fact that each statement or expression can in fact yield multiple valid types. To see why, consider what happens when we attempt to type the field access in the method getDoor in Figure 5.6.

From the declaration result: @unlocked we know the desired labels for the expression being returned by return d. To know the labels of d we inspect its declaration, and find that it has different labels depending on the labels of the owning object, which we need to have either the property @open, or the property @closed. At this point we construct *two* different types of the return d statement: one in which this is assumed to have the label @closed, and one where it is assumed to have the label @open. These assumptions are tracked as either invariants or subtractions in the label signature that is part of the statement's type, depending on whether the label will be lost at a later stage. Depending on the remaining statements to be typed, which would have been the ones preceding the return statement, some number of the constructed types remains viable. The ones that are found not to be viable, because they cannot be satisfied, are dropped. The ones that remain viable may give rise to further combinations. At the point where there are no statements remaining to be typed in the method body, which is immediately in this case, the remaining assumptions are compared with the preconditions provided by the method's signature itself. This corresponds to the expanded prior signature (see Section 4.2.10). In this case this is found to have the property @open, which satisfies one of the two types, so the check passes. In general, the check passes if at least one viable type can be found.

The general idea behind our checking algorithm is as follows. As we have just seen, statements or types can potentially have multiple valid types when inspected locally,

but when they must be made to fit into a context, some types do not fit the applicable constraints, and can thus be ruled out. Checking a method body is the process of finding a valid type for each statement and expression such that the types are compatible with each other and with the surrounding constraints (prior and posterior expanded signature). In order to do this, we traverse each method body in reverse (which corresponds naturally to the sequencing rules in Section 4.2.9) constructing first all possible types for each statement, and then all valid combinations of that statement with the following statements.

Note that in Jardine, Java type safety has already been checked by the time the Poplar type checking stage is run. For this reason there is no need to check Java types in this stage, although we sometimes make use of them when checking the Poplar types. Uniqueness has also already been verified at this stage (by the uniqueness checking stage), which means that this stage only needs to focus on tracing labels.

Algorithm 1 is the abstract, generalised form of the checking procedure for each term. The constant $unitType$ is the type of an effect-free empty statement with no resource mutations, no preconditions and no postconditions. First, the term is broken down into its subterms. Then the potential types of each subterm are computed in isolation. We track each possible assignment of types with respect to each subterm, represented here by the Cartesian product. A term with 3 subterms and n types for each subterm would have a total of n^3 "raw" combinations. In the next step, inconsistent combinations are removed (for instance, if label preconditions cannot be satisfied, i.e. if $(\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}_2, \rho_2)\ \mathsf{ok}$) is false, see Section 4.2.1) and the valid combinations are chained (sequentially composed) together into types for the entire term.

It is important to note that the order of composition of types of the subterms will always correspond to the evaluation order of the term when the program is running. For instance, in an if-statement, the truth condition is always evaluated before any of the branches. This ordering corresponds to what was formalised in Chapter 4.

Algorithm 1 Check term (generalised): $checkTerm(t, IDL, DL)$

$fullTypes \leftarrow \{unitType\}$
for each subterm $st \in t$ **do**
  $stIDL \leftarrow computeIDL(st, DL)$
  $stTypes \leftarrow checkTerm(st, stIDL, DL)$
  $fullTypes \leftarrow fullTypes \times stTypes$
end for
$sequentialTypes \leftarrow \emptyset$
for $ft \in fullTypes$ **do**
  if ft is a sequentially consistent typing **then**
    $sequentialTypes \leftarrow sequentialTypes \cup chain(ft)$
  end if
end for
return $sequentialTypes$

The $chain$ operation here corresponds exactly to the $\oplus$ operation specified in Section 4.2.1.

Throughout, we make use of the notions of *desired labels*, represented by the variable DL and, for expressions only, *immediate desired labels*, represented by the variable IDL. These variables are used to propagate constraints on the expected labels of a term from other terms that are related to it. For example, the judgment TE-VAR

(Section 4.2.7) types a variable when it is being read. In this judgment, the set of labels associated with the variable is unconstrained, so this information must come from somewhere else. This information would be held in IDL when the variable expression is being typed. Depending on the term, we may either reject the typing attempt if the IDL labels do not match what we can assign to the term being typed, accept it if they do match, or pass on the IDL labels for establishment elsewhere.

IDL-labels that are not produced by a term itself are usually moved to the precond(LS) set associated with its type. This means that the fragment as a whole will not be well typed unless these labels eventually originate in a valid source. Similarly, DL is a map $X \to L*$ that maps variables to the corresponding IDL set for that variable. This map is updated and propagated throughout the typing of a method body. It can be thought of as the precond(LS) set for the fragment that has been typed so far.

5.8.3 Selected checking routines

In principle, each typing judgment for statements and expressions presented in Sections 4.2.9 and 4.2.7 has its own variation of Algorithm 1. For the sake of brevity, rather than present each variation of the algorithm here, we will describe some general cases as well as the cases with unusual requirements. Instead of making calls to the abstract $checkTerm$ function presented in Algorithm 1 we will use the two routines: $checkStat(s, DL)$ and $checkExpr(e, IDL, DL)$ for statements and expressions, respectively. They are fundamentally the same as $checkTerm$. Each of these functions will dispatch the checking to the appropriate case algorithm depending on the concrete statement or expression being typed, for instance $checkIf$, $checkFieldWrite$, $checkVar$ and so on. Below we show some of the special case routines that $checkExpr$ and $checkStat$ can dispatch to. All routines return a set of valid typings on success, and an empty set on failure.

Sequences are checked by Algorithm 2. We use the notation $statements.head$ to refer to the first statement in a list, and $statements.tail$ to refer to the remaining statements after the first has been removed.

Algorithm 2 Check sequence: $checkSeq(statements, DL)$

if $size(statements) = 1$ **then**
  return $checkStat(statements.head, DL)$
end if
$headTypes \leftarrow checkStat(statements.head, DL)$
$tailTypes \leftarrow checkSeq(statements.tail, DL)$
for $ft \in (headTypes \times tailTypes)$ **do**
  $sequentialTypes \leftarrow \emptyset$
  if ft is a sequentially consistent typing **then**
    $sequentialTypes \leftarrow sequentialTypes \cup chain(ft)$
  end if
end for
return $sequentialTypes$

Sequences where the first statement is a field write are checked by Algorithm 3. This gives an example of the rewriting of ρ and LS that is carried out when an expression flows from one variable or field to another. First we check that the mutations that have already been found for the rhs expression will be acceptable for the field it

is flowing to, using the judgment $\Delta; \Gamma \vdash e!r$ ok (Section 4.2.6.) Then, for each "tail type" where the mutations could be accepted, we rewrite the corresponding LS and ρ so that the label pre- and postconditions and mutations follow the expression to the new name. This corresponds to the $lflow$ and $rflow$ functions in the formalisation (Section 4.2.9). Sequences where the first statement is a variable write are checked in almost exactly the same way as for the ones where the first statement is a field write, save for trivial changes.

Algorithm 3 Check sequence with field write: $checkSeqFW(statements, DL)$

if $size(statements) = 1$ **then**
  return $checkStat(statements.head, DL)$
end if
$fieldWriteExpr \leftarrow statements.head$
$tailTypes \leftarrow checkSeq(statements.tail, DL)$
$acceptableTailTypes \leftarrow \emptyset$
for $tt \in tailTypes$ **do**
  if $acceptableMutation(fieldWriteExpr.target, tt.mutations)$ **then**
    $acceptableTailTypes \leftarrow acceptableTailTypes \cup tt$
  end if
end for
$finalTypes \leftarrow \emptyset$
for $att \in acceptableTailTypes$ **do**
  LS $\leftarrow att.ls[fieldWriteExpr.rhs \mapsto fieldWriteExpr.target]$
  $\rho \leftarrow att.\rho[fieldWriteExpr.rhs \mapsto fieldWriteExpr.target]$
  $finalTypes \leftarrow finalTypes \cup \{(\text{LS}, \rho)\}$
end for
return $finalTypes$

If-statements are checked by Algorithm 4. The statement being checked has the form if(cond) { branch1 } else { branch2 }. In this algorithm we make use of the disjunctive composition operation since either branch can execute (see Section 4.2.1). Any disjunctive combination of a valid type of $branch1$ and a valid type of $branch2$ can represent the if-statement together with the branch condition. The IDL set for the branch condition is empty, since the if-statement is not permitted to have any particular constraints on it.

Access to fields in resources are checked by Algorithm 5. This case is the main reason why statements and expressions may have multiple types. Consider again the example shown in Figure 5.6. If we are typing access to the field d in a context where the main constraint is that d must have the property @unlocked, then there are two possible sets of labels that the owning object may have in order for the field to satisfy the constraints. This gives rise to multiple different ways that methods with such field accesses may be typed. In the algorithm shown here we use the function $fieldLabels(JT, field, labels)$ to give the labels of the field $field$ in the Java type JT when the owning object has the labels $labels$ - a single set. The function $fieldLabels^{-1}$ is not a true inverse function, as it returns a set of sets - all the possible combinations of labels that the owning object should have. For example, considering again Figure 5.6, $fieldLabels(House, d, \{@unlocked\})$ returns $\{\{@closed\}, \{@open\}\}$.

Method invocations are checked by Algorithm 6. Constructor invocations are very

Algorithm 4 Check if-statement: $checkIf(cond, branch1, branch2, DL)$

$condTypes \leftarrow checkExpr(cond, \emptyset, DL)$
$b1types \leftarrow checkSeq(branch1, DL)$
$b2types \leftarrow checkSeq(branch2, DL)$
$paraTypes \leftarrow \emptyset$
for $pair \in (b1types \times b2types)$ **do**
  $paraTypes \leftarrow paraTypes \cup disjunctiveCompose(pair)$
end for
$sequentialTypes \leftarrow \emptyset$
for $ft \in (condTypes \times paraTypes)$ **do**
  if ft is a sequentially consistent typing **then**
    $sequentialTypes \leftarrow sequentialTypes \cup chain(ft)$
  end if
end for
return $sequentialTypes$

Algorithm 5 Check resource field access: $checkField(owner, field, DL, IDL)$

$ownerJT \leftarrow javaType(owner)$
$ownerDLs \leftarrow fieldLabels^{-1}(ownerJT, field, IDL))$
$ownerTypes \leftarrow \emptyset$
for $ownerDL \in ownerDLs$ **do**
  $ownerTypes \leftarrow ownerTypes \cup checkExpr(owner, ownerDL, DL)$
end for
for $ownerType \in ownerTypes$ **do**
  $FL \leftarrow fieldLabels(ownerJT, field, ownerType.labels)$
  $accessTypes \leftarrow accessTypes \cup \{(javaType(field), \mathsf{Unique}, FL, ownerType.ls, \emptyset)\}$
end for
return $accessTypes$

similar. This also shows how the formalisation given in Chapter 4, which, for simplicity, only considers methods with a single argument, is extended to n arguments. The $invsub$ function from Section 4.2.6 is used to produce the version of the method signature where the receiver and the arguments have been replaced with the concrete expressions to be used.

Algorithm 6 Check method invocation: $checkInvoke(receiver, m, params, IDL, DL)$

$boundSignature \leftarrow invsub(m.poplarSignature, receiver, args)$
{Check receiver}
if not $acceptableMutation(receiver, boundSignature.\rho[receiver])$ **then**
  return $\emptyset$
end if
$recTypes \leftarrow checkExpr(receiver, DL[receiver], DL)$
$allCombinations \leftarrow recTypes$
{Check parameters}
for $p \in params$ **do**
  if not $acceptableMutation(p, boundSignature.\rho[p])$ **then**
    return $\emptyset$
  end if
  $argTypes_p \leftarrow checkExpr(p, DL[p] \cup boundSignature.preconditions[p], DL)$
  $allCombinations \leftarrow allCombinations \times argTypes_p$
end for
$finalTypes \leftarrow \emptyset$
for $typing \in allCombinations$ **do**
  if $typing$ is sequentially consistent **then**
    if $chain(typing).labels \subseteq IDL$ **then**
      $finalTypes \leftarrow finalTypes \cup chain(typing)$
    end if
  end if
end for
return $finalTypes$

Method bodies are checked by Algorithm 7. This is not a special case for $checkExpr$ or $checkStat$, but a top level case that is invoked on each method or constructor body in each class that is checked. Here we make use of prior and posterior expanded signatures (Section 4.2.10) as well as the mutation summary ρ of the surrounding method to make sure that the method body has a valid typing. A pseudo-statement that contains the method's preconditions as effects is prepended to the body before it is checked. This will produce the expected LS and no mutations when passed to $checkStat$.

5.8.4 Discussion

The checking routines that we have just shown exemplify all the different cases that can be encountered during the Poplar type checking of a method. The remaining routines are very similar to the ones shown here. For instance, sequencing with a variable write is very similar to sequencing with a field write, and constructor invocation checking is very similar to method invocation checking. Queries are typed as simple statements that assume nothing about their environment and provide the labels specified in the

Algorithm 7 Check method body: $checkMethod(m)$

$priorExp \leftarrow priorExpanded(m)$
$postExp \leftarrow posteriorExpanded(m)$
$startCond \leftarrow pseudo(priorExp.postconditions)$ {Pseudo statement}
$types \leftarrow checkSeq((startCond; m.body), postExp.preconditions)$
if $types \neq \emptyset$ and $\exists t \in types.(t.\rho \prec m.\rho)$ **then**
  return true
else
  return false
end if

query. (We will return to the interaction between planning and Poplar method checking in Section 5.9.3).

Resource fields and their owners, for which multiple typings may satisfy a given constraint, give rise to most of the complexity in the Poplar type checking. The number of types that may be found for a single method body is bounded by

$$O(n^f)$$

where n is the maximum number of disjoint solution sets available for $fieldLabels^{-1}(JT, field, labels)$, and f is the number of resource field reads in the method body. The time and space required to type a single method body is bounded by

$$O(sn^f)$$

where s is the number of statements (recursively descending into constructs such as if-statements) in the method body.

Unless field specifications are very complex, these should be acceptable constraints. The time taken to type a class is linear in the number of methods it has. Poplar method typing is decidable, since only a finite number of types need to be computed and tested.

5.9 Query Solving Stage

The query solving stage performs the integration procedure. For each method and constructor declaration, it traverses the body, looking for integration queries. When a query is found, partial order planning is used to look for a solution.

An important design decision was the point in the pipeline where the query solving stage should be inserted. If queries are solved early, then the solutions can be output in a simpler format where the representation is closer to the source code. On the other hand, less information about the surrounding context and the available data is available at an early stage. If queries are solved at a later stage, then a lot of information about the context is available, but at a late stage each class has been transformed and processed by several stages, and any code that we generate would have to conform to the invariants expected by the compiler at that point. We chose to execute our additional stages after JKit's type checking stage. At this point types have been resolved, checked and propagated to all expressions. When we solve queries we have access to poplar types, resolved labels and uniqueness kinds, since these stages are performed first. After we have generated solutions, we simply do a second, slightly weakened pass of type

propagation and scope resolution, so that the generated code will appear valid to the later compiler stages.

Methods that contain queries may be used to satisfy queries in other methods. However, we can clearly not allow any circular dependence here, since otherwise an infinite loop would result. We solve this by initially marking each query-containing method as unfinished, and marking it as finished once all queries in it have been solved. Only finished methods can be used to solve other queries. This constraint gives rise to a search tree: the order in which we finish methods is significant. If we fail to find solutions for all queries while trying to solve a class' methods in a certain order, we backtrack and attempt a different order.

Note that in theory, more solutions could be found if all orderings of all queries in all the classes being compiled were considered simultaneously. However, currently Jardine must solve the queries in each class in isolation. This means that there cannot be mutual interdependence between queries and methods in classes A and B. Queries in class A can use members from class B in their solutions, but then queries in class B cannot simultaneously use members from class A.

When a satisfactory solution has been found for each query, each solution is converted into a series of JIL statements, The query is removed from the method body, and the solution's JIL statements are inserted in the same location.

Algorithm 8 Solve all queries in a class

$source \leftarrow actionSource(class)$
for $method \in class$ **do**
  for $statement \in method$ **do**
    if $statement$ is $query$ **then**
      $solution \leftarrow solve(query, source)$
      $method \leftarrow method[query \rightarrow solution]$ {Replace the query with its solution.}
    end if
  end for
end for

5.9.1 Planning

The AI planning problem is the problem of identifying a sequence of actions in some domain that transform an initial state into a goal state. There is a large amount of literature on AI planning. Very broadly, AI planning algorithms may be divided into *plan space* search algorithms and *state space* search algorithms. The former explore a state of all possible plans; the latter explore a space of all possible states. We discuss related work in the area of planning in section 7.1.3.

In principle, any planning or search algorithm can be used to find solutions in Jardine. The choice of algorithm and heuristics is mainly a matter of performance. We have used Partial order planning (POP), a plan space search algorithm, as the main search algorithm and found it to produce good performance. It is also relatively easy for humans to understand and reason about.

We use a number of plans and classes to represent plans and actions. The following is a list of the most important classes.

Action Represents an action template, i.e. a single Java statement. FieldAction represents field accesses. MethodAction represents method or constructor invocation. Each action describes its preconditions and effects in general terms, which are made more specific (bound) when the action is inserted into a plan and connected to other actions. When we insert an action into a plan, we *instantiate* it, giving it a unique id number, to distinguish it from potential additional uses of the same action in the plan.

Label Represents a label, i.e. a property or a tag. The Label class also describes the resources that this label is sensitive to, if it is a property, and its declaring class.

Subject Represents a variable. Variables can either be referenced by name using the LocalVariable class, or by their role in a method signature (Receiver, Parameter or Result).

VarCondition A VarCondition is the basic unit of state in the plan representation. It is a triplet $(subject, type, label)$ which indicates that the given subject is a variable of the given type and has the given label. Note that a VarCondition can satisfy both a produce query (constraining type and label) and a transform query (constraining subject and label).

CausalLink The CausalLink class is used both to represent open preconditions, which represent unsatisfied conditions in plans, and actual causal links, which link effects of preceding actions to satisfied (closed) preconditions of successor actions.

Plan The main plan representation class. In addition to containing actions and causal links, it contains various auxiliary data, such as ordering constraints beyond causal links (which are implicitly ordering constraints) and variable binding info (to be described below).

