Mostly-Static Decentralized Information Flow Control

Andrew C. Myers

January 1999

© Massachusetts Institute of Technology 1999

This research was supported in part by DARPA contract F00014-91-J-4136, monitored by the Office of Naval Research, and by DARPA contracts F30602-96-C-0303 and F30602-98-1-0237, monitored by USAF Rome Laboratory. The author was supported by an Intel Foundation Fellowship.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts, USA

Mostly-Static Decentralized Information Flow Control

by

Andrew C. Myers

Abstract

The growing use of mobile code in downloaded programs such as applets and servlets has increased interest in robust mechanisms for ensuring privacy and secrecy. Common security mechanisms such as sandboxing and access control are either too restrictive or too weak—they prevent applications from sharing data usefully, or allow private information to leak. For example, security mechanisms in Java prevent many useful applications while still permitting Trojan horse applets to leak private information. This thesis describes the *decentralized label model*, a new model of information flow control that protects private data while allowing applications to share data. Unlike previous approaches to privacy protection based on information flow, this label model is *decentralized*: it allows cooperative computation by mutually distrusting principals, without mediation by highly trusted agents. Cooperative computation is possible because individual principals can *declassify* their own data without infringing on other principals' privacy. The decentralized label model permits programs using it to be checked statically, which is important for the precise detection of information leaks.

This thesis also presents the new language *JFlow*, an extension to the Java programming language that incorporates the decentralized label model and permits static checking of information flows within programs. Variable declarations in JFlow programs are annotated with labels that allow the static checker to check programs for information leaks efficiently, in a manner similar to type checking. Often, these labels can be inferred automatically, so annotating programs is not onerous. Dynamic checks also may be used safely when static checks are insufficiently powerful. A compiler has been implemented for the JFlow language. Because most checking is performed statically at compile time, the impact on performance is usually small.

Keywords: constraint solving, covert channels, integrity, Java, labels, principals, privacy, programming languages, role hierarchy, security, static checking, trojan horse, trusted computing base, type systems, verification

This report is a minor revision of the dissertation of the same title submitted to the Department of Electrical Engineering and Computer Science on January 7, 1999, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in that department. The thesis was supervised by Professor Barbara Liskov.

Acknowledgments

Many people deserve thanks for helping me make this work, my degree, and my love of research, a reality.

First, I must thank my advisor, Barbara Liskov, for her continuous support and the inspiration she has provided me. If I am a better writer or speaker because of my time at MIT, it is largely due to her. I hope that someday I will approach her clarity of thought and expression, and her ability to focus on the important.

My thesis committee provided many excellent suggestions on both the form and content of this work. Martín Abadi has given thoughtful and extremely useful comments at several phases of this work; I greatly appreciate his support and his meticulous reading of this thesis. John Guttag suggested many useful clarifications. Butler Lampson had several insightful suggestions as I was developing this work and helped me figure out how to make some of the central ideas of this thesis more intuitive.

Many other people also have provided useful feedback on this thesis and on the work as a whole. Chandrasekhar Boyapati, Miguel Castro, and Stephen Garland deserve special thanks for their in-depth readings of my work and helpful suggestions over the last couple of years. Nick Mathewson provided much useful code that I was able to borrow.

It has been a pleasure to be a graduate student in the Programming Methodology Group, and I would like to thank the group members for creating such a lively and thought-provoking environment. Atul Adya, Joseph Bank, Phillip Bogle, Chandra, Miguel, Mark Day, Sanjay Ghemawat, Jason Hunter, Umesh Maheshwari, Quinton Zondervan, and many others made life richer, not only because of their considerable abilities, but also because they were willing to make movie runs, share dinners, suggest good books, play ping-pong, and swap evil puzzles.

Other friends have helped over the years. Ulana Legedza has always been there to help refine talks and papers. In addition to his useful feedback on all of my work, Jim O'Toole was a stimulating housemate in my early years at MIT. Joe Boyt and Paul Isherwood often flew into town and enlivened my graduate life.

Many other people have made me a better researcher and computer scientist. Greg Nelson of SRC made me appreciate the value of a well-written specification. I learned about user interfaces and good design from Rob Myers of Silicon Graphics, and Brendan Eich, Kipp Hickman, Bruce Karsh, and others from SGI made me a better programmer and collaborator. I would also like to thank Franklyn Prendergast, S. Vuk-Pavlovic, and Zeljko Bajzer of the Mayo Clinic, who introduced me to research and helped me realize that I wanted to become a scientist.

My parents and the rest of my family have been incredibly supportive over the years of my quest for a doctoral degree. My parents always have pushed me to achieve to the limits of my ability and have done what they could to ensure that I had the best education possible. I owe them much gratitude.

Finally, my greatest asset for the last three years has been my wife, Kavita Bala. She is always emotionally supportive, but she is also my best reviewer and the person I first bounce ideas off of. Even while she worked to complete her own doctoral degree, she helped me unstintingly in every conceivable way, right from the start in a Barcelona park where she helped nurture an offbeat idea into my thesis topic.

Contents

1 Introduction **11**
- 1.1 Example . . . 12
- 1.2 Existing security techniques . . . 14
- 1.3 Decentralized information flow control . . . 15
  - 1.3.1 Decentralized label model . . . 15
  - 1.3.2 Static information flow analysis . . . 16
- 1.4 Trusted computing base . . . 18
- 1.5 Applications . . . 19
- 1.6 Limitations . . . 19
- 1.7 Outline . . . 20

2 The Label Model **21**
- 2.1 Basic model . . . 22
  - 2.1.1 Principals . . . 22
  - 2.1.2 Labels . . . 22
  - 2.1.3 Relabeling by restriction . . . 23
  - 2.1.4 Computation and label join . . . 24
  - 2.1.5 Relabeling by declassification . . . 25
  - 2.1.6 Channels . . . 26
- 2.2 Examples . . . 27
  - 2.2.1 Tax preparer example . . . 27
  - 2.2.2 Hospital example . . . 28
- 2.3 Extending and interpreting labels . . . 30
  - 2.3.1 Limitations of the subset relabeling rule . . . 30
  - 2.3.2 Interpreting labels . . . 32
  - 2.3.3 Formalizing the principal hierarchy . . . 33
  - 2.3.4 Label interpretation function . . . 33
  - 2.3.5 Flow set constraints . . . 34
  - 2.3.6 Label functions . . . 35
- 2.4 Checking relabeling statically . . . 37
  - 2.4.1 Annotations . . . 38
  - 2.4.2 Static correctness condition . . . 39
  - 2.4.3 A sound and complete relabeling rule . . . 40
  - 2.4.4 Static checking . . . 44
- 2.5 Output channels . . . 50
- 2.6 Generalizing labels and principals . . . 51
  - 2.6.1 Integrity policies . . . 51

2.6.2 Combining integrity and privacy . . . 54
2.6.3 Generalizing principals and the acts-for relation . . . 55
2.7 Summary . . . 59

3 The JFlow Language 60
3.1 Static vs. dynamic checking . . . 62
3.2 Language support for information flow checking . . . 63
3.2.1 Labeled types . . . 63
3.2.2 Implicit flows . . . 64
3.2.3 Termination channels . . . 65
3.2.4 Run-time labels . . . 66
3.2.5 Reasoning about principals . . . 68
3.2.6 Declassification . . . 69
3.2.7 Run-time principals . . . 70
3.3 Interactions with features of Java . . . 70
3.3.1 Method declarations . . . 71
3.3.2 Default labels . . . 72
3.3.3 Method constraints . . . 73
3.3.4 Exceptions . . . 75
3.3.5 Parameterized classes . . . 75
3.3.6 Arrays . . . 78
3.3.7 Run-time type discrimination . . . 79
3.3.8 Authority declarations . . . 79
3.3.9 Inheritance and constructors . . . 80
3.4 Examples . . . 82
3.4.1 Example: passwordFile . . . 82
3.4.2 Example: Protected . . . 84
3.5 Limitations . . . 84
3.6 Grammar extensions . . . 85
3.6.1 Label expressions . . . 85
3.6.2 Labeled types . . . 86
3.6.3 Class declarations . . . 86
3.6.4 Method declarations . . . 87
3.6.5 New statements . . . 88
3.6.6 New expressions . . . 88

4 Statically Checking JFlow 90
4.1 Correctness . . . 90
4.2 Static checking framework . . . 91
4.2.1 Type checking vs. label checking . . . 92
4.2.2 Environments . . . 92
4.2.3 Exceptions . . . 93
4.2.4 Additional notation conventions . . . 96
4.2.5 Environment bindings . . . 97
4.2.6 Representing principals . . . 98
4.2.7 Representing labels and components . . . 98
4.2.8 Representing types . . . 100
4.2.9 Invariant vs. covariant types . . . 100

4.3 Basic rules . . . 102
4.3.1 Reasoning about principals . . . 102
4.3.2 Reasoning about labels . . . 102
4.3.3 Class scope and environments . . . 103
4.3.4 Reasoning about subtypes . . . 106
4.4 Checking Java statements and expressions . . . 108
4.4.1 Simple rules . . . 108
4.4.2 Arithmetic . . . 109
4.4.3 Local variables . . . 110
4.4.4 Variable access . . . 110
4.4.5 Variable assignment . . . 112
4.4.6 Compound statements . . . 113
4.4.7 Goto-like statements . . . 114
4.4.8 Exceptions . . . 115
4.4.9 Dynamic type discrimination . . . 116
4.5 Checking new statements and expressions . . . 117
4.5.1 Testing the principal hierarchy . . . 117
4.5.2 Declassification . . . 118
4.5.3 Run-time label tests . . . 119
4.6 Method and constructor calls . . . 120
4.6.1 Generic checking . . . 120
4.6.2 Specific rules for checking calls . . . 122
4.7 Checking classes and methods . . . 123
4.7.1 Checking classes . . . 123
4.7.2 Class authority . . . 125
4.7.3 Method signature compatibility . . . 125
4.7.4 Method declarations . . . 127

5 Constraint Solving and Translation 133
5.1 Constraint solving . . . 133
5.1.1 Integrating static checking and constraint solving . . . 134
5.1.2 Constraint equations . . . 134
5.1.3 Solving constraints . . . 136
5.1.4 Determining the meet of two components . . . 137
5.1.5 Handling dynamic constraints . . . 138
5.1.6 Recursion in dynamic components . . . 140
5.1.7 Ordering the relaxation steps . . . 141
5.1.8 Empirical comparisons . . . 144
5.2 Translation . . . 146
5.2.1 Principal values and the actsFor statement . . . 146
5.2.2 Label values and the switch label statement . . . 148

6 Related Work 149
6.1 Access control . . . 149
6.2 Limitations of discretionary access control . . . 150
6.3 Information flow control . . . 151
6.4 Static enforcement of security policies . . . 155
6.5 Modeling principals and roles . . . 156

6.6 Cryptography 157
6.7 Covert channels 157

7 Conclusions **159**
7.1 Decentralized label model 159
7.2 Static analysis of information flow 160
7.3 Future work 161

Chapter 1

Introduction

Computer security is becoming increasingly important, as the result of several ongoing trends. Computers everywhere are becoming inextricably connected to the Internet. Increasingly, computation and even data storage are distributed to geographically remote and untrusted sites, and both programs and data are becoming highly mobile. Sensitive personal, corporate, and government data is being placed online and is routinely accessed over networks. The number of users and other interacting entities also continues to increase rapidly, and trust relationships among these entities are growing increasingly complex. In short, there is more to protect and it is more difficult to protect it.

It is difficult even to characterize what protection is needed. Abstractly, the goal of computer security is to ensure that all computations obey some set of policies, but there are two central goals of computer security: private or secret data should not be leaked to parties that might misuse it, and valuable data should not be damaged or destroyed by other parties. These complementary goals will be referred to here as *privacy* and *integrity*. This thesis focuses on the protection of privacy, though integrity is also considered briefly. Protecting privacy and secrecy of data has long been known to be a very difficult problem, and existing security techniques do not provide satisfactory solutions to this problem.

Systems that support the downloading of distrusted code are particularly in need of better protection for privacy. For example, Java [GJS96] supports downloading of code from remote sites, creating the possibility that the downloaded code will transfer private data to those sites. Suppose a user computes his taxes using a downloaded applet. The user cannot ensure that the applet will not transfer his tax information back to the applet provider. Java attempts to prevent improper transfers by using a compartmental security model called the *sandbox model* [FM96, MF96], but this approach largely prevents applications from sharing data, while still permitting privacy violations like the one just described. A key problem is that information must be shared with downloaded code, while preventing that code from leaking the information.

There is no generally accepted definition of what it means to protect privacy. A distinction sometimes has been drawn between privacy and other security goals such as *secrecy* or *confidentiality*. Sometimes privacy is identified with the weaker goal of *anonymity*: protecting the identity of various parties, as in a medical protocol, rather than their data, as in [Swe96]. However, in this work the terms privacy and secrecy

are considered to be synonymous; they both refer to the ability to control information leakage of any kind. The use of the term privacy emphasizes that in a decentralized environment, no generally accepted notion of the sensitivity of data exists. Users generally consider their own data to be private, and are naturally less concerned with the privacy of the data of other users. However, the privacy requirements of all users are treated as equally important.

In general, security enforcement mechanisms may be internal or external to the computing system. Internal mechanisms attempt to prevent security violations by making them impossible; external mechanisms, such as the threat of legal action, attempt to convince users not to initiate computation that would violate security. Current security mechanisms, both internal and external, are becoming less viable as the computing system becomes large, decentralized, anonymous, and international.

With the widespread downloading of code, dealing with untrusted programs becomes a greater issue for security than in the past. Conventionally, the focus is placed on protecting the operating system from buggy or malicious programs, and on protecting users from each other. On most computer systems, the programs that might be used to violate user privacy are programs already installed on the system, and purchased from some vendor. Since the source of the program is known, some form of external redress is available if the program is found to violate privacy. When programs such as Java applets are dynamically downloaded and executed, the ability to identify and exact redress from the supplier of privacy-violating code is reduced. Therefore, the goal of this work is to develop better internal mechanisms, preventing *programs* from violating security policies rather than convincing *users* not to.

In another sense, the goal of this work is to reduce the cost of ensuring security—a cost that is passed on to users. If a user downloads a free application, the user accepts either the risk that a program will violate security, or the considerable cost of ensuring that a program does not violate security. This observation applies to commercial software as well; a company providing an application must ensure that it does not violate user security, or else be liable in cases where it violates security, at least in the sense that the reputation of the company may suffer. With both kinds of software, the cost is passed on to the users of that application. Better internal mechanisms that can be applied either by end-users or by software developers should reduce this cost.

1.1 Example

Figure 1.1 depicts an example with security requirements that cannot be satisfied using existing techniques. This scenario contains mutually distrusting principals that must cooperate to perform useful work. In the example, the user Bob is preparing his tax form using both a spreadsheet program and a piece of software called “WebTax”. Bob would like to be able to prepare his final tax form using WebTax, but he does not trust WebTax to protect his privacy. The computation is being performed using two programs: a spreadsheet that he trusts and grants his full authority to, and the WebTax program, which he does not trust. Bob would like to transmit his tax data from the spreadsheet to WebTax and receive a final tax form as a result, while

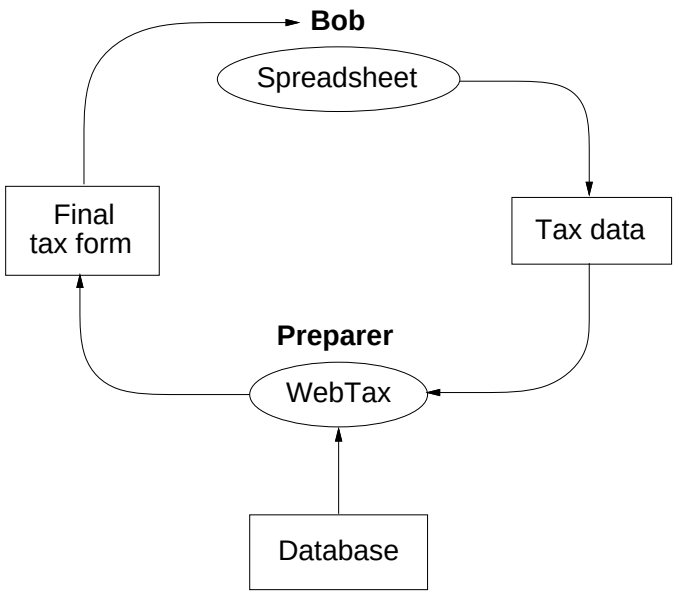

Figure 1.1: A simple example

being protected against WebTax leaking his tax information.

In this example, there is another principal named Preparer that has privacy interests. The principal Preparer represents a firm that distributes the WebTax software. The WebTax application computes the final tax form using a proprietary database, shown at the bottom, that is owned by Preparer. This database might, for example, contain algorithms for minimizing tax payments. Since this principal is the source of the WebTax software, it trusts the program not to distribute the proprietary database through malicious action, though the program might leak information because it contains bugs.

In principle, it may be difficult to prevent some information about the database contents from leaking back to Bob, particularly if Bob is able to make a large number of requests and then carefully analyze the resulting tax forms. This information leak is not a practical problem if Preparer can charge Bob a per-form fee that exceeds the value of the information Bob obtains through each form.

To make this scenario work, the Preparer principal needs two pieces of functionality. First, it needs protection against accidental or malicious release of information from the database by paths other than through the final tax form. Second, it needs the ability to *sign off* on the final tax form, confirming that the information leaked in the final tax form is sufficiently small or scrambled by computation that the tax form may be released to Bob.

It is worth noting that Bob and Preparer do need to trust that the execution platform has not been subverted. For example, if WebTax is running on a computer that Bob completely controls, then Bob will be able to steal the proprietary database. Clearly, Preparer cannot have any real expectation of privacy or secrecy if its private data is manipulated in unencrypted form by an execution platform that it does not trust!

In this thesis, it is assumed that the execution platform is trusted, even though the programs running on

that platform may not be. The issue of trust in the execution platform is discussed further in Section 1.4. Even with this assumption, this scenario cannot be implemented satisfactorily or even modeled using existing security techniques. With current techniques, Bob must carefully inspect the Webtax code and verify that it does not leak his data; in general, this task is difficult. The techniques described in this thesis allow the security goals of both Bob and Preparer to be met without this inspection; Bob and Preparer then can cooperate in performing useful computation. In another sense, this work shows how both Bob and Preparer can inspect the Webtax program efficiently and simply to determine whether it violates their security requirements.

1.2 Existing security techniques

Let us now briefly consider the application of existing security techniques to this problem; for a more in-depth discussion, see Chapter 6. When most people think of computer security, they think of well-established security techniques such as access control. Typical access control mechanisms (which support *discretionary access control*) do not protect privacy well when programs are not trusted: access control prevents unauthorized information release but does not control information propagation once the information has been accessed. For example, if a program A is allowed to read user B's data, B cannot control how A distributes the information it has read.

A less well-known approach to protecting privacy is *information flow control*. In information flow techniques (such as *mandatory access control*), every piece of data has an attached *sensitivity label*. The labels are typically from a small ordered set such as $\{\mathsf{unclassified}, \mathsf{classified}, \mathsf{secret}, \mathsf{top\ secret}\}$. The labels remain attached to data as it propagates through the system, preventing it from being released improperly even if it is released to an untrusted program. Data may be *relabeled* to further restrict its use (such as a relabeling from secret to top secret). However, relabeling data from top secret to secret (or allowing top secret data to affect secret data) would be *declassification* or *downgrading*, which could lead to an information leak.

Intuitively, information flow control protects privacy much more directly than access control does, but practical problems with information flow control have prevented its widespread adoption. Sensitivity labels are usually maintained dynamically, causing substantial loss of performance. Dynamic labels impose even greater run-time and storage overheads than access control mechanisms do, because for every primitive operation, the label of the result must be computed. Another limitation is that sensitivity labels are implicitly *centralized*: they express the privacy concerns of a single principal (typically, the government). If one considers providing privacy in a more decentralized setting, such as the community of Web users, it is clear that no universal notion of secret sensitivity can be established.

All practical information flow control systems provide the ability to *declassify* or *downgrade* data because strict information flow control is too restrictive for writing real applications. Declassification in these systems lies outside the model: it performed by a *trusted subject*, which is code possessing the authority of a

highly trusted principal. However, the notion of a highly trusted principal does not extend to a decentralized system. Traditional information flow models do not support workable declassification for a decentralized environment.

Another important issue for information flow systems is the *precision* of the detection of information flow. Information is assumed to flow from one program value to another if there is any dependency between the values. Any unidentified dependency would create a potential information leak. However, it is also important to avoid false dependencies, since a false dependency results in data being overly restrictively labeled, and thus not usable in situations where it ought to be. To provide a precise determination of data dependencies, particularly dependencies arising from *implicit flows*, static analysis is required [DD77]. Dynamic enforcement of information flow control, as in mandatory access control systems [DOD85], can determine data dependencies conservatively—even dependencies arising from implicit flows—but results in false dependencies and overly restrictive labels.

1.3 Decentralized information flow control

The central goal of this work is to make information flow control a viable technique for providing privacy in a complex, decentralized world with mutually distrusting principals. This work has involved two major components, each of which is independently useful.

1.3.1 Decentralized label model

The first component is the development of a new model for labeling data that supports situations involving mutual distrust. This model allows users to control the flow of their information without imposing the rigid constraints of a traditional multilevel security system. It provides security guarantees to users and to groups rather than to a monolithic organization—in essence, it provides every principal with its own multilevel security.

The decentralized information flow model differs from previous work on information flow control: it introduces a notion of ownership of data, and allows users to explicitly declassify data that they own. When data is derived from several sources, all the sources own the data and must agree to release it. Previous work on information flow allowed declassification only by a *trusted agent* or *trusted subject* with essentially arbitrary powers of declassification; the notion of a universally trusted agent is clearly inapplicable to a decentralized environment. Declassification in this model provides a safe escape hatch from the rigid restrictions of strict information flow checking. Deciding when declassification is appropriate is outside the scope of this model; work in *inference controls* and *statistical databases* has developed some applicable methods [Den82].

The decentralized label model has a number of important properties that are discussed further in Chapter 2:

- It allows individual principals to attach flow policies to pieces of data. The flow policies of all principals are reflected in the *label* of the data, and the system guarantees that all the policies are obeyed simultaneously. Therefore, the model works even when the principals do not trust each other.
- The model allows a principal to declassify data by modifying the flow policies in the attached label. Arbitrary declassification is not possible because flow policies of other principals are still maintained. Declassification permits the programmer to remove restrictions when appropriate; for example, the programmer might determine that the amount of his information being leaked is acceptable using techniques from information theory [Mil87].
- The model is compatible with static checking of information flow.
- It allows a richer set of safe relabelings than in previous label models [Den76, MMN90] by fully exploiting information about relationships between different principals.
- It has a formal semantics that allows a precise characterization of what relabelings are legal.
- The rule for static checking is shown to be both sound and complete with respect to the formal semantics: the rule allows only safe relabelings, and it allows all safe relabelings.
- In this model, labels form a lattice-like structure that helps make static checking of programs effective.
- The model can be applied in dual form to yield decentralized integrity policies.

1.3.2 Static information flow analysis

The second component of this work is a collection of new techniques for static analysis of information flow in programs. These techniques have been incorporated in the new language *JFlow*, an extension of the Java language [GJS96] that allows information in the program to be annotated with decentralized labels. These annotations can then be checked statically, allowing more precise, fine-grained determination of information flows within programs than in previous languages allowing static checking of information flow. Like other recent approaches [PO95, VSI96, ML97, SV98, HR98, Mye99], JFlow treats static checking of flow annotations (*label checking*) as an extended form of type checking. Programs written in JFlow can be checked statically by the JFlow compiler, which detects any information leaks through covert storage channels. JFlow is intended to support the writing of secure servers and applets that manipulate sensitive data.

An important philosophical difference between JFlow and other work on statically checking information flow is the focus on a usable programming model, avoiding the unnecessary restrictiveness of earlier systems for static flow analysis. JFlow provides a more practical programming model than earlier work does. The goal of this work is to add enough power to the static checking framework to allow reasonable programs to be written in a natural manner.

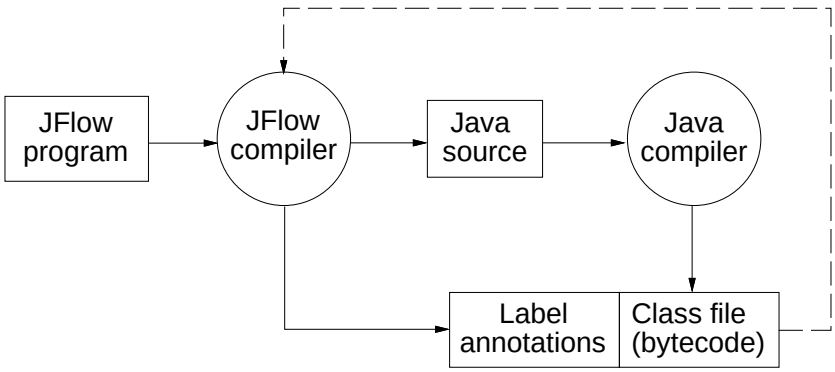

Figure 1.2: JFlow compiler

Adding this power has required several new contributions. Because JFlow extends a complex programming language, it supports many language features that have not been integrated previously with static flow checking, including mutable objects (which are more complex than function values), subclassing, dynamic type tests, access control, and exceptions.

JFlow also provides powerful new features that make information flow checking less restrictive and more convenient than in previous models:

- *Label polymorphism* allows the writing of code that is generic with respect to the security class of the data it manipulates.
- Run-time label checking and first-class label values create a dynamic escape in cases where static checking is too restrictive. Run-time checks are statically checked to ensure that information is not leaked by the success or failure of the run-time check itself.
- Automatic label inference makes it unnecessary to write many of the annotations that would be required otherwise.
- A statically-checked declassification operator allows safe declassification as described by the decentralized label model.

The JFlow compiler is structured as a source-to-source translator; its output is a standard Java program that can be compiled by any Java compiler. The operation of the compiler is depicted in Figure 1.2. The input to the compiler is the text of a JFlow program and the compiled bytecode for any external program modules used by the program. This model of compilation is exactly that of Java. Using this information, the compiler checks JFlow programs and translates them into an equivalent Java program, which is converted to executable form by a standard Java compiler. In addition, the JFlow compiler generates an auxiliary file containing information about label annotations found within the program. This auxiliary file is used in conjunction with the compiled bytecode file whenever this program is used as an external module for the purpose of compiling other code that depends on it, as shown by the dashed arrow.

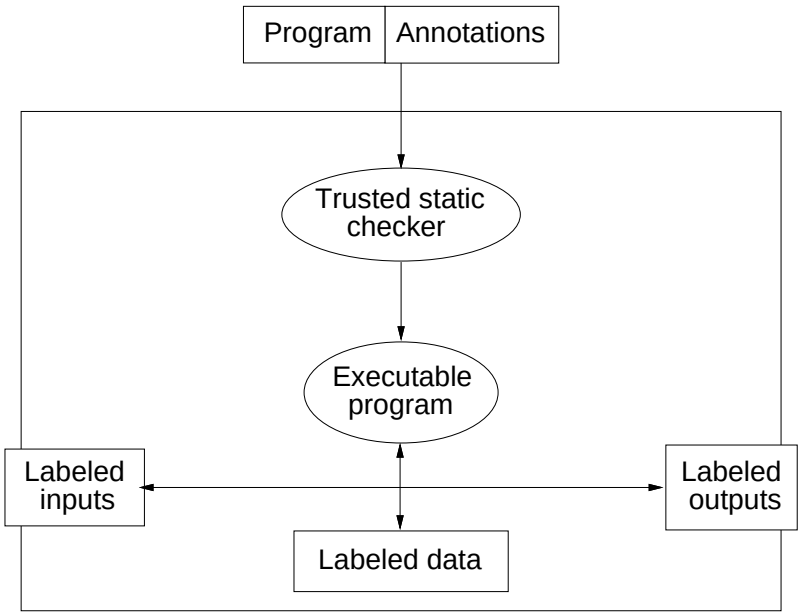

Figure 1.3: Trusted execution platform

For the most part, translation involves removal of the static annotations in the JFlow program (after checking them, of course). For this reason, there is little code space, data space, or run time overhead, because most checking is performed statically.

1.4 Trusted computing base

An important aspect of any security mechanism is the identification of the *trusted computing base* (TCB): the set of hardware and software that must function correctly in order for security to be maintained. In this work, the trusted computing base includes many of the usual trusted components: hardware that has not been subverted, a trustworthy underlying operating system, and a reliable authentication mechanism.

With conventional security mechanisms, all programs are part of the trusted computing base with respect to the protection of privacy, since there is no internal mechanism ensuring that programs respect privacy. For privacy to be protected, it is necessary that programs not transfer information in ways that violate it. In this work, the model is that a static checker rejects programs containing information flows that violate privacy. The static checker may be a compiler that statically checks the information flows in a program and then digitally signs the program, or else a verifier that checks the work of such a compiler.

Together, these trusted components make a trusted execution platform. Figure 1.3 depicts a trusted execution platform, into which code may enter only if it has been checked statically to ensure that it may be trusted to obey the label model. Data in the system is labeled, as are inputs to and outputs from the system.

When this trusted computational environment is constructed from trusted nodes connected by a network, the communication links between the nodes also must be trusted, which can be accomplished through phys-

ical security or by encrypting and digitally signing communication between nodes. Unrelated third parties are assumed to be unable to violate privacy and integrity by snooping on or subverting channels directly; the question addressed here is how to prevent the intended receiver of an information transfer from violating privacy.

1.5 Applications

The goal of this new information flow control system is to support secure distributed computation, including the following useful applications:

- A node could share information with a downloaded program, yet prevent the mobile code from leaking the information; additionally, the program could be protected from leaking its private information to other programs running on the same node. This kind of security for mobile code would be useful both for clients, which download applet code from servers, and for servers, which upload servlet code and data from clients for remote evaluation.
- Secure servers and other heavily-used applications can be written in programming languages extended with information flow annotations, adding confidence that sensitive information is not revealed to clients of the service through programming errors.
- Trusted parties can provide *secure computation servers* that allow mutually distrusting parties to carry out computations securely and privately, even though neither trusts that the programs of the other will respect its security. This architecture is a solution to the problem that arises when neither party trusts the execution platform of the other, and might be used in the tax preparation example. A trustworthy platform for computation becomes a service with economic value for which the provider might charge.

The annotations used in the JFlow programming language could be used to extend many conventional programming languages, intermediate code (such as Java Virtual Machine bytecode [LY96]), or machine code, where the labeling system defined here makes a good basis for easily checkable security proofs as in proof-carrying code [Nec97]. A good approach to producing proof annotations is for the compiler to generate them as a by-product of static checking; this approach has been shown to work for checkable type-safe machine code [MWCG98], and ought to be applicable to information flow labels as well.

1.6 Limitations

The static analysis techniques developed in Chapters 3 through 5 are intended to control covert and legitimate storage channels. These techniques do does not deal with timing channels, which are harder to control. Because the static analysis is applied to the program being executed, it cannot identify covert channels that do not exist at the level of abstraction presented by the programming language. These covert channels are

mostly timing channels that are ruled out in a single-threaded system. However, in a multi-threaded system, information may be communicated by covert channels such as cache miss timing. Covert channels of this sort cannot be identified by analysis of a program in source code form, because the source code is at too high a level of abstraction.

1.7 Outline

The remainder of this thesis is structured as follows. Chapter 2 describes the decentralized label model and demonstrates its formal properties. Chapter 3 presents the JFlow programming language, which extends the Java language with support for information flow control. Chapter 4 shows how information flow in the JFlow language can be checked statically through a process similar to type checking, though certain aspects of static checking and source-to-source translation are deferred until Chapter 5. Other security techniques and related work on privacy protection are discussed in Chapter 6. Chapter 7 concludes and offers some thoughts on extensions to this work.

Chapter 2

The Label Model

This chapter describes the decentralized label model. It has been presented earlier [ML97, ML98] but is developed further in this thesis. The key new feature of the decentralized label model is that it supports computation in an environment with mutual distrust. The ability to handle mutual distrust is achieved by attaching a notion of ownership to information flow policies. These policies then can be modified safely by their owners—a form of safe *declassification*. Arbitrary declassification is not possible because flow policies of other principals remain in force.

The decentralized label model also supports a richer set of safe relabelings than earlier models. For example, it enables every user to define a personal set of sensitivity levels, so that a data value can be relabeled upward in sensitivity independently for each user. It also allows information flow policies to be defined conveniently in terms of groups and roles. The rule for relabeling data is also shown to be both *sound* and *complete* with respect to a simple formal semantics for labels: the rule allows only safe relabelings, and it allows *all* safe relabelings.

The decentralized label model also has the important property that it supports static checking of information flow, including the ability to infer many information flow labels automatically. Discussion of static checking and how the model is integrated into a programming language is deferred until Chapters 3 and 4. However, this chapter does demonstrate that the model has the necessary properties to support this integration.

This chapter has the following structure: in Section 2.1, the essentials of the label model are presented. Section 2.2 provides some examples showing how the label model is applied to applications. The following sections develop the model more carefully. Section 2.3 gives a formal semantics of labels in the system, and Section 2.4 uses this semantics to develop more powerful rules for manipulating labels. Output channels are discussed in Section 2.5. Section 2.6 discusses ways that labels and principals can be generalized to allow more convenient modeling of security requirements.

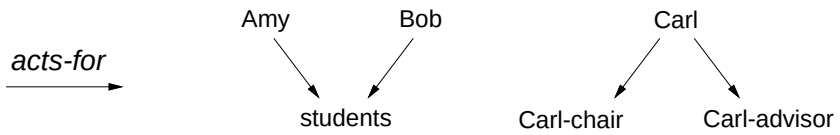

Figure 2.1: Principal hierarchy examples

2.1 Basic model

This section presents the essentials of the decentralized label model: principals, which are the entities whose privacy is protected by the model, and labels, which are the way that principals express their privacy concerns. The rules that must be followed as computation proceeds in order to avoid information leaks are then described, including the mechanism for safe declassification within this model.