The following auxiliary classes are also important.

BoundVariableInfo For a given action and subject pair, BoundVariableInfo tracks information about the subject's binding state. This includes whether the variable is bound, what its name is, and whether it was generated by Jardine or supplied in the query context. Plans contain maps that map $(action, subject)$ pairs to BoundVariableInfo instances. Because the binding information is contained in plans rather than in actions, actions can be shared between plans that contain the same action in different binding states, leading to simpler algorithms and improved memory efficiency.

ActionSource ActionSource classes supply Action templates to the planner. Currently there are two main action sources being used: the JILActionSource, which supplies actions from (previously compiled) JIL classes, and the TreeActionSource, which supplies actions from the AST class that is currently being compiled. In addition, plans can also act as action sources by themselves. When they do, they simply supply the actions that are already present in the plan.

The search algorithm presented here is not novel, but a straightforward implementation of the partial order planning algorithm [40]. Algorithm 9 is the main search algorithm. It makes use of the auxiliary Algorithm 10 and Algorithm 11.

It should be noted that $plans$ is always kept ordered according to the number of actions in the plans, with small plans first. This is our main search heuristic. It effectively ensures that plans able to satisfy a query with a smaller number of separate actions are preferred over larger plans.

Algorithm 9 Main plan search algorithm

$start \leftarrow makeAction(assumptions)$
$goal \leftarrow makeAction(goals)$
$plans \leftarrow makePlan(start, goal)$
while $plans \neq \emptyset$ and $plans.first.openPreconditions \neq \emptyset$ **do**
  $plan_c \leftarrow plans.first$
  $plans = plans \setminus plan_c$
  $plans \leftarrow plans \cup successors(plan_c)$
end while
if $plans.first.openPreconditions = \emptyset$ **then**
  return Success($plans.first$)
else
  return Failure
end if

Algorithm 10 Successor construction procedure successors

$cond \leftarrow pickCondition(heuristic, plan)$
$newActions \leftarrow actionsFor(aSource, cond)$
$oldActions \leftarrow actionsFor(plan, cond)$ {Pick suitable actions already in the plan}

$newPlans \leftarrow \emptyset$
for all $a \in newActions \cup oldActions$ **do**
  $plan_r \leftarrow$ Insert a into plan before $searchAction$
  $newPlans \leftarrow newPlans \cup resolveConflicts(plan_r)$
end for
return $newPlans$

Algorithm 11 Conflict resolution procedure resolveConflicts

$resolutions \leftarrow \emptyset$
for all $(action, cLink) \in conflicts(plan)$ **do**
  $r_1 \leftarrow$ new plan with $action$ before $cLink$ if possible
  $r_2 \leftarrow$ new plan with $action$ after $cLink$ if possible
  $resolutions \leftarrow resolutions \cup \{r_1, r_2\}$
end for
return $resolutions$

5.9.2 Decidability of planning

The plan search procedure carried out by Jardine is decidable. This follows from the fact that we have a progress measure: with each new candidate plan, we can tell whether or not it has made progress. Progress is tracked in terms of the types, labels and uniquenesses of the open preconditions of plans that have previously been seen in a plan search problem. A plan p_2 is an *improvement* of a previous plan p_1 if at least one of the following conditions is true:

- The preconditions of p_1 contain a variable that does not exist in the preconditions of p_2.
- For some variable, the preconditions of p_1 contain labels that are a superset of the corresponding labels in p_2.
- Some variable is associated with a more permissive uniqueness kind in p_2 than in p_1. For instance, Unique is more permissive than Normal since it can be used in more contexts. The ordering is Uniquer $\prec$ Unique $\prec$ Normal $\prec$ Maintainr $\prec$ Maintain .

We only retain candidate plans that have made progress compared with the previous best progress state. It is possible to track multiple "progress branches" simultaneously, remembering the best achieved state of each one. It is only when a new candidate is equal to or weaker than some other best progress state that it is discarded. In this way, for a given set of available actions (members in classes on the classpath) and a given query in a given context, all possibility of further progress is eventually exhausted and the plan search stops. Thus, our planning problem is decidable.

5.9.3 Ensuring the safety of solutions in a context

It is necessary to ensure that the solutions to queries do not interfere with Poplar types that have previously been found. Consider the following example.

```
1  class C {
2    resource r { properties @a, @b;
3      void m() this: ++@a. { ... }
4      void n() this: ++@b. { ... }
5    }
6    void q() this: +@a, +@b. {
7      m();
8      #transform(this, @b);
9    }
10 }
```

The method m establishes the property @a, but erases @b. The method n erases @a and establishes @b. Naively, a solver might attempt to satisfy the query in q by using the method n. However, this is not correct as such a method invocation would destroy the label @a, which is needed to satisfy the overall signature of the method. We address this problem by propagating information from the Poplar type checking stage to the query solving stage. When a valid Poplar typing for a method has been found, we track all label flows that flow across a query, that is, from the statements before it to the statements after it (considering the start and end of the method to be pseudo-statements). When we solve such a query, we first set up these label flows as predefined causal links, which may not be broken. This guarantees that the solutions will be correct with respect to such cross-query label flows. Thus, in the example given here, there is no valid solution, and this fact can correctly be identified.

5.10 A Future Extension: Verification of Integration Links

We mentioned in Section 5.2 that an ideal Poplar compiler would perform three tasks: checking of method contracts, construction of integration links, and verification of integration links. Jardine, the compiler described in this chapter, only performs the former two tasks, but it would be straightforward to extend it to check integration links as well.

When a Java program causes the JVM to load a class dynamically, the class that is actually loaded may be a future version, which could be different from the version that the program was compiled against. However, the JVM will check the class as it is being loaded to verify that it has the expected declarations and that they have the correct signatures, in addition to doing bytecode verification (CITE). This basic compatibility check can be thought of as a form of binary compatibility. If Poplar type checking is a stricter form of Java type checking, then verification of integration links would resemble a stricter form of binary compatibility. The purpose of such verification would be to identify when a method or field contract has evolved to a future incompatible version, for instance by weakening its postcondition.

Standard Java class files store information in attributes. Java compilers are permitted to define new such attributes [72]; unknown attributes will not interfere with existing compilers. This facility can be used in a natural way to aid Poplar. At each integration (query) site, the Poplar compiler should store, in new attributes in the relevant class files, the following information:

- The query itself.
- Minimal assumptions about the method contracts used as part of the solution to the query.

In service components, which provide the declarations that can form part of solutions to queries, the Poplar compiler should store, in new attributes in the relevant class files, the full contracts of all methods that are being made available to Poplar. With this information in place, it is easy to check whether the exported contracts in a future version of a class file match the expectations declared by client classes. Such a check would amount to checking the valid subtyping and overriding relation specified in Chapter 4.

Ideally, this check should be performed not by a static tool but by the JVM at runtime. It should be straightforward to modify a JVM to carry out this additional check as part of classloading; however, an even simpler way of achieving the same result might be to use a custom classloader [10, p. 438]. Such a classloader modifies the JVM's standard mechanism for loading classes and may perform additional checks before permitting a class to be loaded in the usual way.

5.11 Conclusion

In this chapter, we discussed the design and implementation of Jardine, a Poplar compiler. Jardine is based on JKit, an existing Java compiler framework. We established three functions that a Poplar compiler will ideally perform: query solving (integration), Poplar type checking and integration link verification. We described our implementation of the former two functions, and showed how the third may be implemented as a future extension.

Jardine retains the Java type checking system from JKit with slight modifications, and so it remains for Jardine to verify uniqueness of variables, verify label flow, and perform query solving. These problems are solved in separate stages that have been added to the pipeline.

The main source of complexity in Poplar checking is the possibility of having multiple valid types of fields and owners of fields with respect to the constraints on the fields themselves. Unless the methods in question are not very large, and unless the specifications of the fields in question are extremely complicated, we do not expect that this will pose a significant problem. Poplar checking is decidable, since only a finite number of potential types for each method body must be constructed and tested.

Query solving is the main procedure that performs our automatic integration. In Jardine it is based on partial order planning. Thanks to a measure of established progress, query solving is decidable. Information propagation from the Poplar checking stage to the query solving stage guarantees that the previously established Poplar types will not be invalidated by any query solutions.

In the following chapter, we carry out a case study by applying Jardine to an existing software system.

6 Evaluation and Discussion

In this section we apply Poplar to the problem of refactoring a large real-world software library. Our purpose is to verify that Poplar can be used in practice and yield useful results. In this study, we cannot claim to be exhaustive. For a Java software system, there is rarely such a thing as a final, perfect design, and refactoring is often a matter of taste. Nevertheless there is some agreement on good practice and bad practice. Fowler et al discuss the subject at length [34]. In addition to giving a list of commonly used refactorings, they identify what they term *bad code smells* - patterns that indicate that there might be a need for a particular refactoring. However, what is a bad smell in one design, going against the grain, might be a sign of success in another where it goes with the grain. We have selected some refactorings that are recommended by Fowler and that we believe could realistically occur, if the component under study were to be redesigned. We also discuss the theoretical application of Poplar to all of Fowler's refactorings in Section 6.2.

6.1 Case Study: Refactoring JFreeChart

As the subject of our study we have chosen the open source charting library JFreeChart [59]. This is a relatively large library of 583 top-level classes and more than 216 000 lines of source code (version 1.0.13). As of 28 September 2011, it had been downloaded more than 430 000 times. It is also an interesting library to study for the reason that in order to use its functionalities fully, it is sometimes necessary to create instances of multiple different classes and make them interact; the core API is not confined to a single class.

The experiments described in this section have been run on a Mac Pro with a 2.8 GHz Quad-Core Intel Xeon CPU using Mac OS X 10.6.8. The runtime environment consisted of Oracle Java 1.6.0_26 and Scala 2.9.0.1.

First, we will show a simple use case for JFreeChart, and then incrementally evolve the library through a series of refactorings. We will take advantage of Poplar to make the client remain compatible with the evolved library component. This demonstrates the descriptive capabilities of the Poplar language as well as Jardine's ability to generate solutions. After each refactoring we will generate a solution and generate a diagram that represents it, as well as show the generated code. The diagrams are automatically generated by Jardine in the GraphViz format but have been adjusted manually for layout and readability.

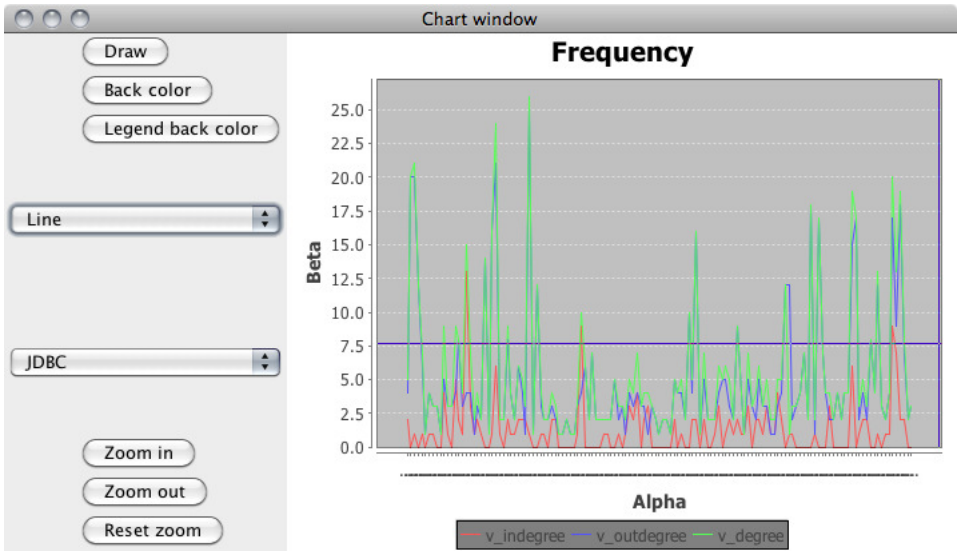


Figure 6.1: Chart window produced by ChartClient

6.1.1 A JFreeChart application

Throughout our study we will use a simple program that makes use of JFreeChart to display data in a chart. This chart is then displayed in a standard Swing JFrame. We developed the example application specifically for this study, but we believe that it represents a typical nontrivial usage of the JFreeChart library. Our application has the following features:

- Selectable data source - a JDBC database or CSV data.
- Support for five different chart types - Pie chart, Line chart, Bar chart, 3D Bar chart and XY bar chart.
- Zoom buttons.
- Configurable colours.

Thus, our application is interacting with both the data model, the chart creation/instantiation process, and the layout/theme parts of JFreeChart. A screenshot of the application is shown in Figure 6.1.

We will now describe our chart application in detail, since it is the starting point for a number of refactorings that we will carry out. In order to do this, we must first describe the relevant parts of the JFreeChart API. We give a simplified class diagram in Figure 6.2. With respect to this class hierarchy, we may note the following.

Multiple plot types. The JFreeChart class is the central chart class, but it delegates most of the actual drawing to an object of type Plot. The Plot class has multiple subclasses, which are different depending on which chart type has been requested. The subclasses we have included here are CategoryPlot, PiePlot and XYPlot, although more exist.

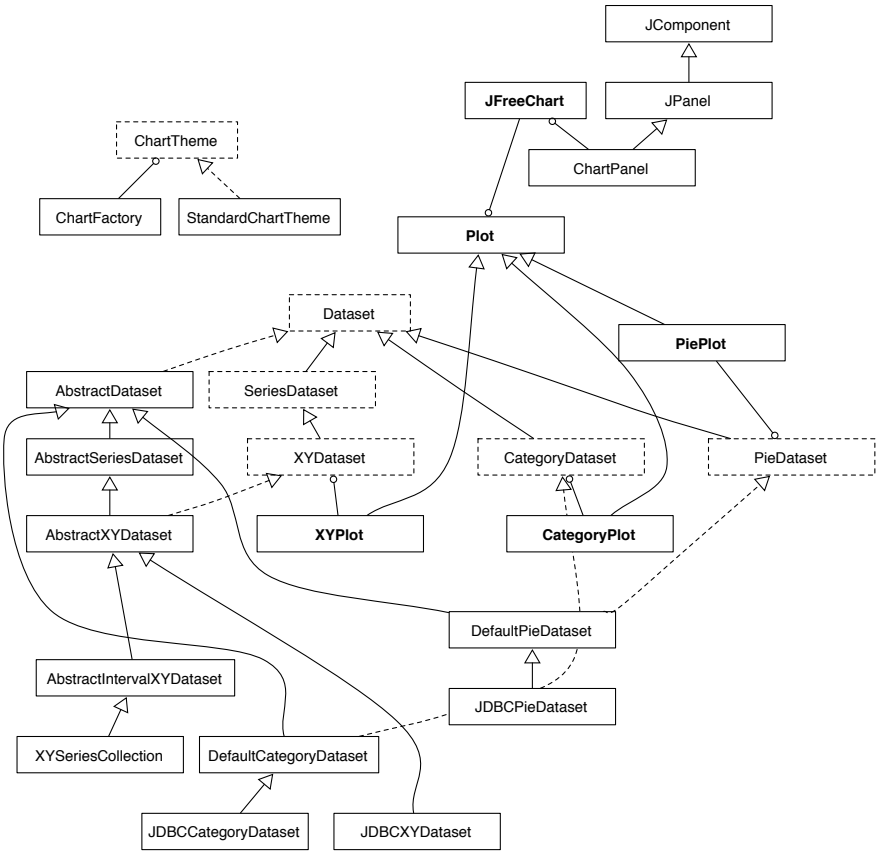


Figure 6.2: Simplified JFreeChart class diagram. Circled arrows denote delegation, normal arrows inheritance and implementation.

Multiple data set types. The Dataset implementations correspond generally to the various plot types, so that XYPlot needs an XYDataset, CategoryPlot needs a CategoryDataset and so on. A range of dataset classes are available for each plot type. We may also note the JDBC reading dataset classes – JDBCXY-Dataset, and so on – which have built-in functionality for database access.

Our application supports five chart types. For XYBarChart, an XYDataset is needed. For PieChart, a PieDataset is needed. The other three chart types are LineChart, BarChart and 3DBarChart. These three all need a CategoryDataset. Note that the different chart types do not each have their own class. All charts are implemented by the JFreeChart class, although the Plot and Dataset objects it contain will be of different types, depending on which kind of chart is created.

Most of the source code of our chart application is given on the following pages. We have omitted uninteresting, static code, such as GUI setup routines.

```
1  //Source code of our JFreeChart client application.
2  public class ConfigPanel implements ActionListener {
3    JFreeChart chart:(any);
4    Color bgPaint:(tBackgroundPaint) = Color.white;
5    Color legendBgPaint:(tLgBackgroundPaint) = Color.gray;
6    //Other GUI fields omitted.
7
8    public ConfigPanel() result: ++any. {
9      //GUI setup code omitted
10   }
11   public void actionPerformed(ActionEvent e) mutates any(JFreeChart).
         ext[Plot].zoomState, this.data:
12     this: -@notReady. {
13     if (e.getSource() == drawButton) {
14       redrawChart();
15     } else if (e.getSource() == bgColorButton) {
16       bgPaint = JColorChooser.showDialog(panel, "Background color",
             bgPaint);
17     } else if (e.getSource() == legendBgColorButton) {
18       legendBgPaint = JColorChooser.showDialog(panel, "Legend
             background color", legendBgPaint);
19     } else if (e.getSource() == zoomInButton) {
20       if (chart != null) { zoomIn(); }
21     } //etc. for zoomOut and zoomReset
22   }
23
24   void zoomIn() {
25     double zf:(zoomDelta) = 0.9;
26     JFreeChart jc = #produce(JFreeChart, @zoomed);
27   }
28   void zoomOut() {
29     double zf:(zoomDelta) = 1.1;
30     JFreeChart jc = #produce(JFreeChart, @zoomed);
31   }
32   void zoomReset() {
33     double zf:(zoomDelta) = 0;
34     JFreeChart jc = #produce(JFreeChart, @zoomed);
35   }
36
37   void redrawChart() mutates this.data: this: -@notReady. {
38     JFreeChart newChart = makeChart();
39
40     if (chartPanel != null) {
41       panel.remove(chartPanel);
42     }
43     chartPanel = makeChartPanel(newChart);
44     panel.add(chartPanel, BorderLayout.CENTER);
45     panel.validate();
46     chart = newChart;
47   }
48
49   JFreeChart makePieChart() this: @inited. {
50     JFreeChart ch = #produce(JFreeChart, tPieChart, themed);
51     return ch;
52   }
53   JFreeChart makeLineChart() this: @inited. {
54     JFreeChart ch = #produce(JFreeChart, tLineChart, themed);
55     return ch;
56   }
57   JFreeChart makeBarChart() this: @inited.  {
58     JFreeChart ch = #produce(JFreeChart, t2D, tBarChart, themed);
59     return ch;
```

```
60  }
61  JFreeChart makeXYBarChart() this: @inited. {
62    JFreeChart ch = #produce(JFreeChart, tXYBarChart, themed);
63    return ch;
64  }
65  JFreeChart makeBar3DChart() this:@inited. {
66    JFreeChart ch = #produce(JFreeChart, t3D, tBarChart, themed);
67    return ch;
68  }
69
70  resource data {
71    properties @notReady, @inited;
72    DefaultCategoryDataset myCategoryData:((@inited) -> (@populated));
73    DefaultPieDataset myPieData:((@inited) -> (@populated));
74    XYSeriesCollection myXYData:((@inited) -> (@populated));
75
76    String url:((any) -> (jdbcUrl)) = "jdbc:sqlite:data.db";
77    String driver:((any) -> (jdbcDriver)) = "org.sqlite.JDBC";
78    String user:((any) -> (jdbcUser)) = "";
79    String pass:((any) -> (jdbcPasswd)) = "";
80    String query:((any) -> (sqlQuery)) = "select p_id, v_indegree,
          v_outdegree, v_degree from versions where g_id=1;";
81  }
82
83  private JFreeChart makeChart() mutates this.data:
84    this: -@notReady, +@inited. {
85    makeData();
86    String ctype = (String) typeCombo.getSelectedItem();
87    if (ctype == "Line") {
88      this.chart = makeLineChart();
89    } else if (ctype == "Bar") {
90      this.chart = makeBarChart();
91    } else if (ctype == "XYBar") {
92      this.chart = makeXYBarChart();
93    } else if (ctype == "Bar3D") {
94      this.chart = makeBar3DChart();
95    } else if (ctype == "Pie") {
96      this.chart = makePieChart();
97    }
98    return chart;
99  }
100 void makeData() mutates this.data:
101   this: -@notReady, ++@inited. {
102   makeCategoryData();
103   makePieData();
104   makeXYData();
105 }
106 void makeCategoryData() {
107   String st = getSourceType();
108   if (st == "CSV") {
109     myCategoryData = categoryCSV();
110   } else {
111     myCategoryData = categoryJDBC();
112   }
113 }
114 //Two similar methods makePieData() and makeXYData() follow
115
116 DefaultCategoryDataset categoryCSV() this: any. {
117   FileReader fr:(tCSVreader) = new FileReader("data.csv");
118   DefaultCategoryDataset r = #produce(DefaultCategoryDataset,
          @populated);
119   return r;
```

```
120     }
121     DefaultPieDataset pieCSV() this: any. {
122       FileReader fr:(tCSVreader) = new FileReader("data.csv");
123       DefaultPieDataset r = #produce(DefaultPieDataset, @populated);
124       return r;
125     }
126     XYSeriesCollection XYCSV() this: any. {
127       FileReader fr:(tCSVreader) = new FileReader("data.csv");
128       XYSeriesCollection r = #produce(XYSeriesCollection, @populated);
129       return r;
130     }
131     DefaultCategoryDataset categoryJDBC() this: any. {
132       DefaultCategoryDataset r = #produce(DefaultCategoryDataset,
              @populated);
133       return r;
134     }
135     DefaultPieDataset pieJDBC() this: any. {
136       DefaultPieDataset r = #produce(DefaultPieDataset, @populated);
137       return r;
138     }
139     XYSeriesCollection XYJDBC() this: any. {
140       XYSeriesCollection r = #produce(XYSeriesCollection, @populated);
141       return r;
142     }
143   }
144
145   //For each chart type, we have a "Maker" class, similar to this one.
146   public class LineChartMaker {
147     public LineChartMaker() result: ++any. {}
148     public JFreeChart useFactoryIndirect(DefaultCategoryDataset data)
149     data:@populated; result: +tLineChart, +t2D. {
150       String title:(tChartTitle) = "Frequency";
151       PlotOrientation po:(tPlotOrientation) = PlotOrientation.VERTICAL;
152       String f:(tXAxisLabel, tCategoryAxisLabel) = "Alpha";
153       String a:(tYAxisLabel, tValueAxisLabel) = "Beta";
154       boolean da:(tWithDateAxis) = false;
155       boolean gu:(tGenUrls) = false;
156       boolean tt:(tGenTooltips) = true;
157       boolean lr:(tReqLegend) = true;
158       JFreeChart c = #produce(JFreeChart, tLineChart, t2D);
159       return c;
160     }
161   }
```

It can be seen that the class ConfigPanel has 14 different queries. The 3 first queries are in zoomIn, zoomOut and zoomReset. The following 5 are in methods that make charts - makePieChart and so on. Finally, there are 6 queries that make datasets in categoryCSV and the following methods. We refer to these three groups of queries as the *zooming queries*, the *chart creators* and the *dataset creators*. Before we describe our refactorings, we will describe the initial solutions to these various queries. As we perform the refactorings, the solutions will change, but no manual intervention will be needed in the client application.

6.1.2 Initial service API annotations

We now give the initial Poplar annotations made to the JFreeChart API. Over time these will change to reflect refactorings, but the client side annotations will remain constant.

```
1  public abstract class ChartFactory {
2  //For each chart type, the corresponding factory method has been
       annotated with labels in a similar fashion to this one.
3      public static JFreeChart createPieChart(String title,
4                                              DefaultPieDataset dataset,
5                                              boolean legend,
6                                              boolean tooltips,
7                                              boolean urls)
8    title: tChartTitle;
9    dataset: @populated;
10   legend: tReqLegend;
11   tooltips:tGenTooltips;
12   urls: tGenUrls;
13   result: ++tPieChart, ++t2D. { ... }
14   //...
15 }
16
17 public class StandardChartTheme implements ChartTheme, Cloneable,
18         PublicCloneable, Serializable {
19
20   composite @configured=(@rbpSet, @lbpSet);
21
22   resource rBackgroundPaint {
23     properties @rbpSet;
24     private transient Paint chartBackgroundPaint;
25
26     public void setChartBackgroundPaint(Paint paint) paint:
           tBackgroundPaint; this: ++@rbpSet. {
27       if (paint == null) {
28         throw new IllegalArgumentException("Null 'paint' argument.");
29       }
30       this.chartBackgroundPaint = paint;
31     }
32    }
33   //Similar resource and property for setLegendBackgroundPaint, @lbpSet
34   //...
35 }
36
37 public class CSV {
38   public DefaultCategoryDataset readCategoryDataset(Reader in) throws
         IOException
39    in: tCSVreader; result: +@populated. { ... }
40   //...
41 }
42
43 public class DefaultCategoryDataset extends AbstractDataset {
44   resource data {
45     properties @populated, @empty;
46     /** A storage structure for the data. */
47     private DefaultKeyedValues2D data;
48   }
49   //...
50 }
51
52 public class JDBCCategoryDataset extends DefaultCategoryDataset {
53   public JDBCCategoryDataset(String url, String driverName,
54     String user, String passwd, String query)
55   throws ClassNotFoundException, SQLException
56   mutates this.data:
57   url: jdbcUrl; driverName: jdbcDriver; user: jdbcUser; passwd:
         jdbcPasswd;
58       query: sqlQuery; result: +@populated. { ... }
```

```
59
60    public void executeQuery(Connection con, String query) throws
          SQLException
61    mutates this.data: this: ++@populated. { ... }
62    //...
63  }
64  //Similar annotations for JDBCPieDataset, JDBCXYDataset
65
66  public class JFreeChart {
67    public Plot getPlot() result: ++any. {
68      return this.plot;
69    }
70    //...
71  }
72
73  public class Plot {
74    tags(double) zoomDelta;
75
76    //Using an external resource to provide a property for JFreeChart in
          Plot
77    resource[JFreeChart] zoomState {
78      properties @zoomed;
79
80      public void zoom(double percent, JFreeChart chart) percent:
            zoomDelta; chart: ++@zoomed. {
81        zoom(percent);
82      }
83      public void zoom(double percent) { ... }
84    }
85  //...
86  }
```

6.1.3 Initial solutions

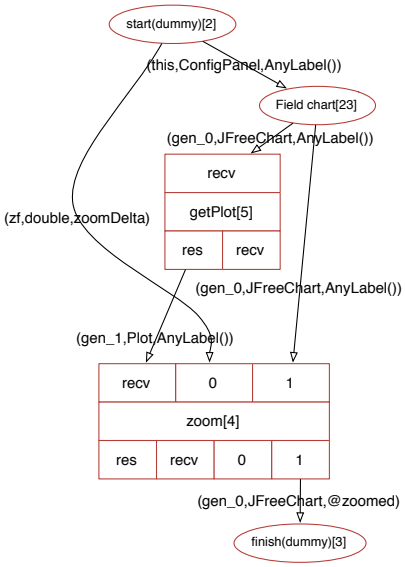
We have now given the initial annotations both in the client application and in the JFreeChart library. We can now show the initial solutions to the client's queries.

How to read the diagrams

Jardine generates compiled Java code and GraphViz diagrams that describe the generated solutions. We give both the diagrams and the corresponding generated code for each solution that we describe. In the diagrams, blue boxes are static methods, red boxes are nonstatic methods, and ellipses are dummy actions or fields. Recv identifies the receiver of a method, and res identifies the return value. Numbers $0, \ldots n-1$ identify each argument of a method. Black arrows connect input and output variables. Each such arrow is labelled with a triplet (name, type, label) that describes the condition that matched. Black arrows also imply an ordering of actions. Red arrows are orderings of actions without any corresponding condition.

Zooming

We give the solution to zoomIn in Figure 6.3. The other two queries have similar solutions.




```
1  JFreeChart gen_0 = this.chart;
2  Plot gen_1 = gen_0.getChart();
3  gen_1.zoom(zf);
```

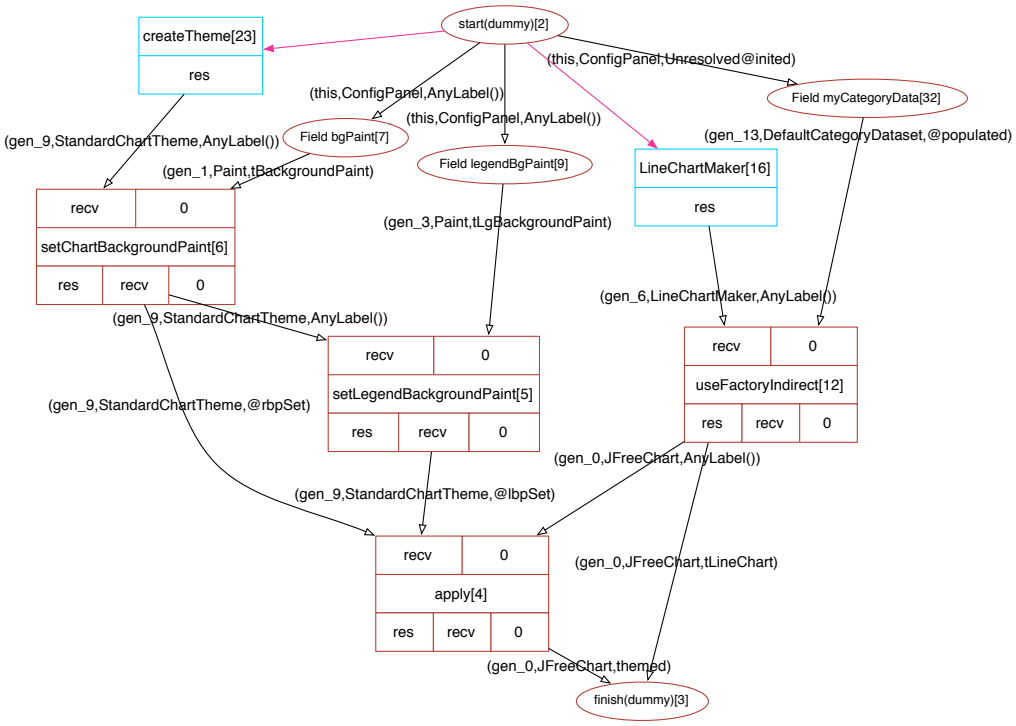
Figure 6.3: Solution to the zoomIn query.

Chart creators

We give the solution to the makeLineChart query in Figure 6.4. It makes use of the class LineChartMaker, the solution of whose query is shown in Figure 6.5. The other chart types have similar solutions.

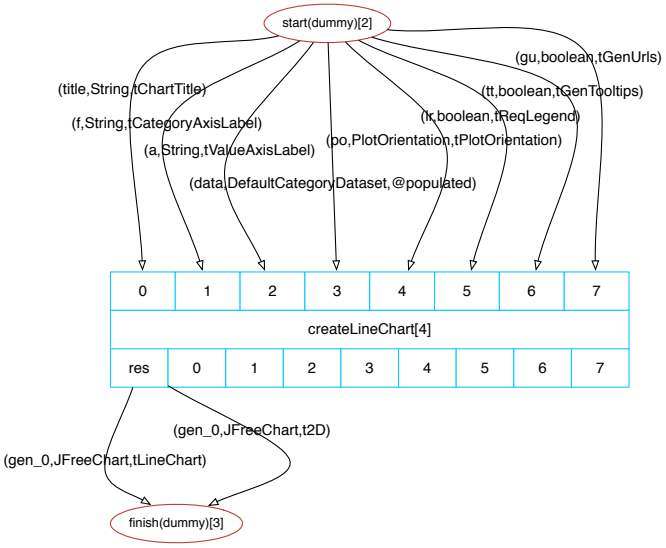
Dataset creators

We give the solutions of categoryCSV and categoryJDBC in Figure 6.6 and Figure 6.7, respectively. The other dataset types have similar solutions.



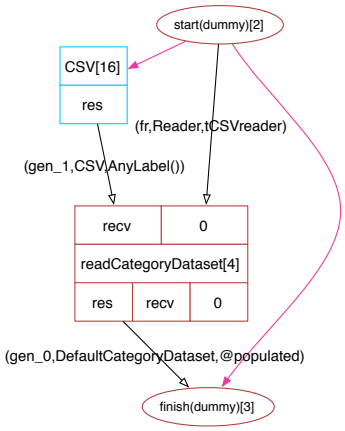

```
1  StandardChartTheme gen_9 = ChartFactory.createTheme();
2  Paint gen_1 = this.bgPaint;
3  Paint gen_3 = this.legendBgPaint;
4  LineChartMaker gen_6 = new LineChartMaker();
5  DefaultCategoryDataset gen_13 = this.myCategoryData;
6  gen_9.setBackgroundPaint(gen_1);
7  gen_9.setLegendBackgroundPaint(gen_3);
8  JFreeChart gen_0 = gen_6.useFactoryIndirect(gen_13);
9  gen_9.apply(gen_0);
```

Figure 6.4: Solution to the makeLineChart query.



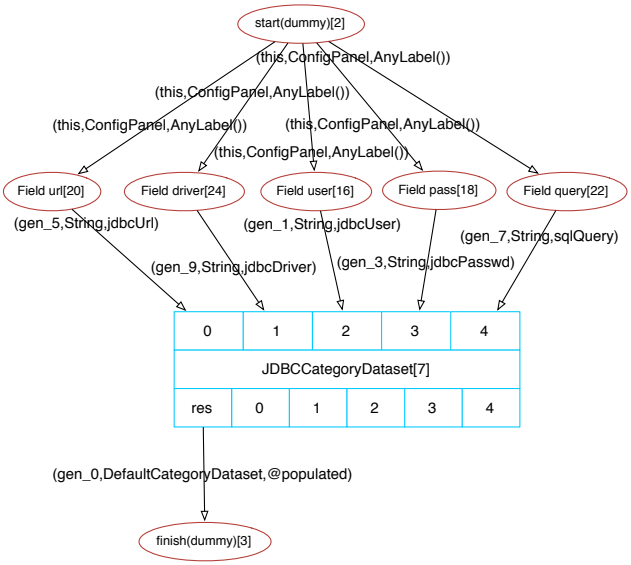

```
1  JFreeChart gen_0 = ChartFactory.createLineChart(title, f, a, data, po,
       lr, tt, gu);
```

Figure 6.5: Solution to the LineChartMaker query.




```
1  CSV gen_1 = new CSV();
2  gen_1.readCategoryDataset(fr);
```

Figure 6.6: Solution to the categoryCSV query.




```
1  String gen_5 = this.url;
2  String gen_9 = this.driver;
3  String gen_1 = this.user;
4  String gen_3 = this.pass;
5  String gen_7 = this.query;
6  JDBCCategoryDataset gen_0 = new JDBCCategoryDataset(gen_5, gen_9, gen_1
       , gen_3, gen_7);
```

Figure 6.7: Solution to the categoryJDBC query.

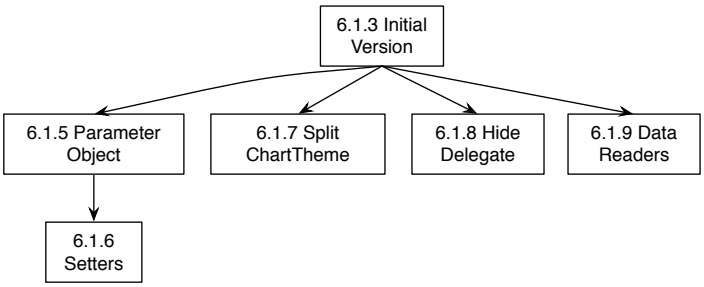


Figure 6.8: Refactorings to be carried out. The numbers reference sections in this chapter.