2.1.1 Principals

In the decentralized label model, information is owned by, updated by, and released to *principals*: users and other authority entities such as groups or roles. For example, both users and groups in Unix would be modeled as principals.

In this model, some principals are authorized to *act for* other principals The *acts for* relation is reflexive and transitive, defining a hierarchy or partial order of principals. This relation is similar to the *speaks for* relation [LABW91]; the principal hierarchy is also similar to a *role hierarchy* [San96].

The acts-for relation can be used to model groups and roles conveniently, as shown in Figure 2.1. Arrows in the figure indicate acts-for relations. A group, such as students, is modeled by authorizing all of the principals representing members of the group (Amy and Bob) to act for the group principal. A role, which is a restrictive form of a user's authority, is modeled by authorizing the user's principal to act for the role principal. In the figure, the roles Carl-chair and Carl-advisor are roles that the principal Carl can fill.

Information about the structure of the principal hierarchy is maintained in a secure database. Although the principal hierarchy changes over time, revocations are assumed to occur infrequently. The handling of revocation is discussed later, in Section 3.2.5.

This simple model of principals is easily generalized to provide more complete modeling of groups, roles, and other entities; these extensions are explored later, in Section 2.6.3.

2.1.2 Labels

Every value used or computed in a program execution has an associated *label*. As we will see later, the label of a value functions as a kind of type, so program expressions can also be said to have a label. A label is a set of *policies* that express privacy requirements. A privacy policy has two parts: an owner, and a set of readers, and is written in the form *owner*: *readers*. The owner of a policy is a principal whose data was observed in order to construct the value labeled by this policy. The readers of a policy are a set of principals who are

permitted by the owner to read the data. It is also implicitly understood that the owner of the policy permits itself to read the data, even if it is not explicitly a reader. Other principals are not permitted to read the data. The intuitive meaning of a label is that every policy in the label must be obeyed as data flows through the system, so labeled information is released only by the consensus of all of the owners. A principal may read the data only if it is a reader or owner for every policy in the label. Because the intersection of all of the policies is enforced, adding more policies to a label only restricts the propagation of the labeled data.

An example of an expression that denotes a label L is the following: $L = \{o_1 : r_1, r_2;\ \ o_2 : r_2, r_3\}$, where o_1, o_2, r_1, r_2 denote principals. Semicolons separate two policies within the label. The owners of these policies are o_1 and o_2, the reader sets for the policies are $\{r_1, r_2\}$ and $\{r_2, r_3\}$, respectively. A policy with no readers means that only the owner of the policy is to be able to read the data. An example of a label containing such a policy is $\{o_1 :\ \}$, which is equivalent to the label $\{o_1 : o_1\}$.

If a label does *not* contain any policy owned by a principal p, the effect is that p does not care how the data propagates. It is as if there were a policy for p that listed *all* possible principals as readers. The least restrictive label possible is a label containing no policies, because no principal has expressed an interest in restraining the data with this label. This label is written as an empty set, $\{\}$. If a label contains two or more policies with the same owner p, the policies are enforced independently just as other policies are: a principal may read the labeled data only if all the policies permit that principal as a reader.

If a policy K is part of the label L (that is, $K \in L$), then the notation $\mathbf{o}(K)$ denotes the owner of that policy, and the notation $\mathbf{r}(K)$ denotes the *set* of readers specified by that policy. The functions $\mathbf{o}$ and $\mathbf{r}$ completely characterize a label, with types $\mathsf{policy} \rightarrow \mathsf{principal}$ and $\mathsf{policy} \rightarrow \mathsf{set[principal]}$, respectively. For compactness, single-argument functions like $\mathbf{o}$ and $\mathbf{r}$ will often be expressed without parenthesizing the arguments; for example, as $\mathbf{o}K$ rather than $\mathbf{o}(K)$. In the equations in this chapter, the letters I, J, K always denote label policies.

2.1.3 Relabeling by restriction

As a program computes, the information it manipulates will not leak as long as the labels of that information obey certain rules. We can now begin to consider these rules, beginning with arguably the simplest computation that can be performed by a program: assignment of a value into a variable.

In this model, every variable has a label that applies to the data within the variable. When a value is read from a variable, it has the same label as the variable. When a value is stored into a variable, the label of the value is forgotten; effectively, it acquires the label of that variable into which it is stored. Thus, assignment of a value to a variable causes a *relabeling* of the copy of the value that is assigned. To avoid leaking information, the label of the copied value (which is the label of the variable) must be at least as restrictive as the original label of the value. This kind of relabeling is therefore termed a *restriction*.

The expression $L_1 \sqsubseteq L_2$ means that the label L_1 is either less restrictive than or equal to the label L_2 (alternatively, L_2 is at least as restrictive as L_1), and that values can be relabeled from L_1 to L_2. Using this definition, an assignment from a value x into a variable v is legal if $L_x \sqsubseteq L_v$, where L_x and L_v are the labels

of x and v, respectively.

A relabeling is a restriction if all of the policies in the old label are guaranteed to be enforced in the new label. A policy J in L_1 is guaranteed to be enforced by a policy in K if the two policies have the same owner, and the reader set of K is a subset of the reader set of J. This observation leads to the *subset relabeling rule*:

Relabeling by restriction: subset rule

$$\frac{\forall (J \in L_1)\ \exists (K \in L_2)\ (\mathbf{o}K = \mathbf{o}J \ \ \wedge \ \ \mathbf{r}K \subseteq \mathbf{r}J)}{L_1 \sqsubseteq L_2}$$

The following relabelings are restrictions under this rule, assuming the letters A–E denote principals:

$$\begin{array}{rcl} \{A:B,C\} & \sqsubseteq & \{A:B\} \\ \{A:B\} & \sqsubseteq & \{A:\ ;\ D:E\} \\ \{A:B,C\} & \sqsubseteq & \{A:B;\ A:C\} \\ \{\} & \sqsubseteq & \{A:B\} \end{array}$$

The subset relabeling rule is sound and captures relabelings that are safe regardless of the principal hierarchy. However, if some knowledge of the principal hierarchy is available, additional relabelings can be determined to be safe. However, presentation of a more permissive relabeling rule must wait until a formal semantics for labels has been developed in Section 2.3, defining what it means for a relabeling to be safe.

In this model, variables are *statically bound* to their labels, and a value loses its label upon assignment. This approach to supporting variables differs from the *dynamic binding* approach used in some systems [MMN90, MR92], where the label of a variable is automatically made more restrictive when a restricted value is written into it. Dynamic binding requires run-time overhead and prevents static analysis. It also can lead to *label creep*, in which a variable becomes gradually more restrictive until it is unusable. In JFlow, the type Protected, described in Chapter 3, can provide the behavior of a dynamically labeled variable if it is needed.

2.1.4 Computation and label join

During computation, values are derived from other values. Because a derived value may contain information about its sources, its label must reflect the policies of each of its sources. For example, if we multiply two integers, the product's label must be at least as restrictive as the labels of both operands.

To avoid unnecessarily restricting the result of a computation, the result should have the *least* restrictive label that is at least as restrictive as the labels of the operand; that is, the *least upper bound* or *join* of the operand labels with respect to the relation $\sqsubseteq$. The join of the operands, which is constructed simply by taking the union of the sets of policies in the operand labels, ensuring that all of the policies of the operands

are enforced in the result. For example, the join of the labels $\{A : B\}$ and $\{C : A\}$ is $\{A : B;\ C : A\}$. For any two labels L_1 and L_2, their join is written as $L_1 \sqcup L_2$ and is defined as follows:

Join rule

$$L_1 \sqcup L_2 = L_1 \cup L_2$$

This rule ensures that the policies in the label of a value propagate to the labels of all other values that it affects, protecting the privacy of data even when it is used for computation. However, sometimes this rule is too restrictive, and a way to relax these policies is needed.

2.1.5 Relabeling by declassification

Because labels in this model contain information about the owners of labeled data, these owners can retain control over the dissemination of their data, and relax overly restrictive policies when appropriate. This is a safe form of *declassification* that provides a second way of relabeling data.

The ability of a process to declassify data depends on the authority possessed by the process. At any moment while executing a program, a process is authorized to act on behalf of some (possibly empty) set of principals. This set of principals is referred to as the *authority* of the process. If a process has the authority to act for a principal, actions performed by the process are assumed to be authorized by that principal. Code running with the authority of a principal can declassify data by creating a copy in whose label a policy owned by that principal is relaxed. In the label of the copy, readers may be added to the reader set, or the policy may be removed entirely, effectively allowing all readers.

Because declassification applies on a per-owner basis, no centralized declassification process is needed, as it is in systems that lack ownership labeling. Declassification is limited because it cannot affect the policies of owners the process does not act for; declassification is safe for these other owners because reading occurs only by the consensus of all owners.

The declassification mechanism makes it clear why the labels maintain independent reader sets for each owning principal. For example, if a label consisted of just an owner set and a reader set, information about the individual flow policies would be lost, reducing the power of declassification.

Because the ability to declassify depends on the run-time authority of the process, it requires a run-time check for the proper authority. As shown in Chapter 4, the overhead of this run-time check can be reduced in the proper static framework.

Declassification can be described more formally. A process may weaken or remove any policies owned by principals that are part of its authority. Therefore, the label L_1 may be relabeled to L_2 as long as $L_1 \sqsubseteq L_2 \sqcup L_A$, where L_A is a label containing exactly the policies of the form $\{p :\}$ for every principal p in the current authority. The rule for declassification may be expressed as an inference rule:

Relabeling by declassification

$$\frac{L_A = \bigsqcup_{(p \text{ in current authority})} \{p : \ \} \qquad L_1 \sqsubseteq L_2 \sqcup L_A}{L_1 \text{ may be declassified to } L_2}$$

This inference rule builds on the rule for relabeling by restriction. The subset rule for relabeling L_1 to L_2 states that for all policies J in L_1, there must be a policy K in L_2 that is at least as restrictive. The declassification rule has the intended effect because for policies J in L_1 that are owned by a principal p in the current authority, a more restrictive policy K is found in L_A. For other policies J, the corresponding policy K must be found in L_2, since the current authority does not have the power to weaken them. This rule also shows that a label L_1 always may be declassified to a label that it could be relabeled to by restriction, because the restriction condition $L_1 \sqsubseteq L_2$ implies the antecedent $L_1 \sqsubseteq L_2 \sqcup L_A$.

2.1.6 Channels

In this model, users are assumed to be external to the system on which programs run. Information is leaked only when it leaves the system. Giving private data to an untrusted program does not create an information leak—even if that program runs with the authority of another principal—as long as that program obeys all of the label rules described here. Information can be leaked only when it leaves the system through an *output channel*, so output channels are labeled to prevent leaks. Information can enter the system through an *input channel*, which also is labeled to prevent leaks. It is safe for a process to manipulate data even though no principal in its authority has the right to read it, because all the process can do with the data is write it to a variable or a channel with a label that is at least as restrictive.

Input and output channels are half-variables; like variables, they have an associated label and can be used as an information conduit. However, they only provide half of the functionality that a variable provides: either input or output. As with a variable, when a value is read from an input channel, the value acquires label of the input channel. Similarly, a value may be written to an output channel only if the label of the output channel is at least as restrictive as the label on the value; otherwise, an information leak is presumed to occur.

Obviously, the assignment of labels to channels is a security-critical operation. It is important that the channel's label reflect reality. For example, if the output of a printer can be read by a number of people, it is important that the output channel to that printer identify all of them, because otherwise an information leak is possible. If two computers communicate over channels, it is important that the labels of the matching output and input channels agree; otherwise, labels can be laundered by a round trip.

Typically, an output or input channel has a label containing a single policy, though multiple-policy channels work too. For an output channel, the owner of the policy can be thought of as a guarantor that the data will be released to *at most* the principals listed in the reader set of that policy. As will become clear, the

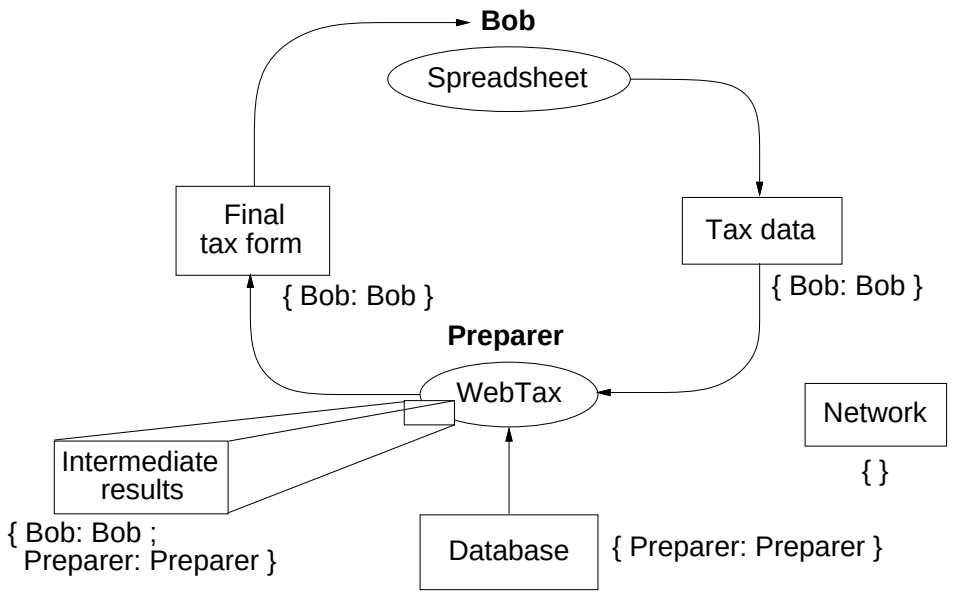

Figure 2.2: Annotated Tax Preparation Example

data of a principal p can be written to an output channel only if p trusts the owner of the output channel, and the readers of the output channel are a subset of the readers that p allows. Conversely, the owner of an input channel is a principal who demands that data arriving from the channel may be released only to the listed readers. This policy may be overridden only by the owner or by a principal who can act for the owner. For multiple-policy channels, each policy acts as an additional requirement for the release of the data.

2.2 Examples

Let us now consider two examples in which the decentralized label model is helpful in protecting privacy. These examples illustrate the intuitions behind the model and demonstrate that it can capture the security needs of interesting, useful computations.

2.2.1 Tax preparer example

The tax preparer example, illustrated in Figure 2.2, is identical to the example from Chapter 1, except that all data in the example has been annotated with labels to protect the privacy of Bob and Preparer. It can be seen that these labels obey the rules given and meet the security goals set out in Chapter 1 for this scenario.

In the figure, ovals indicate programs executing in the system. A boldface label beside an oval indicates the authority with which a program acts. In this example, the principals involved are Bob and Preparer, as we have already seen, and they give their authority to the spreadsheet and WebTax programs, respectively. Arrows in the diagrams represent information flows between principals; square boxes represent information that is flowing, or databases of some sort.

First, Bob applies the label $\{\mathsf{Bob}\colon\ \mathsf{Bob}\}$ to his tax data. This label allows no one to read the data

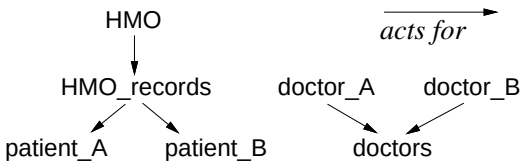

Figure 2.3: The hospital principal hierarchy

except Bob himself. With this label applied to it, tax data cannot be sent to an untrusted network location, represented as an output channel with label $\{\}$, because it is not the case that $\{\mathsf{Bob}\colon \mathsf{Bob}\} \sqsubseteq \{\}$. Bob can give this data to the WebTax program with reasonable confidence that it cannot be leaked, because WebTax will be unable to remove the $\{\mathsf{Bob}\colon \mathsf{Bob}\}$ policy from the tax data or any data derived from it.

The WebTax program uses Bob's tax data and its private database to compute the tax form. Any intermediate results computed from these data sources will have the label $\{\mathsf{Bob}\colon \mathsf{Bob};\ \mathsf{Preparer}\colon \mathsf{Preparer}\}$. Because the reader sets of this label disagree, the label prevents both Bob and Preparer (and everyone else) from reading the intermediate results. This joint label is generated by the rule for join:

$$\{\mathsf{Bob} : \mathsf{Bob}\} \sqcup \{\mathsf{Preparer} : \mathsf{Preparer}\ \} = \{\mathsf{Bob} : \mathsf{Bob}\ ;\ \mathsf{Preparer} : \mathsf{Preparer}\ \}$$

Preparer is protected by this label against accidental disclosure of its private database through programming errors in the WebTax application.

Before being released to Bob, the final tax form has the same label as the intermediate results, and is not readable by Bob, appropriately. In order to make the tax form readable, the WebTax application *declassifies* the label by removing the $\{\mathsf{Preparer}\colon \mathsf{Preparer}\}$ policy. The application can do this because the Preparer principal has granted the application its authority. This grant of authority is reasonable because Preparer supplied the application and presumably trusts that it will not use the power maliciously.

The authority to act as Preparer need not be possessed by the entire WebTax application, but only by the part that performs the final release of the tax form. By limiting this authority to a small portion of the application, the risk of accidental release of the database is reduced. However, it is important that this part of the application not be exposed as a generally accessible external interface, because this exposure might allow Bob and other parties to misuse the interface to declassify data owned by Preparer.

2.2.2 Hospital example

In this example, there are three parties with privacy concerns: a patient obtaining medical services, a doctor providing the services, and a health maintenance organization (HMO) that serves as an intermediary. There are principals in the system for patients, *e.g.*, patient_A, and doctors, *e.g.*, doctor_B; additionally, all doctors can act for a principal doctors that represents the group of doctors within the HMO. Two HMO principals also exist: HMO, representing maximum authority within the HMO, and HMO_records, representing authority over the record-keeping functions of the HMO; HMO can act for HMO_records, and HMO_records

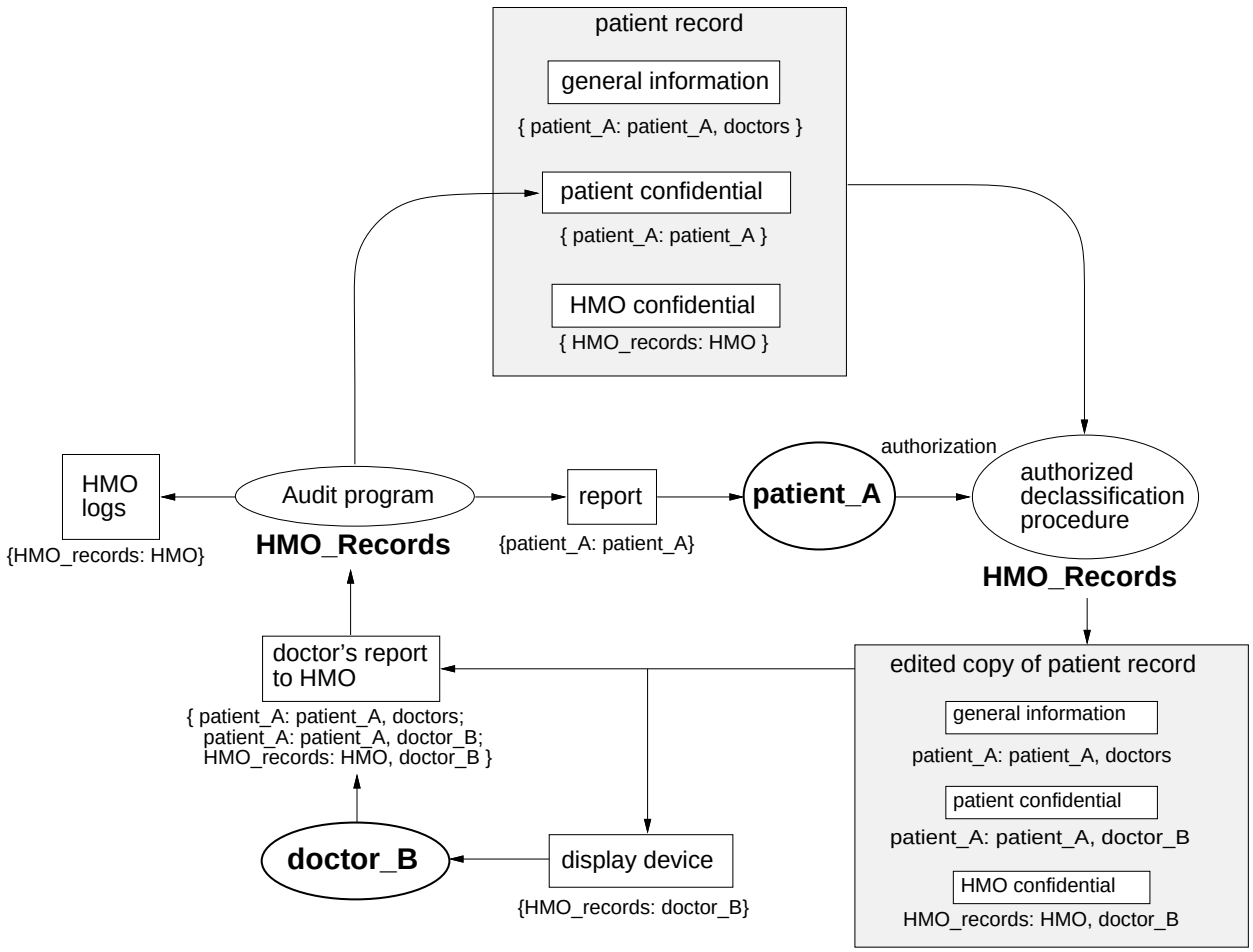

Figure 2.4: The hospital example

can act for patients: each patient must trust the HMO to keep track of its records. The resulting principal hierarchy is shown in Figure 2.3.

Figure 2.4 shows the hospital example, which shows how information flows as the patient receives medical services. The HMO maintains the patient's medical history, which has three parts: general information, which is controlled by the patient but is readable by any doctor, private information (such as the medical history of the patient), which is normally not readable by doctors, and confidential information that the HMO does not release to patients.

The first step in a patient/doctor interaction is for the doctor to obtain a copy of the patient's record. The record is declassified so that the doctor can read it; this can happen only with the authorization of the patient. The patient, represented in the diagram by the dark oval labeled patient_A, makes an authenticated request to an existing program running with the authority of HMO_records; this program uses the patient's authority to provide the doctor with an edited version of the patient's private information and of the HMO's confidential information.

The doctor is represented by the dark oval labeled doctor_B. To read the information, the doctor requires

an output channel to a display device with the single reader, doctor_B. This display device is certified by HMO_records as a secure device that only doctor_B is reading from. In principle, all of the information in the patient records should be safe to write to this display device, though the subset relabeling rule will not permit it. Thus, this example motivates the development of a better relabeling rule, which is developed in the following sections. Writing the information to the display device is safe because HMO_records can act for all of the owners of the data in the patient records (patient_A and HMO_records), so its certification should be good enough. In addition, various parts of the patient record are released to doctors or doctor_B, and the actual reader, doctor_B, can act for both these principals. Note that the patient information cannot be written to a channel that has any readers other than doctor_B, and that there is no way the doctor can declassify the patient information.

Eventually, the doctor sends a report to the HMO of services rendered. In addition to the comments of the doctor, the report contain information from all three components of the patient's record, so it acquires a joint label reflecting all these sources. Note that the general patient information does not explicitly permit doctor_B as a reader. Using the subset relabeling rule, the first policy owned by patient_A in the resulting joint label prevents the doctor from reading his own report. This example of unnecessary restrictiveness also arises from the subset relabeling rule and is fixed by the more flexible relabeling rule developed later.

The audit program runs with the authority of the HMO_records principal and thus can store the information with the appropriate labels both in the log and in the patient record database. It can also send a report to the patient; as in the tax preparer example, the designer of the audit program must use mechanisms outside the scope of information flow control to determine either that no HMO-confidential information is leaked or that the leak is acceptably small.

2.3 Extending and interpreting labels

The hospital example presented in the previous section shows that the basic model is not powerful enough, and a more permissive relabeling rule is needed that takes the principal hierarchy into account. This section formalizes the notions of labels and principal hierarchies and then defines a condition for judging whether a relabeling rule is correct.

2.3.1 Limitations of the subset relabeling rule

One way to think about whether a relabeling rule is safe is by considering *incremental relabelings* that can make a label more restrictive, or leave it equally restrictive. The relabeling rules discussed in this thesis can be understood in terms of the incremental relabelings they allow. For example, the subset relabeling rule allows the following two kinds of *incremental relabelings*, which make a label more restrictive (or possibly have no effect).

- **Removing a reader**. Removing a reader from a policy will restrict the propagation of the labeled data further, if it has any effect at all.

- **Adding a policy**. Similarly, adding a new policy only can restrict the data further, because all policies in a label are enforced.

Any sequence of such relabelings will also result in a label that is at least as restrictive as the original. To compare two labels and see whether a sequence of such incremental relabelings can be found is trivial.

The subset relabeling rule defined earlier is clearly sound, in that it only permits a value to be relabeled to a more restrictive label. However, it prevents valid relabelings. There are three kinds of such relabelings, which are based on the existence of an acts-for relationship between principals:

- **Adding readers.** It should be possible to add a reader r' to a policy if the policy already allows a reader r that r' acts for. This rule is safe because if r' acts for r, it has all of the privileges of r. Allowing r to read the data also allows all principals that act for r to read.

- **Replacing owners.** It should be possible to replace an owner o with some principal o' that acts for o. This rule is safe because the new label allows only processes that act for o' to declassify it, while the original label also allows processes with the weaker authority of o to declassify it.

- **Self-authorization.** If a principal o is the owner of a policy, it is safe to add as a reader any principal r that acts for o. We already consider the owner of a policy to be a reader, so it is reasonable to allow the owner to be added explicitly to the list of readers. Similarly, the addition of readers that act for the owner should be allowed.

If readers may be added, the doctor in the example is able to view his own report. The confidential patient information has the label $\{$patient_A: patient_A,doctors$\}$, which allows any doctor to view the data item, and therefore it should be possible to relabel the item explicitly to allow a particular doctor to view it, *e.g.*, $\{$patient_A: patient_A,doctor_B$\}$. The doctor doctor_B then can view the report, because doctor_B is a reader in every policy in the joint label.

If owners may be replaced, the output channel in the hospital example (Figure 2.4) will work as intended. The output channel is labeled as $\{$HMO_records: doctor_B$\}$, which means that the HMO records division has certified that doctor_B is the only reader on this channel. With this label, the display device can be used to display all the information in the patient's record, since the principal HMO_records acts for patient_A. There is no global notion of the principals that can read from the output channel; data owned by an owner o can be written to this channel only if o trusts the HMO records division (that is, HMO_records can act for o).

The self-authorization rule does not add any significant power to the label model, since the policy owner always can be added explicitly as a reader of the policy. However, it does make the expression of many common labels more concise.

If the subset relabeling rule is used, then relabelings that add readers or replace owners can be done only by a process with sufficient authority, using the declassification mechanism. However, because these

relabelings are restrictions, it would be safe for any process to perform them regardless of its authority. Direct support for the relabelings is therefore consistent with the principle of least privilege [Sal74], since it avoids unnecessarily vesting excessive privilege in processes.

Extending the label model with support for these relabelings also facilitates the modeling of some desirable security policies. For example, suppose that a user wants to define security classes in a multi-level fashion: their own personal unclassified, classified, and secret classes for protecting their data. With these extensions, these three security classes can be represented as principals in the system, where the secret principal can act for classified, and classified for unclassified. The user then can assign security classes to other principals in the system by allowing them to act for one of these three principals; the user correspondingly marks each data item as readable by the appropriate security class principal.

It is not trivial to extend the relabeling rule to permit these relabelings, because we want to preserve the ability to analyze information flow statically. As pointed out by Denning and Denning [DD77], information flow should be checked statically (*e.g.*, at compile time) to avoid leaks through *implicit flows*, which are discussed later in Section 3.1. The new relabelings above depend on the principal hierarchy as it exists at run time. The principal hierarchy that exists at run time is likely to differ from the principal hierarchy at compile time, so the rule for relabeling must work when the principal hierarchy changes. The trick is to check relabelings statically using a rule that ensures that the relabelings are safe for *all* hierarchies that might be encountered at run time at that point in the program.

This problem is addressed in two steps. The remainder of this section presents a formal model for labels that allows a precise definition of legal relabelings. Section 2.4 then defines the rules for static checking and shows that they are both sound and complete.

2.3.2 Interpreting labels

A relabeling is allowed if it does not create new ways for the relabeled information to flow. However, to characterize this rule precisely, we need a way to *interpret* a label: that is, to decide what information flows are described by a label. It is useful to think of a label as describing a set of *flows*, where a flow is an (owner, reader) pair. The set of denoted flows is the label's *interpretation*. A flow (o, r) represents a flow of information from the owner o to the reader r; if the interpretation of a label contains a flow (o, r), it means that according to the principal o, the labeled data may be read by the principal r. In general, the interpretation of a label includes flows not explicitly stated in the label.

The subset relabeling rule corresponds to a very literal interpretation of a label as a set of flows: if a label L has a policy K, then this interpretation of L contains flows $(\mathbf{o}K, r)$ for every reader r in the set $\mathbf{r}K$. However, if a principal o' is not an owner in the label, the interpretation of L contains flows (o', r) for every principal r. In other words, o' permits flows to every principal because it has not expressed a flow policy for the labeled data and does not care how it flows. For example, in a system containing three principals A, B, and C, the label $\{A : B;\ C :\ \}$ is interpreted as the set of flows $\{(A, B), (B, A), (B, B), (B, C)\}$. There are flows from B to every other principal because it is not an owner, but no flows from C, since it allows

no readers. If a principal o is an owner of multiple policies K_i, then the label only describes flows (o, r) for readers r in the intersection of all the sets $\mathbf{r}K_i$. This interpretation is a function that maps labels into sets of flows, and is called $\mathbf{X}_0$. For any label L, the expression $\mathbf{X}_0(L)$ is a simple, literal interpretation of L as a set of flows.

We have seen already that the subset relabeling rule is too restrictive to support certain safe relabelings, because it does not take the principal hierarchy into account. A more flexible relabeling rule requires an interpretation function that, unlike $\mathbf{X}_0$, does take the principal hierarchy into account.

Despite the limitations of the $\mathbf{X}_0$ interpretation, it has a use here as a shorthand for expressing sets of flows, precisely because it is so literal. Writing down sets of flows is inconvenient because the sets of flows are usually large and contain uninteresting flows, such as the many flows from principals that are not owners. However, a set of flows can be expressed unambiguously in a manner that is independent of the principal hierarchy by writing a label whose interpretation by $\mathbf{X}_0$ is that set of flows. For every set of flows that is of interest, a label can be constructed easily whose interpretation by $\mathbf{X}_0$ is that set of flows; in this chapter, these labels are given in place of much longer sets of flows that have the same meaning.

2.3.3 Formalizing the principal hierarchy

To express a richer interpretation precisely, it is necessary to clarify the idea of the principal hierarchy. If x can act for y, it is denoted formally by the expression $x \succeq y$. The binary relation $\succeq$ is reflexive and transitive, but not anti-symmetric: two distinct principals may act for each other, in which case the principals are said to be equivalent. A relation of this sort is called a pre-order. The notation $P \vdash x \succeq y$ indicates that the principal x can act for the principal y in the principal hierarchy P. A principal hierarchy is a pre-order on principals, and can therefore be treated as a set of ordered pairs of principals that specifies all relations that exist. With this interpretation, $P \vdash x \succeq y$ is equivalent to $(x, y) \in P$. When one principal hierarchy P' contains more acts-for relations than another, P, we say that P' *extends* P, which is written as $P' \supseteq P$.

The space of principals is assumed to be infinite, immutable, and pre-existing. Of course, a real implementation must be finite and will allow the creation of new principals. In this model, the creation of a new principal is treated as the assignment of new meaning to some already existing (but unused) principal. The advantage of this treatment is that a principal hierarchy P is just a set of acts-for relations; it does not specify the set of its principals as well.

2.3.4 Label interpretation function

The idea behind a richer interpretation is that actual flows denoted by the label depend on the principal hierarchy. The label interpretation function has the form $\mathbf{X}(L, P)$, where $\mathbf{X}$ is a function yet to be defined, L is the label being interpreted, and P is the principal hierarchy in which it is being interpreted. Taking the current principal hierarchy as an implicit argument for now, the set of flows $\mathbf{X}L$ is the interpretation of the label L.

Informally, the function $\mathbf{X}$ is defined as follows: a flow (o, r') is denoted by a label L if every policy I whose owner can act for o permits the flow—either explicitly, because r' is either a member of the reader set of I or the owner of I; or implicitly, because some principal r is a member of the reader set (or the owner), and $r' \succeq r$. Also, if there is no policy I whose owner can act for o, the flow is permitted because o does not care how the data propagates.

There are two intuitions behind this new interpretation. First, if a policy lists a reader r as a reader, that policy implicitly authorizes as readers all principals r' such that $r' \succeq r$. This implicit authorization makes sense because such an r' should possess every power than r does. Second, suppose there is a policy I in the label owned by a principal o'. In this case, it is as if the label contains policies owned by every principal o that o' acts for, and these policies have reader sets identical to that of the policy I. In other words, the policies dictated by o' apply to every principal o that it acts for. In the following sections, the basis for interpretation function $\mathbf{X}$ is developed more carefully, formally specifying $\mathbf{X}$ and showing how it is constructed. This more complex interpretation is then used to develop a less restrictive relabeling rule.

2.3.5 Flow set constraints

If we consider the label as a set of flows, we can see that there are two constraints that a set of flows ought to satisfy in a particular principal hierarchy—one constraint on readers, and one on owners. A set of flows makes sense only if it satisfies both of these constraints. As we will see, these constraints underlie the label interpretation function just described.