6.1.4 Refactorings to be carried out

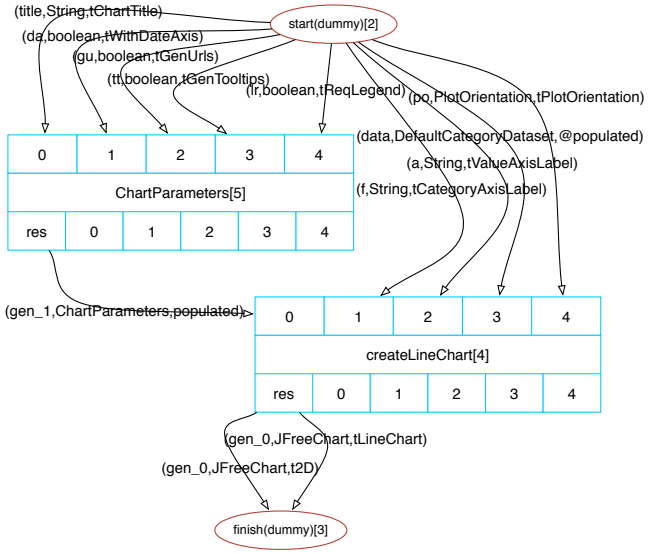
From this point onwards, we will gradually refactor the JFreeChart library, which will automatically push changes into the client application we have just shown. The roadmap for our refactorings is given in Figure 6.8. We start from the initial version of the chart application, which we have just described. In Section 6.1.5, we introduce a parameter object that replaces many of the parameters taken by the ChartFactory methods. In Section 6.1.6 and Section Z we modify the ChartFactory with setter methods, and then show how to use a local overriding method to influence the outcome of the solution search. We then go back to the initial application and split the StandardChartTheme class into several subclasses, which we describe in Section 6.1.7. This affects how visual styles are applied to the charts. In Section 6.1.8 we hide the chart's Plot delegate by adding some forwarding methods to the JFreeChart itself. This affects zooming. Finally, in Section 6.1.9, we refactor the Dataset hierarchy, introducing the concept of data readers. This refactoring separates the JDBC reading functionality from the dataset hierarchy completely, introducing instead a new delegate hierarchy.

6.1.5 Introducing a parameter object

In Figure 6.5, we may note that the factory method createLineChart takes a large number of arguments. In fact, most of the factory methods in ChartFactory take at least 8 arguments, and many of them have arguments in common. This is one of the situations in which Fowler suggests [34, p. 295] that the refactoring known as *Introduce parameter object* may be used. It may be desirable to introduce a new class that wraps some of the common parameters of these methods, and then pass an instance of that class instead of passing each parameter separately to each method. With Poplar, and using the kind of integration query just shown, we may carry out such a change without changing API clients at all. First, we introduce a class called ChartParameters, shown in figure **??**. We also modify the ChartFactory API accordingly. Here, the tag populated indicates that the members of the ChartParameters object have been filled out. Since the tag is immutable, we cannot track here whether the parameter object's fields are overwritten after they have been initialised. We could do so if we put these fields in a resource and instead used a property, a technique that we will demonstrate later.

```
1  public class ChartParameters {
2
3    tag(ChartParameters) populated;
4
5    String chartTitle;
6    boolean withDateAxis, urls, tooltips, legend;
7
8    public ChartParameters(String chartTitle, boolean withDateAxis,
9        boolean urls, boolean tooltips, boolean legend)
10     result: ++populated; chartTitle: tChartTitle;
11     withDateAxis: tWithDateAxis; urls: tGenUrls;
12     tooltips: tGenTooltips; legend: tReqLegend. {
13     this.chartTitle = chartTitle; this.withDateAxis = withDateAxis;
14     this.urls = urls; this.tooltips = tooltips;
15     this.legend = legend;
16
17   }
18 }
19
20 public class ChartFactory {
21   //...
22     public static JFreeChart createLineChart(ChartParameters cp,
23   String categoryAxisLabel, String valueAxisLabel,
24   DefaultCategoryDataset dataset, PlotOrientation orientation)
25   cp: populated; categoryAxisLabel: tCategoryAxisLabel;
26   valueAxisLabel: tValueAxisLabel; dataset: @populated;
27   orientation: tPlotOrientation;
28   result: ++tLineChart, ++t2D. {...}
29
30 //...
31 }
```

We add ChartParameters to the source path and invoke Jardine again, using the unchanged client code from Figure **??**. The resulting solution is shown in Figure **??**. We are at liberty to divide parameters between the createXYBarChart and the ChartParameters object any way we like. The only condition that must be satisfied for an integration to be successful is that there should exist a conflict-free path, obtained by assembling methods and fields in sequence, from the starting conditions to the goal conditions.




```
1  ChartParameters gen_1 = new ChartParameters(title, da, gu, tt, lr);
2  JFreeChart gen_0 = ChartFactory.createLineChart(gen_1, f, a, data, po);
```

Figure 6.9: Solution to the LineChartMaker query with a parameter object. Compare with Figure 6.5.

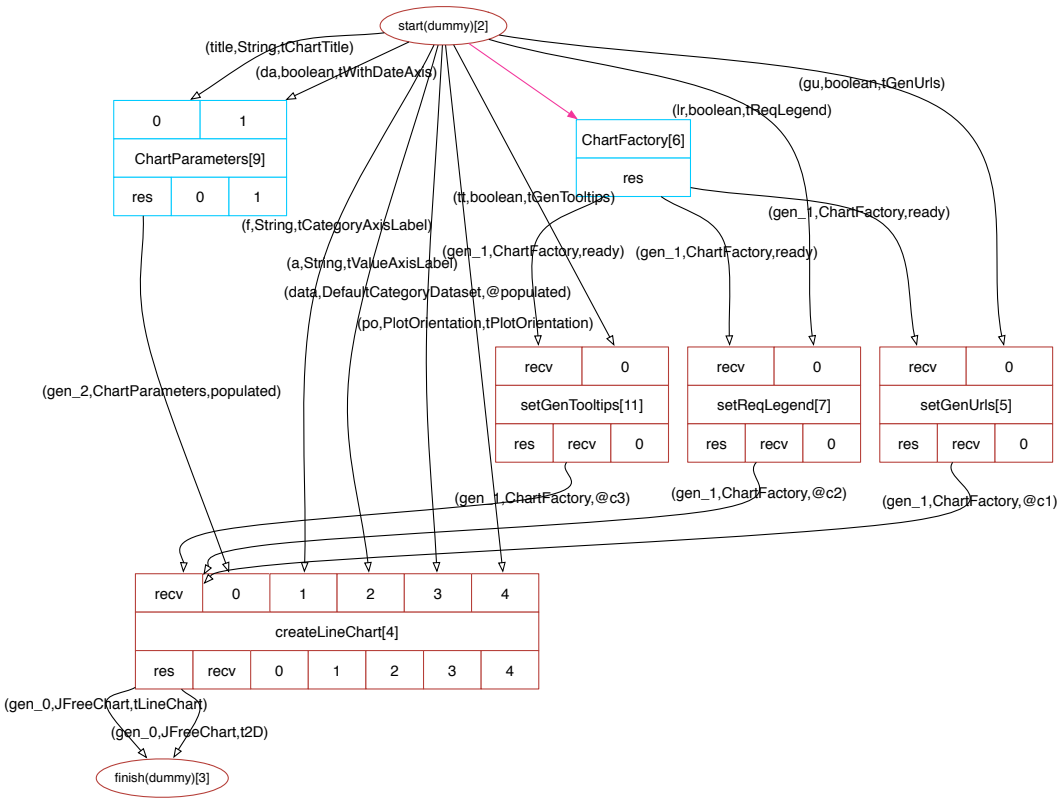
6.1.6 Converting parameters to state

For our next example, we decide that rather than passing in the reqLegend, reqTooltips and genUrls parameters each time we construct a chart, we would like the factory to remember these parameters as part of its mutable state. Each time a new chart is created, the ChartFactory should consult the current default values for these parameters and use them.

To perform this change, we first add these fields to the ChartFactory. We put each one in a corresponding resource. We could have put these properties in a common resource, but then we would have had to insist that they should be initialised in a specific order. We have also removed the newly added fields from the ChartParameters class, simplifying it. Note that we have added a composite property @factoryConfigured to refer to the state of all of @c1, @c2, @c3 being established. This lets us conveniently request all three properties with a single keyword in the createXYBarChart method specification. We also make the methods of ChartFactory nonstatic, since we now require the factory to carry a significant amount of state. The changes are as follows.

```
1  public class ChartFactory {
2    public ChartFactory() result: ++any. { }
3
4    composite @factoryConfigured = (@c1, @c2, @c3);
5    resource urlConfig {
6      properties @c1;
7      private boolean currentUrls;
8      public  void setGenUrls(boolean urls)
9        urls: tGenUrls;
10       this: maintain, any, ++@c1. {
11       currentUrls = urls;
12     }
13   }
14   resource legendConfig {
15     properties @c2;
16     private boolean currentLegend;
17     public  void setReqLegend(boolean legend)
18       legend: tReqLegend;
19       this: maintain, any, ++@c2. {
20       currentLegend = legend;
21     }
22   }
23   //Similar resource for tooltipsConfig with @c3
24
25   public JFreeChart createLineChart(ChartParameters cp,
26     String categoryAxisLabel, String valueAxisLabel,
27     DefaultCategoryDataset dataset, PlotOrientation orientation)
28   cp: populated; categoryAxisLabel: tCategoryAxisLabel;
29   valueAxisLabel: tValueAxisLabel; dataset: @populated;
30   this: @factoryConfigured; orientation: tPlotOrientation;
31   result: ++tLineChart, ++t2D. { ... }
32   //Similar annotations for other createXChart methods
33 }
```

The solutions that were found in the various ChartMaker classes are similar to the one shown in Figure 6.10.




```
1  ChartParameters gen_2 = new ChartParameters(title, da);
2  ChartFactory gen_1 = new ChartFactory();
3  gen_1.setGenTooltips(tt);
4  gen_1.setReqLegend(lr);
5  gen_1.setGenUrls(gu);
6  JFreeChart gen_0 = gen_1.createLineChart(gen_2, f, a, data, po);
```

Figure 6.10: Solution to the LineChartMaker query with a parameter object and pre-configured state. Compare with Figure 6.9.

6.1.7 Splitting ChartTheme

This refactoring is based on one of the ”big refactorings” in Fowler's book, *Extract Hierarchy* [34, p. 375]. This refactoring replaces a single class with a hierarchy of classes in a case where the single class is doing a considerable amount of work involving conditional statements. The conditionals are also natural opportunity to introduce polymorphism, as in the *Replace Conditional With Polymorphism* refactoring [34, p. 255]. In JFreeChart we have identified a natural opportunity to apply this refactoring in the class StandardChartTheme. This class is responsible for applying visual styles to charts. It contains persistent state that describes the visual style, as well as knowledge about how to apply this style to different concrete plot types.

Before the refactoring, when StandardChartTheme is applied to a chart, it tests what concrete plot type the chart contains:

```
1  public void apply(JFreeChart chart) this: @configured; chart: ++themed.
       {
2    //...
3    Plot plot = chart.getPlot();
4    if (plot != null) {
5      applyToPlot(plot);
6    }
7    //...
8  }
9
10 protected void applyToPlot(Plot plot) {
11   if (plot == null) {
12     throw new IllegalArgumentException("Null 'plot' argument.");
13   }
14   if (plot instanceof PiePlot) {
15     applyToPiePlot((PiePlot) plot);
16   }
17   else if (plot instanceof CategoryPlot) {
18     applyToCategoryPlot((CategoryPlot) plot);
19   } else if //...
20 }
```

Specialised methods such as applyToCategoryPlot contain the logic that applies to a particular plot type. Our refactoring splits the StandardChartTheme into several subclasses, one for each plot type. We then introduce a new factory method that creates the appropriate subclass that applies to a given chart. This is shown as a class diagram in Figure 6.11. We introduce the subclasses PieChartTheme, XYChartTheme and CategoryChartTheme. These are all we need to cover the charts that are created in our example application, but it would have been straightforward to introduce even more subclasses to cover all the chart types that JFreeChart supports. We move as much type-specific code as possible down from the superclass into the new subclasses. This involves the refactorings *Push Down Method* [34, p. 328] and *Extract Class* [34, p. 149].

The extracted subclasses and the factory method are similar to the following.

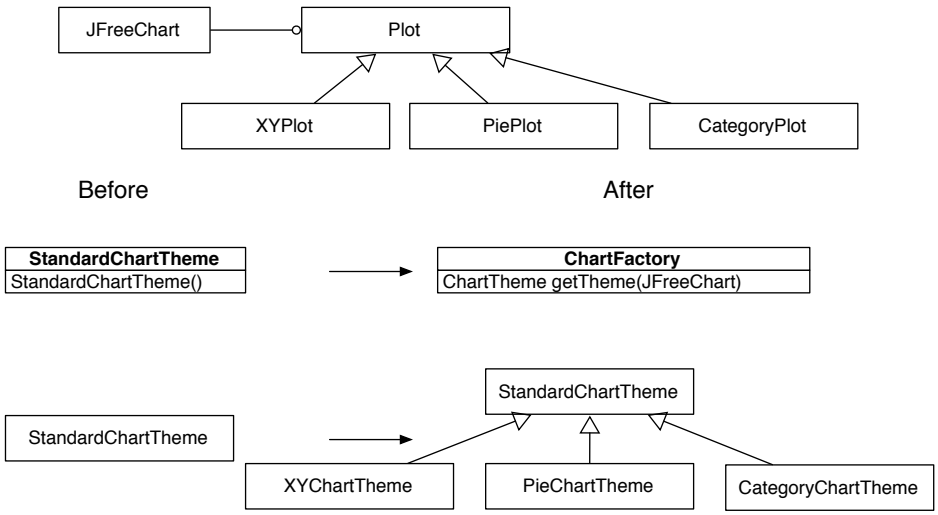
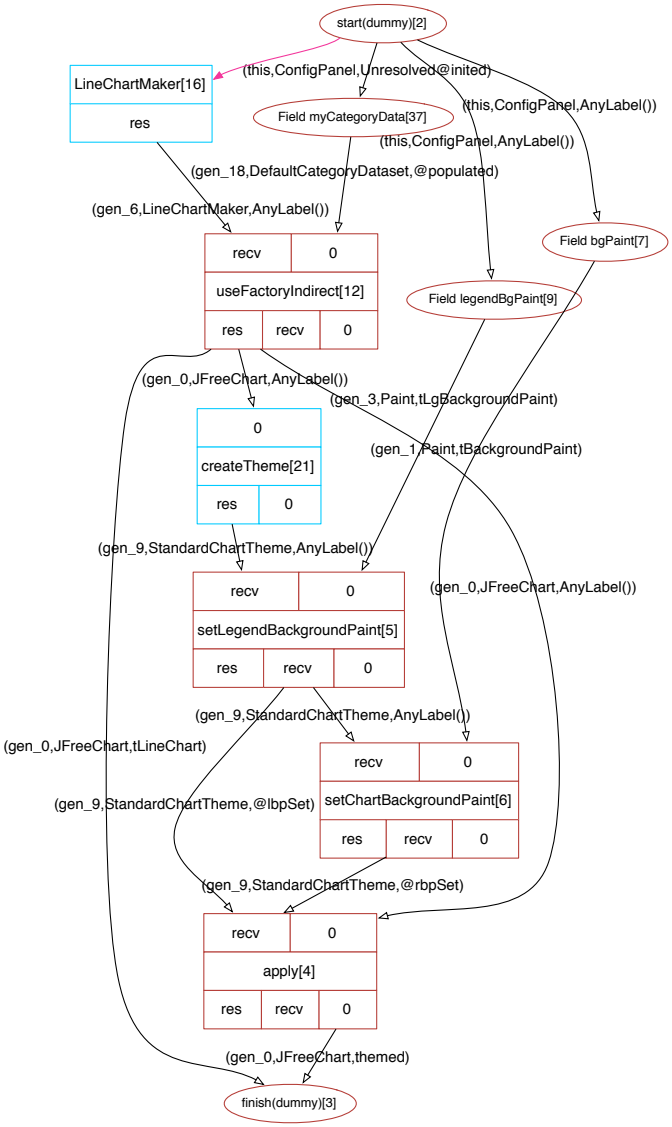


Figure 6.11: Class diagram for the Split ChartTheme refactoring

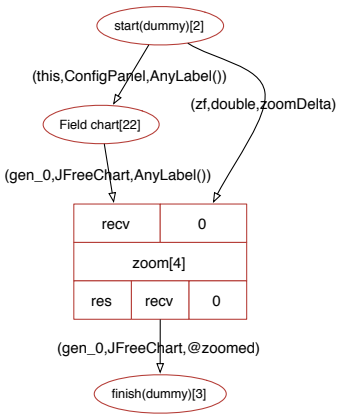
```
1  class StandardChartTheme {
2    //...
3    //Empty method to be overridden by subclasses
4    void applyToPlot(Plot plot) this:@configured; plot:any. { }
5    //...
6  }
7  class CategoryChartTheme extends StandardChartTheme {
8    void applyToPlot(Plot plot_) this:@configured; plot_:any. {
9          CategoryPlot plot = (CategoryPlot) plot_;
10           plot.setAxisOffset(this.axisOffset);
11     //Code to apply a theme to a CategoryPlot
12   }
13   //Other methods that are needed for CategoryPlot
14 }
15 class ChartFactory {
16     public static StandardChartTheme getTheme(JFreeChart chart) chart:
           any; result: ++any.{
17       if (chart.getPlot() instanceof CategoryPlot) {
18         return new CategoryChartTheme();
19       } else if (chart.getPlot() instanceof PiePlot) {
20         return new PieChartTheme();
21       } else //...
22     }
23 }
```

Note that when we add a factory method, it provides an additional way to obtain a value of type StandardChartTheme, but one which takes more arguments than the old constructor. This means that using the factory method will be a larger solution than using the constructor, and therefore the solver will not give priority to the factory method. We solve this by removing the Poplar annotations from the StandardChartTheme constructor, and from those of its subclasses, which means that the factory method will be the only solution available for this subproblem. After we have performed these changes, the solutions in the ChartMaker classes are as shown in Figure 6.12.




```
1  LineChartMaker gen_6 = new LineChartMaker();
2  DefaultCategoryDataset gen_18 = this.myCategoryData;
3  Paint gen_3 = this.legendBgPaint;
4  Paint gen_1 = this.bgPaint;
5  JFreeChart gen_0 = gen_6.useFactoryIndirect(gen_18);
6  StandardChartTheme gen_9 = ChartFactory.createTheme(gen_0);
7  gen_9.setLegendBackgroundPaint(gen_3);
8  gen_9.setChartBackgroundPaint(gen_1);
9  gen_9.apply(gen_0);
```

Figure 6.12: Solution to the makeLineChart query with a split chart theme. Compare with Figure 6.4.




```
1  JFreeChart gen_0 = this.chart;
2  gen_0.zoom(zf);
```

Figure 6.13: Solution to the zoomIn query with a hidden delegate. Compare with Figure 6.3. The other zoom methods have similar solutions.

6.1.8 Hiding a delegate

The JFreeChart class delegates many operations to the Plot class or to one of its subclasses. In order to perform an operation such as zooming, it is necessary to first obtain the Plot from the JFreeChart and then zoom. In this kind of situation, it is possible to perform the *Hide Delegate* refactoring [34, p. 157]. The reverse of this refactoring is *Remove Middle Man* [34, p. 160] and the same principles apply. Introducing this refactoring with Poplar is extremely simple. We add a forwarding method to JFreeChart:

```
1  class JFreeChart {
2  //...
3  resource zoomState {
4    properties @zoomed;
5
6    void zoom(double factor) factor: zoomDelta; this: ++@zoomed. {
7      plot.zoom(factor);
8    }
9  }
10 //...
11 }
```

We have also added a property and a resource to track whether the chart has been zoomed. From the perspective of our client application, the JFreeChart class is more accessible than Plot, so forwarding methods like this one always take precedence, leading to smaller solutions, as can be seen in Figure 6.13. Once sufficient forwarding methods have been introduced, one may disable direct access to the Plot class if this is desirable. In the reverse refactoring, one may remove all pure forwarding methods once access to the delegate object has been granted, provided that the new, longer solutions will not see competition from some other possible solution.

6.1.9 Introducing data readers

In this final section we consider another of Fowler's "big refactorings": *Tease Apart Inheritance* [34, p. 362]. In the original JFreeChart library, we may note that multiple data sets have JDBC reading functionality. Thus, data set classes are described by two adjectives: what kind of data set they are, and whether they support JDBC. Thus we have JDBCCategoryDataset, DefaultCategoryDataset, JDBCPieDataset, and so on. In this situation, the class hierarchy is doing "two jobs at once", in Fowler's terms, and it is appropriate to extract one of these functionalities into a separate hierarchy of delegate classes. Thus, we introduce the concept of a DataReader, which is a delegate that has some functionality for obtaining data from a source. We pass this delegate to the constructor of the top level AbstractDataset class. Concrete subclasses will then need to decide how to make use of it. We also add new functionality by turning the CSV class, which had been separate (producing a CategoryDataset only) into a second kind of data reader.

Substeps of this refactoring include refactorings such as *Extract Class* and *Move Method*. The class diagram before and after our changes can be seen in Figure 6.14.

```
1  class AbstractDataset {
2    resource data {
3      properties @populated;
4      private DataReader reader;
5    }
6    //...
7    AbstractDataset(DataReader reader) reader:@dataLoaded; result: +
         @populated. {
8      this();
9      this.reader = reader;
10     readInitialData(); //Subclasses implement this method in different
           ways
11   }
12   //...
13 }
14
15 abstract class DataReader {
16   resource data {
17     properties @dataLoaded;
18   }
19
20   abstract Collection<String> getRowKeys();
21   abstract Collection<String> getColumnNames();
22   abstract Collection<Double[]> getDataValues();
23 }
24
25 class JDBCDataReader {
26   JDBCDataReader(String url, String driverName,
27     String user, String passwd)
28     throws ClassNotFoundException, SQLException
29     result: ++any; url: jdbcUrl; driverName: jdbcDriver;
30     user: jdbcUser; passwd: jdbcPasswd. {
31
32     Class.forName(driverName);
33     this.connection = DriverManager.getConnection(url, user, passwd);
34   }
35
36   void executeQuery(String query) throws SQLException query: sqlQuery;
37     this: ++@dataLoaded. { ... }
38 } //Similar code for CSVDataReader
```

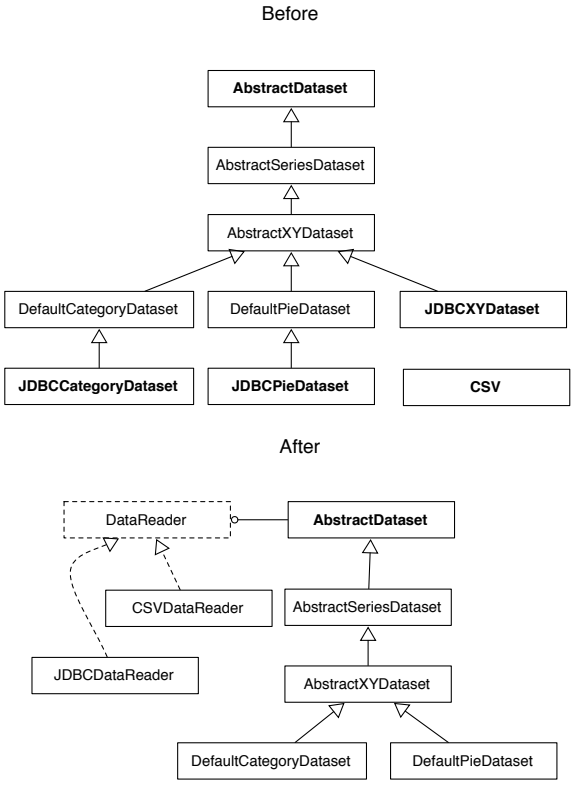
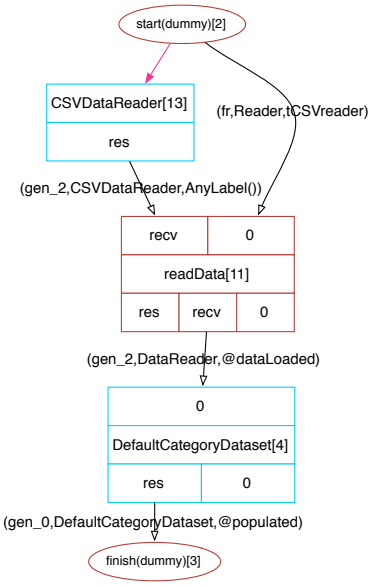


Figure 6.14: Class diagram for the DataReader refactoring

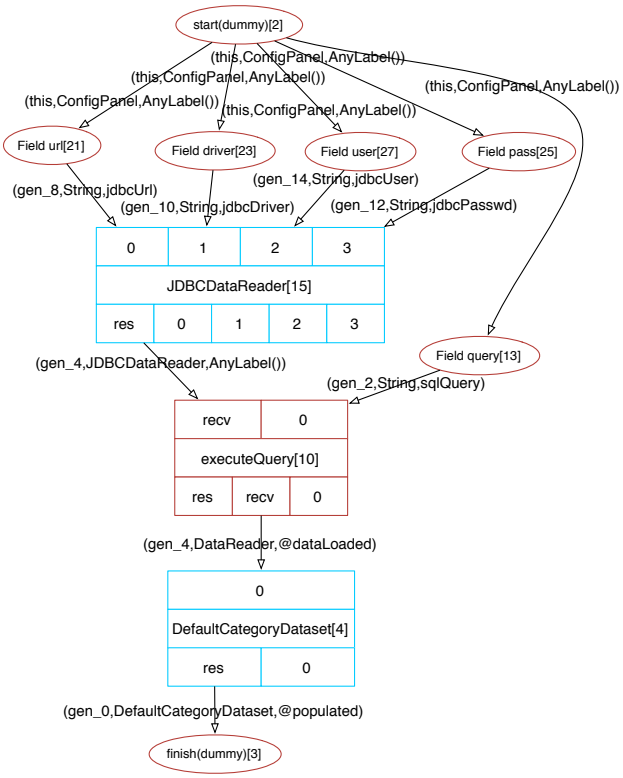



```
1  CSVDataReader gen_2 = new CSVDataReader();
2  gen_2.readData(fr);
3  DefaultCategoryDataset gen_0 = new DefaultCategoryDataset(gen_2);
```

Figure 6.15: Solution to the categoryCSV query with data readers. Compare with Figure 6.6.

In Figure 6.15 and Figure 6.16, solutions can be seen for the categoryCSV and categoryJDBC methods, respectively. Similar solutions were produced for other methods, such as pieCSV and pieJDBC, etc.

We may note that in this case, constructing the CSVReader is simpler than constructing JDBCReader from a Poplar perspective, since the former requires fewer actions. In methods such as categoryJDBC (Section 6.1.1) it is only the absence of a FileReader with the label tCSVreader that forces the construction of the JDBC equivalent. If one wants additional certainty that the correct kind of reader is being constructed, additional labels may be used for disambiguation.




```
1  String gen_8 = this.url;
2  String gen_10 = this.driver;
3  String gen_14 = this.user;
4  String gen_12 = this.pass;
5  String gen_2 = this.query;
6  JDBCDataReader gen_4 = new JDBCDataReader(gen_8, gen_10, gen_14, gen_12
       );
7  gen_4.executeQuery(gen_2);
8  DefaultCategoryDataset gen_0 = new DefaultCategoryDataset(gen_4);
```

Figure 6.16: Solution to the categoryJDBC query with data readers. Compare with Figure 6.7.

6.2 Application to Fowler's Refactorings

We now discuss the application of Poplar to all of the refactorings found in Fowler's book. The structure of this discussion mirrors Fowler's own way of structuring his catalogue of refactorings, with one section for each of his chapters.

We will use the symbol ○ to indicate that Poplar is generally useful with respect to a given refactoring, △ to indicate that it is sometimes useful, and × to indicate that it is generally not useful.

In the absence of other heuristics, Poplar always prefers short solutions over long ones. For this reason, it is important to know whether a particular refactoring makes solutions larger or smaller. If an existing solution becomes larger, it might no longer be selected. If a new solution becomes smaller than existing ones, it will always be selected instead of those. We call the minimum number of steps from a starting context to a goal the *path length* of a solution. In the following tables, we have indicated whether particular refactorings make paths longer or shorter. If the results are not as desired, interactive experimentation may be needed. It should be noted that in the case where a component supplier is supplying his own labels, which are not shared with other components, they are also able to control all the paths that lead to solutions for those labels, which eliminates the need for experimentation.

6.2.1 Composing methods

These refactorings are generally not intended to cross the component boundary, exposing changes in a service component to clients. Instead they are intended to help the developer of a component to structure his code better for his own benefit. Even though this is not the main focus of Poplar, there are still some situations where it can help.

Refactoring	Feas.	Technique	Comments	Ex.
Extract method	○	A query may be used to invoke the new method		
Inline method	×			
Inline temp	×		Queries cannot replace individual variables	
Replace temp with query	×		Queries cannot replace individual variables	
Introduce explaining variable	×			
Split temporary variable	×			
Remove assignments to parameters	×			
Replace method with method object	○	A single query may be used to construct the object and invoke a method on it		
Substitute algorithm	×		Poplar cannot be used to aid readability	

Those refactorings that focus on introducing, removing or changing temporary variables (”temps”) are at too fine grained a level for Poplar to be useful. Poplar queries replace statements, not expressions such as variables. However, in the case of refactorings such as *Extract method* and *Replace method with method object*, queries can be used to invoke the resulting method or method object. This adds flexibility, since the programmer can easily perform changes such as adding or removing parameters of the extracted method afterward.

6.2.2 Moving features between objects

The refactorings in this section are a very good fit for Poplar. We call them "structural", since they change the structure and exported interfaces of classes without essentially changing any of the overall capabilities of a component or set of classes.

Refactoring	Feas.	Technique	Comments	Ex.
Move method	○	Invoke method with query		
Move field	○	Access field with query		
Extract class	○	Access members in the new class using queries		
Inline class	△	Access members in the destination class using queries	Clients must not depend on the old class name explicitly	
Hide delegate	△	Access members in the destination class using queries	Clients must not depend on the old class name explicitly. Paths become shorter.	6.1.8
Remove middle man	△	Extra protocol steps may be used to introduce calls to the delegate		6.1.8
Introduce foreign method	△	Queries may possibly be used to construct and to invoke the foreign method		
Introduce local extension	△	Poplar may help in automatically instantiating a wrapper (but not a subclass)		

Poplar is generally very applicable to these refactorings. However, one of the main limitations is that client queries must not be dependent on class names that will disappear. Both *Inline class* and *Hide delegate* involve removing dependencies on a specific class. With a query such as

```
C x = #produce(C, @p);
```

a dependency on the class C is exposed to clients, and this cannot automatically be removed. This is also true if the clients declare a variable of type C. On the other hand, if the class C is only used as an intermediate step to achieve something else (and if these steps are automatically generated by Poplar), then C can easily be inlined or hidden.

With the *Hide delegate* refactoring, members that are now accessed through the outer class instead (from the client's perspective) become available on shorter paths (using smaller solutions) than before. This means that if the same members are both available through the delegate and through the outer class, the outer class will always be favoured. The shortening of paths in general may also change some existing solutions to other queries.

6.2.3 Organising data

Poplar is generally useful for these refactorings. When data items are reorganised, the smallest unit that can be identified is generally a field or the return values of methods. Poplar can describe these data elements using labels. As long as queries are used to access these items, changes in data organisation and structure are mostly transparent. One exception is that Poplar does not yet have explicit collection or array support, something that could be added in the future.

Refactoring	Feas.	Technique	Comments	Ex.
Self encapsulate field	○	Describe field with labels, access it using queries	Questionable benefit	
Replace data value with object	○		Paths become longer	
Change value to reference	×	Poplar may help in replacing constructor calls with factory method calls		
Change reference to value	×	Poplar may help in replacing factory method calls with constructor calls		
Replace array with object	×		Poplar does not yet support arrays	
Duplicate observed data	×			
Change unidirectional association to bidirectional	△	If queries are used, the new reference can be used automatically where possible	Paths become shorter	
Change bidirectional association to unidirectional	△	If queries are used, access can be redirected where possible	Paths become longer	
Replace magic number with symbolic constant	△	If the constant is labelled, Poplar may be used to access it in some cases		
Encapsulate field	○	Label or constrain the field. Use queries to access it.		
Encapsulate collection	×		Poplar does not yet have collection support.	
Replace record with data class	×	Poplar may help in accessing the individual members of the data class.		
Replace type code with class	○	Poplar helps when method signatures that use the type code need to change to use the new class		
Replace type code with subclasses	○	As above		
Replace type code with state/strategy	○	As above		
Replace subclass with fields	○	Poplar may help in invoking a new factory method instead of explicit constructors	Clients must not depend on the old class name explicitly	

6.2.4 Simplifying conditional expressions

Poplar is generally a poor fit for the refactorings described in this chapter. For the most part, they do not cross the service/client component boundary, and they operate at the fine grained level of individual variables. More generally, the goal of most of the refactorings in this chapter is to improve code readability, which is something Poplar cannot do, and which also falls outside the problem of the composability-evolvability conflict.

Refactoring	Feas.	Technique	Comments	Ex.
Decompose conditional	×		Poplar cannot be used to replace individual variables	
Consolidate conditional expression	×		As above	
Consolidate duplicate conditional fragments	×		Poplar cannot be used for intra-method code movement	
Remove control flag	×		Poplar is not applicable	
Replace nested conditional with guard clauses	×		Poplar does not generate branches	
Replace conditional with polymorphism	△	Poplar may help in replacing a factory method with a constructor		6.1.7
Introduce null object	×		Poplar cannot be used to replace specific values	
Introduce assertion	×		Poplar has no clear benefit	

6.2.5 Making method calls simpler

Changes in method signatures are one of the most suitable refactoring types for Poplar. Poplar can easily be applied to most of the refactorings in this chapter, as we have demonstrated in several examples.

Refactoring	Feas.	Technique	Comments	Ex.
Rename method	○	If the method is invoked using a query, we can guarantee correct replacement.	Poplar makes method and field names entirely irrelevant.	
Add parameter	○	Parameters can be found if method invoked using a query.	Paths become longer	6.1.5
Remove parameter	○	Parameters can be removed if method invoked using a query.	Paths become shorter	6.1.5
Separate query from modifer	○	A separate property should express the state of having been modified.	Here, *query* is a Java method that queries a value, not a Poplar query	
Parameterise method	○	The parameter can be found if method invoked using a query.	Paths become longer	
Replace parameter with explicit methods	○	Additional labels should be used to disambiguate between methods.	Paths become shorter	
Preserve whole object	○	Parameters can be replaced if method invoked using a query.	Paths become shorter	
Replace parameter with method	○	Automatic if the inner access is done using a query.		
Introduce parameter object	○	Parameters can be replaced if method invoked using a query.	Paths become shorter	6.1.5
Remove setting method	×	Poplar adds no benefit		
Hide method	×	Poplar adds no benefit		
Replace constructor with factory method	△		Factory method may compete with constructor	6.1.7
Encapsulate downcast	×		Poplar adds no benefit	
Replace error code with exception	×	Poplar does not support exceptions yet		
Replace exception with test	×	Poplar does not support exceptions yet		

6.2.6 Dealing with generalisation

When features are moved between different levels in the class hierarchy, or when classes are removed or introduced, interface compatibility often breaks in a similar manner as in Section 6.2.5. Poplar can be used in many cases to ease the transition. However, the same caveat as in Section 6.2.2 applies: if clients explicitly reference a class name, whether through a variable type or a query, this class cannot suddenly disappear without breaking the client.

Refactoring	Feas.	Technique	Comments	Ex.
Pull up field	×		Does generally not make the API incompatible	
Pull up method	×		As above	6.1.7
Pull up constructor body	×		As above	
Push down method	△	If queries are used, the new method can generally be found automatically.	Sometimes, a downcast or type change of a variable might be needed.	6.1.7
Push down field	△	If queries are used, the new field can generally be found automatically.	As above	
Extract subclass	△	Queries may help in creating and using the new class.	As above	6.1.7
Extract superclass	×		Does generally not make the API incompatible	
Extract interface	×		Does generally not make the API incompatible. Poplar has no interface support yet.	
Collapse hierarchy	△	Queries may help in creating and using the new class.	Sometimes, manual casts or type changes may be needed.	
Form template method	×		Does generally not make the API incompatible	
Replace inheritance with delegation	○	Poplar can help create and use the new delegate objects if queries are used		6.1.9
Replace delegation with inheritance	○	Poplar can help create objects with new constructor signatures		6.1.9

6.2.7 Big refactorings

In big refactorings, many steps, if not all, can be supported by Poplar. We have given examples of two of these refactorings with our JFreeChart application.