The *reader constraint* corresponds to the first intuition just described: if a set of flows contains a flow (o, r), and r' is a principal that can act for r, then the set must also contain the flow (o, r'). For example, the label $\{$patient_A: doctors$\}$ is equivalent to the label $\{$patient_A: doctors, doctor_B$\}$, since the principal doctor_B can act for the principal doctors. The reader constraint can be stated more formally as follows, using the symbol $\rightarrow$ for implication:

$$r' \succeq r \ \wedge \ (o, r) \in \mathbf{X}L \rightarrow (o, r') \in \mathbf{X}L$$

However, the reader constraint is not sufficient, because we also want to allow relabelings that change the label's owners. Consider the relabeling from $\{$patient_A: doctor_B$\}$ to $\{$HMO_records: doctor_B$\}$. This relabeling effectively transfers the responsibility of controlling the flow of the data from the principal patient_A to the principal HMO_records. This transfer restricts the data's flow, since HMO_records can act for patient_A. The key insight to allowing this kind of relabeling is the *owner constraint*:

$$o' \succeq o \ \wedge \ (o', r) \notin \mathbf{X}L \rightarrow (o, r) \notin \mathbf{X}L$$

The interpretation of this constraint is that when a superior owner states that a flow must not occur, this flow is removed from the reader sets of all inferior owners (principals that the superior owner acts for). Restrictions applied by superior owners apply to inferior owners as well. However, if a superior owner does

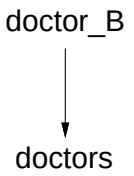

Figure 2.5: A small principal hierarchy

not try to prevent a flow, inferior owners may still prevent it. Thus, the inferior owner's policy must be at least as restrictive as the superior owner's policy.

Using this constraint, the label $\{\mathsf{HMO_records}\colon\ \mathsf{doctor_B}\}$ is equivalent to the label $\{\mathsf{HMO_records}\colon \mathsf{doctor_B};\ \mathsf{patient_A}\colon \mathsf{doctor_B}\}$, in the principal hierarchy of Figure 2.3. While the first label would seem to allow flows from patient_A to all readers, the only flow it allows from patient_A is (patient_A, doctor_B), because $\mathsf{HMO_records} \succeq \mathsf{patient_A}$ and the HMO_records policy only allows a flow to doctor_B.

2.3.6 Label functions

To help construct the label interpretation function $\mathbf{X}$, two functions are defined that establish the reader and owner constraints. First, the function $\mathbf{R}$ expands the set of readers in a policy I to include the readers implicitly allowed by the reader constraint, as well the owner of the policy I and any principals that can act for it. Given a policy I, the function produces an expanded policy $\mathbf{R}I$. Using the notation $\langle \mathbf{o}I : \mathbf{r}I \rangle$ to denote the policy with owner $\mathbf{o}I$ and readers $\mathbf{r}I$, the function is defined as follows:

$$\mathbf{R}I = \langle \mathbf{o}I : \{r' \mid r' \succeq \mathbf{o}I \ \vee\ \exists (r \in \mathbf{r}I)\ r' \succeq r\} \rangle$$

This function is expressed concisely using a function $\mathbf{r}^+$ that yields the reader set of a policy, plus its owner:

$$\begin{aligned} \mathbf{r}^+I &= \mathbf{r}I \cup \{\mathbf{o}I\} \\ \mathbf{R}I &= \langle \mathbf{o}I : \{r \mid \exists (r' \in \mathbf{r}^+I)\ r \succeq r'\} \rangle \end{aligned}$$

For convenience, the application of the function $\mathbf{R}$ to an entire label is defined as the label produced by applying $\mathbf{R}$ to each of its individual policies: $\mathbf{R}L = \{\mathbf{R}I \mid I \in L\}$. Suppose $\mathbf{R}$ is applied to the two-policy label $L_1 = \{\mathsf{doctors} : \mathsf{patient_A};\ \mathsf{doctor_B} : \mathsf{patient_A}, \mathsf{patient_B}\}$, in a principal hierarchy containing only the single relation $\mathsf{doctor_B} \succeq \mathsf{doctors}$, as shown in Figure 2.5. In this case, we have $\mathbf{R}L_1 = \{\mathsf{doctors} : \mathsf{patient_A}, \mathsf{doctors}, \mathsf{doctor_B};\ \mathsf{doctor_B} : \mathsf{patient_A}, \mathsf{patient_B}, \mathsf{doctor_B}\}$. Note that doctors self-authorizes itself as a reader in the first policy, and that doctor_B is therefore a reader because it acts for doctors.

To establish the owner constraint, the function $\mathbf{O}$ converts a label into a set of flows by restricting it. It generates a flow (o, r) only if all operative policies in the label (those policies I for which $\mathbf{o}I \succeq o$) allow the flow. The intuitive effect of $\mathbf{O}$ is to remove flows that would violate the owner constraint.

$$\mathbf{O}L = \{(o, r) \mid \forall (I \in L)\ \mathbf{o}I \succeq o \rightarrow r \in \mathbf{r}I\}$$

The function also generates a flow (o, r) if there are *no* policies in the label for which $\mathbf{o}I \succeq o$, since in that case the implication is vacuously true for all policies I in L. These flows capture the intuition that if a principal does not own a policy, it allows flows to all possible readers.

For example, consider applying $\mathbf{O}$ to $\mathbf{R}L_1$, from the previous example. The set of flows that results is the interpretation of the label $\{\mathsf{doctors} : \mathsf{patient_A}, \mathsf{doctor_B};\ \mathsf{doctor_B} : \mathsf{patient_A}, \mathsf{patient_B}, \mathsf{doctor_B}\}$ by $\mathbf{X}_0$. Notice that this set of flows includes the flow $(\mathsf{doctor}, \mathsf{doctor_B})$ but not $(\mathsf{doctors}, \mathsf{doctors})$, even though the first policy in $\mathbf{R}L_1$ seems to specify the latter flow. The flow $(\mathsf{doctors}, \mathsf{doctors})$ is eliminated by $\mathbf{O}$ because the owner of the second policy, $\mathsf{doctor_B}$, does not allow a flow to $\mathsf{doctors}$, and $\mathsf{doctor_B}$ acts for the owner of the first policy, $\mathsf{doctors}$.

As we would expect, $\mathbf{R}$ is monotonic with respect to reader sets that it is applied to, in the following sense: if $\mathbf{r}I_1 \supseteq \mathbf{r}I_2$ and $\mathbf{o}I_1 = \mathbf{o}I_2$, then $\mathbf{rR}I_1 \supseteq \mathbf{rR}I_2$. $\mathbf{O}$ is also monotonic in reader sets; if L_1 and L_2 are two labels that differ only in the reader sets of their respective policies I_1 and I_2, with $\mathbf{o}I_1 = \mathbf{o}I_2$ and $\mathbf{r}I_1 \supseteq \mathbf{r}I_2$, then $\mathbf{O}L_1 \supseteq \mathbf{O}L_2$.

However, the functions differ in their behavior as the principal hierarchy changes. To show this, the principal hierarchy P must appear as an explicit argument to the functions. If the principal hierarchy P' is an extension of P (that is, $P' \supseteq P$), then the following relations hold:

$$\begin{aligned} \mathbf{rR}(I, P') &\supseteq \mathbf{rR}(I, P) \\ \mathbf{O}(L, P') &\subseteq \mathbf{O}(L, P) \end{aligned}$$

Unlike $\mathbf{R}$, the function $\mathbf{O}$ is anti-monotonic in its argument P.

By composing the $\mathbf{R}$ and $\mathbf{O}$ functions, we obtain the label interpretation function $\mathbf{X}$, which maps a label to a set of flows, given a particular principal hierarchy.

Definition of the interpretation function X

$$\begin{aligned} \mathbf{X}L &= \mathbf{OR}L = \mathbf{O}\{\mathbf{R}I \mid I \in L\} = \\ &= \{(o, r) \mid \forall (I \in L)\ \mathbf{o}I \succeq o \rightarrow r \in \mathbf{rR}I\} \\ &= \{(o, r) \mid \forall (I\ \in L)\ \mathbf{o}I \succeq o \rightarrow [\ r \succeq \mathbf{o}I\ \vee\ \exists (r' \in \mathbf{r}I)\ r \succeq r'\ ]\} \end{aligned}$$

The result of $\mathbf{X}L$ satisfies both the reader and owner constraints, since $\mathbf{O}$ preserves the reader constraint established in each policy by $\mathbf{R}$. The result is that this formula has the same meaning as the informal definition for $\mathbf{X}$ presented earlier in Section 2.3.4. We have already seen an example of the application of $\mathbf{X}$ to the label $\{\mathsf{doctors} : \mathsf{patient_A};\ \mathsf{doctor_B} : \mathsf{patient_A}, \mathsf{patient_B}\}$, because the earlier examples applied $\mathbf{R}$ and $\mathbf{O}$ sequentially to it, just as in the definition of $\mathbf{X}$.

The function $\mathbf{X}$ can now be used to express the *correctness condition* for relabeling in the presence of an arbitrary principal hierarchy. The relabeling from L_1 to L_2 in principal hierarchy P is valid as long as no new flows are added. Making the principal hierarchy an explicit argument to $\mathbf{X}$, the correctness condition is the following:

Correctness condition

$$\frac{\mathbf{X}(L_1, P) \supseteq \mathbf{X}(L_2, P)}{\text{Relabeling from } L_1 \text{ to } L_2 \text{ is safe in } P}$$

We can apply this rule to show the validity of the relabeling from $L_1 = \{\mathsf{patient_A}\colon \mathsf{doctors}\}$ to $L_2 = \{\mathsf{HMO_records}\colon \mathsf{doctor_B}\}$, using the principal hierarchy of Figure 2.3. Applying $\mathbf{X}$ to L_2 gives us a set containing the flow ($\mathsf{HMO_records}$, $\mathsf{doctor_B}$) and the flows (p, $\mathsf{doctor_B}$) for every patient p (since HMO acts for all patients), as well as other flows (o, r) for unrelated owners o and all readers r. Applying $\mathbf{X}$ to L_1 gives us a set containing all these pairs and more: ($\mathsf{HMO_records}$, r) for every r, for example. Because $\mathbf{X}L_1 \supseteq \mathbf{X}L_2$, the relabeling from L_1 to L_2 is safe.

Because the function $\mathbf{X}$ is a composition of $\mathbf{R}$ and $\mathbf{O}$, it is monotonic with respect to reader sets in L, but neither monotonic nor anti-monotonic with respect to P. It also has some other interesting properties. We can interpret the set produced by applying $\mathbf{X}$ to a label as a label itself (although one that is too large to write down!); this is the label in which every flow is mentioned explicitly, even the flows from owners that allow all readers. With this interpretation, we can see that like $\mathbf{O}$ and $\mathbf{R}$, the function $\mathbf{X}$ is idempotent; that is, $\mathbf{X}L = \mathbf{X}\mathbf{X}L$.

2.4 Checking relabeling statically

Static checking of programs containing label annotations is desirable because it allows precise, fine-grained analysis of information flows and can capture implicit flows properly [DD77], whereas dynamic label checks create information channels that must be controlled through additional static checking [ML97]. However, the correctness condition ($\mathbf{X}L_1 \supseteq \mathbf{X}L_2$) derived in Section 2.3 cannot be used directly in static checking; it depends on the principal hierarchy at the time that the relabeling takes place, but static checking is done earlier, perhaps as part of compilation. The principal hierarchy may have changed between compilation and execution, so the full run-time principal hierarchy is not available when relabeling is checked. Therefore, relabeling must be checked using only partial information about the principal hierarchy.

In this section, a general rule is developed for checking relabelings statically, using partial information about the principal hierarchy. Section 2.4.1 begins by giving a sketch of how programs are annotated. Section 2.4.2 demonstrates that defining a sound relabeling rule for static environment is non-trivial. Then, Section 2.4.3 defines a relabeling rule for static checking and shows that it is both sound and complete. Finally, Section 2.4.4 shows that the label model has the lattice properties needed to support label checking and automatic label inference in a static environment.

```
int{patient: doctors} x;
int{patient: doctor_B} y;
actsFor (doctor_B, doctors) { y = x; }
```

Figure 2.6: Assignment using the static principal hierarchy

2.4.1 Annotations

Programs are statically annotated with information about the labels of data that they manipulate. A static label checker uses these annotations to analyze information flows within these programs and determine whether the program follows the information flow rules that have been described.

In Chapters 3 and 4, a set of language annotations is described that permits static information-flow checking. The following summarizes the features that are important for understanding how static analysis affects the model:

- All variables, arguments, and procedure return values have labeled types. For example, a labeled integer variable might be declared as `int{patient_A: doctors} x;`. The label may be omitted from a local variable, causing it to be inferred automatically. If the label is omitted from a procedure argument, it is an implicit parameter, and the procedure is generic with respect to it.

- The statement actsFor(p_1, p_2) S allows a run-time test of the structure of the principal hierarchy. The statement S is executed only if the principal p_1 can act for principal p_2. The label checker then uses the knowledge that $p_1 \succeq p_2$ when checking relabelings that occur within S. The statement also has an optional else clause that is executed if the specified relationship does not exist.

- The expression declassify(e, L) relabels the value e with the label L. The label L may add readers to the label of e for some owners o_i, or remove some owners o_i; the statement is legal only if it is statically known that the process can act for each of the o_i.

- Procedures are assigned a principal when they are compiled; this principal derives from the user who is running the compilation. When a procedure is called it always runs under this authority. Code that calls a procedure also can grant the called procedure the authority to act for one or more principals the caller acts for, but this grant must be made explicitly.

For example, the assignment from x to y in Figure 2.6 is legal because within the body of the actsFor statement, the checker knows that doctor_B can act for doctors.

For each program statement that the label checker verifies, some acts-for relations can be determined to exist, based on the lexical nesting of the actsFor statements. These relations form a subset of the true principal hierarchy that exists at run time; all that is known statically is that the true principal hierarchy contains the explicitly stated acts-for relations.

Using this fairly general model for programming with static information flow annotations, the challenge is to define a sound (conservative) rule for checking relabelings.

2.4.2 Static correctness condition

When a program assigns a value to a variable, it relabels the data being assigned, because the value's label is changed to be the same as the label on the variable. This relabeling is sound as long as it does not create new ways for the assigned data to flow. One example of a sound relabeling rule is the original subset relabeling rule of Section 2.1.3. For this rule, the monotonicity of **X** guarantees that the correctness condition holds, regardless of the run-time principal hierarchy. However, the subset relabeling rule, as we've seen, is excessively restrictive. We would like a rule that uses the information about the principal hierarchy that is available statically.

Let P be a principal hierarchy that contains only the acts-for relations that are statically known based on the containing actsFor statements. This principal hierarchy is called the *static principal hierarchy*. The actual principal hierarchy at run time is an extension of P; it must contain all of the acts-for relations in P, but may contain additional relations. If P' is the actual principal hierarchy, we have $P' \supseteq P$. Using this notation, and introducing the principal hierarchy as an explicit argument to the function **X**, the *static correctness condition* says that it is safe to relabel from L_1 to L_2 in P if the following condition holds at the time of static checking:

Static correctness condition

$$\frac{\forall (P' \supseteq P)\ \mathbf{X}(L_1, P') \supseteq \mathbf{X}(L_2, P')}{\text{Relabeling from } L_1 \text{ to } L_2 \text{ is statically safe in } P}$$

It is interesting to note that a more restrictive static correctness condition, $\forall (P)\ \mathbf{X}(L_1, P) \supseteq \mathbf{X}(L_2, P)$, is almost the same as checking the subset relabeling rule (the difference is that is allows self-authorization). The subset relabeling rule expresses the requirement that a relabeling be safe in all principal hierarchies, but what we want is a relabeling rule that takes advantage of information about the run-time principal hierarchy, as expressed by the condition $P' \supseteq P$ in the static correctness condition.

One might expect that to check whether a relabeling is valid, we could check a weaker condition, which simply applies the correctness condition directly to the static hierarchy P:

$$\mathbf{X}(L_1, P) \supseteq \mathbf{X}(L_2, P)$$

By construction, this rule allows all valid relabelings to take place; if a relabeling is not allowed by this rule, then it creates new flows in the principal hierarchy P. Therefore, this rule is necessary but not sufficient. The following example will show that this rule is not sound.

Consider the following (bad) relabeling from L_1 to L_2, where L_1 is the same label that was used in the examples of Section 2.3.6:

L_1 = { doctors: patient_A; doctor_B: patient_A, patient_B }
L_2 = { doctors: staff, patient_A ; doctor_B: patient_A, patient_B }

Now, consider what happens when we apply **X** to each of these labels while assuming that the principal hierarchy P contains a single relation doctor_B $\succeq$ doctors that is known to hold at compile time; in other words, the principal hierarchy shown in Figure 2.7(a). The result of **X** when applied to each label is a set of flows, which is written as a label for brevity, using the $\mathbf{X}_0$ interpretation:

$\mathbf{X}L_1$ = { doctors: patient_A, doctor_B; doctor_B: patient_A, patient_B, doctor_B }
$\mathbf{X}L_2$ = { doctors: patient_A, doctor_B; doctor_B: patient_A, patient_B, doctor_B }

Note that $\mathbf{X}L_2$ does not contain the flow (doctors, staff) because the superior owner doctor_B rules it out. It would seem that the relabeling is safe because these two label interpretations are equal. However, suppose that the run-time principal hierarchy is the one shown in Figure 2.7(b); that is, patient_B is also a staff member (patient_B $\succeq$ staff). Applying **X** to each label using this hierarchy leads to a quite different conclusion:

$\mathbf{X}L_1$ = { doctors: patient_A, doctor_B; doctor_B: patient_A, patient_B, doctor_B }
$\mathbf{X}L_2$ = { doctors: patient_B, patient_A, doctor_B; doctor_B: patient_A, patient_B, doctor_B }

The relabeling is invalid under the principal hierarchy P', because it adds the flow (doctors, patient_B). This example shows that the correctness condition cannot be applied directly as a static relabeling rule.

2.4.3 A sound and complete relabeling rule

Now let us examine a relabeling rule that does work. If L_1 can be relabeled to L_2 under principal hierarchy P, it will be written as $P \vdash L_1 \sqsubseteq L_2$, an expression that is defined formally as follows:

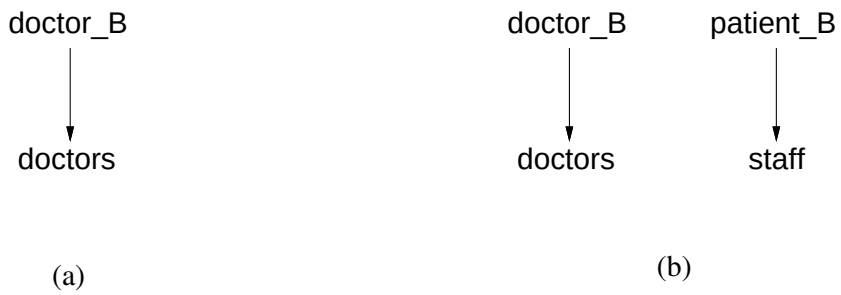

Figure 2.7: Two small principal hierarchies

Definition of the complete relabeling rule ($\sqsubseteq$)

$$
\begin{aligned}
(P \vdash L_1 \sqsubseteq L_2) &\equiv \forall(I \in L_1)\, \exists(J \in L_2)\; P \vdash I \sqsubseteq J \\
(P \vdash I \sqsubseteq J) &\equiv P \vdash \mathbf{o}J \succeq \mathbf{o}I \;\wedge\; \mathbf{r}J \subseteq \mathbf{rR}(I,P) \\
&\equiv P \vdash \mathbf{o}J \succeq \mathbf{o}I \;\wedge\; \forall(r_j \in \mathbf{r}J)\,[P \vdash r_j \succeq \mathbf{o}I \;\vee\; \exists(r_i \in \mathbf{r}I)\; P \vdash r_j \succeq r_i] \\
&\equiv P \vdash \mathbf{o}J \succeq \mathbf{o}I \;\wedge\; \mathbf{r}^+J \subseteq \mathbf{rR}(I,P) \\
&\equiv P \vdash \mathbf{o}J \succeq \mathbf{o}I \;\wedge\; \forall(r_j \in \mathbf{r}^+J)\, \exists(r_i \in \mathbf{r}^+I)\; P \vdash r_j \succeq r_i
\end{aligned}
$$

The rule for checking a relabeling from label L_1 to label L_2 is straightforward: for every policy I in L_1, there must be a corresponding policy J in L_2 that is at least as restrictive as I. If the policy J is at least as restrictive as I in the principal hierarchy P, it will be expressed as $P \vdash I \sqsubseteq J$, which also is defined formally in the figure. This condition will also be described informally as "J *covers* I"; informally, the relabeling rule says that any policy may be replaced by a policy that covers it.

The policy covering rule is stated four different ways. The second and fourth statements of the policy covering rule are simply expansions of the first and third, respectively, but it may not be obvious why the first and third definitions are equivalent. The first definition contains the condition $\mathbf{r}J \subseteq \mathbf{rR}(I,P)$, and the third replaces this condition with $\mathbf{r}^+J \subseteq \mathbf{rR}(I,P)$. The first definition implies the third because $P \vdash \mathbf{o}J \succeq \mathbf{o}I$ implies $o \in \mathbf{rR}(I,P)$, which implies $\mathbf{r}^+J \subseteq \mathbf{rR}(I,P)$ in conjunction with $\mathbf{r}J \subseteq \mathbf{rR}(I,P)$. The third definition implies the first because the statement $\mathbf{r}J \subseteq \mathbf{r}^+J$ transitively implies $\mathbf{r}J \subseteq \mathbf{rR}(I,P)$. Therefore, the two definitions are equivalent. When the complete relabeling rule is used in the following sections, the most convenient definition for each use will be selected.

The difference between this relabeling rule and the unsafe relabeling rule of Section 2.4.2 can be explained simply. The rule here says that for every policy I in L_1, a *single* policy J in L_2 must cover it. The earlier, unsafe rule effectively allows *multiple* policies in L_2 to cover a policy in L_1. When the principal hierarchy is extended, these policies can interact in unexpected ways and fail to cover I.

The binary relation $\sqsubseteq$ is defined on labels for any principal hierarchy P. The relation is a *pre-order*: it is transitive and reflexive, but not anti-symmetric, since two labels may be equivalent without being equal. If A and B are equivalent, we write $A \approx B$ to mean $A \sqsubseteq B \;\wedge\; B \sqsubseteq A$. For example, with the hierarchy of Figure 2.3, the labels $\{\mathsf{HMO}\colon \mathsf{doctors}\}$ and $\{\mathsf{HMO}\colon \mathsf{doctors},\ \mathsf{doctor_A}\}$ are equivalent. Every principal hierarchy generates a pre-order on labels, defining the legal relabelings.

The nature of the relabeling rule can be understood by considering the incremental relabelings that it permits. We have already seen in Section 2.3.1 that the subset relabeling rule can be characterized by two incremental relabeling rules. The new relabeling rule also allows the three additional relabelings described in Section 2.3.1 that the subset relabeling rule does not permit. The result is that this new rule allows an arbitrary sequence of any of the following five kinds of relabelings, each of which is sound individually:

- A reader may be dropped from some owner's reader set.
- A new owner may be added to the label, with an arbitrary reader set.
- A reader may be added if it acts for a member of the reader set.
- An owner may be replaced by an owner that acts for it.
- A reader may be added if it acts for the owner.

Interestingly, these incremental relabelings also capture *all* of the sound relabelings. In other words, the rule for $\sqsubseteq$ on page 41 is both sound and complete, and therefore is called the *complete relabeling rule*. The rule is complete in the sense that it exactly captures the set of valid relabelings, with respect to the static correctness condition defined in Section 2.4.2, and using our assumptions about the static checking environment. Now let us consider the proofs of these statements, which are given in Figures 2.8 though 2.10. (The relabeling rule has also been checked for soundness using Nitpick, a counter-example generator [JD96].)

Soundness. If the rule is sound, then if the relabeling rule holds for some principal hierarchy P, the correctness condition holds for all possible extensions P':

$$(P \vdash L_1 \sqsubseteq L_2) \rightarrow [\forall (P' \supseteq P)\, \mathbf{X}(L_1, P') \supseteq \mathbf{X}(L_2, P')]$$

A formal proof of this statement is given in Figure 2.8, using the definition of $\sqsubseteq$ for policies given on page 41. Some comments about the proof notation are in order. In this proof, the introduction of a hypothesis is indicated by an increase in the level of indentation. The notation $x \Rightarrow y$ is used in the right-hand columns when y is substituted for x in some statement. This step happens when a formula $\exists x\ P(x)$ is replaced by $P(y)$, where y is a fresh variable, as at step 8; it also happens when a formula $\forall x\ P(x)$ is instantiated on an existing expression y, producing $P(y)$, as at step 20.

The proof can be argued informally as follows. Soundness is proved by contradiction. Suppose that L_1 can be relabeled to L_2 in P, $P' \supseteq P$, and $\mathbf{X}(L_1, P')$ does not contain some flow (o, r). We will show that (o, r) cannot be in $\mathbf{X}(L_2, P')$ either, and that therefore the relabeling is safe. If (o, r) is not in $\mathbf{X}(L_1, P')$, there must be some policy I_1 in L_1 that suppresses it (*i.e.*, $r \notin \mathbf{rR}(I_1, P')$ and $P' \vdash \mathbf{o}I_1 \succeq o$). Because $P \vdash L_1 \sqsubseteq L_2$, there is a policy J_1 in L_2 that covers I_1: $\mathbf{r}^+ J_1 \subseteq \mathbf{rR}(I_1, P)$ and $P \vdash \mathbf{o}J_1 \succeq \mathbf{o}I_1$. Since $P \vdash \mathbf{o}J_1 \succeq \mathbf{o}I_1$, we have $P' \vdash \mathbf{o}J_1 \succeq \mathbf{o}I_1$, and transitively $P' \vdash \mathbf{o}J_1 \succeq o$.

Now, assume the flow (o, r) is a member of $\mathbf{X}(L_2, P')$. We will show that this generates a contradiction. Because $P' \vdash \mathbf{o}J_1 \succeq o$, there must be some reader r_2 in $\mathbf{r}^+ J_1$ such that $P' \vdash r \succeq r_2$. Since $\mathbf{r}^+ J_1 \subseteq \mathbf{rR}(I_1, P)$, r_2 must also be a member of $\mathbf{rR}(I_1, P)$. There must be another reader r_1 in $\mathbf{r}^+ I_1$ such that $P \vdash r_2 \succeq r_1$, which means that $P' \vdash r_2 \succeq r_1$, and transitively, $P' \vdash r \succeq r_1$. But this contradicts the statement that $r \notin \mathbf{rR}(I_1, P')$.

By contradiction, we conclude $(o, r) \notin \mathbf{X}(L_2, P')$. Because flows not in $\mathbf{X}(L_1, P')$ are not in $\mathbf{X}(L_2, P')$ either, every flow in $\mathbf{X}(L_2, P')$ is also in $\mathbf{X}(L_1, P')$. Therefore, the relabeling rule is sound.

$P \vdash L_1 \sqsubseteq L_2$	(Assumption)	(1)
$P' \supseteq P$	(Assumption/arbitrary P')	(2)
$(o, r) \in \mathbf{X}(L_2, P')$	(Assumption/arbitrary o, r)	(3)
$(o, r) \notin \mathbf{X}(L_1, P')$	(Assumption)	(4)
$\forall (I \in L_1)\ \exists (J \in L_2)\ P \vdash I \sqsubseteq J$	(1, Defn. of $\sqsubseteq$)	(5)
$\forall (I \in L_2)\ P' \vdash \mathbf{o}I \succeq o \rightarrow r \in \mathbf{rR}(I, P')$	(3, Defn. of $\mathbf{X}$)	(6)
$\exists (I \in L_1)\ P' \vdash \mathbf{o}I \succeq o\ \wedge\ r \notin \mathbf{rR}(I, P')$	(4, Defn. of $\mathbf{X}$)	(7)
$I_1 \in L_1\ \wedge\ P' \vdash \mathbf{o}I_1 \succeq o\ \wedge\ r \notin \mathbf{rR}(I_1, P')$	(7, $I \Rightarrow I_1$)	(8)
$\forall (r' \in \mathbf{r}^+ I_1)\ \neg (P' \vdash r \succeq r')$	(8, Defn. of $\mathbf{R}$)	(9)
$\exists (J \in L_2)\ P \vdash I_1 \sqsubseteq J$	(5, 8)	(10)
$P \vdash I_1 \sqsubseteq J_1$	(10, $J \Rightarrow J_1$)	(11)
$P \vdash \mathbf{o}J_1 \succeq \mathbf{o}I_1\ \wedge\ \mathbf{r}^+ J_1 \subseteq \mathbf{rR}(I_1, P)$	(11, Defn. of $\sqsubseteq$)	(12)
$P' \vdash \mathbf{o}J_1 \succeq \mathbf{o}I_1$	(2, 12)	(13)
$P' \vdash \mathbf{o}J_1 \succeq o$	(8, 13)	(14)
$P' \vdash \mathbf{o}J_1 \succeq o \rightarrow r \in \mathbf{rR}(J_1, P')$	(6, $I \Rightarrow J_1$)	(15)
$r \in \mathbf{rR}(J_1, P')$	(14, 15)	(16)
$\exists (r' \in \mathbf{r}^+ J_1)\ P' \vdash r \succeq r'$	(16, Defn. of $\mathbf{R}$)	(17)
$r_2 \in \mathbf{r}^+ J_1\ \wedge\ P' \vdash r \succeq r_2$	(17, $r' \Rightarrow r_2$)	(18)
$\forall (r_j \in \mathbf{r}^+ J_1)\ \exists (r_i \in \mathbf{r}^+ I_1)\ P \vdash r_j \succeq r_i$	(12, Defn. of $\mathbf{R}$)	(19)
$\exists (r_i \in \mathbf{r}^+ I_1)\ P \vdash r_2 \succeq r_i$	(18, 19, $r_j \Rightarrow r_2$)	(20)
$r_1 \in \mathbf{r}^+ I_1\ \wedge\ P \vdash r_2 \succeq r_1$	(20, $r_i \Rightarrow r_1$)	(21)
$P' \vdash r_2 \succeq r_1$	(2, 21)	(22)
$P' \vdash r \succeq r_1$	(18, 22)	(23)
$\neg (P' \vdash r \succeq r_1)$	(9, 21)	(24)
contradiction	(23, 24)	(25)
$(o, r) \in \mathbf{X}(L_1, P')$	(4, 25)	(26)
$\forall (o, r)\ (o, r) \in \mathbf{X}(L_2, P') \rightarrow (o, r) \in \mathbf{X}(L_1, P')$	(3, 26)	(27)
$\mathbf{X}(L_1, P') \supseteq \mathbf{X}(L_2, P')$	(27)	(28)
$\forall (P' \supseteq P)\ \mathbf{X}(L_1, P') \supseteq \mathbf{X}(L_2, P')$	(2, 28)	(29)
$P \vdash L_1 \sqsubseteq L_2 \rightarrow \forall (P' \supseteq P)\ \mathbf{X}(L_1, P') \supseteq \mathbf{X}(L_2, P')$	(1, 29)	(30)

Figure 2.8: Proof of soundness

Completeness. We must show the converse:

$$\left[\forall (P' \supseteq P)\ \mathbf{X}(L_1, P') \supseteq \mathbf{X}(L_2, P')\right] \rightarrow (P \vdash L_1 \sqsubseteq L_2)$$

We prove this statement by contradiction: if a relabeling is rejected by the rule ($L_1 \not\sqsubseteq L_2$), we can find a P' such that $P' \supseteq P$ but $\mathbf{X}(L_1, P') \not\supseteq \mathbf{X}(L_2, P')$. In other words, if a relabeling is rejected, it might result in a leak. This proof is given formally in Figures 2.9 and 2.10. Part 1 shows how to construct the new principal hierarchy P', and Part 2 shows that the relabeling is unsound in that principal hierarchy. The argument goes as follows:

If $\neg(P \vdash L_1 \sqsubseteq L_2)$, there must be some policy I_1 in L_1 such that for every policy J in L_2 where $\mathbf{o}J \succeq \mathbf{o}I_1$, $\mathbf{r}J \not\subseteq \mathbf{rR}I_1$. Consider an arbitrary such policy J in L_2. If there is no such J, the relabeling leaks even in P. For each such policy J, it must have a reader r_j where $r_j \in \mathbf{r}J$ but $r_j \notin \mathbf{rR}I_1$. We will now use the readers r_j of every such J to construct a principal hierarchy P' that extends P and results in a leak.