Refactoring	Feas.	Technique	Comments	Ex.
Tease apart inheritance	○	Objects in the hierarchy should be created and interacted with using queries.		6.1.9
Convert procedural design to objects	○	As above		
Separate domain from presentation	○	As above		
Extract hierarchy	○	As above		6.1.7

6.2.8 New refactorings

The refactorings in this section are not discussed by Fowler, but become easy to perform with Poplar, and we believe that they may be useful in many situations.

Refactoring	Feas.	Technique	Comments	Ex.
Introduce preaction	○	Introduce a mandatory method invocation prior to accessing a method or field. This can be done using *Split protocol state* below.	May be used for logging, debugging etc	
Introduce postaction	×		Poplar cannot be used to require a method invocation *after* something else has been achieved. Possible future extension.	
Collapse protocol state	○	Collapse properties or labels into one		
Split protocol state	○	Introduce a new property or label, and require it in order to produce another		
Replace parameter with setter	○	Introduce setting method, associate with a property (new or existing)		6.1.6
Replace setter with parameter	○	Remove setting method		6.1.6

6.2.9 Summary

In total, Martin Fowler provides 72 different refactorings, although many of them are each other's inverses. Out of these, Poplar can generally always be applied to 27 refactorings (○), and can be applied in certain circumstances to another 14 refactorings (△). It is non-applicable to 31 refactorings (×), although this includes refactorings that Poplar is not intended to be used with, such as "simplifying conditional expressions", a type of refactoring that does not affect interfaces. Poplar is highly applicable to a wide range of refactorings. Obstacles to applying Poplar often involve the explicit dependency on a class name. A general consideration when applying Poplar with a certain refactoring is whether solution paths become shorter or longer; this may influence what generated results are to be expected when the Poplar solver is run.

6.3 Discussion

In the experiments we have just shown, we performed a number of successive transformations of a client of the JFreeChart API. In each case, when it was possible, Jardine was able to catch up with the change and correctly generate new integrating code. In some cases no valid solution existed, and Jardine could correctly tell the user that there was a problem, and what the problem was.

The range of transformations we used is not exhaustive, and does not show what the limit of Poplar's capabilities is, but it does show a lower bound. We were able to successfully use it for a range of transformations, many of which are refactorings recommended by Fowler [34]. Others were selected specifically to demonstrate Jardine's abilities, though we do not believe that they are contrived or far-fetched.

What we have confirmed above all is that Jardine successfully decouples superficial, structural variance from true variance in a component's capabilities. In the transformations we carried out, JFreeChart's abilities were not reduced, but its interfaces and their constraints, temporal and otherwise, were drastically changed. Jardine was able to find the new, updated method invocation sequences and arguments required to retain the same functionality as the superficial structure of the component changed. Given that refactorings are such an essential part of software maintenance, the value of this ability is clear.

The case study we have just carried out has also demonstrated that it is feasible, at least in this case, to take an existing Java library and retrofit Poplar annotations into it. This means that Poplar would be valuable not only for new software development, but also as a tool that can help manage existing code bases.

In all of these cases, Jardine found solutions in under 1.5 seconds, and the search space for plans was below 50 candidates. In addition, memory usage was stable, never exceeding 300 MB.

We also discussed the possible application of Poplar to the refactorings in Martin Fowler's book on the topic, and found that Poplar could be applied to most of them. Poplar is especially strong in the categories *Moving features between objects*, *Making method calls simpler*, *Dealing with generalisation*, and *Big refactorings*. On the other hand, some of Fowler's refactorings concern rearranging data or code inside methods, rather than changes that are visible in the boundary between components. Such fine grained changes are outside of the scope of Poplar. In general, Poplar has very good applicability to the problem that it targets, namely the evolvability-composability conflict.

6.3.1 Limitations

An important limitation of Poplar is that control flow constructs are not generated. For instance, we do not generate loops or if-statements. This means that when flow control is required, as in the case of an iterator which needs to repeatedly test a truth condition in a loop, this information should be communicated externally. For an iterator, separate queries could be used for the truth condition and for the loop body.

Poplar assumes, in #produce queries, that the type of the desired value is known. The more specific this type is, the easier the query will be to resolve. However, this may also couple the client strongly to a particular implementation, in the case where the same functionality is supplied by different components that have no shared type hierarchy.

In Section 4.4.1 and Section 4.6 we discussed some limitations that stem from the basic formalism. The simple, restrictive aliasing scheme that we use is a source both of inability to deploy Poplar (if the program cannot be described our uniqueness kinds) and of imprecision in the analysis, which leads to an overestimation of the potential resource mutations of a code fragment.

For instance, the AWT GUI toolkit, and by extension the Swing toolkit, has a problematic design feature in the java.awt.Frame class. When a new instance of Frame is created, it registers an alias of itself in a class called AppCon. This behaviour can be seen in the OpenJDK6 implementation of Frame, and probably in other versions as well. The effect is that instances of Frame can never be unique, as they are aliased by construction. This applies to javax.swing.JFrame as well, since it is a subclass of the former class. Both of these toolkits are major parts of the official Java platform libraries. And in fact, creating a new alias in a constructor, and registering it somewhere, is not such a controversial thing to do: we may expect that many frameworks and libraries do this. However, a consequence from a Poplar perspective is that we may never view a Frame or a JFrame as being unique.

One way out of this is to apply the distinction between ”Poplar methods” and ”plain methods” described in Section 5.3. If this is done, uniqueness kinds need only apply to those references that are directly visible to Poplar. However, using this strategy increases the burden on the programmer, who has to manually ensure that the guaranteed uniqueness property is upheld, and a more sophisticated solution is desirable.

Our treatment of basic establishers, which initialise properties, means that properties established by superclasses may not be used by the methods that establish the corresponding subclass properties, which may be too restrictive for some programs. Also, we cannot precisely identify objects that are mutated by external resources. As we have seen in this chapter, these limitations do not preclude all realistic uses of Poplar, but they would be the main limitations that need to be addressed.

6.3.2 Reliability

Developers using Poplar might be concerned about the fact that generated code can vary from time to time. Our basic means of ensuring sensible outcomes of code generation are as follows. Firstly, variables with the same type and labels must be truly equivalent. If they are not substitutable for each other, more labels should be added to disambiguate. Second, protection spans should be used to protect those resources that may not be mutated. In the case of handwritten code that surrounds queries, developers may need to identify resources that need to be protected manually.

The outcome of a code generation attempt is dictated by the choices of algorithm and heuristics. In general, we think that a good algorithm should strive to favour short valid solutions over longer valid solutions, once other needs have been taken into account. Thus, one situation that may lead to unexpected outcomes is if a component supplier makes it *easier than before* to produce a given type/label type, thereby making the altered protocol a simpler means of production than existing protocols. In this case, at the next regeneration, the shortened code fragment may out-compete some existing solutions in the quest to be shortest, which could have adverse consequences if annotations are not precise enough.

6.3.3 Adoptability

In adopting Poplar for use in an existing Java code base, it is necessary to add annotations concerning mutations, uniqueness, protocols, resources and so on. We have seen that information such as mutation summaries and uniqueness can easily be inferred locally on a class by class basis, and such an inference tool should be simple to make. In addition, if one sets up a temporary mapping between unmanaged fields and abstract resources in a class (to be discarded afterwards), it should be easy to infer a partial set of resource definitions. It is even possible to infer protocols, as in [75]. In addition, when Poplar is introduced into an existing code base, it should be possible to start with a small number of queries and annotations, and gradually expand the use of Poplar within the code base.

6.3.4 Developing new Poplar components

We have seen in this chapter that it is feasible to retrofit Poplar into existing Java code and use Jardine to integrate the resulting components. Development of new Poplar components from scratch, on the other hand, might pose slightly different requirements. For one thing, one is able to influence the search process through the design of interfaces.

If Jardine is used as the principal integration mechanism, API design no longer needs to only take human users into account. Designing an API for Jardine might be done with a specific purpose in mind. For instance, if methods are very fine grained, each one performing only a minimal amount of functionality, then one obtains a fine grained language that can be used to interact with the resulting component. A large number of combinations of fine grained methods is possible, resulting in a larger search space and a larger number of possible plans, and more integration flexibility. On the other hand, more coarse grained methods, which establish many properties together, reduce the size of the search space and may remove certain code fragments from the final integration capabilities. Further work is needed in order to establish the best design principles, but tentatively we believe that a maximally fine grained API is the best approach, given that the performance of Jardine is already fully acceptable, and likely can be optimised much more.

6.4 Conclusion

In this chapter, we have performed a case study on an application based on the well known JFreeChart library, as well as discussed the application of Poplar to all of Martin Fowler's well known refactorings. Our case study demonstrated the wide range of refactorings that could be performed on service components in practice with the help of Poplar, entirely without manual changes to client code. This shows the practical potential of the Poplar approach for evolution of component based software. The theoretical discussion of Fowler's refactoring catalog demonstrated that Poplar should always be applicable to 27 of Fowler's 72 refactorings, and conditionally applicable to 14 refactorings. Many of the remaining 31 refactorings do not affect interfaces and thus fall outside Poplar's focus. In the following chapter we will discuss general related work and conclude this thesis.

7 Related Work and Conclusion

In this chapter, we discuss related work, conclude the thesis and comment on directions for future work.

7.1 Related Work

In this section we discuss related work in general. Typestate checking, effect system and alias confinement is discussed in the context of our formalisation, in Section 4.5.

7.1.1 Behavioural specifications

Hoare logic was a seminal formalism for method specification, which expressed pre-conditions (P) and postconditions (R) together with program fragments: P {Q} R. There is a large amount of extensions of this idea of specifying behaviours together with fragments. Hatcliff, Leavens et al have provided a survey of behavioural interface specification languages [69]. Poplar, too, is fundamentally based on this idea. What Poplar and most other typestate and protocol related formalisms emphasise is a relative simplicity of the formalism used to specify Q and R. Many other specification languages use first-order logic or languages of similar richness, for which automated theorem proving is not decidable.

Design by contract [79] is a name for a family of practices where client and service components are designed according to a shared functionality agreement. Such agreements are often expresses in some kind of formal specification language. The programming language Eiffel [56, p. 57] integrates a sophisticated assertion system for contracts with the language itself. JML [67] is a markup language that supports extended specifications, including pre- and postconditions for design by contract, for Java. A JML compiler is able to compile JML into Java code that includes assertions that verify the contracts as required. There are also tools such as the extended Java static checker ESC/2, which can perform a wide range of static analyses from JML input.

Separation logic [89, 88] solves the *frame problem* by guaranteeing that procedures do not modify any state that has not been explicitly mentioned in its precondition. Separation logic provides formalisms for reasoning about *disjoint heaps*, which are guaranteed not to reference each other. This makes specifications modular and composable.

7.1.2 Labelled argument selection

To the best of our knowledge, there is currently no major imperative programming language that supports argument selection based on labels. The technique has been studied in the context of labelled lambda calculus, however[1, 39, 38]. Labelled lambda calculus selects appropriate function arguments based on the labels of expressions. Haack has extended ML with semantic symbols [48] to perform automated integration. They are more sophisticated than mere labels, since they also contain axioms for reasoning about concepts such as sets and orderings.

Languages such as ADA [68] and Common Lisp [106] support a notion of labelled arguments, but this is used to allow the programmer to reorder or omit arguments, and not as a basis for selecting the arguments to be passed from some kind of context. Thus, while this mechanism can simplify maintenance programming, it cannot serve as a foundation for an automated integration process. Objective-C uses argument labels to help identify the method (called a selector) itself: reordering or omission of arguments is not permitted. This mechanism provides no additional convenience compared with Java beyond a more readable method naming [54, p. 16].

7.1.3 AI planning

Ghallab, Nau and Traverso have written an overview of the field [40]. Russell and Norvig discuss AI planning in the broader context of artificial intelligence [100].

The first discussion of partial order planning was by McAllester and Rosenblitt [77]. For a friendly introduction, see [40, p. 99]. The algorithm lost popularity to other planning algorithms for some time due to concerns about its scalability. However, recently Nguyen et al. were able to show that the techniques that made other planning algorithms successful could also be applied to make POP efficient [83]. Ireland and Stark have combined proof plans with partial order planning to synthesise imperative programs [57]. However, this is the only previous application of planning at the level of code synthesis in a programming language that we are aware of. Ireland and Stark synthesise programs from mathematical specifications bottom-up, which also means that their domain logic is complex compared with Poplar.

There are many specialised languages for expressing planning problems. One example is PDDL [100, p. 367], derived from STRIPS. The availability of such languages means that it is in theory easy to replace planners and heuristics independently of other parts of a system that uses planning algorithms. Our reason for developing our own planner was the relative ease of development and the possibility of making changes to the planner freely. Also, Poplar is not yet at a stage where a high performance planner is needed.

7.1.4 Code synthesis and component generation

Poplar carries out a form of code generation based on AI planning. There is a rich variety of existing work on code generation and program synthesis. In 1963, Church [23] presented a survey of ongoing work in finite automata at the time. He identified three central automata problems: the decision problem, the synthesis problem, and the simplification problem. The synthesis problem concerns itself with whether it is possible to construct an automaton satisfying a given specification. A substantial amount of research has been done into the field since the time, yielding various approaches with

varying degrees of success in special cases. However, in the general case, the problem is undecidable.

Approaches to program synthesis can broadly be classified in three categories: deductive, inductive, and transformational program synthesis. *Inductive* synthesis formulates hypotheses from examples and then generates a corresponding program. We may think of this as "programming by example". Jha, Gulwani et al recently presented a combined deductive and inductive approach that uses an SMT solver and a library of components to generate a program after first having been trained on examples [60]. In *deductive* synthesis, automated theorem proving is used to deduce the form of a program. Given a specification such as

$$f(a) \Leftarrow \text{ find z such that } Q[a, z].$$

where f is the function to be synthesized, a is the input, z is the output, and Q is a logical sentence of some background language, we use the prover to prove the following theorem.

$$(\forall a)(\exists z)Q[a, z].$$

In proving this, with each proof step, we gradually construct a method for finding z in terms of a. Manna and Waldinger give an introduction to the subject in their 1992 paper [76].

Several systems designed for deductive automated program development have been constructed, such as SNARK [108], NUT [119], Amphion [74] and KIDS [104]. Amphion composes subroutine libraries using a theorem prover, operating fundamentally at the same level of granularity as Poplar, but using a more complex search strategy. It must be trained to fit a particular domain (such as libraries of scientific subroutines) using proof tactics and heuristics.

Finally, *transformational* synthesis concerns itself with gradual refinement of a program. Each transformation step improves the program in terms of performance while preserving its observable behaviour. Thus this synthesis method requires some initial, possibly inefficient version of the program to exist as a specification. An example of this approach is the well-known Bird-Merteens formalism, described in [41]. Another example is Darlington's system, which successively applies transformations to increase program performance [25].

In addition to these general approaches, it is also possible to take various highly specialised approaches to program synthesis, such as program synthesis from statechart [49] specifications [121]. Srivastawa, Gulwani et al recently presented an approach that treats synthesis as generalised program verification [105]. Liu, Fu, Zhang et al. combine code patterns with deductive synthesis to generate software specifically for embedded systems [11]. Pnueli and Rosner have discussed how to synthesise reactive modules [95].

Lämmermann applies structural synthesis of programs, which is a form of deductive synthesis based on intuitionistic logic, to runtime composition of services in Java [66]. The main contrast between his system and Poplar would be the focus on services, which are relatively coarse grained components assumed not to have any externally visible side effects, and the focus on runtime composition rather than compile time composition. Because it composes individual Java statements, Poplar is operating at a more fine grained level and emphasises containment of side effects.

GenVoca generators [13] were discovered by Batory and O'Malley in several independently developed domain-specific component systems. They were later formalised

by Batory and Geraci[12] as extensions of the language P++, a superset of C++. GenVoca components are parameterised, composable software generators that can be combined hierarchically to generate concrete component implementations. Depending on the precise details of a component generation scenario, the concrete interfaces of the components (in GenVoca, too, sets of classes) may be different. This can be thought of, on some level, as the inverse of the approach taken in Poplar. GenVoca generators push flexibility to component users, top-down, by providing the possibility of generating interfaces that have a range of different (subjective) concrete forms. Poplar component clients pull flexibility from components by interpreting the fine grained building blocks they offer in a way that is appropriate for the client context (through the planning/search algorithm). In both cases, a compositional subjectivity may be realised. One important difference is that although Batory and Geraci provide algorithms for verifying the correctness of the generated components, we do not believe that GenVoca generators are well suited to handling component evolution. Using GenVoca generators involves manual work in finalising the integration of the generated components and the client code, and this would need to be turned into an automated task in order to adapt integrations to changes as easily as in Poplar.

7.1.5 Empirical studies of software evolution

Zenger et al proposed a taxonomy [132] of software evolution events based on four axes: temporal properties (*when* does the change occur?), object of change (*where* does the change occur?), temporal properties(*when*), and change support(*how*).

Dig and Johnson found that a large proportion of breaking changes are due to refactorings [28].

Vasa et al quantitatively studied component evolution in several software systems [120].

Component-based software usually involves a network of inter-component dependencies, which necessitates synchronisation of developers when multiple interdependent components are being developed simultaneously. De Souza et al studied how developer teams handle software dependencies and changes in practice [26].

7.1.6 Component matching, discovery and retrieval

There is much existing work that approaches the problem of component integration as being fundamentally a problem of *matching specifications*. In such an approach, one assumes a library of well-specified components. The problem to be solved is then: how do we find the best component to use in order to satisfy a well specified functional requirement? The seminal work in the area may be that of Zaremski and Wing [127, 126]. They define a lattice of different ways that a specification can match a component query. Poplar queries fundamentally correspond to the *plug-in match*.

Some component retrieval systems use sophisticated automated theorem proving in order to assure that the retrieved components are correct. In order to simplify the computational complexity of this approach, Schumann and Fischer proposed [102] an approach with a multi-layered proof filter. Higher level layers, which are simpler to prove, are first used to filter out a large number of components, and only later are more expensive, sophisticated filters used to assure the full correctness of a match.

A separate problem from that of formalised, automated matching done by tools is interactive component search for end users. Reiss presents an interactive semantics-based search with a web interface[97]. Users may enter examples of input-output pairs that they want a valid component to match.