$\neg(P \vdash L_1 \sqsubseteq L_2)$	(Assumption)	(1)
$\exists(I \in L_1)\, \forall(J \in L_2)\, \neg(P \vdash I \sqsubseteq J)$	(1, Defn. of $\sqsubseteq$)	(2)
$I_1 \in L_1 \ \wedge\ \forall(J \in L_2)\, \neg(P \vdash I_1 \sqsubseteq J)$	(2, $I \Rightarrow I_1$)	(3)
$\forall(J \in L_2)\, P \vdash \mathbf{o}J \succeq \mathbf{o}I_1 \rightarrow \exists(r_j \in \mathbf{r}J)\, r_j \notin \mathbf{rR}(I_1, P)$	(3, Defn. of $\sqsubseteq$)	(4)
Now, let F be a Skolem function that maps from any J such that $J \in L_2$ and $P \vdash \mathbf{o}J \succeq \mathbf{o}I_1$ to a corresponding r_j, as described in step 4: $\forall(J \in L_2)\, P \vdash \mathbf{o}J \succeq \mathbf{o}I_1 \rightarrow FJ \notin \mathbf{rR}(I_1, P)$	(Define F)	(5)
$L_2' = \{J \mid J \in L_2 \ \wedge\ P \vdash \mathbf{o}J \succeq \mathbf{o}I_1\ \}$	(Define L_2')	(6)
Let r be a fresh principal with no relation in principal hierarchy P to any owners or readers in L_1 or L_2.	(Define r)	(7)
$R_{all} = (\bigcup_{I \in L_1} \mathbf{r}^+ I) \cup (\bigcup_{J \in L_2} \mathbf{r}^+ J)$	(Define R_{all})	(8)
$\forall(r' \in R_{all})\, \neg(P \vdash r \succeq r') \ \wedge\ \neg(P \vdash r' \succeq r)$	(7, 8)	(9)
$P' = P \cup \{(r, r') \mid \exists(J \in L_2')\, P \vdash FJ \succeq r'\}$	(Define P')	(10)
$\forall(r' \in R_{all})\, (P' \vdash r \succeq r' \rightarrow \exists(J \in L_2')\, P \vdash FJ \succeq r')$	(9, 10)	(11)

Figure 2.9: Proof of Completeness, part 1

Consider a principal hierarchy P' that is exactly like P, except that there is an additional principal r that in P is unrelated to any of the owners or readers in L_1 and L_2. It is assumed that new principals always can be added to the principal hierarchy after static checking, so such a principal always potentially exists. We form P' by adding a relation (r, r_j) for each r_j and taking the transitive closure:

$$P' = P \cup \{(r, r') \mid \exists r_j : (r_j, r') \in P\}$$

Note that since P is a pre-order, the relation (r, r) is already a member of P. Because P' is a transitive closure of a reflexive relation, it is a pre-order too. Using this definition for P', we find that $(\mathbf{o}I_1, r) \in \mathbf{X}(L_2, P')$ but $(\mathbf{o}I_1, r) \notin \mathbf{X}(L_1, P')$: the relabeling causes a leak in P'. Therefore, the relabeling rule is complete.

This completeness result can be strengthened further. This rule is complete even in the presence of negative information about relationships in the principal hierarchy. In fact, negative information is available in the else clause in the actsFor statement. Because actsFor tests whether one principal can act for another, in the body of the else clause it is known statically that the specified principal relationship does *not* exist. This static information could be used to establish an *upper bound* on the dynamic principal hierarchy, just as the static principal hierarchy establishes a lower bound. However, an upper bound is not useful in checking relabelings: the proof for completeness still holds in the presence of an upper bound on P', because we can choose an arbitrary r that is not mentioned in the upper bound.

2.4.4 Static checking

The label model must have certain lattice properties in order to support static checking. Checking of assignments has already been explained by the complete relabeling rule. But the labels being compared may be

$(\mathbf{o}I_1, r) \in \mathbf{X}(L_1, P')$	(Assumption)	(12)
$\forall(I \in L_1)\ P' \vdash \mathbf{o}I \succeq \mathbf{o}I_1 \rightarrow r \in \mathbf{rR}(I, P')$	(12, Defn. of $\mathbf{X}$)	(13)
$P' \vdash \mathbf{o}I_1 \succeq \mathbf{o}I_1 \rightarrow r \in \mathbf{rR}(I_1, P')$	(3, 13, $I \Rightarrow I_1$)	(14)
$r \in \mathbf{rR}(I_1, P')$	(13, Reflexivity)	(15)
$\exists(r' \in \mathbf{r}^+ I_1)\ P' \vdash r \succeq r'$	(15, Defn. of $\mathbf{R}$)	(16)
$r_2 \in \mathbf{r}^+ I_1\ \wedge\ P' \vdash r \succeq r_2$	(16, $r' \Rightarrow r_2$)	(17)
$\exists(J \in L_2')\ P \vdash FJ \succeq r_2$	(11, 17)	(18)
$J_1 \in L_2\ \wedge\ P \vdash \mathbf{o}J_1 \succeq \mathbf{o}I_1\ \wedge\ P \vdash FJ_1 \succeq r_2$	(6, 18, $J \Rightarrow J_1$)	(19)
$FJ_1 \notin \mathbf{rR}(I_1, P)$	(5, 19)	(20)
$\forall(r' \in \mathbf{r}^+ I_1)\ \neg(P \vdash FJ_1 \succeq r')$	(20, Defn. of $\mathbf{R}$)	(21)
$\neg(P \vdash FJ_1 \succeq r_2)$	(17, 21)	(22)
contradiction	(19, 22)	(23)
$(\mathbf{o}I_1, r) \notin \mathbf{X}(L_1, P')$	(12, 23)	(24)
$(\mathbf{o}I_1, r) \notin \mathbf{X}(L_2, P')$	(Assumption)	(25)
$\exists(I \in L)\ \mathbf{o}I \vdash \mathbf{o}I_1 \succeq\ \wedge\ r \notin \mathbf{rR}I\}$	(25, Defn. of $\mathbf{X}$)	(26)
$J_4 \in L_2\ \wedge\ P' \vdash \mathbf{o}J_4 \succeq \mathbf{o}I_1\ \wedge\ r \notin \mathbf{rR}(J_4, P')$	(26, $I \Rightarrow J_4$)	(27)
$\neg(P \vdash \mathbf{o}J_4 \succeq \mathbf{o}I_1)$	(Assumption)	(28)
$(\mathbf{o}J_4, \mathbf{o}I_1) \in (P' - P)$	(27, 28)	(29)
$\mathbf{o}J_4 = r$	(10, 29)	(30)
contradiction	(7, 30)	(31)
$P \vdash \mathbf{o}J_4 \succeq \mathbf{o}I_1$	(28,31)	(32)
$r_4 = FJ_4$	(5, 31, define r_4)	(33)
$r_4 \in \mathbf{r}^+ J_4\ \wedge\ r_4 \notin \mathbf{rR}(I_1, P)$	(4, 5, 33)	(34)
$P' \vdash r \succeq r_4$	(10, 34)	(35)
$\forall(r' \in \mathbf{r}^+ J_4)\ \neg(P' \vdash r \succeq r')$	(27, Defn. of $\mathbf{R}$)	(36)
$\neg(P' \vdash r \succeq r_4)$	(34, 36)	(37)
contradiction	(35, 37)	(38)
$(\mathbf{o}I_1, r) \in \mathbf{X}(L_2, P')$	(25, 38)	(39)
$\mathbf{X}(L_1, P') \not\supseteq \mathbf{X}(L_2, P')$	(24, 39)	(40)
$\exists(P' \supseteq P)\ X(L_1, P') \not\supseteq \mathbf{X}(L_2, P')$	(10, 40)	(41)
$\neg(P \vdash L_1 \sqsubseteq L_2) \rightarrow \exists(P' \supseteq P)\ X(L_1, P') \not\supseteq \mathbf{X}(L_2, P')$	(1, 41)	(42)
$[\forall(P' \supseteq P)\ X(L_1, P') \supseteq \mathbf{X}(L_2, P')] \rightarrow (P \vdash L_1 \sqsubseteq L_2)$	(42)	(43)

Figure 2.10: Proof of Completeness, part 2

the results of joins (to account for computations), and meets (which occur during the process of automatic label inference). Therefore, join and meet also must be defined. Join was defined earlier in Section 2.1.4, but it is revisited here in the context of the new definition of the relation $\sqsubseteq$.

Labels form a pre-order rather than a lattice or even a partial order, because two labels can be equivalent without being equal. However, labels do preserve the important properties of a lattice that make static reasoning about information flow feasible: any pair of elements possesses least upper bounds and greatest lower bounds. Because labels form a pre-order, these bounds are equivalence classes of labels rather than single labels. The set of labels also has a bottom element ($\bot$), which is the label $\{\}$. For mathematical completeness, the set of labels is considered to have a top element, $\top$, which is more restrictive than any other label. In addition, the join and meet operations distribute over each other.

The definitions of join and meet have the desirable properties that join and meet are easy to evaluate and that the resulting labels are easy to deal with when applying the complete relabeling rule.

Join. Using the new definition for the relation $\sqsubseteq$, we can now revisit the definition for the join, or least upper bound, of two labels. The join is useful in assigning a label to the result of an operation that combines several values, such as adding two numbers. The result of adding two numbers ought in general to be restricted at least as much as the numbers being added. However, we would also like not to restrict the sum unnecessarily; therefore, it is assigned the *least* restrictive label that is at least as restrictive as both input labels. In a lattice, there is a unique least label; however, uniqueness is not important for our purposes. Any label within an equivalence class is acceptable as long as it can be relabeled to every label that is at least as restrictive as the input labels.

The join of two label expressions can be defined quite simply; the definition of Section 2.1.4 still holds with the complete relabeling rule:

Definition of join

$$L_1 \sqcup L_2 = L_1 \cup L_2$$

The following are examples of join expressions, where A, B, and C are principals unrelated by the acts-for relation:

$$\{A : B\} \sqcup \{B : C\} = \{A : B; B : C\} \tag{2.1}$$

$$\{A : B\} \sqcup \{A : B, C\} = \{A : B\} \tag{2.2}$$

$$\{A : B\} \sqcup \{A : C\} = \{A : B; A : C\} \tag{2.3}$$

After doing a join, a compiler often can simplify the label expression by removing redundant policies, so that future checking steps run more efficiently. This simplification has been performed in the second example, whereas neither policy is redundant in the third example. A policy is redundant if the relabeling rules behave identically for the label regardless of whether the policy is present. One policy I makes another policy J redundant in static principal hierarchy P if I covers J (that is, $P \vdash J \sqsubseteq I$). In the second join example, the relation $\{A : B, C\} \sqsubseteq \{A : B\}$ is true, so the former policy is redundant in the join result.

We can now see why it is important that owners be repeatable in labels: it completes the lattice of equivalence classes. If repeated owners were not allowed, there would be no least upper bound for many pairs of labels. Consider the third example again, but disallowing repeated owners. If A' is another principal with $A' \succeq A$, and it is the only such principal, then the least restrictive labels that both $\{A\text{: } B\}$ and $\{A\text{: } C\}$ could be relabeled to would include $\{A\text{: }\}$, $\{A\text{: } B;\ A'\text{: } C\}$, and $\{A'\text{: } B;\ A\text{: } C\}$, none of which can be relabeled to any other. There would be three upper bounds in different equivalence classes, but no *least upper bound* for these two labels.

The join operation just described produces the least upper bound of two labels. This can be seen by interpreting a join result as a set of flows, in an extended principal hierarchy P'. It follows directly from the definition of $\mathbf{X}$ that for all such hierarchies P',

$$\mathbf{X}(A \sqcup B, P') = \mathbf{X}(A, P') \cap \mathbf{X}(B, P')$$

This result follows because $\mathbf{X}L$ takes the intersection of the sets of flows generates by each of the policies in the label L. This equation means that there is no label less restrictive than $A \sqcup B$ that both A and B can be relabeled to. The result of the join operator can be relabeled to every label that both A and B can be relabeled to, and every label that has this property is in the same equivalence class as the result of the join operator, since it has the same interpretation as a set of flows. This equivalence class defines the least upper bound of the two labels.

Declassification. In Section 2.1.5, the rule for declassification was presented as follows: the label L_1 may be relabeled to L_2 as long as $L_1 \sqsubseteq L_2 \sqcup L_A$, where L_A is a label containing exactly the policies of the form $\{p :\}$ for every principal p that the process can act for. This definition continues to have the intended effect with the complete relabeling rule, and can be performed statically if there is a static notion of the process authority, which is called the *static authority* here.

Because L_1 must be capable of relabeling to $L_2 \sqcup L_A$, every policy in L_1 must be covered by some policy in $L_2 \sqcup L_A$. However, the policies in L_1 that are owned by a principal in the static authority are automatically covered by policies in L_A. Only policies in L_1 *not* owned by any principal in the static authority need be covered by L_2, so the effect is that policies in L_1 that are owned by the static authority may be weakened arbitrarily by declassification.

Reasoning about joins. Policies in a join independently can be relabeled or declassified. This property is important because it allows checking of code that is generic with respect to some of the labels that appear in it. In the case of declassification, there are no surprises for the declassifying principal: the set of flows that are added by declassifying a join is always a subset of the set of flows that would be added by declassifying the individual policies. There are no interactions between the two parts of the join that create new, unexpected flows.

For example, if label L_1 can be relabeled to L_2, then $L_1 \sqcup L_3$ can be relabeled to $L_2 \sqcup L_3$, regardless of what L_3 is. L_3 may be an unknown label, or even a label that is determined at run time, without invalidating the relabeling. Similarly, if L_1 can be declassified to L_2, then $L_1 \sqcup L_3$ can be declassified to $L_2 \sqcup L_3$. These relabelings and declassifications work because the join guarantees that all policies in L_3 will be respected.

Meet. The meet or greatest lower bound of two labels is the most restrictive label that can be relabeled to both of them. The meet of two labels is not produced by computations during the program's execution, but it is useful in defining algorithms for automatic label inference [DD77, ML97]. The meet is useful for

Definition of meet

$$\frac{A = \bigsqcup_i a_i \qquad B = \bigsqcup_j b_j}{A \sqcap B = \bigsqcup_{i,j} (a_i \sqcap b_j)}$$

Figure 2.11: The meet of two labels

inferring the labels of inputs automatically, just as the join is useful for producing the labels of outputs. For example, in the following code, the most restrictive label x could have can be expressed by using a meet:

```
int x;
int{A} y;
int{B} z;
y = x;
z = x;
```

In this example, the variables y and z have labels of A and B respectively. The variable x can be assigned any label C so long as it can be relabeled to both A and B. Therefore, $\mathsf{A} \sqcap \mathsf{B}$ is an upper bound on the label for x. The algorithm for inferring variable labels that is described in Chapter 5 uses a succession of meet operations in this fashion, refining unknown variable labels downward until either all variables have consistent assignments or a contradiction is reached.

To construct the meet of two labels, let us first consider the meet of two policies J and K. If there is no statically known relation between the owners of these policies, the meet is $\{\}$ because no other label can be relabeled to both J and K. This result is obtained when either J or K is uninterpreted (*e.g.*, is a label parameter), or when both have known owners but no relationship is known statically to exist between them (by some containing actsFor statement). Otherwise, suppose that $J = \{o : r_1 \ldots r_n\}$ and $K = \{o' : r'_1 \ldots r'_{n'}\}$. If o' can act for o or they are equal, the meet of the two policies is $\{o : \ r_1 \ldots r_n, r'_1 \ldots r'_{n'}\}$. If o' is equivalent but not equal to o, the meet of the two policies is $\{o : \ r_1 \ldots r_n, r'_1 \ldots r'_{n'};\ o' : \ r_1 \ldots r_n, r'_1 \ldots r'_{n'}\}$. This label is equivalent to other, simpler labels such as $\{o : \ r_1 \ldots r_n, r'_1 \ldots r'_{n'}\}$, but it is chosen because it is symmetrical with respect to the two policies.

Now, consider the meet of two arbitrary labels. Because a label containing several policies is the join of these policies, the meet can be computed by distributing the meet over both joins. The result of the meet, shown in Figure 2.11, is the join of all pairwise meets of policies, using one policy from each label. In the figure, labels A and B are composed of policies a_i and b_j, respectively. Some of these pairwise meets $a_i \sqcap b_j$ may produce the label $\{\}$, which of course can be dropped from the join.

As with join, the validity of this formula for meet can be seen by using the interpretation function $\mathbf{X}$. If P' is some extension of the principal hierarchy used to compute the meet of labels A and B, then the following relation holds:

$$\mathbf{X}(A \sqcap B, P') \supseteq \mathbf{X}(A, P') \cup \mathbf{X}(B, P')$$

Unlike the formula for join, the definition of meet does not always produce the most restrictive label for all possible extensions P', though it produces the most restrictive label existing in the static principal hierarchy. This result occurs because the rule for joining two policies returns $\{\}$ when the owners are not known statically to have a relationship, though in the run-time hierarchy, a relationship may exist. The practical effect is that label inference must be conservative in some cases. These cases do not seem to be a significant problem since even explicit label declarations do not work in those cases: any explicitly declared label more restrictive than $\{\}$ would cause static checking to fail.

Distribution properties. It can also be shown straightforwardly that join and meet distribute over each other in the expected way for distributive lattices, producing equivalent labels:

$$\begin{aligned} A \sqcap (B \sqcup C) &\approx (A \sqcap B) \sqcup (A \sqcap C) \\ A \sqcup (B \sqcap C) &\approx (A \sqcup B) \sqcap (A \sqcup C) \end{aligned}$$

This means that a static checker doing label inference as described elsewhere [ML97] can rely on the properties of meet and join to simplify label expressions.

The first equation follows trivially from the definition of meet:

$$\begin{aligned} A \sqcap (B \sqcup C) &= (\bigsqcup_i a_i) \sqcap ((\bigsqcup_j b_j) \sqcup (\bigsqcup_k c_k)) \\ &= (\bigsqcup_{i,j} a_i \sqcap b_j) \sqcup (\bigsqcup_{i,k} a_i \sqcap c_k) \\ &= (A \sqcap B) \sqcup (A \sqcap C) \end{aligned}$$

Proving the second equation is only slightly harder:

$$\begin{aligned} (A \sqcup B) \sqcap (A \sqcup C) &= ((\bigsqcup_i a_i) \sqcup (\bigsqcup_j b_j)) \sqcap ((\bigsqcup_i a_i) \sqcup (\bigsqcup_k c_k)) \\ &= (\bigsqcup_{i,i'} a_i \sqcap a'_i) \sqcup (\bigsqcup_{i,j} a_i \sqcap b_j) \sqcup (\bigsqcup_{i,k} a_i \sqcap c_k) \sqcup (\bigsqcup_{j,k} b_j \sqcap c_k) \\ &= (\bigsqcup_i a_i) \sqcup (\bigsqcup_{i \neq i'} a_i \sqcap a'_i) \sqcup (\bigsqcup_{i,j} a_i \sqcap b_j) \sqcup (\bigsqcup_{i,k} a_i \sqcap c_k) \sqcup (\bigsqcup_{j,k} b_j \sqcap c_k) \\ &\approx (\bigsqcup_i a_i) \sqcup (\bigsqcup_{j,k} b_j \sqcap c_k) \\ &= A \sqcup (B \sqcap C) \end{aligned}$$

The fourth step is a bit tricky, relying on an absorption property for policies a and b: $a \sqcup (a \sqcap b) \approx a$. Because of this property, the term $(\bigsqcup_i a_i)$ makes redundant other terms containing meets with a_i.

The absorption property follows directly from the definition of meet for policies, because in any label containing both the policies a and $a \sqcap b$, the latter term will be redundant. To see why, consider the three

possible cases for the result of the expression $a \sqcap b$, where $a = \{o : r_1, \ldots, r_n\}$ and $b = \{o' : r'_1, \ldots, r'_{n'}\}$. In the first case, the meet may be $\{\}$, in which case the absorption property holds since $a \sqcup \{\} = a$. The second case is $o = o'$ or $o' \succeq o$ (but o and o' are not equivalent); in that case,

$$a \sqcup (a \sqcap b) = \{o : r_1, \ldots, r_n;\ o : r_1, \ldots, r_n, r'_1, \ldots, r'_n\} \approx a$$

because the second policy is weaker than or equal to the first. The absorption property also holds in the third case, where o and o' are equivalent:

$$\begin{aligned} a \sqcup (a \sqcap b) &= \{o : r_1, \ldots, r_n;\ o : r_1, \ldots, r_n, r'_1, \ldots, r'_n;\ o' : r_1, \ldots, r_n, r'_1, \ldots, r'_n\} \\ &\approx a \end{aligned}$$

Again, the second and third policies are redundant.

2.5 Output channels

It is assumed the private information is not leaked by computation, even computation performed by untrusted programs, as long as the label discipline is observed. Information is leaked only through transmission outside the region where labels are enforced. Note that the region of enforcement may include many computers and networks, but that there is no control over humans, who may choose to violate the rules. The reader-set component of an output channel policy is the characterization of the part of the outside world that the output channel leads to. It is essential that the output channel be labeled properly, because information is transmitted through an output channel based on whether its label can be relabeled to that of the output channel.

Because the output channel has a decentralized label, there does not need to be any universally accepted notion of the readers on an output channel. The effect of the relabeling rules is that a principal p effectively accepts the reader set of a policy only if the owner of the policy acts for p. In fact, the process of creating labeled output channels can be described rather neatly with almost no additional mechanism. The only additional mechanism needed is the ability to create a *raw output channel*: an output channel with the label $\{\}$. Data can be written to such a channel only if it has no privacy restrictions, so the creation of such a channel is a safe operation: the channel cannot leak any private data.

Labeled output channels can be constructed on top of raw channels in a straightforward manner. A labeled output channel is simply a function that accepts data with label L and performs the following three steps:

1. an optional transformation of the data, such as encryption with a public key,
2. declassification of the transformed data to the label $\{\ \}$,
3. and transmission over the raw output channel.

Step 2 can be performed only if a function runs with the authority of all the owners of the label L. In other words, the labeling system ensures that the owners of all the policies that the output channel claims to enforce must have granted their authority to the process that creates the output channel; these owners explicitly trust the output channel. How these owners decide to grant their authority to the output channel is outside the scope of this thesis, but the granting of authority should be based on the belief that the channel delivers data to at most the listed readers. Two possible reasons for this belief are the following:

- The physical connection that the raw channel models is known to be a secure connection to at most the listed readers.
- Data being sent on the channel is encrypted in such a way that only the intended recipients will be able to decrypt it.

2.6 Generalizing labels and principals

There are several interesting ways to extend the basic label model described so far. In this section, a few of them will be considered.

2.6.1 Integrity policies

We have seen that the decentralized label model supports labels containing privacy policies. All of the structure that has been developed to this point can now be applied to integrity policies. Integrity policies [Bib77] are the dual of privacy policies. Just as privacy policies protect against data being *read* improperly, even if it passes through or is used by untrusted programs, integrity policies protect data from being improperly *written*. An integrity label keeps track of all the *sources* that have affected a value, even if those sources only affect the value indirectly. It prevents untrustworthy data from having an effect on trusted storage.

The structure of a *decentralized integrity policy* is identical to that of a decentralized privacy policy. It has an *owner*, the principal for whom the policy is enforced, and a set of *writers*: principals who are permitted to affect the data. A label may contain a number of integrity policies on behalf of various owners. The intuitive meaning of an integrity policy is that it is a guarantee of quality. A policy $\{o : w_1, w_2\}$ is a guarantee by the principal o that only w_1 and w_2 will be able to affect the value of the data. The most restrictive integrity label is the label containing no policies, $\{\ \}$. This is the label that provides no guarantees as to the contents of the labeled value, and can be used as the data input only when the receiver imposes no integrity requirements.

Using an integrity label, a variable can be protected against improper modification. For example, suppose that a variable has a single policy $\{o : w_1, w_2\}$. A value labeled $\{o : w_1\}$ may be written to this variable, because that value has been affected only by w_1, and the label of the variable permit w_1 to affect it. If the value were labeled $\{o : w_1, w_3\}$, the write would not in general be permitted, because the value

was affected by w_3, a principal not mentioned as an allowed writer in the label of the variable. (It would be permitted if $w_3 \succeq w_2$.) Finally, consider a value labeled $\{o : w_1; o' : w_3\}$. In this case, the write is permitted, because the first policy says that o believes only w_1 has affected the value. That the second policy exists on behalf of o' does not affect the legality of the write to the variable; it is a superfluous guarantee of quality.

Just as with privacy policies earlier, assignment relabels the value being copied into the variable, and to avoid violations of integrity, the label of the variable must be more restrictive than the label of the value. In the preceding sections, a relabeling rule has been developed for privacy. We will now see that this work also can be applied to integrity labels. In Section 2.4.3, it was said that any legal relabeling for privacy policies can be characterized by a set of five incremental relabelings. This characterization was attractive because it is easier to judge the correctness of an incremental relabeling. For an integrity label, there are also five incremental relabelings:

- *A writer may be added to a policy.* This addition is safe because an additional writer in an integrity policy is an additional warning of contamination and can make the value only more restricted in subsequent use.

- *A policy may be removed.* An integrity policy may be thought of as an assurance that at most the principals in a given set (the writers) have affected the data. Removing such an assurance is safe and restricts subsequent use of the value.

- *In a policy, a writer w' may be replaced by a writer w that it acts for.* Because w' has the ability to act for w, a policy permitting w as a writer permits both w and w' as writers, whereas a policy permitting w' does not, in general, permit w. Therefore, replacing w' by w really adds writers, a change that is safe.

- *A policy J may be added that is identical to an existing policy I except that $\mathbf{o}I \succeq \mathbf{o}J$.* The new policy offers a weaker integrity guarantee than the existing one, so the value is not made less restrictive by the addition of this policy.

- *Any principal that acts for the owner of a policy may be removed as a writer.* The most restrictive integrity policy that any principal o would want to express is that only o (or principals that can act for o) could write to the labeled variable. Therefore, the owner of a policy (and any principal that acts for the owner) is implicitly considered to be a writer, and need not be expressed explicitly as such. This rule is the equivalent of self-authorization for privacy policies.

These five kinds of relabelings turn out to capture exactly the inverse of the relabelings that are allowed by the incremental rules for privacy labels, described in Section 2.4.3. To see why, consider each of the incremental rules above in turn. The effect of each of these rules can be reversed by applying the privacy rules:

- *Adding a writer.* The privacy rules permit removing a reader.
- *Removing a policy.* The privacy rules permit adding an arbitrary policy.
- *Replacing a writer w' with w, where $w' \succeq w$.* The privacy rules allow a reader r' to be added is r if also a reader, with $r' \succeq r$. The reader r then can be removed.
- *Adding a policy J identical to an existing policy I, with an inferior owner ($\mathbf{o}I \succeq \mathbf{o}J$).* The privacy rules allow the owner of J to be replaced with $\mathbf{o}I$, making the two policies identical.
- *Removing the owner of a policy from the writer set.* The owner of a policy may be added to the reader set of a policy.

Similarly, the effect of each of the privacy rules may be reversed by applying the integrity rules.

If L_1 and L_2 are privacy labels, and L_1 can be relabeled to L_2, then there is a sequence of incremental privacy relabelings that converts L_1 into L_2. Suppose that L_1' and L_2' are integrity labels with the same form as L_1 and L_2. There must be a sequence of incremental integrity relabelings leading from L_2' to L_1'. Therefore, if $L_1 \sqsubseteq L_2$, then $L_2' \sqsubseteq L_1'$. The ordering relations for privacy and integrity labels are perfect duals.

This property means that all of the rules for integrity can be derived directly from the rules for privacy. We have just seen that for privacy labels L_1 and L_2 and corresponding integrity labels L_1' and L_2',

$$P \vdash L_1 \sqsubseteq L_2 \longleftrightarrow P \vdash L_2' \sqsubseteq L_1'$$

This logical equivalence defines the complete relabeling rule for integrity in terms of the corresponding rule for privacy that was given in Section 2.4.3.

The rules for the meet and join of two integrity labels are similarly expressed in terms of their privacy label counterparts. These rules follow directly from the dual relationship of the ordering relation $\sqsubseteq$ for the two kinds of labels.

$$\begin{aligned} L_3 \approx L_1 \sqcup L_2 &\quad\longleftrightarrow\quad L_3' \approx L_1' \sqcap L_2' \\ L_3 \approx L_1 \sqcap L_2 &\quad\longleftrightarrow\quad L_3' \approx L_1' \sqcup L_2' \end{aligned}$$

Operationally, the *meet* of two integrity labels is performed by simply concatenating their policies, as if the *join* of the corresponding privacy labels were being evaluated, and the *join* of integrity labels corresponds to the *meet* of the corresponding privacy labels. In other words, the meet of two labels is the most restrictive label that is less restrictive than (contains all the policies of) the labels, so it is performed by taking a union of the policies. Similarly, the join of two integrity labels can contain only policies enforced by both labels.

Declassification. An analogue to declassification also exists for integrity labels. For privacy labels, the declassification mechanism allows privacy policies to be removed in cases where reasoning outside the scope of strict dependency analysis (as in the tax-preparer example) suggests that the policy is overly strict. The dual action for integrity policies is to *add* new integrity policies in situations where the data has higher integrity than strict dependency analysis might suggest. If a principal adds a new integrity policy to a label, or removes writers from an existing policy, it represents a vote of confidence in the integrity of the data, and allows that data to be used more freely subsequently. Just as with declassification for privacy, however, the reasons why a principal might choose to do so lie outside the scope of this model.

Adding new policies is safe because the new policy may be added only if the current process of the authority to act for the owner of the policy. Other principals will not be affected unless they trust the policy owner (and by extension, the process performing the declassification) to act for them.

Declassification can be described more formally: declassification of integrity label L_1 to a label L_2 is permitted when $L_2 \sqcap L_A^I \sqsubseteq L_1$, where L_A^I is an integrity label in which there is a policy for every principal in the authority of the process. Each such policy lists all principals in the system as writers. Note the duality of this rule to the rule for declassification of privacy labels.

Code labels. Integrity labels do introduce one new issue: code can damage integrity without access to any extra labeled resource. For example, the routine alleged to add two numbers might perform a different computation, destroying integrity. To keep track of this effect, an integrity label must be assigned to each function in a program, and joined with any value computed by the function. In a program expression like $f(x, y)$, all three sub-expressions (f, x, and y) have an associated integrity label.

Code labels could be applied to privacy as well, and would have some utility in the case where the code itself were a secret. For both privacy and integrity the natural default code label is $\{\ \}$. However, this default label has quite different effects for the two kinds of labels. The label $\{\ \}$ is the *least* restrictive privacy label and has no effect when joined with another label. As an integrity label, it is the *most* restrictive label, since it offers no guarantee about the integrity of the data computed by the function.

Because an integrity label offers a quality guarantee, some authority is needed to label code with it—specifically, the authority to act for the owners of any integrity policies in the label. One would expect that the owner of the integrity label typically would not be the author of the code. Instead, the author would appear as a *writer* in the integrity label.

2.6.2 Combining integrity and privacy

The set of all privacy labels, which will be called S_P, and the set of all integrity labels (S_I), each form a pre-order with ordering relations $\sqsubseteq_P$ and $\sqsubseteq_I$, respectively. These two kinds of labels can be used to generate a system of combined labels that enforce privacy and integrity constraints simultaneously.

A combined label is written as a sequence of privacy and integrity policies. To disambiguate the two kinds of policies, privacy policies are written in the form $\{o \rightarrow r_1, r_2, \ldots\}$, and integrity policies are written

in the form $\{o \leftarrow w_1, w_2, \ldots\}$, where the arrows suggest the direction of information flow. A combined label can be considered as a pair $\langle L_P, L_I \rangle$, which is a member of the set $S_P \times S_I$. The ordering relation on combined labels and the join and meet operations are easily defined in the usual way for product spaces of ordered sets:

$$\begin{array}{rcl} \langle L_P, L_I \rangle \sqsubseteq \langle L'_P, L'_I \rangle & \equiv & L_P \sqsubseteq_P L'_P \wedge L_I \sqsubseteq_I L'_I \\ \langle L_P, L_I \rangle \sqcup \langle L'_P, L'_I \rangle & = & \langle L_P \sqcup_P L'_P, L_I \sqcup_I L_I \rangle \\ \langle L_P, L_I \rangle \sqcap \langle L'_P, L'_I \rangle & = & \langle L_P \sqcap_P L'_P, L_I \sqcap_I L_I \rangle \end{array}$$

Similarly, a combined label $\langle L_P, L_I \rangle$ can be declassified to another combined label $\langle L'_P, L'_I \rangle$ if both components can be declassified. Here, L_A^P is used to refer to the label called L_A earlier.

$$\frac{\begin{array}{c} L_P \sqsubseteq_P (L'_P \sqcup L_A^P) \\ (L_I \sqcap_I L_A^I) \sqsubseteq_I L'_I \end{array}}{\langle L_P, L_I \rangle \text{ can be declassified to } \langle L'_P, L'_I \rangle}$$

In summary, for all of these rules for combined labels, the integrity and privacy policies are independently enforced and do not interact.

2.6.3 Generalizing principals and the acts-for relation

Principals and the principal hierarchy are more powerful concepts that might be apparent. Principals can be used to represent a broader range of entities than users, groups, and roles. When used as readers or writers in policies, principals may also represent input and output devices, user-defined privacy or integrity levels, and compartments. Also, it is not necessary that owners and readers (and writers) are the same kinds of entities.

Using the notation of Section 2.6.2, an external connection to a user A through a cable might be represented as an output channel with the single-policy privacy label $\{\mathsf{root} \rightarrow \mathsf{A}, \mathsf{cable}\}$, where root is a trusted principal. Information that is not marked as readable by the cable principal will be prevented from transmission on the cable. Having the cable principal as one of the readers of the output channel is a way of reflecting the danger that the cable may leak information in some way. Similarly, if the cable is used as an input channel it might be assigned the integrity policy $\{\mathsf{root} \leftarrow \mathsf{A}, \mathsf{cable}\}$ to indicate that data from this input channel passed through the cable on its way into the system and was conceivably damaged in transit.

The principal hierarchy can be used to establish categories of such devices. If the principal cable acts for another principal secure-channel, it effectively becomes one of the secure-channel devices, and will interoperate with labels that are expressed in terms of secure-channel rather than in terms of specific devices. Also, a user can express trust in secure channels by allowing the secure-channel principal to act for the user's principal; this trust will allow any data that lists the user as a reader to be sent to the channel, assuming the policy owners have the required degree of trust. We will see in a moment that less trust in the secure-channel principal is needed than one might expect.

Users can establish their own abstract privacy levels by introducing new role principals to represent these privacy levels. The acts-for relation among these principals expresses the information flows allowed among

the levels, in the absence of the use of declassification. For example, a user might have two jobs whose information should by default be kept compartmentalized. Suppose Amy is both a manager and a committee chair. Her compartmentalization concerns are addressed by introducing two new principals: Amy_manager and Amy_chair, as shown in Figure 2.12. As long as Amy does not assume the full power of the Amy principal, data will not be allowed to move between the compartments. However, the declassification mechanism is always available for explicit use in cases where she deems it appropriate. Roles can be introduced to represent user-specific integrity levels in a similar fashion.

One unsatisfactory but repairable aspect of the model described so far is that the acts-for relation appears to give too much power. For example, the approach that has been described for modeling a group principal is for each of the members of the group to act for the group principal. This structure allows group members to read anything that can be read by the group principal. However, it also gives them the additional power to declassify and redistribute publicly anything *owned* by the group. This added power violates the principle of least privilege.

What we would like is to introduce different kinds of acts-for relations, so that group members have the power to read group data but not to declassify it. Suppose that Amy and Bob are group members; Amy is a group administrator with the power to declassify data owned by the group, whereas Bob is a group member who is able merely to read data that can be read by the group. This scenario can be modeled as shown in Figure 2.13. As the diagram shows, Bob has the right to *read for* the group, whereas Amy has the full power to act for the group, which implies the ability to read for and also to *declassify for* the group. Both of these new, weaker relations are transitive: if x reads for y and y reads for z, then x reads for z; declassifies-for behaves similarly.

To understand the implications of the extended acts-for relations, it is not necessary to develop a new theory of labels, because a system containing extended acts-for relations can be translated into the original model. A principal hierarchy P_E supporting these extended relations can be translated into as another principal hierarchy P that contains only the simple acts-for relation; a label that names principals in P_E also may be translated into a corresponding label that names principals in P. The semantics for the extended system P_E are determined simply by applying the existing rules for relabeling, join, and meet to the translated forms of the labels in P.

The translation from P_E to P is performed as follows. Each principal p in P_E corresponds to three principals in P named p_o, p_r, and p_w, with the acts-for relations shown in Figure 2.14: both p_r and p_w

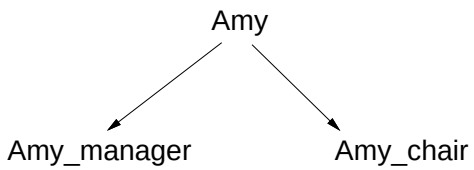

Figure 2.12: Compartments through hierarchy

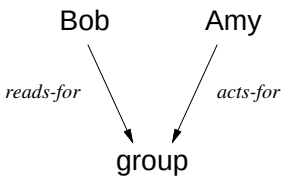

Figure 2.13: Modeling a group

act for p_o. As the names suggest, each of the principals p_o, p_r, and p_w is used in only one of the three possible positions it might occupy in a label: as an owner, reader, or writer, respectively. A privacy label $\{\mathsf{Bob}\text{: }\mathsf{group}\}$, which mentions principals in P, is translated to the label $\{\mathsf{Bob}_o\text{: }\mathsf{group}_r\}$, because Bob is being used as an owner, and group as a reader. Because p_r always acts for p_o, a principal is automatically authorized to read data that it owns. Process authority also must be translated from P_E to P. A process running with authority of p actually runs with the authority of the principal p_o; the authority of the principals p_r and p_w is never given to a process.

Figure 2.15 shows how the principal hierarchy of Figure 2.13 is translated into the simpler model. In the figure, Bob has power only over the principal group_r, giving him the right to read. The ability of Amy to act for both the group_o and group_r principals means that she both can declassify data owned by the group and can read data readable by the group.

There is a third relationship that Amy can have to the group: the *self-reads* relationship, which means that Amy can read any data owned by the group. By itself, this relationship does not mean that Amy can read data readable by the group, or that she can declassify group data. The self-reads relationship is weaker than the other two relationships, because the abilities of Amy to read for and to declassify for group each imply by transitivity that Amy self-reads group.

These three different kinds of acts-for relations in the P_E hierarchy between two principals p' and p are translated as follows to the P hierarchy:

The principal p' *reads for* the principal p.	$p'_r \succeq p_r$
The principal p' *declassifies for* the principal p.	$p'_o \succeq p_o$
The principal p' *is self-authorized to read for (self-reads)* p.	$p'_r \succeq p_o$

These three relations also correspond to three of the incremental relabeling rules defined in Section 2.3.1: reads-for corresponds to the rule for adding readers, declassifies-for corresponds to the rule for replacing owners, and self-reads corresponds to the rule for self-authorization.

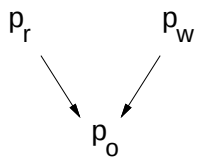

Figure 2.14: Splitting principals

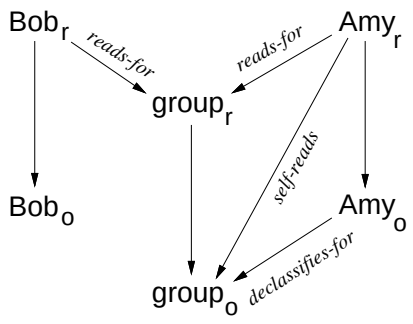

Figure 2.15: Modeling a group with split principals

We can see from this that the extended principal hierarchy P_E supports five new relations that are indicated by writing appropriate subscripts after the $\succeq$ sign.

- *declassifies-for* $p' \succeq_o p \equiv p'_o \succeq p_o$
- *reads-for*: $p' \succeq_r p \equiv p'_r \succeq p_r$
- *writes-for*: $p' \succeq_w p \equiv p'_w \succeq p_w$
- *self-reads*: $p' \succeq_{ro} p \equiv p'_r \succeq p_o$
- *self-writes*: $p' \succeq_{wo} p \equiv p'_w \succeq p_o$

The three relations that affect privacy (declassifies-for, reads-for, and self-reads) correspond exactly to the three ways that the $\succeq$ relation is used in the second definition of the relation $\sqsubseteq$ on page 41. In that definition, the expression $\mathbf{o}J \succeq \mathbf{o}I$ compares two owners, and is therefore a test of the declassifies-for relation. The expression $r_j \succeq \mathbf{o}I$ compares a reader to an owner, so it is a test of the self-reads relation. Finally, $r_j \succeq r_i$ compares two readers, and is a test of the reads-for relation. The complete relabeling rule therefore can be expressed in the P_E system in such a way that enforcing this new rule directly has the same effect as enforcing the original complete relabeling rule on the translated labels. The new version of the complete relabeling rule is as follows:

$$P \vdash I \sqsubseteq J \equiv (P \vdash \mathbf{o}J \succeq_o \mathbf{o}I) \ \wedge \ \forall(r_j \in \mathbf{r}J) \ [P \vdash r_j \succeq_{ro} \mathbf{o}I \ \vee \ \exists(r_i \in \mathbf{r}I) \ P \vdash r_j \succeq_r r_i]$$

By using this rule, the model with extended acts-for relations can be enforced directly in the P_E hierarchy, without reference to the transformation of P_E into the original model.

These five acts-for relations ($\succeq_o, \succeq_r, \succeq_w, \succeq_{ro}, \succeq_{wo}$) can be viewed as access control lists [Lam71]. For each principal p and distinct kind of acts-for relation, there is a list of principals that p allows to act for it in that manner. The relations are similar to access control lists in that there is an appropriate notion of ownership: a principal (typically) has the power to change which other principals are in its lists. These acts-for relations are not complete: for example, one privilege that a principal might usefully grant another is the ability to modify these lists, changing the principal hierarchy. Such privileges and their management, though important, are outside the scope of this work.

The relations differ from the usual concept of access control lists in that certain axioms connect the relations. One axiom is that authorization is transitive: if p reads for q and q reads for r, then p reads for

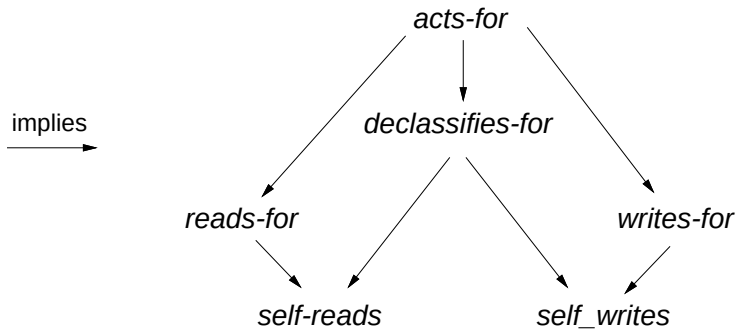

Figure 2.16: Partial order on the extended acts-for relations

r. In addition, some of these relations imply others; there is a partial order on the relations, as shown in Figure 2.16. The original relation acts-for, which gives one principal the full privileges of another, implies all five of the new relations.

2.7 Summary

The decentralized label model is a promising approach to specifying information flow policies for privacy and integrity. It provides considerable flexibility by allowing individual principals to attach flow policies to individual values manipulated by a program. These flexible labels then permit values to be declassified by an owner of the value. This declassification is safe because it does not affect the secrecy guarantees to other principals who have an interest in the secrecy of the data. This support for multiple principals makes the label model ideal for mutually distrusting principals.

One important feature of the decentralized label model is the complete relabeling rule, which precisely captures all the legal relabelings that are allowed when knowledge about the principal hierarchy is available statically. The rule is both sound and complete, and easy to apply. The rule is formalized as a pre-order relation with distributive lattice properties: join and meet operators are defined on these labels, so a compiler or static checker can use them to check information flow. When information flow is checked statically, run-time overhead is avoided. The compile-time overhead of checking these rules also is small.

The new rules for relabeling, join, and meet make the decentralized label model more practical and more usable. They also make it easier to model common security paradigms. For example, information flow can be described concisely in a system with group or role principals. Individual principals can model their own multilevel security classes in a decentralized fashion, and the rules also can be used in their dual form to protect integrity, or to protect both privacy and integrity simultaneously.

Chapter 3

The JFlow Language

The preceding chapter discusses the decentralized label model with only a little consideration about how to apply it to a programming language. This chapter presents JFlow, a new programming language that extends the Java language [GJS96] and permits static checking of flow annotations. A shorter description of the JFlow language also has been published elsewhere [Mye99]. JFlow is intended to support the writing of secure servers and applets that manipulate sensitive data.

Like other recent approaches to static information-flow checking [VSI96, SV98, HR98], JFlow treats static checking of flow annotations as an extended form of type checking. Programs written in JFlow can be checked statically by the JFlow compiler, which detects any information leaks through covert storage channels. If a program is type-safe and flow-safe, it is translated by the JFlow compiler into an equivalent Java program that can be converted into executable code by a standard Java compiler. The static checker does not, however, detect leaks through covert timing channels.

JFlow is the most practical programming language developed to date that allows static information flow checking. An important philosophical difference between JFlow and other work on statically checking information flow is the focus on a usable programming model. Despite a long history, static information flow analysis has not been accepted widely as a security technique. One major reason is that previous models of static flow analysis were too limited or too restrictive to be used in practice. The goal of this work has been to add enough power to the static checking framework to allow reasonable programs to be written in a natural manner.

This work has involved several new contributions. Because JFlow extends a complex, object-oriented programming language, it supports many language features that have not been integrated with static flow checking previously, including mutable objects, subclassing, dynamic type tests, access control, and exceptions. JFlow also provides powerful new features that make information flow checking less restrictive and more convenient than in previous models:

- The decentralized label model presented in Chapter 2 is supported, allowing multiple principals to protect their privacy even in the presence of mutual distrust. JFlow also supports the safe, statically-checked *declassification* mechanism described in Chapter 2, which permits a principal to relax its own

privacy policies, but not to weaken the policies of other principals.

- *Label polymorphism* allows the expression of code that is generic with respect to the security class of the data it manipulates.
- Run-time label checking and first-class label values provide a dynamic escape when static checking is too restrictive. Run-time checks are statically checked to ensure that information is not leaked by the success or failure of the run-time check itself.
- Automatic label inference makes it unnecessary to write many of the annotations that would be required otherwise.

The goal of type checking is to ensure that the apparent, static type of each expression is a supertype of the actual, run-time type of every value it might produce; similarly, the goal of label checking is to ensure that the apparent label of every expression is at least as restrictive as the actual label of every value it might produce. In addition, label checking guarantees that, except when declassification is used, the apparent label of a value is at least as restrictive as the actual label of every value that might *affect* it. In principle, the actual label could be computed precisely at run time. Static checking ensures that the apparent, static label is always a conservative approximation of the actual label. For this reason, it is typically unnecessary to represent the actual label at run time.

However, the two kinds of static checking differ in at least one important way. With type checking, it is not as important to achieve a language that can be checked entirely statically. Limitations in static type checking can be worked around by resorting to dynamic type checking, as in Java, or by simply trusting that programmers understand the types in their programs better than the static checker does, as in C++. These fallback positions are not available when checking information flow, because dynamic information flow checking itself creates a new information channel. It is for this reason that the language mechanisms in JFlow that support static checking of information flow are more elaborate than the usual language mechanisms for static type checking.

The JFlow compiler is structured as a source-to-source translator, so its output is a standard Java program that can be compiled by any Java compiler. For the most part, translation involves removal of the static annotations in the JFlow program after checking them; there is little code space, data space, or run time overhead, because most checking is performed statically.

JFlow is not completely a superset of Java. Certain features have been omitted to make information flow control tractable. Also, JFlow does not eliminate all possible information leaks. Certain covert channels (particularly, various kinds of *timing channels*) are difficult to eliminate. These limitations of JFlow are enumerated later, in Section 3.5.

```
int{public} x;
boolean{secret} b;
...
int x = 0;
if (b) {
    x = 1;
}
```

Figure 3.1: Implicit flow example

3.1 Static vs. dynamic checking

Information flow checks can be viewed as an extension to type checking. For both kinds of static analysis, the compiler determines that certain operations are not permitted on certain data values. Type checks may be performed at compile time or at run time, though compile-time checks usually are preferred when applicable because they impose no run-time overhead.

By contrast, fine-grained information flow control is practical only with some static analysis. This claim may sound odd; after all, any check that can be performed by the compiler can be performed at run time as well. The difficulty with run-time checks is exactly the fact that they can *fail*. In failing, they may communicate information about the data that the program is running on. Unless the information flow model is properly constructed, the fact of failure (or its absence) can serve as a covert channel. By contrast, the failure of a compile-time check reveals no information about the actual data passing through a program. A compile-time check only provides information about the program that is being compiled. Similarly, link-time and load-time checks provide information only about the program, and may be considered to be static checks for the purposes of this work.

For example, consider the code segment of Figure 3.1. By examining the value of the variable x after this segment has executed, we can determine the value of the secret boolean b, even though x has been assigned only constant values. This flow of information from b into x is called an *implicit flow*, because information is transferred through the program control structure rather than through a direct assignment. The problem is the assignment $x = 1$, which should not be allowed.

Static analysis is required in order to make this program work safely. A run-time check easily can detect that the assignment $x = 1$ communicates information improperly, and abort the program at this point. Consider, however, the case where b is false: no assignment to x occurs within the context in which b affects the flow of control. The fact that the program aborts or continues implicitly communicates information about the value of b. This information can be used in at least the case where b is false.

Most multilevel-secure systems handle such programs safely by restricting all writes that follow the if statement, on the grounds that once the process has observed b, it is irrevocably tainted. Every value the process computes is tainted by the label of b, even if it does not depend on the conditional in any way. A label is associated with the process, and becomes more restrictive with every value that the process observes. The

problem with this approach is that it is too coarse-grained: the process label easily can become so restrictive that every value the process computes is unusable.

We could imagine inspecting the body of the if statement at run time to see whether it contains disallowed operations, but in general this requires the evaluation of all possible execution paths of the program, which is clearly infeasible at run time. The advantage of compile-time checking is that in effect, static analysis efficiently constructs proofs that *no* possible execution path contains disallowed operations. We will see shortly how static analysis can be used to check this small program properly.

3.2 Language support for information flow checking

The next two sections present an overview of the JFlow language. This section concentrates on the new features added to the JFlow language and the rationale for their addition. The following section examines interactions between information flow control and complex programming language features such as objects, methods, and inheritance. In both sections, ordinary Java semantics are not discussed, because Java is widely known and well-documented [GJS96].

3.2.1 Labeled types

In a JFlow program, a label is denoted by a *label expression*, which is a set of *component expressions*. These expressions may take the form seen in Section 2.1.2: a label expression may be a series of policy expressions, separated by semicolons, such as $\{ o_1\colon r_1, r_2; o_2\colon r_2, r_3\}$. In this case, the two component expressions are both policy expressions. JFlow supports only privacy policies, although it would be straightforward to add combined privacy and integrity policies of the sort described in Section 2.6.2.

As in Chapter 2, the component expression *owner*: *reader*, *reader*, ... denotes a policy. In a program, a component expression may take a few additional forms. One added component form is a variable name, which denotes the *set* of policies in the label of the variable named. For example, the label expression {a} contains a single component expression; this label means that value it labels should be as restricted as the contents of a are. The label expression {a; o: r} contains two component expressions, indicating that the labeled value should be as restricted as a is, and also that the principal o restricts the value to be read by at most r. Other kinds of label components will be introduced later.

In JFlow, every value has a *labeled type* that consists of two parts: an ordinary Java type such as int, and a *label* that describes the ways that the value can propagate. Any type expression *t* may be labeled with any label expression *l*. This labeled type expression is written as *t*{*l*}; for example, the labeled type int{p:} represents an integer that principal p owns and, because no readers are listed, that only p can read. A labeled type may occur in a JFlow program in most places where a type may occur in a Java program. For example, variables may be declared with labeled type:

```
int{p:} x;
int{x} y;
int z;
```

The label usually may be omitted from a labeled type, as in the declaration here of the variable z. When a label is omitted, a default label is automatically provided in a manner that depends on the context. For example, when the label of a local variable is omitted, the label is inferred automatically from the uses of the variable. When the label of an instance variable (also known as a *field* or *member variable*) is omitted, the default label is the label $\{\}$. As in Chapter 2, this label is the least restrictive possible label because it contains no components to restrict the data it labels. There are several other cases in which default labels are assigned; however, these cases are discussed later.

The type and label parts of a labeled type act largely independently. The notation $S \leq T$ is used here to mean that the type S is a subtype of the type T. The intuitive behavior of subtyping is that it operates independently on the type and label: for any two types S and T and labels L_1 and L_2, $S \leq T \wedge L_1 \sqsubseteq L_2 \longleftrightarrow S\{L_1\} \leq T\{L_2\}$ (as in [VSI96]). However, this rule is really true only in an environment in which there is no mutation, such as a functional programming language. In this thesis, subtyping is a relation only on types, not on labeled types.

3.2.2 Implicit flows

In JFlow, the label of an expression's value varies depending on the evaluation context. This somewhat unusual property is needed to prevent leaks through implicit flows: channels created by the control flow structure itself. To prevent information leaks through implicit flows, the compiler associates a *program-counter label* (pc) with every statement and expression, representing the information that might be learned from the knowledge that the statement or expression was evaluated. The the idea of the program-counter label is due to Fenton [Fen74]. For example, consider the program of Figure 3.1 again, assuming that no information can be learned from the fact that the program is executed (that is, initially $\underline{\mathsf{pc}} = \{\}$). In this case, the value of pc during the consequent of the if statement is $\{\mathsf{b}\}$. After the if statement, it is again true that $\underline{\mathsf{pc}} = \{\}$, because no information about b can be deduced from the fact that the statement after the if statement is executed. (It is not true in general that the value of pc reverts after if statements, but is true here because this if statement always terminates normally.) The label of a literal expression (e.g., 1) is the same as its pc, or $\{\mathsf{b}\}$ in this case. The unsafe assignment in the example is prevented because the label of the variable being assigned ($\{\mathsf{public}\}$) is not at least as restrictive as the label of the value being assigned ($\{\mathsf{b}\}$, or $\{\mathsf{secret}\}$). The label of a variable is the same as its declared label, joined with the pc at the point of its declaration. The label of a variable expression (such as b) is the join of the variable label and the pc at the point where the expression occurs. The label of the expression 1 is $\{\mathsf{b}\}$, so the assignment is in general not permitted: the condition $\{\mathsf{b}\} \sqsubseteq \{\mathsf{x}\}$ translates to $\{\mathsf{secret}\} \sqsubseteq \{\mathsf{public}\}$, which is not true in general.

One way of thinking of the program-counter label is that there is a distinct pc for every basic block in the program. In general, the flow of control within a program depends on the values of certain expressions.

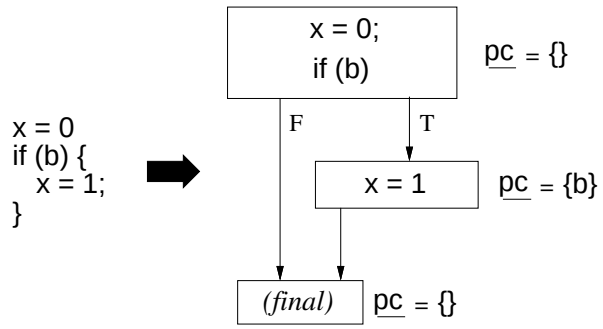

Figure 3.2: Basic blocks for an if statement

At any given point during execution, various values v_i have been observed in order to decide to arrive at the current basic block; therefore, the labels of these values affect the current pc:

$$\underline{\text{pc}} = \bigsqcup_i \{v_i\} = \{v_1\} \sqcup \{v_2\} \sqcup \ldots$$

Any mutation (that is, assignment) potentially can leak information about the observed values v_i, so the variable that is being mutated must be at least as restricted as the labels on all these variables; in other words, its label must be at least as restrictive as the label pc.

This label $\bigsqcup_i \{v_i\}$ can be determined through straightforward static analysis of the program's basic block diagram. The decision about which exit point to follow from a basic block B_i depends on the observation of some value v_i. The label pc for a particular basic block B is the join of some of the labels $\{v_i\}$. A label $\{v_i\}$ is included in the join if it is possible to reach B from B_i, and it is also possible to reach the final node from B_i without passing through B. If all paths from B_i to the final node pass through B, then arriving at B conveys no information about v_i.

This rule for propagating labels through basic blocks is equivalent to the rule of Denning and Denning [DD77]. JFlow does not apply this rule directly. Instead, the rules for determining the pc of a statement or expression are expressed as static inference rules in Chapter 4. Usually the static inference rules generate the same pc label as the rule based on basic block analysis, though there are cases in which the inference rules generate a more restrictive label, resulting in a loss of precision. This loss of precision occurs in code that throws and catches exceptions in a complex manner; it does not appear to be a problem in practice.

3.2.3 Termination channels

Information can be transmitted by the termination or non-termination of a program. Consider the execution of a "while" statement, which creates a loop in the basic block diagram. This situation is illustrated in Figure 3.3. Using the basic block rule just given or the static inference rules that will be presented later, it is the case that after the statement terminates, $\underline{\text{pc}} = \{\}$, using the same reasoning as for the "if" statement. This labeling might seem strange, because we know the value of b when we arrive at the final block. However, arriving at the final block gives no information about the value of b before the code started.

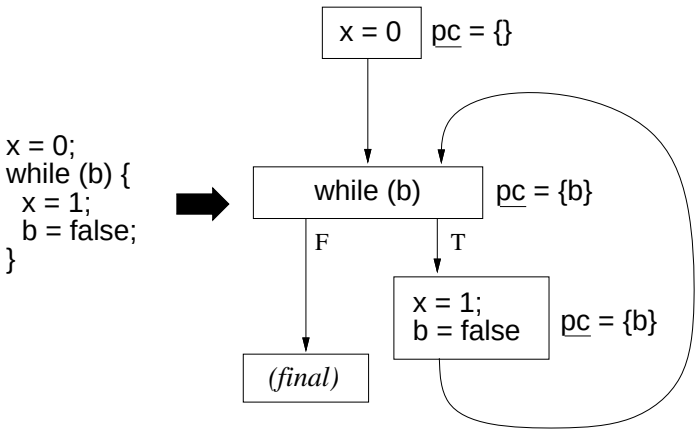

Figure 3.3: Basic blocks for a while statement

There is no way to use code of this sort to transmit information improperly as long as all programs terminate, or at least if there is no way to derive information from non-termination of a program [DD77, AR80]. The way one decides that a program has not terminated is to time its execution, either explicitly or through asynchronous communication with another thread. As discussed later, JFlow does not attempt to control information transfers through timing channels, termination channels, or asynchronous communication between threads.

3.2.4 Run-time labels

In JFlow, labels are not purely static entities; they may also be used as values. First-class values of the new primitive type label represent labels. This functionality is needed when the label of a value cannot be determined statically. For example, if a bank stores a number of customer accounts as elements of a large array, each account might have a different label expressing the privacy requirements of the individual customer. To implement this example in JFlow, each account can be labeled by an attached dynamic label value.

A variable of type label may be used both as a first-class value and as a label for other values. For example, methods can accept arguments with run-time labels, as in the following method declaration:

```
static float{*lb} compute(int x{*lb}, label lb)
```

In this example, the component expression *lb denotes the label contained *in* the variable lb, rather than the label *of* the variable lb. To preserve safety, variables of type label (such as lb) may be used to construct labels only if they are immutable after initialization; in Java terminology, if they are final.

The important power that run-time labels add is the ability to be examined at run time, using the switch label statement, an example of which is shown in Figure 3.4. The code in this figure attempts to transfer an integer from the variable x to the variable y. This transfer is not necessarily safe, because x's label, lb, is not known statically. The statement examines the run-time label of the expression x, and executes one of

```
label{L} lb;
int{*lb} x;
int{p:} y;
switch label(x) {
   case (int{y} z) y = z;
   else throw new UnsafeTransfer();
}
```

Figure 3.4: Switch label

several case statements, or an optional else statement. The statement executed is the first whose associated label is at least as restrictive as the expression label; that is, the first statement for which the assignment of the expression value to the declared variable (in this case, z) is legal. If it is the case that $\{*\mathsf{lb}\} \sqsubseteq \{\mathsf{p}:\}$, the first arm of the switch will be executed, and the transfer will occur safely via z. Otherwise, the else clause will be executed and an exception thrown.

The statement appears superficially like a typecase statement as in Modula-3 [Nel91]; however, it does not permit any discrimination on the actual (run-time) type of the expression. The types of the variables declared in each of the arms of the statement must all be supertypes of the apparent type of the expression. In this example, the apparent type of x is int, so the declared type of z must also be int.

Because lb is a run-time value, information may be transferred through it; in the example, one might observe which of the two arms of the switch are executed and infer the value of lb accordingly. However, this information channel is not covert. To prevent this information channel from becoming an information leak, the pc in the first arm is augmented to include lb's label, L. The assignment from z to y is permitted only if $\mathsf{L} \sqsubseteq \{\mathsf{y}\}$. Thus, the ordinary label-checking rules are used to control this information channel.

As we have seen, this run-time test of the labels {*lb} and {y} gives information about the contents of the variable lb. If the principal p is a final local variable of type principal, the run-time test may give information about the contents of p as well. Thus, the assignment is permitted only if $\{\mathsf{p}\} \sqsubseteq \{\mathsf{y}\}$, because information about both lb and p affects the possibility of executing that first arm. Note that if p is not a run-time principal, then $\{\mathsf{p}\} = \{\}$, and the condition $\{\mathsf{p}\} \sqsubseteq \{\mathsf{y}\}$ is trivially true.

A switch label statement may contain several case arms. In each arm, the fact that it is executed gives information about the labels of all previous case clauses, because the earlier clauses are known *not* to have been executed. Therefore, the pc in each arm, including the final, optional else clause, is as restrictive as the labels of *all* of the labels that the previous case arms tested against. In this example, the pc of the else clause is as restrictive as both {L} and {p}.

Run-time labels can be manipulated statically, though conservatively; they are treated as an unknown but fixed label. The presence of such opaque labels is not a problem for static analysis, because of the lattice properties of these labels. For example, given any two labels L_1 and L_2 where $L_1 \sqsubseteq L_2$, it is the case for any third label L_3 that $L_1 \sqcup L_3 \sqsubseteq L_2 \sqcup L_3$. This implication makes it possible for an opaque label L_3 to

appear in a label without preventing static analysis. Thus, unknown labels, including run-time labels, can be propagated statically.

3.2.5 Reasoning about principals

JFlow contains a mechanism for determining the authority of a running process that is both dynamically and statically checked. This authority mechanism is similar to that in other systems supporting more complex access control mechanisms. In JFlow, a method executes with some authority that has been granted to it. The authority is essentially the capability to act for some set of principals, and controls the ability to declassify data. This simple authority mechanism can be used to build more complex access control mechanisms, though the focus of this work is on using authority only to control declassification.

At any given point within a program, the static checker understands the code to be running with the ability to act for some set of principals, which is the static authority of the code at that point. The actual authority may be greater, because those principals may be able to act for other principals. The static authority can never exceed the actual authority unless revocation occurs while the program is running.

Static principal hierarchy. The static checker maintains a notion of the *static principal hierarchy* at every point in the program. The static principal hierarchy is a set of acts-for relations that are known to exist. The static principal hierarchy is a subset of the acts-for relations that exist in the true principal hierarchy.

The static authority of a procedure may be augmented by testing the principal hierarchy dynamically. The principal hierarchy is tested using the new actsFor statement. The statement actsFor(p_1, p_2) *S* executes the statement *S* if the principal p_1 can act for the principal p_2 in the current principal hierarchy. Otherwise, the statement *S* is skipped. The statement *S* is checked statically using the knowledge that the tested acts-for relation exists: for example, if the static authority includes p_1, then during *S* it is augmented to include p_2.

In addition, the actsFor statement may also have an else clause, just as if it were an if statement. The else clause is executed when the tested relationship does not exist. However, the else clause is statically checked without any additional knowledge. As Section 2.4.3 showed, negative information about acts-for relations cannot be used to augment static checking.

The authority of a process can be viewed simply as part of the principal hierarchy. The process represents a transient principal within the hierarchy. When authority is granted to the process, either by a principal in the system or by calling code that explicitly grants the authority, it can be thought of as a transient acts-for relation.

Revocation. It is possible that while an actsFor statement is being executed, the principal hierarchy may change in a way that would cause the test in the statement to fail. In this case, it may be desirable to revoke the code's permission to run with that authority, and it is assumed that the underlying system can do this, by halting the process that is executing the code at some point after the hierarchy changes. If a running program is halted because of a revocation, information may be leaked about what part of the program was

```
int b;
int y = 0;
if (b) {
    declassify ({y}) y = 1;
}
```

Figure 3.5: A declassify statement

being executed. This leak is a covert channel, but probably one that can be made slow enough that it is impractical to use.

Another strategy for dealing with asynchronous revocation is to run the program as a series of transactions. The principal hierarchy is checked at the time that the transaction commits to ensure that no actsFor statements were executed using principal hierarchy information that was invalidated by the time that the transaction committed. If invalid acts-for relations were used, the transaction is aborted and all of its changes are rolled back, preventing improper information flows. In this framework, handling revocation properly becomes a by-product of the isolation from asynchronous modification that transaction systems normally provide.

The current JFlow implementation does not attempt to invalidate execution because of revocation. However, there is one form of revocation that requires no extra support: the revocation that occurs when a method that has been granted authority terminates. As described in the preceding section, such a method can be considered a transient principal within the system. Revocation of the privileges of this principal is safe because the principal itself no longer exists after revocation; there is no way to name the principal corresponding to an executing method.

3.2.6 Declassification

A program can use its authority to declassify a value according to the model of Section 2.4.4. The expression declassify(e, L) relabels the result of an expression e with the label L. Declassification is checked statically, using the static authority at the point of declassification. The declassify expression may relax only policies owned by principals in the static authority.

A program also can use its authority to declassify the program-counter label. This functionality is provided by the new statement declassify(L) S, which executes the statement S using the program-counter label L. This form of declassification is also checked statically. For example, Figure 3.5 contains an example of a declassify statement. Assuming that the label of y is not more restrictive than the label of b, this program declassifies the implicit flow from b into y. For the duration of the assignment into y, the program-counter label is relaxed until it is no more restrictive than y itself. The legitimacy of the declassification is statically checked using the label of y and the static authority of the program at this point. Note that the labels of b and y are both automatically inferred in this example; these automatically inferred labels are not a problem for checking declassification statically.

```
class Account {
    final principal customer;
    String{customer:} name;
    float{customer:} balance;
}
```

Figure 3.6: Bank account using run-time principals

3.2.7 Run-time principals

Like labels, principals may also be used as first-class values at run time. The type principal represents a principal that is a value. A final variable of type principal may be used as if it were a real principal. For example, an explicit policy may use a final variable of type principal to name an owner or reader. These variables may also be used in actsFor statements, allowing static reasoning about parts of the principal hierarchy that may vary at run time. When labels are constructed using run-time principals, declassification may also be performed on these labels.

Run-time principals are needed in order to model systems that are heterogeneous with respect to the principals in the system, without resorting to declassification. For example, a bank might store bank accounts with the structure shown in Figure 3.6, using run-time principals rather than run-time labels. With this structure, each account may be owned by a different principal (the customer whose account it is). The security policy for each account has similar structure but is owned by the principal in the member variable customer. Code can manipulate the account in a manner that is generic with respect to the contained principal, but can also determine at run time which principal is being used. The principal customer may be manipulated by an actsFor statement, and the label {customer:} may be used by a switch label statement.

3.3 Interactions with features of Java

One novel aspect of JFlow is its integration of information flow analysis into a practical, object-oriented programming language. Java has complex features such as mutable objects, inheritance, subtyping and exceptions, and these features interact with label checking. This section describes how some of these Java language constructs have been extended or modified to support information flow control.

JFlow is an object-oriented language and supports inheritance and subtyping. Classes in JFlow are largely an extension of classes in Java. They may contain methods, static methods, and instance variables. Instance variables are declared with labeled types, just like local variables within methods.

Some class-related features of Java are not supported in JFlow: neither inner classes nor static instance variables are supported. Inner classes are not supported because they are a complication that is unnecessary for the goals of this work. Static instance variables are not supported because they would create covert channels, as discussed later in Section 3.5. However, non-static instance variables usually can substitute for static instance variables.