7.1.7 Component frameworks and techniques

The notion of a software component goes back to 1968. At the time, McIlroy contrasted [78] the software industry with the manufacturing of physical goods, which benefited from a supply chain and from well-defined reusable components that could be selected from a wide range of supplier offerings and assembled in a wide variety of ways. At the time, there was a perception of a ”software crisis”, as the true difficulties of large scale software development were starting to become apparent.

Component frameworks

Besides languages and tools, one can identify a third class of solutions to the integration problem: frameworks, whose main parts usually consist of a library written in the same host language that the user is developing software in, together with an API and coding style designed to enable generally useful functionality that the language itself does not provide. Component frameworks often serve to help define and isolate the notion of a component and to provide integration, change and upgrade mechanisms. The scope is not always strictly in-language: some frameworks also include auxiliary tools.

In this section, we discuss some widespread component frameworks. Generally speaking, while they help discover and integrate components and manage the component lifecycle in large software systems, they do not address the problem of fragility caused by the information encoded interfaces. These frameworks augment programming languages; they do not interfere with fundamental mechanisms or concepts intrinsic to the language.

OSGI , Open Services Gateway Interface [116], is a service and component system for Java, which is implemented as a library and a runtime environment. It emphasises lifecycle management of components. For instance, components explicitly identify whether they are running, starting, stopping, stopped, installed or uninstalled. Through callbacks and events, components can become aware of their own and other components' state changes. In contrast with models such as CORBA, OSGI targets components that run in the same process (same JVM) only. OSGI has been developed by the OSGI alliance, a coalition of companies, and the first major release appeared in 2000. Its most widespread success is perhaps as a foundation for the Eclipse rich client platform.

COM+ COM is a fundamental component model of applications and frameworks found mainly in Microsoft operating systems and software. Like CORBA, it emphasises inter-language object and wiring management. COM+ later extended it by adding declarative (qualitative) attributes to components, which enable platform-dependent dependency injection. It has now largely been superseded by the .NET framework [114, p. 356].

CORBA (Common Object Request Broker Architecture) [47] is a platform that was originally released in 1991 by the Object Management Group, a non-profit consortium seeking to create a standard for interoperability of heterogeneous components. Components are described in an interface definition language (OMG IDL) for which bindings exist for many programming languages, including Java, Smalltalk and C++. Object requests go through the centralised request broker. A special compiler generates proxy objects that appear to be ”real” local objects, but forward all messages to a remote object.

The large amount of indirection necessary in order to integrate such a wide range of platforms and programming languages means that CORBA components often end up sacrificing some performance in order to gain interoperability.

JavaBeans are a connection-oriented component mechanism for Java, defined by Sun (now Oracle) [112]. A JavaBean is a set of classes. Concepts defined by the JavaBean standard include events, properties, introspection, customisation and persistence. A new distinction between design-time and run-time is made: a bean is expected to behave differently while it is being designed, before deployment, and while it is being used, after deployment. Advanced specifications such as persistence and producer-consumer frameworks are also available for JavaBeans.

There are also well-known techniques that are not strictly frameworks but that operate on the same scale. The Java language supports reflection, as we have seen, which allows a Java program to ”become aware” of itself, in a sense, by reasoning about its classes and their member declarations. Reflection and dynamic classloading leads directly to the possibility of certain component techniques within the Java language itself. For instance, JDBC is the standard Java SQL database connection package. Before it can be used, it requires the program to load a driver explicitly, as seen in the following example.

```
1 Class.forName("org.sqlite.JDBC");
2 Connection c = DriverManager.getConnection("jdbc:sqlite:test.db")
```

When the class org.sqlite.JDBC is loaded into a running JVM, its *class initialiser* is run. The class registers itself and tells the DriverManager to begin recognising URLs that begin with jdbc:sqlite. All other interaction with the driver is done through the JDBC API, so there is no other need to directly interact with classes from the org.sqlite package once its initialising class has been loaded.

It should be noted that the notion of *component* that we address in our work is different from the one that has been discussed in this section: we are interested in components as sets of classes in the Java language, running in a single virtual machine.

7.1.8 Handling component evolution

There is a wide range of work on handling evolution in component-based software systems. Broadly, existing approaches may be classified according to whether they are tools or language extensions. This distinction is still somewhat floating, since a tool that depends on program annotations to perform some kind of rewriting or checking can in effect be said to be an extended programming language.

Tools

Many solutions take the form of external tools designed to work with some language or binary format. Diff-Catchup [123] is an interactive Eclipse plugin that can suggest ways for the user to upgrade client code to match a new version of a supplied component. The tool generates suggestions based on the output from UMLDiff [124], a specialised tool for identifying version differences in code based on UML semantics. There are three main threats to the validity of UMLDiff upgrades. API changes that have no syntactic effects, such as an incompatible method contract change, cannot be discovered by the tool. UMLDiff does not necessarily give correct results, although studies show its reliability to be fairly high. Finally, UMLDiff depends on the presence

of ”voluntarily supplied” migration examples within the evolution history of the supplied component; it must incorporate examples as to the new correct usage of its API. Poplar does not have these problems. On the other hand, the novel language elements in Poplar make it harder to adopt: programmers must use our specification model and write corresponding annotations. Diff-Catchup can be adopted directly.

Stephen Kell has developed an integration language, Cake [63, 62], that integrates components using interface relations at the level of object code. As such it is best suited to languages like C and C++ and currently not useful for Java.

Language extensions

UpgradeJ [17] defines a Java extension that supports versioning of classes. When users instantiate a class, they indicate which version to instantiate. Such indications can be either an absolute version number, or the current recommended revision of a given version number, or the latest version of a class. Furthermore, the language has an explicit upgrade statement that performs the upgrade itself. In other words, class upgrading is a first class language feature. Upgrades can take the form of new classes, redefinitions of existing class members, or extending existing classes with new members. The approach supports sound incremental upgrading: every class only needs to be type-checked once, even as new versions of other classes arrive. The scope of this approach is slightly different from Poplar. The focus in UpgradeJ is on versioning of classes, but Poplar focusses on handling interface and usage contract changes. Upgrades in UpgradeJ cannot invalidate or remove existing code: future class versions must be compatible with old ones, according to the usual notion of Java binary compatibility. In addition, UpgradeJ focusses on handling runtime upgrades whereas Poplar is aimed at developers needing a compile time solution.

Languages such as ML have module systems to help programming in the large [118, p. 980] Currently, Java has no module system. Even though packages exist in a hierarchical structure, there is no way to define relationships among packages: a package cannot be contained in another, being visible only to its owner, in the way that a private class is visible only to the class that declares it. However, there is currently ongoing work on an official Java module system [109].

ArchJava is a language extension for programming in the large. It emphasises expressing the large scale architecture of a Java system together with the code. It guarantees *communication integrity*, the property that components only communicate with components that they have been formally connected to [4].

Keris [130] is a Java extension that focuses on safe extension of modules and on automatically inferring new wirings when an extension takes place.

The problem that Poplar focusses on is not strictly extension, but the more general problem of evolution. It is important to note that the kind of evolution we seek to address is not strictly monotonic evolution through addition or overriding of existing entities (classes, class members). We also address evolution that expresses itself through the removal of entities, as well as lateral changes such as renaming and reorganisation.

7.1.9 Other related work

Design patterns are a set of well-known programming idioms for object-oriented imperative programming languages. They were popularised by Gamma, Johnson, Helm

and Vlissides [37]. Many of the patterns strive to reduce coupling, for instance mediator, adapter, proxy and facade. However, as would be the case with any in-language technique, patterns cannot get around the constraints of the programming language itself, i.e. explicit dependencies on interfaces. Poplar can be thought of as a mechanism that generates instances of the *adapter* pattern, one of the most well known patterns, on the fly at compile time.

FeatureHouse [9] is a framework for language-independent composition of software artifacts. Artifacts can range from source code to other entities such as documentation. The authors were able to demonstrate its use with a wide range of different programming languages, including Java, Haskell and C. Artifacts are first broken down into a hierarchical structure, and superimposition of these hierarchical structures yields the final, composed structure.

In Aspect-oriented programming [64], programmers specify pointcuts using an expression language. A compiler (*aspect weaver*) inserts code at the pointcut sites at compile time. The success of this approach means that programmers should be willing to rely on a tool such as Poplar, which also inserts code at predefined sites.

7.2 Conclusion

We have now described, defined, implemented and investigated our approach. The time has now come to conclude the present work and review our results.

Our working hypothesis was that it is possible to extend the Java language with stateful labels in such a way that planning algorithms can be used to construct evolvable integration links between components. We can now confirm this hypothesis.

In Chapter 2, we described the overall design of our language, a combination of typestate, AI planning and effect systems. In Chapter 3 we described the semantics of our language informally.

In Chapter 4, we formalised Poplar as an MJ extension and argued that it provides a sound must-analysis for label establishment and a sound may-analysis for resource mutations. Here we also identified some of the main limitations of our current design: the alias confinement scheme is imprecise, which leads to overestimation of resource mutations in some cases. Furthermore, mutations on external resources cannot be linked to a specific object, which leads to a new programmer responsibility that cannot be statically checked. Our handling of newly established properties is also restrictive in that we do not have the concept of frame properties (which have been initialised in some class frames but not for others). All of these issues can potentially be addressed in future work. The current design specifically aimed for simplicity in order to prove the viability of a novel concept, and as the viability has now been established, increased precision would be a logical next step.

In Chapter 5, we described our implementation of a Poplar compiler and showed that the essential problems it is addressing - Poplar type checking and query solving - are decidable problems. The main source of complexity in Poplar type checking is resource fields with disjunctive specifications.

In Chapter 6, we performed a case study that demonstrates that integration links can indeed be constructed and evolved for a wide range of transformations of a large software library in a realistic application. The results show that a wide range of different transformations can be compensated for. We also discussed the application of Poplar to Martin Fowler's refactorings and found that it is able to simplify the task of performing these in a majority of the cases.

Considering the findings of this thesis, we believe that labelled argument selection, combined with our model of resources and properties, together with a search algorithm such as POP planning, is a viable approach to component integration in imperative object-oriented languages, and for the purposes of component evolution, far more flexible and robust than explicit integration through series of explicit method invocations.

7.3 Future Work

The novelty of our approach means that many different aspects are open for additional investigation, in addition to the precision issues we have already identified.

7.3.1 Runtime composition

In our work, we have only investigated the use of Poplar for compile time composition of components. However, with dynamic classloading and unloading it may be desirable to investigate runtime composition. The most important problem to solve in such a scenario may be that of trust. In a compile time composition scenario, the composition process is supervised, and developers can always, as a last resort, interfere and tweak annotations if they do not obtain the desired results. At runtime, especially if a product has been deployed to a customer site of some kind, there may be no possibility of supervision or interference when new classes are encountered in the wild. Trust policies, sandboxing of some kind, or even more fine grained effect systems may help mitigate these problems.

7.3.2 Java-compatible syntax

The current Poplar syntax is incompatible with standard Java. For example, it introduces new keywords such as unique and resource and uses @ signs to signify properties, a syntax that is normally used by Java annotations [45, p. 270]. In our work, insisting on using our own syntax simplified development, but it might be possible to make Poplar more accessible to many Java developers if, instead of using its own syntax, it expressed all the extra information in valid Java annotations. Java already permits annotations on all declarations, and JSR-308 [30] is an ongoing official Java extension proposal that is designed to permit annotations on individual types. This means that the granularity permitted by Java annotations should be sufficient to express all the necessary Poplar metadata. Queries could be expressed as calls to ”magic” methods that do nothing, but that are intercepted and replaced with solutions at compile time.

7.3.3 Additional language elements

An explicit inclusion modifier. Developers might want to specify labels or types that should definitely be included as part of the solution to a query. Such a preference could be implemented as a search heuristic that gives preference to the indicated label or type, possibly even favouring larger solutions up to a certain limit. Having this ability would let developers specify precisely the details of the integration that are important for their application and let the others be decided according to the state of the components.

Subresources. The Boyland-Greenhouse effect system implements regions and subregions [46], effectively creating a tree of regions for each type. It would be natural

to also implement subresources in Poplar, further increasing the precision of the mutations that may be specified. Subresources lead to natural refinements of the rules for subtyping and overriding. When a resource is mutated, all properties in that resource and in any of its subresources would be lost. This corresponds naturally to Statecharts [49] in the sense that a resource mutation would be a transition to an outer state corresponded to by the mutated resource.

Resource links. This feature would link mutations of one resource to mutations of another, imposing a free hierarchy among all the resources that have been defined. In theory this can be implemented with external resources as they are today, but such an approach would not be scalable. Resource links could potentially also be created and destroyed dynamically, using an analysis similar to the one in FUSION [58]. As an example, Figure 7.1 illustrates how some JDBC classes might be modelled and protected correctly with resource links. Among these classes, a mutation of Connection.state also mutates Statement.connection, which in turn influences ResultSet.results. This feature would be especially powerful if combined with subresources.

Disjunctive conditions. In many cases it is natural to specify more than one contract for a method. For instance, in the time and date example introduced in Section 2.4.1, the Calendar.get method used in Java 1.5 returns a different result depending on its argument. This could easily be specified as a disjunction of conjunctions, where sets of preconditions are declared together with sets of postconditions, in the same way that resource fields can currently be specified. To the Poplar compiler, this would be conceptually equivalent to several separate method declarations with the same name, and thus this feature would not add any new significant complications beyond the new syntax. However, it would add complexity to Poplar checking in the same way that resource fields do.

Resources and properties in interfaces. It would be natural to declare resources and properties in interfaces, and not just in classes. This would permit users to refer to properties by the the interface that they belong to, avoiding the coupling to a specific, implementing type when they are requested. It would also specify in advance the property allocation to resources that implementing classes must later abide by.

Handling a larger set of Java elements. The version of Poplar formalised and implemented here is unaware of many important Java elements. For instance, it is natural to include arrays, and not just fields, in resources as part of their state. Exception handling in generated code also needs to be addressed. A naive approach to exception handling would be to specify in advance exactly what exceptions may be thrown by generated code. This would be sound, but it would be a source of coupling and incompatibility, and it might preclude the integrator from using new, unknown components to satisfy old queries. Further investigation would be needed in order to clarify how exceptions might be handled more gracefully.

7.3.4 Poplar specification mining

We mentioned Prospector [75] as one inspiration for our work in Section 2.3.3. Prospector mines protocol fragments from a corpus of code, assumed to be valid, and uses this

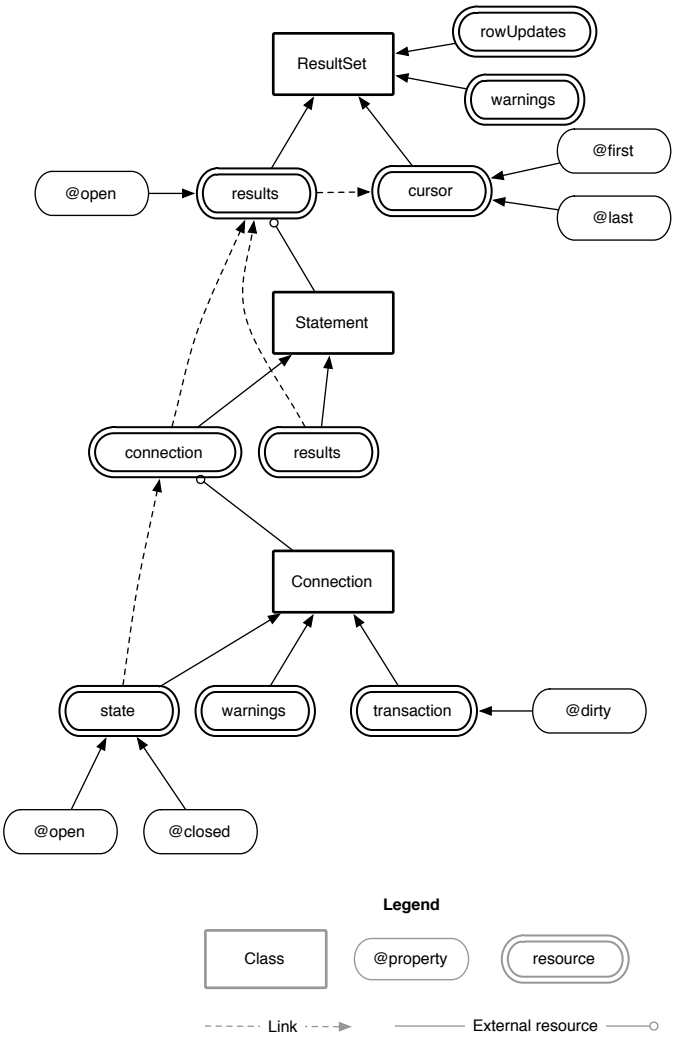


Figure 7.1: A future extension: protecting external state in JDBC with resource links

as a basis for code generation later. Another example of protocol mining is given by Gabel and Su [35]. Pradel and Gross mine protocols from execution traces [96]. Alur extracts specifications using a game-theoretical approach [7]. It should be similarly possible for Poplar, given a valid code base, perhaps seeded with a small number of initial property and resource names, to infer the most conservative specification that matches a given code base and captures its temporal constraints. It might also be possible to discover resources and the methods or fields that mutate them, with the exception of mutation done from native methods, where the precise state that is being mutated cannot be discovered by examining Java code.

7.3.5 Implementation improvements

Jardine, the Poplar compiler described in Chapter 5, currently lacks several features that would be very helpful in widespread practical use.

Subtype validation. Jardine does not currently check whether subtypes override their supertypes in a valid way, although we have formalised this notion.

Java source output. Jardine only outputs compiled Java code currently, and it would be desirable to also optionally output Java source code, which is easier for developers to inspect without having to resort to special tools.

Integration link verification. We discussed in Section 5.10 how integration link verification could be implemented, but implementing this feature remains to do as future work.

7.3.6 Quality parameters

Poplar provides a well-defined notion of a valid integration, which is guaranteed to be bounded above and below in terms of its destructive and constructive effects. The implementation described in Chapter 5 is content to find any valid integration. But it might be desirable to distinguish further between various valid integrations according to how well they fit certain quality criteria. For instance, methods could be annotated with information about their memory consumption, amount of I/O performed, CPU usage, and other such information, and hard or soft constraints could be placed on these quality parameters to further improve the solution search. In this work we have focussed strictly on the notion of a *correct* integration, and thus, a study of the generation of *high quality* integrations might be desirable in the future.