3.3.1 Method declarations

The syntax of a JFlow method declaration has some extensions when compared to Java syntax; there are a few optional annotations to manage information flow and authority delegation. A method header has the syntax shown in Figure 3.7, in the syntax of (and using some definitions from) the Java Language Specification [GJS96].

MethodHeader:
Modifiers$_{opt}$ *LabeledType Identifier*
BeginLabel$_{opt}$ (*FormalParameterList*$_{opt}$) *EndLabel*$_{opt}$
Throws$_{opt}$ *WhereConstraints*$_{opt}$

FormalParameter:
LabeledType Identifier OptDims

Figure 3.7: Grammar of a method header

As this grammar shows, the return value, the arguments, and the exceptions each may be labeled individually. There are two optional labels in a method declaration called the *begin-label* and the *end-label*. The begin-label is used to specify any restriction on pc at the point of invocation of the method. The begin-label allows information about the pc of the caller to be used for statically checking the implementation, preventing assignments within the method from creating implicit flows of information.

Figure 3.8 contains an example of a JFlow class declaration: a JFlow version of the standard Java class Vector. It provides several examples of JFlow method declarations. The setElementAt method in this declaration is prevented from leaking information by its begin-label, {L}. It can be called only if the pc of the caller is no more restrictive than {L}. The labels of the arguments o and i are written as {}, but as discussed in the following section, argument labels automatically include the begin-label, so both arguments also are labeled by {L}.

```
public class Vector[label L] extends AbstractList[L] {
   private int{L} length;
   private Object{L}[ ]{L} elements;

   public Vector() ...
   public Object elementAt(int i):{L; i} throws (IndexOutOfBoundsException) {
         return elements[i];
   }
   public void setElementAt{L}(Object{} o, int{} i) ...
   public int{L} size() { return length; }
   public void clear{L}() ...
   ...
}
```

Figure 3.8: A JFlow version of the class Vector

The end-label of a method specifies the pc at the point of termination of the method, and captures the restrictions on the information that can be learned by observing whether the method terminates normally. Individual exceptions and the return value itself also may have their own distinct labels, allowing static label checking to track information flow at fine granularity. For example, the end-label of the elementAt method in Figure 3.8 means that the pc following normal termination is at least as restrictive as both the label L and the label of the argument i. This end-label is necessary because the index-out-of-bounds exception is thrown because of an observation of the instance variable elements and the argument i. Therefore, knowledge of the termination path of the method may give information about the contents of these two variables.

Unlike in Java, method arguments in JFlow are always implicitly final. This change makes the use of first-class principals and labels more convenient, since arguments of the types label and principal are nearly always desired to be final. This simple change does not remove any significant power from the language, since code that assigns to an argument variable always can be rewritten to use a local variable instead.

3.3.2 Default labels

Figure 3.8 contains examples of JFlow method declarations that demonstrate some of the features of method declarations. Some types in the example are labeled, such as the types of the arguments o and i of the method setElementAt. Other types in this figure are unlabeled, such as the types of the argument and return value of elementAt. Whenever labels are omitted in a JFlow program, a default label is assigned, providing both greater expressiveness and greater convenience. The effect of these defaults is that often methods require no label annotations whatever. This section describes how default labels are assigned.

Labels may be omitted from a method declaration, signifying the use of *implicit label polymorphism*. For example, the argument of the method elementAt is unlabeled. When an argument label is omitted, the method is generic with respect to the label of the argument. The argument label becomes an implicit parameter of the procedure. The method elementAt can be called with any integer i regardless of its label.

Label polymorphism is important for building libraries of reusable code; without it, methods would need to be reimplemented for every argument label ever used. Consider implementing a method cos that evaluates the cosine of its argument. Without implicit label polymorphism, there are two strategies: reimplement it for every argument label ever used, or implement it using run-time labels. The former approach is clearly infeasible. Implicit labels have the advantage over run-time labels that when they provide adequate power, they are easier and cheaper to use. Without implicit labels, the signature of the cos method would be the following:

```
float{*lx} cos (float{*lx} x, label{} lx)
```

Implicit label polymorphism eliminates the run-time overhead and the gratuitous method arguments in this method signature, allowing the simpler signature that would be used in Java:

```
float cos (float x);
```

Other labels are assigned defaults as well. The end-label of a method always includes the begin-label

even if the end-label is not declared explicitly; if the end-label of the method is omitted, it is equal to the begin-label. The default label for the return value of a method is the end-label, joined with the labels of all the arguments. This default makes sense because it is the common case. For the method cos, the default return value label is $\{x\}$, and therefore does not need to be written explicitly. Methods may also return exceptionally, and exceptions may be labeled; the rule for default exception labels is the same as the rule for the end-label.

If the begin-label is omitted, it becomes an implicit parameter to the method. A method with an implicit begin-label parameter can be called regardless of the pc of the caller, because the code of the method is guaranteed not to leak information that is given to it. In general, methods without side-effects can be written in this fashion, which makes them convenient to use and to implement. The static checking rules described in Section 4 place restrictions on the implementation of such a method that limit its ability to cause side effects: local variables may of course be modified, and a method of this sort may mutate objects passed as arguments if appropriately declared, but other side effects will be prevented. Every assignment requires that the label of the variable be more restrictive than the pc at the point of assignment; however, the label of a variable external to the method cannot be proved more restrictive than the begin-label, so such an assignment will be rejected statically.

3.3.3 Method constraints

Unlike in Java, a method may contain a list of *constraints* prefixed by the keyword where:

WhereConstraints:
    where *Constraints*

Constraint:
    authority (*Principals*)
    caller (*Principals*)
    actsFor (*Principal* , *Principal*)

There are three different kinds of constraints:

- authority$(p_1, \ldots, p_n)$ This clause lists principals that the method is authorized to act for. The static authority at the beginning of the method includes the set of principals listed in this clause. The principals listed may be either names of global principals, or names of class parameters of type principal. Every listed principal must be also listed in the authority clause of the method's class, as described later in Section 3.3.8. This authority mechanism obeys the principle of least privilege, because not all the methods of a class need to possess the full authority of the class.

- caller$(p_1, \ldots, p_n)$ Calling code may also dynamically grant authority to a method that has a caller constraint. Unlike with the authority clause, where the authority devolves from the object itself, authority in this case devolves from the caller. A method with a caller clause may be called only if the calling code possesses the requisite static authority.

```
void m1(principal p, ...):{p} throws(AccessDenied)
 where caller(p) {
  actsFor(p, manager) {
     ...
  } else {
     throw new AccessDenied();
  }
}

void m2() where caller(manager) {
        ...
}
```

Figure 3.9: Using the caller constraint

The principals named in the caller clause need not be constants; they may also be the names of method arguments whose type is principal. By passing a principal as the corresponding argument, the caller grants that principal's authority to the code. These dynamic principals may be used as first-class principals; for example, they may be used in labels.

- actsFor (p_1,p_2) An actsFor constraint may be used to prevent the method from being called unless the specified acts-for relationship (p_1 acts for p_2) holds at the call site. When the method body is checked, the static principal hierarchy is assumed to contain any acts-for relationships declared in the method header. This constraint allows information about the principal hierarchy to be transmitted to the called method without any dynamic checking.

The caller mechanism provides a simple access control mechanism that can be checked either statically or dynamically. To check authority dynamically, a method can use a caller constraint to accept a grant of unknown authority, then use the actsFor statement to test that the granted authority is sufficiently powerful. This access control mechanism can be used to build more elaborate access control mechanisms such as access control lists.

For example, consider the method skeletons in Figure 3.9. The method m1 dynamically tests whether the caller has the authority to act for the principal manager. Because of the caller constraint, the caller must pass a principal p for which it can act. The actsFor test then tests whether p, and therefore this method, has the authority to act for the principal manager. If not, the AccessDenied exception is thrown. Note that the end-label of the method is p, because knowing whether the method terminated normally or exceptionally gives information about the principal passed. Thus, authority tests do not leak information through their success or failure.

The method m2 statically enforces the same test of authority that m1 tests dynamically. It can be called only from code that is statically known to act for manager, such as the consequent of the actsFor test in

the method m1, or from within another method like m2 itself. The method m2 is not as flexible as m1, but incurs no dynamic overhead.

3.3.4 Exceptions

Exceptions in JFlow are almost identical to exceptions in Java. There are two changes, one syntactic and one semantic. The syntactic change is that the list of exceptions in a method header must be delimited by parentheses. Parentheses are needed in case the exception is labeled, as in the following declaration.

```
int f(Object a, Object b):{a;b}
  throws (NullPointerException{a}, NotFound)
```

Without parentheses, it cannot be determined unambiguously whether the brace following NullPointerException is the beginning of a label expression or the beginning of the method.

The more substantive change to Java is the treatment of unchecked exceptions. Java allows users to define exceptions that need not be declared in method headers (*unchecked exceptions*), although this practice is described as atypical [GJS96]. In JFlow, only a few specific exceptions are allowed to be unchecked, because unchecked exceptions can serve as covert channels. All other exceptions (such as NullPointerException and IndexOutOfBoundsException) must be declared explicitly in a method header if the method might throw the exception. Only one unchecked exception is allowed: the new exception FatalError, which may not be caught by a catch clause. This exception is used for error conditions such as stack overflow and heap exhaustion. Because it is unchecked, it can serve as a covert information channel. However, since it cannot be caught, the exception FatalError can be used to transmit only one bit of information per program execution.

In JFlow as well as in Java, the catch clause of a try...catch statement is a type discrimination mechanism as well as an exception-handling mechanism. It is also one of the few places in JFlow where a type may not be labeled. As in Java, a catch clause takes the form catch $(C\ v)\ \ S$, where C is an unlabeled class that inherits from Throwable, v is a variable name, and S is a statement to be executed if the clause catches the exception. The decision about which catch clause of a try...catch statement to execute, if any, depends only on the dynamic type of the exception. Within each catch clause, the pc is determined by the labels attached to the exceptions that might be thrown by the statement in the try clause of the statement.

The break and continue statements provide another exception mechanism in Java, since they may specify a statement label to jump to. These statements are structured goto statements. They are supported in JFlow and introduce the simple requirement that the pc at the destination statement is at least as restrictive as the pc at the break or continue statement.

3.3.5 Parameterized classes

Parameterized types have long been known to be important for building reusable data structures. A parameterized class is generic with respect to some set of type parameters. This genericity is particularly useful for building collection classes such as generic sets and maps. It is even more important to have polymorphism in

```
public class Vector[label L] extends AbstractList[L] {
   private int{L} length;
   private Object{L}[ ]{L} elements;

   public Vector() ...
   public Object elementAt(int i):{L; i} throws (IndexOutOfBoundsException) {
         return elements[i];
   }
   public void setElementAt{L}(Object{} o, int{} i) ...
   public int{L} size() { return length; }
   public void clear{L}() ...
   ...
}
```

Figure 3.10: Parameterization over labels

the information flow domain; the usual way to handle the absence of statically-checked type polymorphism is to perform dynamic type casts, but this approach works poorly when applied to information flow, because dynamic tests create new information channels.

In JFlow, class and interface declarations are extended to allow *parameterization*; they may be generic with respect to some number of labels or principals, by including a set of explicitly declared parameters. Parameterized types are important for building reusable data structures in JFlow.

An example of a reusable data structure is the Java Vector class, which may be translated to JFlow as shown in Figure 3.10. This example also appeared earlier, in Figure 3.8. The Vector class is parameterized on a label L that represents the label of the contained elements. Assuming that secret and public are appropriately defined, the types Vector[{secret}] and Vector[{public}] would represent vectors of elements of differing sensitivity. These types are referred to as *instantiations* of the parameterized type Vector. Without the ability to instantiate classes on particular labels, it would be necessary to reimplement Vector for every distinct element label.

A class may also be parameterized over principals, as in the example of Figure 3.11. This class may be instantiated with any two principals p and q. For example, paramCell[Bob,Amy] has a field contents with the label {Bob: Amy}. This functionality provides power similar to that of run-time principals (as in the bank account example of Figure 3.6), but without the run-time or storage overhead that run-time principals can incur.

```
class paramCell[principal p, principal q] {
   int{p: q} contents;
}
```

Figure 3.11: Parameterization over principals

The semantics of class parameters are defined in such a way that class parameters do not need to be represented at run time, because information then cannot be conveyed through class parameters. As a result, class parameters may not be used in run-time tests; for example, label parameters may not be tested in a switch label statement, nor may principal parameters appear in an actsFor test.

When a parameterized or unparameterized type inherits from a superclass, or implements an interface, the supertype may be an instantiation. The instantiation that is inherited from or implemented must be a legal type within the scope of the class that is inheriting from or implementing it. This is a specific instance of a more general rule in JFlow: within a parameterized class or interface, the formal parameters of the class may be used as actual parameters to instantiations of parameterized types within its scope. This rule corresponds exactly to the approach taken in many languages that support parameterization over types [LCD+94, LMM98, OW97].

JFlow does not provide parameterization with respect to types, because it seems unnecessary for investigating static information flow control. It would be straightforward to add *unconstrained parametric polymorphism* in which the implementation of a polymorphic abstraction is unable to use any knowledge of the type parameter. This kind of parametric polymorphism is less expressive than that which appears in similar languages like PolyJ [MBL97, LMM98] or Pizza [OW97]. Constrained parametric polymorphism, as in those languages, creates complications for information flow control, because the parameter can be used as an information channel.

The addition of label and principal parameters to JFlow makes parameterized classes into simple *dependent types* [Car91], because types contain values. To ensure that these dependent types have a well-defined meaning, only final variables may be used as parameters; since they are immutable, their meaning cannot change. An alternative approach would be to allow all variables to be used as parameters; however, in that case two different types that mention the same variable would have different meanings if an assignment to the variable occurred between them.

Note that even if $\{\mathsf{public}\} \sqsubseteq \{\mathsf{secret}\}$, it is not the case that $\mathsf{Vector}[\{\mathsf{public}\}] \leq \mathsf{Vector}[\{\mathsf{secret}\}]$. (The subtype relation is again denoted by $\leq$.) This subtype relation would be unsound because Vector is mutable, an observation that applies to subtyping relations on type parameters as well [DGLM95].

When such a subtype relation is sound, the parameter may be declared as a covariant label rather than as a label. Covariant label parameters are made sound by placing additional restrictions on their use, as follows. A covariant label parameter may not be used to construct the label for a non-final instance variable. It also may not be used as an actual parameter to a class whose formal parameter is a label. However, immutable (final) instance variables and method arguments and return values may be labeled using a covariant parameter.

Within non-static methods and on an instance variable, the variable this may be used to construct labels, where it denotes the label of the object that the method was invoked on, or the object that the instance variable is part of. If an instance variable is labeled by this, it would not be safe to allow an assignment to that variable, since there might be another reference to the object whose label is less restrictive than the

```
class passwordFile authority(root) {
  public boolean check (String user, String password)
   where authority(root) {
    ...
  }
}
```

Figure 3.12: An authority declaration

label of the reference being used for the assignment. This other reference could then be used to observe the assigned value. For this reason, the variable this is treated as an implicit covariant label parameter when used in a label. The use of the label {this} is restricted in the same way that the use of other covariant parameters is restricted: it may not be used to label non-final instance variables.

3.3.6 Arrays

Although JFlow does not support user-defined type parameters, it does support one type with a type parameter: the built-in Java array type, which is used as the type of the instance variable elements in Figure 3.10. In JFlow, arrays are parameterized with respect to both the type of the contained elements and the label of those elements. In the example for Vector, the type of the instance variable elements is Object$\{L\}$[ ] which represents an array of Object where each element in the array is labeled with L. The array type behaves as though it were a type array$[T, L]$ with two parameters: an element type and an element label; in this case $T =$ Object. The label parameter may be omitted, in which case it defaults to $\{\}$. For example, the types int[ ] and int$\{\}$[ ] are equal.

One might wonder why the label on the array itself is not sufficient to protect the array elements. The reason is that arrays are mutable data containers. Suppose that arrays did not have a separate label parameter. In that case, a variable of type int[ ]$\{\}$ could be assigned to a variable with the labeled type int[ ]$\{L\}$ for some more restrictive label L. A value of labeled type $\{L\}$ then could be assigned to an array element in apparent safety; however, that same value could also be observed through the original array with the unrestricted label $\{\}$, laundering its label away. This argument also applies to the type Vector[L] discussed in the preceding section.

The subtyping rule for arrays in JFlow is the same as in Java: if the type S is a subtype of the type T, then the type array$[S,L]$ is a subtype of array$[T,L]$. However, the label parameter is not covariant, so if L_1 and L_2 are labels, then $L_1 \sqsubseteq L_2$ does not imply that array$[T, L_1]$ is a subtype of array$[T, L_2]$.

JFlow arrays offer one additional operation: the pseudo-field length that returns the number of elements in the array. The label of the length field is the same as the label of the array, *not* the element label. This label is safe because the length of a JFlow array (and a Java array) is immutable after array creation.

3.3.7 Run-time type discrimination

Java supports two expressions for run-time type discrimination: run-time casts and the instanceof operator. The expression $(T)\ E$ attempts to cast an expression E to type T, throwing an exception if this is not possible; the expression E instanceof T returns a boolean indicating whether E produced an expression that can be assigned to a variable of type T. Both of these operators are supported by JFlow as well. The result of both expressions is as restricted as the result of the expression E is.

JFlow imposes one limitation on these operators: they may be invoked only with a type T that is not an instantiation. The reason for this restriction is that information about the parameters of T is not available at run time. If information about the parameters were available at run time, it would create an additional information channel to be controlled. However, the use of parameterized types with these operators would be safe if it could be determined statically that the parameters used in the cast match the parameters of the dynamic type of the class. This approach is taken with type parameters in the language Pizza [OW97], because Pizza does not represent type parameters at run time, but it is not currently supported in JFlow.

3.3.8 Authority declarations

Classes in JFlow also support authority declarations. A class may have some authority granted to its objects by the addition of an authority clause to the class header. Figure 3.12 contains a partial example of a class passwordFile that declares the authority of the principal root; its method check then claims the authority of root and can use it within the body of the method.

The authority clause of a class may name principals external to the program (as in this case), or class parameters of type principal. In either case, if a class C has a superclass C_s, any authority in C_s must be covered by the authority clause of C: if C_s has some principal p in its authority clause, C must too. The effect of this rule is that it is not possible to obtain authority by inheriting from a superclass.

The ability to give a class the authority of external principals is useful but also potentially dangerous and therefore must be controlled. If the authority clause of a class names external principals, these principals must permit the creation of the class. This permission can be tested by requiring that the process that installs the class into the system (perhaps the compiler) has been granted the appropriate authority by the principals named.

When the authority clause names a parameter of the class that is of type principal, the code of the class acts for an arbitrary principal that is specified by the instantiator. The static authority at the point of invocation of the class constructor must include the authority of the actual principal parameters that are used in the call to the constructor; this ensures that the authority of the class was received from a process that actually possessed that authority. This rule differs from the rule that is used when external principals are named in the authority clause, because the authority derives from the code that invokes the constructor, rather than from the process that installs the class into the system. Note that static methods of the class do not possess the authority of principal parameters because otherwise the construction-time test would be

bypassed.

This language feature is both powerful and dangerous, because an object created in this manner can be used to capture and retain authority that is granted to a method by a caller; it is a general, free-standing *capability* [DV66, WCC+74] for that authority. In JFlow, there is no way to tell whether authority that is granted to a subsystem has been captured by the subsystem in a capability of this sort; thus, this mechanism can be misused to create *luring attacks*, in which a subsystem acquires authority without the knowledge of its caller [WBF97]. For this reason, most principals should not be permitted even to define a class that places a principal parameter in its authority clause; these classes may be defined only by a highly trusted principal, such as root.

3.3.9 Inheritance and constructors

Like Java classes, a JFlow class may declare that it has some supertypes: a superclass that it inherits from or interfaces that it implements. Inheritance and subtyping have some interactions with the new features of JFlow.

As in Java, methods may be overloaded and are distinguished by their argument types. The signature of a class method must conform to the signatures of the same method in its supertypes, where method identity is determined by the argument type. Signature conformance in JFlow includes the Java requirement that the return types of the two signatures must be identical, but also places restrictions on the labels of the subclass method signature: the labels of method arguments in the subclass must be at least as restrictive as the labels of method arguments in the superclass, and the label of the return value in the subclass may be at most as restrictive as the label of the return value in the superclass.

JFlow classes support constructors, just like Java classes. A constructor for class C behaves like a static method that returns a new object of type C. Constructors do not declare a return label; the label on the returned object is the same as the end-label of the method. Consider this constructor declaration:

```
class C {
  C{Bob:}(int x{}, int y{}) { ... }
  ...
}
```

The constructor declared here has a begin-label and end-label {Bob:}, and the object produced by a call to the new operator that uses this constructor has this same label.

Constructors in Java and JFlow must invoke a superclass constructor if the class inherits from a superclass. JFlow differs from Java in requiring final instance variables of the subclass to be initialized before the call to the superclass constructor, if any. This requirement arises because it is important to prevent final instance variables of type label or principal from being observed before they are initialized. Such an observation might lead to information leaks. Suppose a variable L of type label is used to construct the label of another variable, using the declaration int{L} x. If the variable x is used as an argument to a switch label statement before the variable L is initialized, the statement will not determine the case to execute properly,

```
class Complex {
   public final float real, imaginary;

   public Complex{r;i}(float r, float i) {
      real = r;
      imaginary = i
   }
   ...
}
```

Figure 3.13: Implementation of complex numbers

and may invoke a case that creates an information leak.

The section of the constructor before the superclass invocation is a sequence of arbitrary statements that is referred to here as the *constructor prologue*. Every final instance variable of the class must be initialized in the constructor prologue; it must include an assignment of the form $v = E$; for every final instance variable v and some expression E. In the prologue and in the call to the superclass constructor, the object (this) and its instance variables are not in scope (may not be used), except that they may of course be used on the left-hand side of their own initialization assignments. The purpose of this rule is to prevent uninitialized data from being read, possibly causing information leaks.

An initialization assignment is checked using a more relaxed rule than for other variable assignments. For an ordinary assignment $v = E$, the safety condition is $L_E \sqsubseteq \{v\}$, where L_E is the label of the expression E and takes into account the current pc. For an initialization assignment, the weaker condition $L_E \sqsubseteq \{v; L_R\}$ is enforced, where L_R is the end-label of the constructor, which is the label of the object being constructed. This weaker condition is safe because the instance variable cannot be accessed without using a reference to the object being constructed. Any access to an instance variable through an object reference causes the result to acquire the label of the reference. Thus, the label on the object will protect the instance variable.

This weaker initialization rule is helpful when writing classes that represent immutable abstractions, such as a class representing complex numbers. For example, consider the code in Figure 3.13, which implements a simple complex number abstraction that is convenient to use. The class Complex has a single constructor that takes two arguments r and i. The object returned by the constructor is automatically labeled as restrictively as both r and i, because the end-label of the constructor is $\{\mathsf{r};\ \mathsf{i}\}$. The implementation of the constructor is also particularly simple. This convenient abstraction and others like it are made possible by the weaker initialization rule. The initializations of the instance variables real and imaginary are permitted because the end-label of the constructor, $\{\mathsf{r};\ \mathsf{i}\}$, is at least as restrictive as the labels of the values being assigned, r and i. Without the weaker initialization rule, the assignment would not be permitted, because the label of both instance variables, $\{\}$, is not known to be more restrictive than the implicit label parameters associated with the arguments r and i. However, the weaker initialization rule is safe because any access to

```
class passwordFile authority(root) {
  public boolean check (String user, String password)
   where authority(root) {
    // Return whether password is correct
    boolean match = false;
    try {
      for (int i = 0; i < names.length; i++) {
        if (names[i] == user &&
        passwords[i] == password) {
          match = true;
          break;
        }
      }
    }
      catch (NullPointerException e) {}
      catch (IndexOutOfBoundsException e) {}
    return declassify(match, {user; password});
  }
  private String [ ] names;
  private String { root: } [ ] passwords;
}
```

Figure 3.14: A JFlow password file

the instance variables real and imaginary must be through the object, which is labeled at least as restrictively as the data that was stored into it using r and i.

3.4 Examples

Now that the essentials of the JFlow language have been covered, we are ready to consider some interesting examples of JFlow code.

3.4.1 Example: passwordFile

Figure 3.14 contains a JFlow implementation of a simple password file, in which the passwords are protected by information flow controls. Only the method for checking passwords is shown. This method, check, accepts a password and a user name, and returns a boolean indicating whether the string is the right password for that user. In this method, the label of the local variables match and i are not stated explicitly, and are automatically inferred from their uses.

The if statement is conditional on the elements of passwords and on the variables user and password, whose labels are implicit parameters. Therefore, the body of the if statement has $\underline{\text{pc}} = \{\text{user; password; root:}\}$, and the variable match also must have this label in order to allow the assignment match = true. This label prevents match from being returned directly as a result, because the label of the return

```
class Protected {
  Object{*lb} content;
  final label{this} lb;

  public Protected{LL}(Object{*LL} x, label LL) {
    lb = LL;       // must occur before call to super()
    super();       //
    content = x;   // checked assuming lb == LL
  }
  public Object get(label L):{L} throws (IllegalAccessError) {
    switch label(content) {
      when (Object{*L} unwrapped) return unwrapped;
      else throw new IllegalAccess();
    }
  }
  public label get_label() {
    return lb;
  }
}
```

Figure 3.15: The Protected class

value is the default label, {user; password}. Finally, the method declassifies match to this desired label, using its compiled-in authority to act for root.

More precise reasoning about the possibility of exceptions would make writing the code more convenient. In this example, the exceptions NullPointerException and IndexOutOfBoundsException must be caught explicitly, because the method does not explicitly declare them. However, it is possible to show in this case that the exceptions cannot be thrown.

Otherwise there is very little difference between this code and the equivalent Java code. Only three annotations have been added: an authority clause stating that the principal root trusts the code, a declassify expression, and a label on the elements of passwords. The labels for all local variables and return values are either inferred automatically or assigned sensible defaults. The task of writing programs is made easier in JFlow because label annotations tend to be required only where interesting security issues are present, although a number of novel language features have been needed to make this possible.

In this method, the implementor of the class has decided that declassification of match results in an acceptably small leak of information. Like all login procedures, this method does leak information, because exhaustively trying passwords eventually will extract the passwords from the password file. However, assuming that the space of passwords is large and passwords are difficult to guess, the expected amount of password information gained in each such trial is far less than one bit. Reasoning about when leaks of information are acceptable lies outside the domain of classic information flow control.

3.4.2 Example: Protected

The class Protected provides a convenient way of managing run-time labels, as in the bank account example mentioned earlier. Its implementation is shown in Figure 3.15. As the implementation shows, an object of type Protected is an immutable pair containing a value content of type Object and a label lb that protects the value. Its value can be extracted with the get method, but the caller must provide a label to use for extraction. If the label is insufficient to protect the data, an exception is thrown. A value of type Protected behaves very much like a value in dynamically-checked information flow systems, because it carries a run-time label. A Protected has an obvious analogue in the type domain: a value dynamically associated with a type tag (for example, the Dynamic type [ACPP91]).

One key to making Protected convenient is that because lb is final, it can be labeled simply as $\{\}$. In effect, its label is the same as the label of the containing object. The initialization of lb is allowed by the permissive initialization rule of Section 3.3.9. For the assignment $\mathsf{lb} = \mathsf{LL}$, the initialization rule requires that the formula $\{\mathsf{LL}\} \sqsubseteq \{\} \sqcup \{\mathsf{LL}\}$ be true, which it obviously is. Note that it is not necessary that the instance variable content be final for this code to be correct.

3.5 Limitations

This section summarizes the ways that JFlow is not a superset of Java, and also covert channels that JFlow cannot eliminate. Certain covert channels (particularly, various kinds of timing channels) are difficult to eliminate. Prior work has addressed static control of timing channels, though the resulting languages are restrictive [AR80, SV98]. Other covert channels arise from Java language features that consequently must be removed.

Threads: JFlow does not prevent threads from communicating covertly via the timing of asynchronous modifications to shared objects. This covert channel can be prevented by requiring only single-threaded programs.

Timing channels: JFlow cannot prevent threads from covertly gaining information by timing code with the system clock, except by removing access to the clock.

Hashcode: The built-in implementation of the hashcode method, provided by the class Object, can be used to communicate information improperly, because it gives information about the memory address at which an object has been allocated. This information allows the memory allocator to be used as a covert channel. As a result, in JFlow every class must implement its own hashcode.

Static variables: The order of static variable initialization could be used to communicate information improperly. This covert channel is blocked by ruling out static variables. However, static methods are legal.

Finalizers: Finalizers are run in a separate thread from the main program, and therefore can be used to communicate covertly. Finalizers are not part of JFlow.

Resource exhaustion: An OutOfMemoryError could be used to communicate information covertly, by conditionally allocating objects until the heap is exhausted. JFlow treats this error by converting it to a FatalError exception, preventing it from communicating more than a single bit of expected information per program execution. Other resource exhaustion errors such as stack overflow are treated similarly.

Wall-clock timing channels: A JFlow program can change its run time because of private information it has observed. As an extreme example, it can enter an infinite loop. JFlow does not attempt to control these channels, which are a variety of timing channel because information only leaks if one is able to time the program.

Unchecked exceptions: As described in Section 3.3.4, JFlow has no unchecked exceptions because they could serve as covert channels.

Backward compatibility: JFlow is not backward compatible with Java, since existing Java libraries are not flow-checked and do not provide flow annotations. However, in many cases, a Java library can be wrapped in a JFlow library that provides reasonable annotations.

3.6 Grammar extensions

JFlow contains several extensions to the standard Java grammar, in order to allow information flow annotations to be added. The following productions must be added to or modified from the standard Java Language Specification [GJS96]. As with the Java grammar, some modifications to this grammar are required if the grammar is to be input to a parser generator. These grammar modifications (and, in fact, the code of the JFlow compiler itself) were to a considerable extent derived from those of PolyJ, an extension to Java that supports parametric polymorphism [MBL97, LMM98].

3.6.1 Label expressions

LabelExpr:
{ *Components*$_{opt}$ }

Components:
Component
Components ; *Component*

Component:
Principal : *Principals*$_{opt}$
this
Identifier
* *Identifier*

Principals:

Principal
Principals , *Principal*

Principal: Name

3.6.2 Labeled types

Types are extended to permit labels. The new primitive types label and principal are also added.

LabeledType:
PrimitiveType LabelExpr$_{opt}$
ArrayType LabelExpr$_{opt}$
Name LabelExpr$_{opt}$
TypeOrIndex LabelExpr$_{opt}$

PrimitiveType:
NumericType
boolean
label
principal

The *TypeOrIndex* production represents either an instantiation or an array index expression. Since both use brackets, the ambiguity is resolved after parsing.

TypeOrIndex:
Name [*ParamOrExprList*]

ArrayIndex:
TypeOrIndex
PrimaryNoNewArray [*Expression*]

ClassOrInterfaceType:
Name
TypeOrIndex

ParamOrExprList:
ParamOrExpr
ParamOrExprList , *ParamOrExpr*

ParamOrExpr:
Expression
LabelExpr

ArrayType:
LabeledType [ ]

ArrayCreationExpression:
new *LabeledType DimExprs OptDims*

3.6.3 Class declarations

ClassDeclaration:
Modifiers$_{opt}$ class *Identifier Params*$_{opt}$
Super$_{opt}$ *Interfaces*$_{opt}$ *optAuthority ClassBody*

InterfaceDeclaration:
Modifiers$_{opt}$ interface *Identifier Params*$_{opt}$

ExtendsInterfaces$_{opt}$
Interfaces$_{opt}$ *InterfaceBody*

Params:
[*ParameterList*]

ParameterList:
Parameter
ParameterList , *Parameter*

Parameter:
label *Identifier*
covariant label *Identifier*
principal *Identifier*

Authority:
authority (*Principals*)

3.6.4 Method declarations

MethodHeader:
Modifiers$_{opt}$ *LabeledType Identifier*
BeginLabel$_{opt}$ (*FormalParameterList*$_{opt}$) *EndLabel*$_{opt}$
Throws$_{opt}$ *WhereConstraints*$_{opt}$
Modifiers$_{opt}$ void *Identifier*
BeginLabel$_{opt}$ (*FormalParameterList*$_{opt}$) *EndLabel*$_{opt}$
Throws$_{opt}$ *WhereConstraints*$_{opt}$

ConstructorDeclaration:
Modifiers$_{opt}$ *Identifier BeginLabel*$_{opt}$ (*FormalParameterList*)
EndLabel$_{opt}$ *Throws*$_{opt}$ *WhereConstraints*$_{opt}$

FormalParameter:
LabeledType Identifier OptDims

BeginLabel:
LabelExpr

EndLabel:
: *LabelExpr*

WhereConstraints:
where *Constraints*

Constraints:
Constraint
Constraints , *Constraint*

Constraint:
Authority
caller (*Principals*)
actsFor (*Principal* , *Principal*)

To avoid ambiguity, the classes in a throws list must be placed in parentheses. Otherwise a label might be confused with the method body.

Throws:
throws (*ThrowList*)

3.6.5 New statements

Statement:
StatementWithoutTrailingSubstatement
. . . existing productions . . .
ForStatement
SwitchLabelStatement
ActsForStatement
DeclassifyStatement

The switch label statement executes the first case in which the label of the new variable introduced is at least as restrictive as the label of the expression on which the statement is invoked. This determination is based upon the static comparison of label components that are not run-time representable, and the dynamic comparison of label component that are run-time representable. The new variable (if any) is initialized with the value of the expression. If none of the cases are executed, the else clause, if any, is executed.