7.3.7 Analysis precision

A significant source of limitations is the imprecision of our alias analysis, and by extension our effect system. Many fragments will appear to Poplar as interfering with each other even though in practice they will not. One source of improvements here would be the adoption of some kind of ownership scheme, which would be able to go beyond the simple distinction between Unique and any(T) mutations of resources.

Cherem and Rugina [22] developed an escape and effect analysis that is parameterised in terms of the number of fields per object and the heap depth being tracked. Method summaries are generated automatically, and it is possible to find a good tradeoff between efficiency and precision thanks to the parameters. This approach could possibly be adopted for use with more precise mutation summaries in Poplar.

Bibliography

[1] Hassan Aït-Kaci and Jacques Garrigue. Label-selective λ-calculus syntax and confluence. *Theor. Comput. Sci.*, 151:353–383, November 1995.

[2] J Aldrich, V Kostadinov, and C Chambers. Alias annotations for program understanding. In *OOPSLA '02: Proceedings of the 17th annual ACM SIGPLAN Conference on Object-Oriented Programming: Systems, Languages and Applications*, volume 37, pages 311–330. ACM, 2002.

[3] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *Onward! at OOPSLA '09: Proceedings of the 24th annual ACM SIGPLAN Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 1015–1022. ACM, 2009.

[4] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in ArchJava. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 334–367, London, UK, 2002. Springer-Verlag.

[5] L De Alfaro and T A Henzinger. Interface automata. In *Proceedings of the 9th Annual Symposium on Foundations of Software Engineering (FSE)*, volume pages, pages 109–120. ACM, 2001.

[6] Ernesto J. Alfonso. Automatic protocol-conformance recommendations. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '11, pages 207–208, New York, NY, USA, 2011. ACM.

[7] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 40, pages 98–109. ACM, January 2005.

[8] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[9] Sven Apel, Christian Kastner, and Christian Lengauer. FEATUREHOUSE: Language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 221–231, Washington, DC, USA, 2009. IEEE Computer Society.

[10] Ken Arnold, James Gosling, and David Holmes. *Java^{TM} Programming Language, The (4th Edition)*. Prentice Hall, 4 edition, 2005.

[11] F. Bastani. Deductive glue code synthesis for embedded software systems based on code patterns. In *Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, volume 91109, pages 109–116. IEEE, 2006.

[12] Don Batory and Bart J. Geraci. Composition validation and subjectivity in genvoca generators. *IEEE Transactions on Software Engineering*, 23:67–82, 1997.

[13] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusamble components. *ACM Transactions on Software Engineering and Methodology*, 1992.

[14] Kevin Bierhoff and J Aldrich. Lightweight object specification with typestates. In *FSE*, 2005.

[15] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA '07: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 301–320, New York, NY, USA, 2007. ACM.

[16] Kevin Bierhoff, Jonathan Aldrich, Taekgoo Kim, and Sungwoon Kang. Typestate protocol specification in JML. In *SAVCBS 2009*, 2009.

[17] Gavin Bierman, Matthew Parkinson, and James Noble. UpgradeJ: Incremental typechecking for class upgrades. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 235–259, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] G.M. Bierman, M.J. Parkinson, and A.M. Pitts. MJ: An imperative core calculus for java and java with effects. Technical Report 563, University of Cambridge, 2003.

[19] Joshua Bloch. *Effective Java: Programming Language Guide (Java Series)*. Addison-Wesley, first printing edition, 2001.

[20] John Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium*, 2003.

[21] Mark Carman, L. Serafini, and Paolo Traverso. Web service composition as planning. In *ICAPS 2003 Workshop on Planning for Web Services*, 2003.

[22] Sigmund Cherem and Radu Rugina. A practical escape and effect analysis for building lightweight method summaries, 2007.

[23] Alonzo Church. Logic, arithmetic and automata. In *International Congress of Mathematicians, Stockholm*. Inst. Mittag-Leffler, 1962.

[24] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 48–64, New York, NY, USA, 1998. ACM.

[25] J. Darlington. *An experimental program transformation and synthesis system*, pages 99–121. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1986.

[26] Cleidson R. B. de Souza and David F. Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proceedings of the 13th International Conference on Software Engineering - ICSE '08*, pages 241–250, New York, NY, USA, 2008. ACM.

[27] R. DeLine and M. Fähndrich. Typestates for Objects. In *Lecture Notes in Computer Science*, pages 465–490. Springer-Verlag, 2004.

[28] D. Dig and R. Johnson. The role of refactorings in API evolution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 389–398. IEEE, 2005.

[29] Torbjörn Ekman and Görel Hedin. The JastAdd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[30] Michael D. Ernst. JSR-308 type annotations specification. http://types.cs.washington.edu/jsr308/specification/java-annotation-design.pdf Retrieved 15 december 2011.

[31] J Field, D Goyal, G Ramalingam, and E Yahav. Typestate verification: Abstraction techniques and complexity results. *Elsevier Science of Computer Programming*, 58(1-2):57–82, 2005.

[32] Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology*, 17(2), 2008.

[33] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269, London, UK, 1999. Springer-Verlag.

[34] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1 edition, 1999.

[35] Mark Gabel and Zhendong Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 339–349, New York, NY, USA, 2008. ACM.

[36] Mark Gabel and Zhendong Su. Online inference and enforcement of temporal properties. In *ICSE*, 2010.

[37] Erich Gamma, Richard Helm, John M. Vlissides, and Ralph Johnson. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[38] Jacques Garrigue. *Label-Selective Lambda Calculi and Transformation Calculi*. PhD thesis, University of Tokyo, December 1994.

[39] Jacques Garrigue and Hassan Aït-Kaci. The typed polymorphic label-selective λ-calculus. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 35–47, New York, NY, USA, 1994. ACM.

[40] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory & Practice (The Morgan Kaufmann Series in Artificial Intelligence)*. Morgan Kaufmann, 1 edition, 2004.

[41] Jeremy Gibbons. An introduction to the Bird-Meertens formalism. In Steve Reeves, editor, *Proceedings of the First New Zealand Formal Program Development Colloquium*, pages 1–12, Hamilton, nov 1994.

[42] Daniel Gibson, John Glass, Carole Lartigue, Vladimir Noskov, Ray-Yuan Chuang, Mikkel Algire, Gwynedd Benders, Michael Montague, Li Ma, Monzia Moodie, Chuck Merryman, Sanjay Vashee, Radha Krishnakumar, Nacyra Assad-Garcia, Cynthia Andrews-Pfannkoch, Evgeniya Denisova, Lei Young, Zhi-Qing Qi, Thomas Segall-Shapiro, Christopher Calvey, Prashanth Parmar, Clyde Hutchison, Hamilton Smith, and Craig Venter. Creation of a bacterial cell controlled by a chemically synthesized genome. *Science*, 329(5987):52–56, 2010.

[43] David Gifford, Pierre Jouvelot, J.M. Lucassen, and M.A. Sheldon. FX-87 reference manual. Technical Report MIT/LCS/TR-407, Massachusetts Institute of Technology, September 1987.

[44] T Gojobori, EN Moriyama, and M Kimura. Molecular clock of viral evolution, and the neutral theory. *Proceedings of the National Academy of Sciences of the United States of America*, 87(24):10015–10018, 1990.

[45] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

[46] Aaron Greenhouse and John Boyland. An object-oriented effect system. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming (LNCS 1628)*, 1999.

[47] The Object Management Group. The common object request broker: Architecture and specification (rev. 2.3), June 1999.

[48] Christian Haack, Brian Howard, Allen Stoughton, and Joe B Wells. Fully automatic adaptation of software components based on semantic specifications. In *AMAST '02: Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, pages 83–98, 2002.

[49] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[50] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[51] John Hogg. Islands: aliasing protection in object-oriented languages. *SIGPLAN Not.*, 26:271–285, November 1991.

[52] IBM. Jikes java compiler, 2011. http://jikes.sourceforge.net Retrieved 9 December 2011.

[53] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and GJ. *ACM Trans. Program. Lang. Syst.*, 23:396–450, May 2001.

[54] Apple inc. The Objective-C programming language, 10 2011. http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf Retrieved 14 December 2011.

[55] INRIA. Spoon java program processor. http://spoon.gforge.inria.fr/Spoon/HomePage Retrieved 9 December 2011.

[56] ECMA International. Standard ECMA-367: Eiffel: Analysis, design and programming language. http://www.ecma-international.org/publications/standards/Ecma-367.htm Retrived 14 December 2011.

[57] A. Ireland and J. Stark. Combining proof plans with partial order planning for imperative program synthesis. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, volume 13, pages 65–105. Springer, 2006.

[58] Ciera Jaspan and Jonathan Aldrich. Checking framework interactions with relationships. In *ECOOP*, 2009.

[59] JFree. JFreeChart. http://www.jfree.org/jfreechart Retrieved on 18 Nov 2011.

[60] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010.

[61] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 303–310, New York, NY, USA, 1991. ACM.

[62] Stephen Kell. Rethinking software connectors. In *International workshop on Synthesis and analysis of component connectors in conjunction with the 6th ESEC/FSE joint meeting - SYANCO '07*, pages 1–12, New York, NY, USA, 2007. ACM.

[63] Stephen Kell. The mythical matched modules. In *OOPSLA '09: Proceedings of the 24th annual ACM SIGPLAN Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 881–888, New York, NY, USA, 2009. ACM.

[64] G.J. Kiczales, John Lamping, C.V. Lopes, J.J. Hugunin, E.A. Hilsdale, and C. Boyapati. Aspect-oriented programming. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming (LNCS 1241)*, number June, pages 220–242. Springer-Verlag, October 1997.

[65] Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking using set interfaces and pluggable analyses. In *ACM SIGPLAN notices*, 2004.

[66] Sven Lämmerman. *Runtime Service Composition via Logic-Based Program Synthesis*. PhD thesis, KTH, 2002.

[67] Gary T. Leavens and Yoonsik Cheon. Design by contract with JML, 2004.

[68] Henry Ledgard. *ADA: An Introduction, Ada Reference Manual*. Springer-Verlag, Berlin, 1980.

[69] K. Rustan M. Leino, John Hatcliff, Gary Leavens, M.J. Parkinson, and Peter Müller. Behavioral interface specification languages. Technical Report CS-TR-09-01a, University of Central Florida, 2010.

[70] K. Rustan M. Leino and K. Rustan M." Leino. Data groups: Specifying the modification of extended state. 1998.

[71] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *PLDI*, 2002.

[72] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[73] Barbara H. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 1994.

[74] Michael Lowry, Michael Lowry, Andrew Philpot, Thomas Pressburger, and Ian" Underwood. AMPHION: Automatic programming for scientific subroutine libraries. *INTL. SYMP. ON METHODOLOGIES FOR INTELLIGENT SYSTEMS*, 31:326–335, 1994.

[75] David Mandelin, Lin Xu, Doug Kimelman, and Ratislav Bodík. Jungloid mining: Helping to navigate the API jungle. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 48–61, New York, NY, USA, 2005. ACM.

[76] Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis. *IEEE Trans. Softw. Eng.*, 18(8):674–704, 1992.

[77] David McAllester and David Rosenblitt. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 634–639. AAAI, 1991.

[78] M. D. McIlroy. Mass-produced software components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.

[79] Bertrand Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.

[80] Naftaly Minsky and Naftaly" Minsky. Towards alias-free pointers. *IN EUROPEAN CONFERENCE FOR OBJECT-ORIENTED PROGRAMMING (ECOOP*, 1098:189–209, 1996.

[81] Todd Moon. *Error Correction Coding: Mathematical Methods and Algorithms*. Wiley-Interscience, 2005.

[82] Nomair a. Naeem and Ondřej Lhoták. Typestate-like analysis of multiple interacting objects. In *OOPSLA '08: Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming: Systems, Languages and Applications*, volume 43, New York, NY, USA, October 2008. ACM.

[83] XuanLong Nguyen and Subbarao Kambhampati. Reviving partial order planning. In *Proceedings of the 17th international joint conference on Artificial intelligence - Volume 1*, pages 459–464, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[84] Flemming Nielson, Hanne Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, corrected edition, 2004.

[85] O. Nierstrasz. A survey of object-oriented concepts. *Object-Oriented Concepts, Databases and Applications*, pages 3–21, 1989.

[86] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: an extensible compiler framework for java. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 138–152, Berlin, Heidelberg, 2003. Springer-Verlag.

[87] Johan Nyström-Persson and Shinichi Honiden. Poplar: Java composition with labels and AI planning. In *FREECO '11: The Workshop on Free Composition at Onward! 2011*, 2011.

[88] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *Symposium on Principles of Programming Languages*, pages 75–86, 2008.

[89] M.J. Parkinson. When separation logic met java (by example). In *FTfJP*, 2006.

[90] Terence Parr. ANTLR, 2011. http://www.antlr.org Retrieved on 21 November 2011.

[91] David J. Pearce. JKit documentation, July 2010. http://whiley.org/tools/jkit/ Retrieved 21 November 2011.

[92] David J. Pearce. JKit, 2011. http://homepages.ecs.vuw.ac.nz/~djp/jkit/ Retrieved 21 November 2011.

[93] David J. Pearce. JPure: A modular purity system for java. In *CC: International Conference on Compiler Construction*, 2011.

[94] Benjamin Pierce. *Types and Programming Languages*. The MIT Press, 1 edition, 2002.

[95] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. ACM.

[96] Michael Pradel and Thomas R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 371–382, Washington, DC, USA, 2009. IEEE Computer Society.

[97] Steven P. Reiss. Semantics-based code search. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 243–253. IEEE, May 2009.

[98] Martin Rinard and Alexandru D. Salcianu. Purity and side effect analysis for java programs. In *VMCAI*, 2005.

[99] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004.

[100] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (3rd Edition)*. Prentice Hall, 3 edition, 2009.

[101] Alexandru D. Salcianu. *Pointer analysis for java programs: novel techniques and applications*. PhD thesis, Cambridge, MA, USA, 2006. AAI0818179.

[102] J. Schumann and B. Fischer. NORA/HAMMR: making deduction-based software component retrieval practical. In *Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*, ASE '97, pages 246–, Washington, DC, USA, 1997. IEEE Computer Society.

[103] M. Shaw. Procedure calls are the assembly language of software interconnection. In *Proceedings of the Workshop on Studies of Software Design*. Springer, 1993.

[104] Douglas R. Smith. KIDS: A semiautomatic program development system. *IEEE Trans. Softw. Eng.*, 1990.

[105] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. *SIGPLAN Not.*, 45:313–326, January 2010.

[106] Guy Steele. *Common LISP : The Language (LISP Series)*. Digital Press, 1984.

[107] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[108] Mark Stickel, Mark Stickel, Richard Waldinger, Michael Lowry, Thomas Pressburger, and Ian Underwood. Deductive composition of astronomical software from subroutine libraries. *Proceedings of the 12th International Conference on Automated Deduction*, 814:341–355, 1994.

[109] Rok Strniša, Peter Sewell, and Matthew Parkinson. The java module system: core design and semantic definition. *SIGPLAN Not.*, 42:499–514, October 2007.

[110] RE Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[111] Alexander J. Summers and Peter Mueller. Freedom before commitment: a lightweight type system for object initialisation. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 1013–1032, New York, NY, USA, 2011. ACM.

[112] Sun Microsystems. JavaBeans API specification. http://java.sun.com/products/javabeans.

[113] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in plaid. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 713–732, New York, NY, USA, 2011. ACM.

[114] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming (2nd Edition)*. Addison-Wesley Professional, 2 edition, 2002.

[115] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 1992.

[116] The OSGi Alliance. OSGI service platform release 4 version 4.2 core specification. http://www.osgi.org/Download/Release4V42.

[117] TIOBE. TIOBE programming community index, November 2011. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html Retrieved 21 November 2011.

[118] Franklyn Turbak and David Gifford. *Design Concepts in Programming Languages*. The MIT Press, 2008.

[119] Enn Tyugu. Using classes as specifications for automatic construction of programs in the NUT system. 1994.

[120] R. Vasa, M. Lumpe, and J. Schneider. Patterns of component evolution. In *Lecture Notes in Computer Science*, volume 4829, pages 235–251. Springer-Verlag, 2007.

[121] Andrzej Wasowski. On efficient program synthesis from statecharts. *SIGPLAN Not.*, 38:163–170, June 2003.

[122] Sun World. An interview with the creators of java, July 1995. http://sunsite.uakom.sk/sunworldonline/swol-07-1995/swol-07-java.html Retrieved 21 November 2011.

[123] Zhenchang Xing and Eleni Stroulia. API-evolution support with Diff-CatchUp. *IEEE Transactions on Software Engineering*, 33(12):818–836, December 2007.

[124] Zhenchang Xing and Eleni Stroulia. Differencing logical UML models. *Automated Software Eng.*, 14:215–259, June 2007.

[125] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, March 1997.

[126] A.M. Zaremski. *Signature and specification matching*. PhD thesis, Mass. Inst. Tech, 1996.

[127] A.M. Zaremski and J. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 1997.

[128] Mattias Zenger. Type-safe prototype-based component evolution.

[129] Mattias Zenger. *Programming Language Abstractions for Exensible Software Components*. PhD thesis, Ècole Fédérale Polytechnique de Lausanne (EPFL), Switzerland, 2004.

[130] Mattias Zenger. Keris: Evolving software with extensible modules. *Journal of Software Maintenance and Evolution: Research and Practice*, 2005.

[131] Mattias Zenger. JaCo java compiler (the programming language Keris), 2007. http://lampwww.epfl.ch/~zenger/keris. Retrieved 9 December 2011.

[132] Mattias Zenger, Tom Mens, Jim Buckley, and Awais Rashid. Towards a taxonomy of software evolution. In *Intl Workshop on Unanticipated Software Evolution (USE)*, 2003.

[133] Tian Zhao, Jens Palsberg, and Jan Vitek. Lightweight confinement for featherweight java. *SIGPLAN Not.*, 38:135–148, October 2003.

[134] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and immutability in generic java. In *OOPSLA '10: Proceedings of the 25th annual ACM SIGPLAN Conference on Object-Oriented Programming: Systems, Languages and Applications*, 2010.

Index