SwitchLabelStatement:
switch label (*Expression*) { *LabelCases* }

LabelCases:
LabelCase
LabelCases LabelCase

LabelCase:
case (*Type LabelExpr Identifier*) *OptBlockStatements*
case *LabelExpr OptBlockStatements*
else *OptBlockStatements*

The actsFor statement executes a statement if the first principal can act for the second principal in the current principal hierarchy. The knowledge of the existence of the acts-for relationship is used when statically checking this statement. If the acts-for relationship does not exist, the statement in the else clause, if any, is executed.

ActsForStatement:
actsFor (*Principal* , *Principal*) *Statement OptElse*

The declassify statement executes a statement, but with some restrictions removed from pc.

DeclassifyStatement:
declassify (*LabelExpr*) *Statement*

3.6.6 New expressions

The new label expression produces a new run-time value of type label. The expression must describe a label that is entirely run-time representable; it may not mention any principal or label parameters (implicit or explicit).

Literal:
. . . existing productions . . .
new label *LabelExpr*

The declassify expression evaluates an expression and returns its result, but with a possibly declassified label. The static authority at the point of invocation must be sufficiently strong.

DeclassifyExpression:
declassify (*Expression* , *LabelExpr*)

Chapter 4

Statically Checking JFlow

This chapter shows that the language presented in Chapter 3 can be checked statically in a straightforward manner. It also describes the JFlow language more completely than the previous chapter did, because it shows precisely how static checking is performed, using formal inference rules and function definitions. These rules are also explained informally. The approach taken is to describe the aspects of JFlow that differ from Java. For example, type checking is largely ignored because it is almost identical to that in Java. The execution semantics of the language also are sufficiently close to Java that they are not described formally.

By focusing on information flow checking, the formal rules provide a concise description of many of the interesting aspects of the JFlow compiler implementation. This chapter describes much of the static checking that is done by the JFlow compiler; however, the description of the label inference algorithm and source-to-source translation are found later, in Chapter 5.

4.1 Correctness

Because this chapter presents rules for statically checking the JFlow language, it is useful to consider the criteria for whether these rules are correct.

The notion of correctness in this language is essentially the same as in other recent work on statically checking information flow as a kind of type system [VSI96, SV98, HR98]. For simple JFlow programs that do not use parameters, run-time labels, or subtyping, the rules needed for static checking are essentially the same as the static checking rules presented in that work. However, extra static checking machinery is present in JFlow to support the new language features that are presented in Chapter 3.

The rules are intended to enforce the following two properties:

- The apparent label of every expression is at least as restrictive as the actual label of every value it might produce.
- The actual label of a value is at least as restrictive as the actual label of every value that might *affect* it. (modulo declassification). One value v_1 is considered to affect another, v_2, if a change to v_1 might cause v_2 to change.

The first property expresses the usual idea that static checking must be conservative; the second property enforces the usual definition of correctness for information flow, *non-interference* [GM84]. Intuitively, non-interference says that the low-security outputs of a program may not be affected by its high-security inputs. In Java (and JFlow), objects may exist both before and after the program runs, so they are effectively persistent, and must be considered to be inputs and outputs themselves.

The non-interference condition must be weakened because of the presence of declassification in the language model. Declassification allows higher-security data to interfere with lower-security data, through the explicit action of the principal whose security is affected. The relaxed version of non-interference is that inputs may affect lower-security outputs only with the explicit authorization of a principal able to override the corresponding policies.

To properly define the notion of an *actual label* for each expression, an operational semantics for JFlow could be defined. The argument for correctness would be twofold: the operational semantics enforce the modified non-interference property, and the static checking rules are conservative with respect to the operational semantics.

This approach has been taken for type checking Java [Sym97, Nv98], but is not taken in this thesis because important features in JFlow such as objects, inheritance, and dependent types make formal proofs of correctness difficult at this point. The operational semantics of Java also are defined clearly elsewhere [GJS96, DE97], and the notion of the actual label is clear simply from the static checking rules themselves. Many of the static checking rules, particularly those for standard Java constructs, are seen to be correct by inspection, and are similar to static checking rules seen in other work on information flow [DD77, VSI96, HR98] (except for the support for exceptions). In addition, an attempt is made to argue informally for the correctness of all the rules.

Section 3.5 described several Java features such as threads and the built-in `hashCode` method that have been removed from JFlow, and information channels that have been ignored, such as stack overflow, which can leak one bit of information. The reason for removing these information channels is that they are difficult to characterize with static typing rules without making the language impractically restrictive. Absent these information channels, the information flows in a JFlow program are easily characterized in a local manner for each statement or expression in the language, as this chapter shows.

4.2 Static checking framework

For the sake of clarity, certain simplifications are made when describing the static checking of JFlow programs. In JFlow, as in Java, a class may be named with a fully qualified name, or with only its base name if either the class or its package has been imported. The rules in this chapter ignore this complication because it is orthogonal to information flow checking. For this reason, all classes are assumed to reside in the same package and names are unqualified. Similarly, visibility modifiers such as `public` or `private` also are ignored: all classes and class members are assumed to be public for the purpose of checking information flow. The

standard visibility checking and class name resolution performed by a Java compiler suffices for JFlow as well.

Before presenting the rules for checking the various language constructs, it will be necessary to establish certain notational and semantic conventions to permit the concise expression of these rules. The purpose of this section is to describe this basic framework upon which the static checking rules are built. The static checking rules are then presented in Sections 4.4 through 4.7.

4.2.1 Type checking vs. label checking

The JFlow compiler performs two kinds of static checking as it compiles a program: type checking and label checking. These two aspects of checking cannot be disentangled entirely, because labels are type constructors and appear in the rules for subtyping. However, the checks needed to show that a statement or expression is sound largely can be classified as either type or label checks. This chapter focuses on the rules for checking labels, because type-checking JFlow is almost exactly the same as type-checking Java. However, there are some interesting interactions between the two kinds of checking.

Static type checking is typically expressed as an attempt to prove a *type judgement*. In inference rules for static type checking, the formula $A \vdash E : T$ typically has the meaning that in the environment A, the expression E has the type T. If the expression E is the entire program, this formula expresses the idea that the program is well-typed. The environment A captures information about the context in which the expression E occurs, or about the context in which the entire program is being checked; in a typical compiler, A is the symbol table. In this work, this formula will be written as $A \vdash_T E : T$, with the subscript T indicating a judgement in the type domain.

Since this thesis is about statically checking information flow, the formula $A \vdash E : X$ is used to indicate a judgement in the domain of information flow. By analogy with type checking, one might expect that the letter X in this formula represents a label. However, this is not the case, because of the need to describe exceptions fully. Instead, the letter X is used to represent a set of *path labels*, which capture information flow along all the possible ways in which the expression can terminate. We will return to the structure of path labels in Section 4.2.3.

4.2.2 Environments

Programs in JFlow are checked for correctness in an environment, which is a binding from symbols (names of various entities) to associated information. These symbols may be names of classes, principals, local variables, and other pieces of the static checking context. The environment also contains the static principal hierarchy and the static authority. The letter A is used in the static checking rules to represent an environment. The binding of the symbol *id* in the environment A is written as $A[id]$. New environments are created by the expression form $A[id := B]$, which creates a new environment identical to A except that the symbol *id* is re-bound to B.

A	an environment, which maps from an identifier such as a variable name to its binding
A^g	the global environment, containing all class definitions and environmental information external to the program being checked
$A[id]$	the binding of identifier *id* in A
$A[id := B]$	a new environment with *id* re-bound to B
$A \vdash E : X$	The expression E generates path labels X when evaluated in environment A.
$A \vdash S : X$	Statement S generates path labels X in environment A.
$A \vdash p_1 \succeq p_2$	The principal p_1 is known to act for the principal p_2, based on the knowledge of the principal hierarchy contained in A.
$A \vdash L_1 \sqsubseteq L_2$	The label L_1 is at most as restrictive as the label L_2, given the knowledge of the principal hierarchy contained in A.
$A \vdash L_1 \approx L_2$	L_1 is equivalent to L_2, given the principal hierarchy contained in A.
$A \vdash_T E : T$	The expression E has type T.
$A \vdash_T T_1 \leq T_2$	The type T_1 is a subtype of the type T_2.
$A \vdash y = \mathit{predicate}(x_1, x_2, \ldots)$	The predicate named *predicate* is true in environment A.

Figure 4.1: Environments and judgements

The *global environment*, A^g, contains definitions for all the classes in the system, and any constant part of the principal hierarchy. As code is checked, more complex environments are constructed that extend A^g to contain definitions for local variables, class parameters, and other bindings.

In addition to the judgements just described ($A \vdash E : X$ and $A \vdash_T E : T$), a few more judgements will be used to describe the static correctness of JFlow. For convenience, these judgements and the syntax for environments just described are summarized in the table of Figure 4.1, but will be explained in more detail as they are introduced.

One convention worth explaining is the syntax for proving auxiliary predicates (the final line in Figure 4.1). The convention followed is that the variable or variables y represent outputs and variables x_i represent inputs. Although in a formal sense there is no difference between inputs and outputs in a predicate or an inference rule, in the natural implementation of these rules some predicate arguments are outputs, and it is useful to distinguish them on this basis.

4.2.3 Exceptions

An important limitation of earlier attempts to create languages for static flow checking has been the absence of usable exceptions. For example, in the original work by Denning and Denning on static flow checking [DD77], exceptions terminated the program, because any other treatment of exceptions could leak information. Subsequent work has avoided exceptions entirely.

It might seem unnecessary to treat exceptions directly, because in many languages, a function that generates exceptions can be desugared into a function that returns a *discriminated union* or *oneof*. However, this approach leads to coarse-grained tracking of information flow. The obvious way to treat oneof types is by analogy with record types. Each arm of the oneof has a distinct label associated with it. In addition, there is an added integer field tag that indicates which of the arms of the oneof is active. The problem with this model is that every assignment to the oneof will require that $\{\mathsf{tag}\} \sqsubseteq \underline{\mathsf{pc}}$, and every attempt to use the oneof will read $\{\mathsf{tag}\}$ implicitly. As a result, every arm of the oneof effectively will carry the same label. For modeling exceptions, this is an unacceptable loss of precision.

Another reason why it might seem unnecessary to treat exceptions directly is that exceptions are usually ignored even in treatments of static type checking. However, it is not feasible to ignore exceptions when checking information flow, because an exception ignored by static checking leads to a possible security violation. One reason why static type checking rules often ignore exceptions may be the legacy of the programming language ML [MTH90], which is strongly typed, and also statically typed except when an expression terminates with an exception, which the static type checking rules ignore. Other programming languages such as CLU [LAB+84] and Theta [LCD+94] do statically check exceptions, and languages such as C++ [Sto87], Modula-3 [Nel91], and Java also treat at least some exceptions statically.

In JFlow, all exceptions except FatalError are checked statically. For each expression or statement, the static checker determines its *path labels*, which are the labels for the information transmitted by various possible termination paths such as normal termination, termination through exceptions, termination through a return statement, and so on. This fine-grained analysis avoids the unnecessary restrictiveness that would be produced by desugaring: each exception that can be thrown by evaluating a statement or expression has a possibly distinct label that is transferred to the $\underline{\mathsf{pc}}$ of catch clauses that might intercept it. Even finer resolution is provided for normal termination and for return termination, where the value label of an expression may differ from the path label. Without this differentiation between the value label and the path label, the $\underline{\mathsf{pc}}$ at a given point in the program would become as restrictive as every value computed prior to that point, making JFlow impractically restrictive.

The path labels for a statement or expression are represented as a total map from paths to labels. Each mapping represents a termination path that the statement or expression might take, and the label of the mapping conservatively indicates what information would be learned if this path were known to be the actual termination path. Paths, the domain of the map, may be one of the following:

- The symbol $\underline{\mathsf{n}}$, which represents normal termination.
- The symbol $\underline{\mathsf{r}}$, which represents termination through a return statement.
- The symbols $\underline{\mathsf{nv}}$ and $\underline{\mathsf{rv}}$ represent the labels of the normal value of an expression and the return value of a statement, respectively. They do not represent paths themselves, but it is convenient to include them as part of the map.

X	a set of path labels: a map from symbols s to labels L
$A \vdash E : X$	The expression E generates path labels X when evaluated in environment A.
s	either a class that extends Throwable, one of the special symbols n, nv, r, or rv, or a pair $\langle$goto *label*$\rangle$ for some statement label *label*, associated with termination through a break or continue statement mentioning *label*
$X[s]$	the label corresponding to path s.
$\bot$	the least restrictive label possible. This label is expressed in programs as $\{\}$, *i.e.*, a label containing no policies.
$\top$	the most restrictive label possible. This label cannot be and does not need to be expressed directly in programs.
$\underline{\emptyset}$	a pseudo-label representing a path that cannot be taken. If $X[s] = \underline{\emptyset}$ for some path s, there is no way for the expression or statement to terminate through the corresponding path.
$X[s := L]$	a set of path labels identical to X, except that the label associated with the path s is changed to L.
$X_\emptyset$	a set of path labels describing an expression that does not terminate : $\forall s\ X_\emptyset[s] = \underline{\emptyset}$
$X_1 \oplus X_2$	the join of two sets of path labels, which is simply the join of all corresponding labels: $X = X_1 \oplus X_2 \quad \equiv \quad \forall s\ (X[s] = X_1[s] \sqcup X_2[s])$
exc	This function is useful for creating path labels for expressions that throw exceptions, and is defined as follows, where C represents an exception type (a class that extends Throwable): *exc-label*$(X, C) = \bigsqcup_{C' : (C' \leq C \vee C \leq C')} X[C']$

Figure 4.2: Definitions for path labels

- Names of classes that inherit from Throwable. Such a class represents an exception, and a mapping from the class represents the path of termination through that exception.
- A tuple of the form $\langle$goto $\mathcal{L}\rangle$ represents termination by executing a named break or continue statement that jumps to the target $\mathcal{L}$. A break or continue statement that does not name a target is represented by the tuple $\langle$goto $\epsilon\rangle$. .

Members of the domain of X (paths) are denoted by s. (Unfortunately, the letter p is already heavily overloaded.) The same notation used for environments is also used for path labels: the expression $X[s]$ denotes the label that X maps s to, and the expression $X[s := L]$ denotes a new map that is exactly like X except that the path s is bound to the label L. The range of path labels is not precisely the set of labels; it is the set of labels augmented with the pseudo-label $\underline{\emptyset}$. If a path s is mapped to $\underline{\emptyset}$, it indicates that the statement cannot terminate through the path s. When used in joins, the label $\underline{\emptyset}$ behaves as if it were lower than any other label: $L \sqcup \underline{\emptyset} = L$ for all labels, including the label $\{\}$. Figure 4.2 summarizes this notation and defines some additional notation relating to path labels.

L	a label or the special value $\underline{\emptyset}$.
l	a label expression, which produces a label when intepreted in an environment
T	a type
t	a type expression
τ	a labeled type expression: an expression of the form $t\{l\}$ or t. The function *labeled*(τ) distinguishes between these two cases.
p	a principal or a principal expression (which must be a name)
P	a component (policy) of a label (See Section 4.2.7)
$\mathcal{P}$	a formal parameter of a class
q	an actual parameter of a class, as a program expression
$\mathcal{Q}$	an actual parameter of a class, as part of a type
C	the name of a class
v	the name of a variable
S	a statement
$\mathcal{S}$	a method or field signature
$\mathcal{M}$	a complete method declaration, including its implementation

Figure 4.3: Additional conventions

4.2.4 Additional notation conventions

Certain other conventions that are used in this chapter are worth mentioning at this point. In the rules that follow, the symbols used suggest the kind of type, value, or expression being denoted. These conventions are summarized in Figure 4.3 for easy reference, and are described in more detail when used later.

Sequences of items of the same kind are represented by the notation $..x_i..$. The letters i, j, k, l, and m are used only as indices into such sequences. Items in the sequence are assumed to be separated by the appropriate delimiters (*e.g.*, "," and ";"), though these delimiters will be included in some cases for clarity, as in the expression $..; x_i; ..$. An equation in which an index variable such as i appears holds for all i in its range, which is 1 to $\max(i)$ unless explicitly indicated otherwise. A sequence of items $..x_i..$ is distinct from a sequence $..x_j..$; the subscript is used not only to index the items, but also to distinguish them. This convention is chosen for its compactness, and is inspired by the convention of repeated indices used in relativistic physics.

Optional items are indicated by large brackets, as in the expression $\left[x\right]$. In many rules, these optional expressions denote an implicit variable generated by unification against some syntactic form or component of the environment. For example, consider this rule:

$$\textit{extend}(A, \left[\mathsf{final}\right]\ \tau\ v) = A[v := \langle \mathsf{var}\ \left[\mathsf{final}\right]\ \textit{type-part}(\tau, A)\{\textit{var-label}(\tau, A)\}\ ]\rangle$$

$\langle$var $T\{L\}$ *uid*$\rangle$	the name of a mutable (non-final) variable maps to this tuple, representing a variable of type T and label L
$\langle$var final $T\{L\}$ *uid*$\rangle$	a final variable
$\langle$param principal *uid*$\rangle$	a parameter of type principal
$\langle$param label *uid*$\rangle$	a parameter of type label
$\langle$covariant label *uid*$\rangle$	a covariant parameter of type label
$\langle$class C ... $\{\ldots\}\rangle$	a class. The entire class declaration is stored in the environment.
$\langle$constant principal$\rangle$	a real principal external to the program
$\langle$goto $\mathcal{L}\rangle$	a variable representing the pc of the statement labeled by the break or continue target $\mathcal{L}$

Figure 4.4: Environment mappings

The $\left[\text{final}\right]$ on the right is present whenever the corresponding option is present in the argument to *extend*. Optional items are also used as the condition of an if expression; in this case the condition is understood to be true if the optional item is present. The notation $\left[\ \right]$ is used to represent an empty optional value. In some cases the brackets are written in a subscript, as in $\left[\text{final}\right]_n$. In this case, the subscript is used to distinguish different optional items.

4.2.5 Environment bindings

In the JFlow static checker, environments store a variety of different kinds of information. Certain information is stored in the environment under special symbols. These special symbols are auth, pc, and ph:

A[auth]	the set of principals that the program is known to be authorized to act for at a particular point in the program: the *static authority*
A[pc]	the program-counter label
A[ph]	the static principal hierarchy. This is a set of pairs of principals (p, p'), meaning that p is known to act for p' in the environment A.

The environment also contains mappings for various named entities, such as local variables. The mappings shown in Figure 4.4 are found in the environment. In these bindings and elsewhere in the rules, the notation *uid* represents a unique identifier that is generated during static analysis and that distinguishes program identifiers that share the same name.

As indicated, classes and interfaces are entered in the environment. In order to support mutual references among classes, class and interface bindings are present in the global environment, A^g, from which all other environments are generated by extension. The global environment also contains some other information; the entry A^g[ph] contains a part of the static principal hierarchy that is assumed to be constant. Code compiled against such a global environment will need to be invalidated if the relations described in A^g[ph] are revoked. Similarly, the entry A^g[auth] contains principals willing to grant their authority to the code

being compiled (or more precisely, being added to the system). Again, if any of these principals revoke their grant of authority, the code must be invalidated.

4.2.6 Representing principals

For almost all JFlow entities, including principals, types, and labels, a sharp distinction is drawn between the syntactic expression denoting an entity and the representation of the entity that is used during static checking. For example, principals are named in JFlow programs using identifiers. These identifiers may be the names of principals external to the program, or parameters denoting unknown principals, or names of variables of type principal. However, during static checking, principals are represented by one of three kinds of tuples:

$\langle$pr-external $p\rangle$	a principal external to the program: typically, a username
$\langle$pr-param *uid*$\rangle$	a static principal parameter. Static parameters have no run-time representation.
$\langle$pr-dynamic *uid* $L\rangle$	a run-time principal variable. The label L is the label of this variable, and keeps track of what information is conveyed by knowing which principal this variable denotes.

Principals appearing in a policy expression may take any of these forms. These forms do not appear in the range of the environment map; for example, a variable of type principal maps to a tuple of form $\langle$var final principal$\{L\}$ *uid*$\rangle$ rather than to one of form $\langle$pr-dynamic $\ldots\rangle$. The mapping from principal identifiers to their internal representation is performed by the function *interp-P*, which is short for "interpret principal". This function assumes that an appropriate environment entry has been installed for the identifier in question. How this is done will become clear later.

interp-P(id, A) = case $A[id]$ of
    $\langle$constant principal$\rangle$: $\langle$pr-external $id\rangle$
    $\langle$param principal *uid*$\rangle$: $\langle$pr-param *uid*$\rangle$
    $\langle$var final principal$\{L\}$ *uid*$\rangle$: $\langle$pr-dynamic *uid* $L\rangle$
end

4.2.7 Representing labels and components

Labels are also represented differently during static checking than in program expressions. A label is expressed in a JFlow program as a set of component expressions $\{..; P_i; ..\}$ separated by semicolons. The letter P denotes a component here (P stands for *policy*). These component expressions may be policy expressions, components that name a variable or parameter, or dynamic components. During static checking, the label is represented as a join of components produced by interpreting the corresponding component expressions. A label L is written as $P_1 \sqcup \ldots \sqcup P_n$, or $.. \sqcup P_i \sqcup ..$, or even as $..P_i..$. As with principals, components and component expressions are represented with different notation. There are four possible forms for a component, corresponding to the allowed ways to write a component expression:

interp-L$(o : ..r_i.. , A) = \langle$policy *interp-P*$(o, A) : .., $*interp-P*$(r_i, A), ..\rangle$

interp-L$(v, A) =$
  case $A[v]$ of
    $\langle$var $\left[\text{final}\right]$ $T\{L\}$ *uid*$\rangle : L$
    $\langle$covariant label *uid*$\rangle : \langle$covariant-label *uid*$\rangle$
    $\langle$param label *uid*$\rangle : \langle$label-param *uid*$\rangle$
    $\langle$constant principal$\rangle : \{\}$
    $\langle$param principal *uid*$\rangle : \{\}$
  end

interp-L$(*v, A) =$
  case $A[v]$ of
    $\langle$var final label$\{L\}$ *uid*$\rangle : \langle$dynamic *uid* $L\rangle$
  end

Figure 4.5: Interpreting labels

$\langle$policy $o : .., r_i, ..\rangle$ represents a policy: a label component with an explicit owner o and readers r_i, all of which are principals. This kind of component is generated by a policy expression of the form o: $..r_i..$.

$\langle$label-param *uid*$\rangle$ a fixed but unknown label, corresponding to an explicit class label parameter.

$\langle$covariant-label *uid*$\rangle$ a fixed but unknown label, corresponding to a class parameter of type label that has been declared to be covariant, or to an implicit argument label parameter

$\langle$dynamic *uid* $L\rangle$ the dynamic label contained in a final variable of type label. This kind of component is generated by an expression of the form *v, where v is the variable. The environment A is the environment that exists after the declaration of the variable v.

$\langle$variable *uid*$\rangle$ An undeclared label, resulting from a label that was omitted from the program. A label of this sort is inferred by a constraint solver, as described in Chapter 5. In the definitions later, the function *fresh-variable*() produces new labels containing a single variable component, with a fresh identifier *uid*. Its definition is *fresh-variable*$() = \langle$variable *fresh-uid*$()\rangle$, where the function *fresh-uid*() generates a unique identifier never before used during static checking.

A label expression in a program is converted into a join of components by the function *interp-L*, which interprets the individual component expressions and joins them together:

interp-L$(\{P_1; \ldots; P_n\}, A) =$ *interp-L*$(P_1, A) \sqcup \ldots \sqcup$ *interp-L*(P_n, A)

A component expression is interpreted straightforwardly, producing one of the kinds of policies above. This interpretation process is shown formally in Figure 4.5. Some of the details of label interpretation hold interest. As the first definition shows, a policy is interpreted by recursively interpreting the principals named

in the policy. A component expression consisting of an identifier is interpreted differently depending on the significance of the identifier. An identifier that is the name of a variable simply denotes the label of that variable when used as a component expression. An identifier that is a label parameter denotes that label parameter. Other identifiers such as the names of external principals are not associated with any information flow, and denote the empty label, $\{\}$. Finally, the contents of a variable v of type label may be used to construct a dynamic component using the notation $*v$.

4.2.8 Representing types

Some care must be taken to represent JFlow types unambiguously during static checking. Java has three kinds of type constructors: class types, interface types, and arrays. JFlow adds labels and the ability to instantiate a class on some parameters. The internal representation of a class or interface type is a symbol (the name of the class) followed by a possibly empty sequence of parameters. Basic types such as int are represented in this way, with an empty sequence of parameters: int[ ]. Arrays are represented by the symbol array, followed by two parameters: the type of contained elements, and their common label. Thus, the type int{L}[ ] is represented internally as array[int,L]. As in Java, arrays are the only type that allow another type as a parameter.

The predicate *interp-T* translates a type expression into this internal representation, as shown in Figure 4.6. For convenience in expressing static-checking rules, this predicate is written as if it were a function. When interpreting instantiations of parameterized classes, the predicate *interp-param* is used to interpret the actual parameters used.

The first two rules for *interp-T* show how simple object types are interpreted. The first rule shows interpretation of a non-parameterized class, which is treated exactly like a parameterized class having no parameters. The second rule shows how a parameterized instantiation is interpreted, using the *interp-param* predicate. The third and fourth rules define interpretation of a JFlow array type in accordance with Section 3.3.6. The final three rules show how actual parameters to a parameterized class are interpreted. The only subtle issue for parameter interpretation is that a non-covariant formal label parameter may not be supplied with a covariant actual label parameter, as in the fifth rule. The predicate *invariant* is defined in the next section.

In the static checking rules in this chapter, the symbol τ is used to represent a labeled type expression: an expression of the form $t\{l\}$ or t. For convenience, the functions *labeled*, *type-part*, and *label-part* are used to manipulate labeled type expressions, as defined in Figure 4.7.

4.2.9 Invariant vs. covariant types

The presence of covariant label parameters makes it necessary to distinguish between *invariant* and *covariant* types. Invariant types are types that do not mention any covariant label parameters; the meaning of an invariant type does not vary with the parameter. Covariant types are types that vary with one or more covariant label parameters. A type is invariant as long as all of its actual label parameters are invariant. The

$$\frac{A[C] = \langle \mathsf{class}\ C\ \ldots \{\ldots\}\rangle}{C[\,] = \textit{interp-T}(C, A)}$$

$$\frac{\begin{array}{c}A[C] = \langle \mathsf{class}\ C[..\mathcal{P}_i..]\ \ldots \{\ldots\}\rangle \\ \mathcal{Q}_i = \textit{interp-param}(q_i, \mathcal{P}_i, A)\end{array}}{C[..\mathcal{Q}_i..] = \textit{interp-T}(C[..q_i..], A)}$$

$$\frac{\begin{array}{c}T = \textit{interp-T}(t, A) \\ \textit{invariant}(T)\end{array}}{\mathsf{array}[T, \bot] = \textit{interp-T}(t[\,], A)}$$

$$\frac{\begin{array}{cc}L = \textit{interp-L}(l, A) & T = \textit{interp-T}(t, A) \\ \textit{invariant}(L) & \textit{invariant}(T)\end{array}}{\mathsf{array}[T, L] = \textit{interp-T}(t\{l\}[\,], A)}$$

$$\frac{\begin{array}{c}L = \textit{interp-L}(q, A) \\ \textit{invariant}(L)\end{array}}{L = \textit{interp-param}(q, \mathsf{label}\ \textit{id}, A)}$$

$$\frac{L = \textit{interp-L}(q, A)}{L = \textit{interp-param}(q,\ \mathsf{covariant\ label}\ \textit{id}, A)}$$

$$\frac{p = \textit{interp-P}(q, A)}{p = \textit{interp-param}(q,\ \mathsf{principal}\ \textit{id}, A)}$$

Figure 4.6: Interpreting type expressions

$$\begin{array}{l}\textit{labeled}(t\{l\}, A) = \textit{true} \\ \textit{type-part}(t\{l\}, A) = \textit{interp-T}(t, A) \\ \textit{label-part}(t\{l\}, A) = \textit{interp-L}(l, A)\end{array}$$

$$\begin{array}{l}\textit{labeled}(t, A) = \textit{false} \\ \textit{type-part}(t, A) = \textit{interp-T}(t, A) \\ \textit{label-part}(t, A) = \bot\end{array}$$

Figure 4.7: Definitions for labeled types

$$\frac{\begin{array}{l}\mathsf{case}\ \mathcal{Q}_i\ \mathsf{of} \\ \quad ..\sqcup P_j \sqcup .. : \nexists j, \textit{uid}\ P_j = \langle \mathsf{covariant\text{-}label}\ \textit{uid}\rangle \\ \quad \mathsf{else}\ \textit{true} \\ \mathsf{end}\end{array}}{\textit{invariant}(C[..\mathcal{Q}_i..])}$$

Figure 4.8: Determining type invariance

$$\frac{A \vdash p_1 \succeq p_2 \quad A \vdash p_2 \succeq p_3}{A \vdash p_1 \succeq p_3}$$

$$\frac{uid_1 = uid_2}{\begin{array}{c} A \vdash \langle \textsf{pr-param } uid_1 \rangle \succeq \langle \textsf{pr-param } uid_2 \rangle \\ A \vdash \langle \textsf{pr-dynamic } uid_1\ L_1 \rangle \succeq \langle \textsf{pr-dynamic } uid_2\ L_2 \rangle \\ A \vdash \langle \textsf{pr-external } uid_1 \rangle \succeq \langle \textsf{pr-external } uid_2 \rangle \end{array}}$$

$$\frac{(p_1', p_2') \in A[\underline{\textsf{ph}}] \quad \textit{get-uid}(p_1') = \textit{get-uid}(p_1) \quad \textit{get-uid}(p_2') = \textit{get-uid}(p_2)}{A \vdash p_1 \succeq p_2}$$

Figure 4.9: Inferring the $\succeq$ relation

predicate *invariant*(T), defined in Figure 4.8, uses this simple rule. For a label L to be invariant, it must not contain any components of the form $\langle$covariant-label *uid*$\rangle$. This condition can also be expressed by requiring that the label L for any label parameter may be at most as restrictive as L_{inv}, a label that contains every label component *except* components of the form $\langle$covariant-label *uid*$\rangle$. It is an ordinary member of the set of labels, but one that is too large to write down.

4.3 Basic rules

Using the representations of principals, labels, and types that have just been defined, the basic rules for reasoning about these entities can now be expressed, starting with principals.

4.3.1 Reasoning about principals

In an environment A, the static principal hierarchy is stored in the component $A[\underline{\textsf{ph}}]$, which is a set of pairs of principals (p_1, p_2). The notation $A \vdash p_1 \succeq p_2$ means that given the static knowledge contained in the environment A, the principal p_1 is known to act for the principal p_2. The necessary reflexivity and transitivity of the static principal hierarchy (see Section 2.1.1) is achieved by inference rules that transitively and reflexively extend the set of pairs in $A[\underline{\textsf{ph}}]$. These rules are shown in Figure 4.9. The first rule expresses the transitivity of the acts-for relation. The second rule captures the reflexive property of the acts-for relation. The third rule describes how the static principal hierarchy is accessed to check acts-for relations. The function *get-uid* extracts the *uid* component of a principal.

4.3.2 Reasoning about labels

The rules shown in Figure 4.10 are used for checking label constraints. The first two rules are simply the complete relabeling rule from Section 2.4.3. The next two rules show that non-policy components are

$$\frac{L = \{..P_i..\} \quad L' = \{..P'_j..\} \quad \forall i\ \exists j\ A \vdash P_i \sqsubseteq P'_j}{A \vdash L \sqsubseteq L'}$$

$$\frac{A \vdash o' \succeq o \quad \forall j\ (A \vdash r'_j \succeq o\ \vee\ \exists i\ A \vdash r'_j \succeq r_i)}{A \vdash \langle \mathsf{policy}\ o : ..r_i..\rangle \sqsubseteq \langle \mathsf{policy}\ o' : ..r'_j..\rangle}$$

$$\frac{\textit{true}}{A \vdash \langle \mathsf{label\text{-}param}\ \textit{uid}\rangle \sqsubseteq \langle \mathsf{label\text{-}param}\ \textit{uid}\rangle}$$

$$\frac{\textit{true}}{A \vdash \langle \mathsf{covariant\text{-}label}\ \textit{uid}\rangle \sqsubseteq \langle \mathsf{covariant\text{-}label}\ \textit{uid}\rangle}$$

$$\frac{A \vdash L_1 \sqsubseteq L_2}{A \vdash \langle \mathsf{dynamic}\ \textit{uid}\ L_1\rangle \sqsubseteq \langle \mathsf{dynamic}\ \textit{uid}\ L_2\rangle}$$

$$\frac{A \vdash L \sqsubseteq L' \quad A \vdash L' \sqsubseteq L}{A \vdash L \approx L'}$$

Figure 4.10: Inferring the $\sqsubseteq$ relation

treated as if they were opaque. The final rule reduces reasoning about label equivalence to reasoning about relabeling.

These rules say nothing about label variables: components of the form $\langle \mathsf{variable}\ \textit{uid}\rangle$. The rules in Figure 4.10 cannot be applied fully until all label variables are given satisfying assignments, replacing them with one of the other kinds of components defined in Section 4.2.7.

In the fourth rule, a dynamic component can be relabeled to another dynamic component only if they have the same *uid*; in other words, if they are the contents of the same variable of type label. Otherwise, they correspond to the contents of different variables, and no static relationship can be inferred. The relationship between two such components depends on their contained labels L_1 and L_2. One would expect that these contained labels would be the same, because they are the labels of the same variable. However, such components can acquire different labels during constraint solving, because the label of the variable (of type label) is being automatically inferred. In this case, the contained label is a conservative approximation to the true label of the variable, and different dynamic components may contain different conservative approximations.

4.3.3 Class scope and environments

JFlow is unique among languages that support static checking of information flow because it fully supports objects. It is also unique in its support for parameterization, including parameterization over both labels and

$$
\begin{array}{l}
\textit{class-env}(C[..\mathcal{Q}_i..]) = \\
\quad \mathsf{case}\ A^g[C]\ \mathsf{of} \\
\quad\quad \langle \mathsf{class}\ C\Big[[..\mathcal{P}_i..]\Big] \ldots \rangle : A^g[..\textit{param-id}(\mathcal{P}_i) := \mathcal{Q}_i..] \\
\quad \mathsf{end} \\
\\
\textit{inner-class-env}(C) = \\
\quad \mathsf{case}\ A^g[C]\ \mathsf{of} \\
\quad\quad \langle \mathsf{class}\ C\Big[[..\mathcal{P}_i..]\Big] \ldots \Big[\mathsf{authority}(..p_k..)\Big] \rangle : \\
\quad\quad\quad \mathsf{let}\ A = A^g[..\textit{param-id}(\mathcal{P}_i) := \textit{formal-to-actual}(\mathcal{P}_i)..]\ \mathsf{in} \\
\quad\quad\quad\quad A[\underline{\mathsf{auth}} := \{..\textit{interp-P}(p_k, A)..\}] \\
\quad\quad\quad \mathsf{end} \\
\quad \mathsf{end} \\
\\
\textit{param-id}(\mathcal{P}) = \\
\quad \mathsf{case}\ \mathcal{P}\ \mathsf{of} \\
\quad\quad \Big[\mathsf{covariant}\Big]\mathsf{label}\ \textit{id} : \textit{id} \\
\quad\quad \mathsf{principal}\ \textit{id} : \textit{id} \\
\quad \mathsf{end} \\
\\
\textit{formal-to-actual}(\mathcal{P}) = \\
\quad \mathsf{case}\ \mathcal{P}\ \mathsf{of} \\
\quad\quad \mathsf{covariant\ label}\ \textit{id} : \langle \mathsf{covariant\ label}\ \textit{fresh-uid}() \rangle \\
\quad\quad \mathsf{label}\ \textit{id} : \langle \mathsf{param\ label}\ \textit{fresh-uid}() \rangle \\
\quad\quad \mathsf{principal}\ \textit{id} : \langle \mathsf{param\ principal}\ \textit{fresh-uid}() \rangle \\
\quad \mathsf{end}
\end{array}
$$

Figure 4.11: Modifying an environment for class scope

principals. This section describes several functions and predicates that support these features.

Handling class parameters. A class in JFlow has a possibly empty list of formal parameters that may be instantiated with actual parameters of the appropriate sort. For code both external and internal to the class, it is necessary to create environments in which these formal parameters are bound. Functions for creating these augmented environments are defined in Figure 4.11.

The function *class-env* is used when checking code external to the class, where that code mentions an instantiation of the class. It augments the environment with definitions for the parameters of a class, given some instantiation of the class on parameters, creating a binding from each formal parameter of the class to the corresponding actual parameter used in the instantiation.

The function *inner-class-env* also augments environments with class parameters, creating an environment for checking the code of the class itself. It adds definitions for the parameters of the class, but treats the formal parameters as actual parameters of the appropriate type. Checking the code of a class against these definitions ensures that the class is safe for all possible actual parameters that might be supplied. For

$$\frac{A' = A[\mathsf{this} := \langle \mathsf{var\ final}\ C[..\mathcal{Q}_i..]\{A[\underline{\mathsf{pc}}]\}\ \textit{fresh-uid}()\rangle] \quad A'' = \textit{extend-all-ivars}(A', C[..\mathcal{Q}_i..])}{A'' = \textit{obj-env}(A, C[..\mathcal{Q}_i..])}$$

$$\begin{array}{l}\mathsf{case}\ A^g[C]\ \mathsf{of}\\ \quad \langle \mathsf{class}\ C\left[[..\mathcal{P}_i..]\right]\left[\mathsf{implements}\ .., t_j, ..\right]\{\ldots\}\rangle :\\ \quad\quad A' = A\\ \quad \langle \mathsf{class}\ C\left[[..\mathcal{P}_i..]\right]\ \mathsf{extends}\ t_s\ \ldots\rangle :\\ \quad\quad A' = \textit{extend-all-ivars}(A, \textit{interp-T}(t_s, \textit{class-env}(C[..\mathcal{Q}_i..])))\\ \mathsf{end}\end{array}$$

$$\frac{A'' = \textit{extend-ivars}(A', C[..\mathcal{Q}_i..])}{A'' = \textit{extend-all-ivars}(A, C[..\mathcal{Q}_i..])}$$

$$\frac{\begin{array}{c}A^g[C] = \langle \mathsf{class}\ C \ldots \{\ldots \left[\mathsf{final}\right]_n \tau_n\ v_n \ldots\}\rangle\\ V = \{n \mid \left[\mathsf{final}\right]_n \wedge\ \tau_n = \mathsf{label}\{l_n\}\}\\ V' = \{n \mid \left[\mathsf{final}\right]_n \wedge\ \tau_n = \mathsf{principal}\{l_n\}\}\\ V = \{..n_j..\}\\ V' = \{..n'_k..\}\\ L_n = \textit{fresh-variable}()\\ A' = A[..v_{n_j} := \langle \mathsf{var\ final\ label}\{L_n\}\ \textit{fresh-uid}()\rangle..]\\ A'' = A[..v_{n'_k} := \langle \mathsf{var\ final\ principal}\{L_n\}\ \textit{fresh-uid}()\rangle..]\\ A'' \vdash L_n \approx \textit{interp-L}(l_n, A'')\end{array}}{A'' = \textit{extend-ivars}(A, C[..\mathcal{Q}_i..])}$$

Figure 4.12: Extending the object environment

example, a class parameter of type label is bound to a label containing a single component of the form $\langle$param label *fresh-uid*()$\rangle$, where *fresh-uid*() is the function that generates a previously unused identifier. The static checking rules treat this component as an opaque label about which nothing is known except that it is equivalent to itself. Because this condition holds for any possible label, code parameterized over this label will be sound regardless of what actual parameter that code is instantiated on.

Building object environments. In JFlow, final instance variables of type label and principal may be used to construct dynamic label components and policies when their containing objects are in scope. For example, one instance variable of type label may be used to label another instance variable in the same object. These instance variables may also be used to construct labels within non-static methods of the class.

When performing static checking, the *obj-env* predicate extends the environment to add definitions for final instance variables of type label or principal. Its definition is shown in Figure 4.12. The primary use of *obj-env* is for checking the correctness of a method body. In this context, the variable this also is in scope. Other instance variables do not need to be placed in scope because an ordinary access to an instance variable

x is treated as the expression $\mathsf{this}.x$.

The predicate *extend-all-ivars* ensures that all the appropriate instance variables are added to the environment. It too is defined in Figure 4.12. Instance variables are added to the environment starting from the topmost superclass, and working down. This ordering ensures that any instance variables that shadow superclass definitions are bound correctly.

The predicate *extend-ivars*, also shown in Figure 4.12, adds the final instance variables of type label and principal that are members of the single class that is the second argument. The rule works by extracting the indices of the final variables of type label and principal into variables $..n_j..$ and $..n'_k..$, respectively. These indices are used to select the variables that are entered successively into the augmented environments A' and A''. This process is complicated by the fact that the labels of the instance variables may refer to each other. For each instance variable v_n, a new label variable L_n is used to handle the potential circularity. The label L_n is the label used when entering the variable into the environment, and it is required by the final antecedent to be equivalent to the interpretation of the declared label of the variable (l_n) in the environment in which all of the necessary instance variables are defined (A'').

Instance variable and method signatures. An important part of static checking is looking up the signatures of class members, including members that are inherited from superclasses. These class members include both instance variables and methods.

The judgement $\mathcal{S} = \textit{signature}(T, f)$ has the meaning that the member f of the type T has the signature $\mathcal{S}$. The type T must be a class type. The rules for looking up signatures are given in Figures 4.13. The member f may be either the name of an instance variable, or a method identifier, which is of the form $m(T_i)$, where the T_i are the types of the arguments. If f is the name of an instance variable, the signature has the form $\langle \left[\mathsf{final}\right] \ \tau \ \textit{id} \rangle$. When using the rule provided to look up method signatures, a signature match the argument types T_i only if the argument types in the signature are supertypes of the corresponding types t_i. This condition is the final antecedent of the rule for method signature lookup. However, using this rule, multiple overloaded signatures may satisfy given argument types T_i. In Java, this situation is a static error unless one of the signatures is at least as specific as all the others. The rules given here do not capture this aspect of static checking, for the sake of simplicity.

Methods and fields can also be inherited from superclasses, using the last rule in Figure 4.13. In this rule, t_s represents the type expression for the superclass of C, and T_s represents the superclass of C. The type expression t_s is interpreted in the environment $\textit{class-env}(C[..\mathcal{Q}_i..])$ because it may mention formal parameters of the class C. The same rule holds for methods as well, if $m(T_j)$ is substituted for f.

4.3.4 Reasoning about subtypes

Consider the judgement $A \vdash_T S \leq T$, which is relevant to JFlow, as to all languages with subtyping. Here, S and T are ordinary unlabeled types. The subtype rule is as in Java, except that it handles class parameters. If S or T is an instantiation of a parameterized class, subtyping is invariant in the parameters except when a

$$\frac{A^g[C] = \langle \mathsf{class}\ \Big[[..\mathcal{P}_i..]\Big] \ldots \{\ldots \Big[\mathsf{final}\Big]\ \tau\ f\ \ldots\} \quad A = \textit{obj-env}(A^g, C[..\mathcal{Q}_i..]) \quad T_f = \textit{type-part}(\tau, A) \quad L_f = (\mathsf{if}\ \textit{labeled}(\tau)\ \mathsf{then}\ \textit{label-part}(\tau, A)\ \mathsf{else}\ \bot)}{\langle \Big[\mathsf{final}\Big]\ T_f\{L_f\}\rangle = \textit{signature}(C[..\mathcal{Q}_i..], f)}$$

$$\frac{\begin{array}{l} A^g[C] = \langle \mathsf{class}\ \Big[[..\mathcal{P}_i..]\Big] \ldots \{ \\ \quad \ldots \Big[\mathsf{static}\Big]\ \tau_r\ m\Big[\{I\}\Big](..\tau_j\ a_j..)\Big[:\{R\}\Big]\ \mathsf{throws}(..\tau_k..)\ \mathsf{where}\ \mathcal{K}_l\ \{S\} \ldots \\ \}\rangle \end{array} \quad A = \textit{class-env}(C[..\mathcal{Q}_i..]) \quad A \vdash_T T_j \leq \textit{type-part}(\tau_j, A)}{\langle \Big[\mathsf{static}\Big]\ \tau_r\ m\Big[\{I\}\Big](..\tau_j\ a_j..)\Big[:\{R\}\Big]\ \mathsf{throws}(..\tau_k..)\ \mathsf{where}\ \mathcal{K}_l\rangle = \textit{signature}(C[..\mathcal{Q}_i..], m(T_j))}$$

$$\frac{A^g[C] = \langle \mathsf{class}\ \Big[[..\mathcal{P}_i..]\Big]\ \mathsf{extends}\ t_s\ \ldots \{\ldots\}\rangle \quad f \text{ is not a member of } C \quad T_s = \textit{interp-T}(t_s, \textit{class-env}(C[..\mathcal{Q}_i..])) \quad \mathcal{S} = \textit{signature}(T_s, f)}{\mathcal{S} = \textit{signature}(C[..\mathcal{Q}_i..], f)}$$

Figure 4.13: Looking up field and method signatures

label parameter is declared to be covariant. This subtyping rule is the first one shown in Figure 4.14. Using this rule, Vector[L] (from Figure 3.8) would be a subtype of AbstractList[L'] only if $L \approx L'$.

Checking subtype relations in JFlow is straightforward. If S and T are not instantiations of the same class, it is necessary to walk up the type hierarchy from S to T, rewriting parameters, as shown in the second rule in Figure 4.14. Together, the two rules inductively prove the appropriate subtype relationships, including reflexivity and transitivity. Two instantiations of the same class have a subtype relation if their parameters are equivalent, or if the parameter is a covariant label and the labels have the appropriate relation.

$$\frac{A^g[C] = \langle \mathsf{class}\ C\Big[[..\mathcal{P}_i..]\Big] \ldots \{\ldots\}\rangle \quad (A \vdash \mathcal{Q}_i \approx \mathcal{Q}'_i) \vee (\mathcal{P}_i = \langle \mathsf{covariant\ label}\ \textit{id}\rangle \wedge A \vdash \mathcal{Q}_i \sqsubseteq \mathcal{Q}'_i)}{A \vdash_T C[..\mathcal{Q}_i..] \leq C[..\mathcal{Q}'_i..]}$$

$$\frac{A^g[C] = \langle \mathsf{class}\ C\Big[[..\mathcal{P}_i..]\Big]\ \mathsf{extends}\ t_s \ldots \{\ldots\}\rangle \quad T_s = \textit{interp-T}(t_s, \textit{class-env}(C[..\mathcal{Q}_i..])) \quad A \vdash_T T_s \leq C'[..\mathcal{Q}'_i..]}{A \vdash_T C[..\mathcal{Q}_i..] \leq C'[..\mathcal{Q}'_i..]}$$

Figure 4.14: Subtype rules

$$\frac{\textit{true}}{A \vdash \ ; \ : X_\emptyset[\underline{\mathsf{n}} := A[\underline{\mathsf{pc}}]]}$$

$$\frac{\textit{true}}{A \vdash \textit{literal} : X_\emptyset[\underline{\mathsf{n}} := A[\underline{\mathsf{pc}}], \underline{\mathsf{nv}} := A[\underline{\mathsf{pc}}]]}$$

$$\frac{\begin{array}{c} A \vdash S : X_\emptyset[s := l] \\ s \in \{\underline{\mathsf{n}}, \underline{\mathsf{r}}\} \end{array}}{A \vdash S : X[s := A[\underline{\mathsf{pc}}]]}$$

Figure 4.15: Some simple rules

These rules for checking a subtype relationship between instantiations of parameterized types are similar to the checking performed by the PolyJ compiler, which supports only type parameters [MBL97]. Checking a subtype relation between a class and an interface, or between two interfaces, is done in exactly the same way as between two classes.

4.4 Checking Java statements and expressions

This section presents rules for statically checking information flow in the statements and expressions that JFlow inherits from Java. The semantics for these statements are the same as in Java, so no discussion of their behavior is needed. One kind of Java expression is deferred until Section 4.6: a call to method or constructor, which differs somewhat in JFlow from Java.

4.4.1 Simple rules

Rules for most statement forms can be expressed simply using the definitions provided so far. Figure 4.15 contains some important static-checking rules.

The first rule in the figure is interpreted as follows: an empty statement always terminates normally, with the same $\underline{\mathsf{pc}}$ at its end as at the start. Thus, it simply passes along its $\underline{\mathsf{pc}}$ to any statement that follows it. In the second rule, it is seen that a literal expression such as a numeric constant also terminates normally always, and is labeled with the current $\underline{\mathsf{pc}}$, as described earlier.

The third rule in Figure 4.15 applies to any statement, and is important for relaxing restrictive path labels. The intuitive meaning of this rule is that if a statement can terminate only normally, the $\underline{\mathsf{pc}}$ at the end is the same as the $\underline{\mathsf{pc}}$ at the beginning. The normal termination of the statement gives no new information. The same is true if the statement can terminate only through a `return` statement. This rule is called the *single-path rule*. It would not be safe for this rule to apply to exception paths, so the rule requires that the single path s be either $\underline{\mathsf{n}}$ or $\underline{\mathsf{r}}$. To see why, suppose that a set of path labels formally contains only a single exception path C. However, that path might include multiple paths consisting of exceptions that are

$$\frac{\begin{array}{c}A \vdash E_1 : X_1 \\ A[\underline{\mathsf{pc}} := X_1[\underline{\mathsf{n}}]] \vdash E_2 : X_2 \\ X = X_1[\underline{\mathsf{n}} := \underline{\emptyset}] \oplus X_2\end{array}}{A \vdash E_1 + E_2 : X}$$

$$\frac{\begin{array}{c}A \vdash E_1 : X_1 \\ A[\underline{\mathsf{pc}} := X_1[\underline{\mathsf{n}}]] \vdash E_2 : X_2 \\ X = \mathit{exc}(X_1[\underline{\mathsf{n}} := \underline{\emptyset}] \oplus X_2, X_2[\underline{\mathsf{nv}}], \mathsf{ArithmeticException})\end{array}}{A \vdash E_1/E_2 : X}$$

$$\mathit{exc}(X, L, C) = X[\underline{\mathsf{n}} := X[\underline{\mathsf{n}}] \sqcup L,\ \underline{\mathsf{nv}} := X[\underline{\mathsf{nv}}] \sqcup L,\ C := X[C] \sqcup L]$$

Figure 4.16: Arithmetic rules

subclasses of C. These multiple paths can be discriminated using a try. . . catch statement. Because the Java exception model identifies exceptions with types, and Java supports subtyping, the single-path rule may not be applied safely to exception paths. If exceptions were not identified with types (as in CLU [LAB$^+$84]), the single-path rule could be applied to exceptions too.

4.4.2 Arithmetic

Figure 4.16 gives rules for checking arithmetic operations. Arithmetic operations that cannot throw an exception, such as addition, are covered by the first rule. Java evaluates the second argument to an arithmetic operation only when the first argument terminates normally. Therefore, the second argument is checked statically using a $\underline{\mathsf{pc}}$ of $X_1[\underline{\mathsf{n}}]$. The operation can terminate in any of the ways that E_1 can terminate, except normally, because in that case E_2 would be evaluated. The operation can also terminate in any of the ways that E_2 can terminate. Therefore, the path labels for the whole expression are derived by applying the $\oplus$ operator to the path labels from the individual expressions (X_1 and X_2), with the normal termination path from E_1 removed.

For arithmetic operations that can throw an exception, such as division or modulo, the second rule applies. These operations throw an exception if the second argument is zero. To simplify the description of the static checking, the function *exc* is used. Its definition is repeated at the bottom of the figure. This function creates a set of path labels that are just like the input path labels X, except that they include an additional path, the exception C, with the path label L. If normal termination or the normal termination value are observed, the knowledge that the exception was *not* thrown may leak the same information as the knowledge that it *was* thrown. Therefore, the *exc* applies the label L to these two components ($\underline{\mathsf{n}}$ and $\underline{\mathsf{nv}}$) as well. For example, in the division rule, an arithmetic exception is thrown depending on the value of the denominator; hence, the static rule applies *exc* with $L = X_2[\underline{\mathsf{nv}}]$.

$$\textit{extend}(A, \langle \left[\mathsf{final}\right] \tau\ v\rangle) = A[v := \langle \mathsf{var}\ \left[\mathsf{final}\right]\ \textit{type-part}(\tau, A)\{\textit{var-label}(\tau, A)\}\ \textit{fresh-uid}()\rangle]$$
$$\textit{extend}(A, \langle S_1; S_2\rangle) = \textit{extend}(\textit{extend}(A, \langle S_1\rangle), \langle S_2\rangle)$$
$$\textit{extend}(A, \langle S\rangle) = A \qquad \textit{(for other statements S)}$$

$$\textit{var-label}(\tau, A) = (\mathsf{if}\ \textit{labeled}(\tau)\ \mathsf{then}\ \textit{label-part}(\tau, A) \sqcup A[\underline{\mathsf{pc}}]\ \mathsf{else}\ \textit{fresh-variable}())$$

Figure 4.17: Adding local variable definitions

4.4.3 Local variables

The static checker stores information about local variables in the environment. The function *extend*, defined in Figure 4.17, is used to augment environments with definitions of local variables. When applied to any statement, the function extracts the local variable definitions; it is needed because Java (and JFlow) allow variable definitions at any point within a method. Angle brackets are placed around the statement argument for clarity. For most statement forms, the function *extend* returns an unchanged environment. For local variable definitions, it adds an appropriate binding, as shown in the first case. Note that the label of the variable is interpreted in the environment A; the variable v may not be used in its own label. A sequence of statements is also considered to be a single statement; the second definition recursively applies *extend* to statements in the sequence to accumulate all the definitions.

The function *var-label* creates the appropriate label for a variable declared to have extended type τ. If the variable has a declared label, the true label is the declared label joined with the $\underline{\mathsf{pc}}$ at the point of declaration. Any access to the variable must be tainted by $\underline{\mathsf{pc}}$, so applying a weaker label to the variable would make it immutable.

Argument variable definitions are added to the environment by a different set of rules (see Section 4.7.4).

4.4.4 Variable access

Some simple rules for accessing variables and components of objects are given in Figure 4.18. The first rule covers an expression consisting of a variable name. The value of a variable is labeled with not only the variable's label, but also the current $\underline{\mathsf{pc}}$. Joining the label with the current $\underline{\mathsf{pc}}$ is necessary because the label of every expression includes the $\underline{\mathsf{pc}}$ in which the expression occurs. The label of the variable itself only includes the $\underline{\mathsf{pc}}$ at the point of declaration of the variable.

The second rule covers an array index expression. This rule mirrors the order of evaluation of the expression. First, the array expression (E_a) is evaluated, yielding path labels X_a. If it completes normally, the index expression (E_b) is evaluated, yielding X_b. If this completes normally, two tests are performed. First, the array is checked to make sure it is not null; then, the index is checked to make sure it is in bounds for the array. If either test fails, an appropriate exception is thrown.

The meaning of the final antecedent in this rule is that the label of the array index expression depends on the labels of the array expression, the index expression, and the array elements (L_a). The possible

$$\frac{A[v] = \langle \mathsf{var}\ \left[\mathsf{final}\right]\ T\{L\}\ \mathit{uid}\rangle \quad X = X_\emptyset[\underline{\mathsf{n}} := A[\underline{\mathsf{pc}}], \underline{\mathsf{nv}} := L \sqcup A[\underline{\mathsf{pc}}]]}{A \vdash v : X}$$

$$\frac{\begin{array}{c} A \vdash_T E_a : T\{L_a\}[\,] \\ A \vdash E_a : X_a \\ A[\underline{\mathsf{pc}} := X_a[\underline{\mathsf{n}}]] \vdash E_b : X_b \\ X_1 = \mathit{exc}(X_a \oplus X_b, X_a[\underline{\mathsf{nv}}], \mathsf{NullPointerException}) \\ X_2 = \mathit{exc}(X_1, X_a[\underline{\mathsf{nv}}] \sqcup X_b[\underline{\mathsf{nv}}], \mathsf{OutOfBoundsException}) \\ X = X_2[\underline{\mathsf{nv}} := L_a \sqcup X_2[\underline{\mathsf{nv}}]] \end{array}}{A \vdash E_a[E_b] : X}$$

$$\frac{\begin{array}{c} A \vdash_T E : T \\ L = \mathit{field\text{-}label}(T, f) \\ A \vdash E : X_E \\ X' = \mathit{exc}(X_E, X_E[\underline{\mathsf{nv}}], \mathsf{NullPointerException}) \\ X = X'[\underline{\mathsf{nv}} := L \sqcup X_E[\underline{\mathsf{nv}}]] \end{array}}{A \vdash E.f : X}$$

$$\frac{\begin{array}{c} \langle \left[\mathsf{final}\right]\ \tau\ f\rangle = \mathit{signature}(T, f) \\ A = \mathit{class\text{-}env}(T) \\ L = (\mathsf{if}\ \mathit{labeled}(\tau)\ \mathsf{then}\ \mathit{label\text{-}part}(\tau, A)\ \mathsf{else}\ \bot) \end{array}}{L = \mathit{field\text{-}label}(T, f)}$$

$$\frac{\begin{array}{c} A \vdash_T E : \mathsf{array}[T, L] \\ A \vdash E : X_E \\ X = \mathit{exc}(X_E, X_E[\underline{\mathsf{nv}}], \mathsf{NullPointerException}) \end{array}}{A \vdash E.\mathsf{length} : X}$$

Figure 4.18: Accessing variables and fields

termination paths of an array index expression include all of the normal termination paths of E_a and E_b, plus the two exceptions just mentioned. This rule uses the $\oplus$ operator to coalesce all these paths.

The third rule in Figure 4.18 is for checking accesses to instance variables (fields). It is similar to the rule for checking array index expressions, except that there is no index to be evaluated or tested. Also, the label of the instance variable is obtained by using the predicate *field-label*, defined just below. This predicate ensures that the label L is the label of the field f in the type T, by using the *signature* predicate to obtain the field's signature, and then interpreting the label of that signature. The *field-label* predicate will be useful again shortly.

The final rule checks accesses to the immutable pseudo-field length of arrays. Note that the value of length is not labeled with L, the label of the array elements, because it is immutable.

$$\frac{\begin{array}{c}A \vdash E : X \\ A[v] = \langle \mathsf{var}\ T\{L\}\ \mathit{uid} \rangle \\ A \vdash X[\underline{\mathsf{nv}}] \sqsubseteq L\end{array}}{A \vdash v = E : X}$$

$$\frac{\begin{array}{c}A \vdash E_a : X_a \\ A \vdash_T E_a : T\{L_a\}[\,] \\ A[\underline{\mathsf{pc}} := X_a[\underline{\mathsf{n}}]] \vdash E_b : X_b \\ A[\underline{\mathsf{pc}} := X_b[\underline{\mathsf{n}}]] \vdash E_v : X_v \\ X_1 = \mathit{exc}(X_a \oplus X_b \oplus X_v, X_a[\underline{\mathsf{nv}}], \mathsf{NullPointerException}) \\ X_2 = \mathit{exc}(X_1, X_a[\underline{\mathsf{nv}}] \sqcup X_b[\underline{\mathsf{nv}}], \mathsf{OutOfBoundsException}) \\ X = \mathit{exc}(X_2, X_a[\underline{\mathsf{nv}}] \sqcup X_v[\underline{\mathsf{nv}}], \mathsf{ArrayStoreException}) \\ A \vdash X_v[\underline{\mathsf{nv}}] \sqcup X[\underline{\mathsf{n}}] \sqsubseteq L_a\end{array}}{A \vdash E_a[E_b] = E_v : X}$$

$$\frac{\begin{array}{c}A \vdash_T E_1 : T \\ L = \mathit{field\text{-}label}(T, f) \\ A \vdash E_1 : X_1 \\ A[\underline{\mathsf{pc}} := X_1[\underline{\mathsf{n}}]] \vdash E_2 : X_2 \\ X = \mathit{exc}(X_1 \oplus X_2, X_1[\underline{\mathsf{nv}}], \mathsf{NullPointerException}) \\ A \vdash X[\underline{\mathsf{nv}}] \sqsubseteq L\end{array}}{A \vdash E_1.f = E_2 : X}$$

Figure 4.19: Assignment rules

4.4.5 Variable assignment

Figure 4.19 contains various rules for assignment. The first rule covers the simple assignment of an expression E to a non-final local variable v. The termination paths of the statement are exactly those of the expression E. The only restriction is that the label of the variable must be more restrictive than the label of the result being assigned ($A \vdash X[\underline{\mathsf{nv}}] \sqsubseteq L$).

The rules for assignment to array elements and object fields are complicated by the fact that Java defers checking the validity of the variable being assigned until the right-hand side is fully evaluated. The rule for array element assignment is similar to the rule for array element access. First, the array expression E_a is evaluated, yielding path labels X_a. If it completes normally, the index expression E_b is evaluated, yielding X_b. Then, the assigned value is evaluated. Java checks for three possible exceptions before performing the assignment. Finally, avoiding leaks requires that the label on the array elements (L_a) is at least as restrictive as the label on the information being stored ($X_v[\underline{\mathsf{nv}}] \sqcup X[\underline{\mathsf{n}}]$).

Assignment to an instance variable also is similar to access to an instance variable. As in that earlier rule, the predicate *field-label* obtains the label of the instance variable. This label is compared against the label of the assigned information to prevent leaks.

$$\frac{\begin{array}{c} A \vdash S_1 : X_1 \\ \mathit{extend}(A, S_1)[\underline{\mathsf{pc}} := X_1[\underline{\mathsf{n}}]] \vdash S_2 : X_2 \\ X = X_1[\underline{\mathsf{n}} := \underline{\emptyset}] \oplus X_2 \end{array}}{A \vdash S_1; S_2 : X}$$

$$\frac{\begin{array}{c} A \vdash E : X_E \\ A[\underline{\mathsf{pc}} := X_E[\underline{\mathsf{nv}}]] \vdash S_1 : X_1 \\ A[\underline{\mathsf{pc}} := X_E[\underline{\mathsf{nv}}]] \vdash S_2 : X_2 \\ X = X_E[\underline{\mathsf{n}} := \underline{\emptyset}] \oplus X_1 \oplus X_2 \end{array}}{A \vdash \mathsf{if}\ (E)\ S_1\ \mathsf{else}\ S_2 : X}$$

$$\frac{\begin{array}{c} L = \mathit{fresh\text{-}variable}() \\ A' = A[\underline{\mathsf{pc}} := L,\ \langle \mathsf{goto}\ \epsilon \rangle := L] \\ A' \vdash E : X_E \\ A'[\underline{\mathsf{pc}} := X_E[\underline{\mathsf{nv}}]] \vdash S : X_S \\ A \vdash X_S[\underline{\mathsf{n}}] \sqsubseteq L \\ X = (X_E \oplus X_S)[\langle \mathsf{goto}\ \epsilon \rangle := \underline{\emptyset}] \end{array}}{\begin{array}{c} A \vdash \mathsf{while}\ (E)\ S : X \\ A \vdash \mathsf{do}\ S\ \mathsf{while}\ (E) : X \end{array}}$$

$$\frac{A \vdash \{S_1;\ \mathsf{while}\ (E_1)\ \{S_3;\ S_2\}\} : X}{A \vdash \mathsf{for}\ (S_1;\ E_1;\ S_2)\ S_3 : X}$$

Figure 4.20: Compound statement rules

4.4.6 Compound statements

Figure 4.20 presents rules for checking some compound statements. The first rule is for the simplest statement containing other statements: a sequence of two statements. The second statement is executed only if the first statement terminates normally, so the $\underline{\mathsf{pc}}$ is augmented to include the information of its normal termination ($X_1[\underline{\mathsf{n}}]$). The environment of the second statement also includes any local variables that were defined in the first. The possible termination paths of the sequence include all the termination paths of S_2, plus the abnormal termination paths of S_1. Note that the statement sequence operator (;) is assumed to be associative; this rule works even when S_1 and S_2 are sequences of statements themselves.

The next rule shows how to check an if statement. First, the path labels X_E of the expression are determined. Since execution of S_1 or S_2 is conditional on E, the $\underline{\mathsf{pc}}$ for these statements must include the value label of E, $X_E[\underline{\mathsf{nv}}]$. Finally, the statement as a whole can terminate through any of the paths that terminate E, S_1, or S_2—except normal termination of E, because normal termination would cause one of S_1 or S_2 to be executed. If the statement has no else clause, the statement S_2 is considered to be an empty statement, and the second rule in Figure 4.15 is applied.

The third rule, for the while statement, is more subtle because of the presence of a loop. This rule introduces a label variable L to represent the information carried by the continuation of the loop through

$$\frac{A \vdash E : X_E \qquad X = X_E[\underline{\mathsf{n}} := \underline{\emptyset}] \oplus X_\emptyset[\underline{\mathsf{r}} := X_E[\underline{\mathsf{n}}], \underline{\mathsf{rv}} := X_E[\underline{\mathsf{nv}}]]}{A \vdash \mathsf{return}\ E : X}$$

$$\frac{A \vdash A[\underline{\mathsf{pc}}] \sqsubseteq A[\langle \mathsf{goto}\ \mathcal{L} \rangle]}{\begin{array}{c} A \vdash \mathsf{continue}\ \mathcal{L} : X_\emptyset[\langle \mathsf{goto}\ \mathcal{L} \rangle := \top] \\ A \vdash \mathsf{break}\ \mathcal{L} : X_\emptyset[\langle \mathsf{goto}\ \mathcal{L} \rangle := \top] \end{array}}$$

$$\frac{\begin{array}{c} L = \textit{fresh-variable}() \\ A' = A[\langle \mathsf{goto}\ \mathcal{L} \rangle := L] \\ A' \vdash S_1 : X_1 \\ A'[\underline{\mathsf{pc}} := X_1[\underline{\mathsf{n}}] \sqcup L] \vdash S_2 : X_2 \\ X = (X_1[\underline{\mathsf{n}} := \underline{\emptyset}] \oplus X_2)[\langle \mathsf{goto}\ \mathcal{L} \rangle := \underline{\emptyset}] \end{array}}{A \vdash S_1;\ \mathcal{L} :\ S_2 : X}$$

Figure 4.21: Checking goto-like statements in JFlow

various paths. The label L is a loop invariant on $\underline{\mathsf{pc}}$; its value is discovered by the constraint solver described in Chapter 5. It may carry information from exceptional termination of E or S, or from break or continue statements that occur inside the loop. An entry is added to the environment for $\langle \mathsf{goto}\ \epsilon \rangle$ to capture information flows from any break or continue statements within the loop. The rules for checking break and continue, presented in the next section, use these environment entries to apply the proper restriction on information flow.

4.4.7 Goto-like statements

Figure 4.21 gives the rules for checking statements that transfer control non-locally. First is a rule for a return statement. A return statement can terminate either by abnormal termination of the expression evaluated, or by the $\underline{\mathsf{r}}$ path. Thus, the rule shown results. If there is no expression to return, the proper path labels are simply $X = X_\emptyset[\underline{\mathsf{r}} := A[\underline{\mathsf{pc}}]]$. These are the same path labels generated by the return of a constant, except that there is no return value label ($\underline{\mathsf{rv}}$).

The break and continue statements are handled by using a special entry in the environment that keeps track of the label containing all information transferred to their targets. In the rule for while, in Figure 4.20, we saw an example of such an entry for break and continue statements lacking a specific target. Since break and continue transfer information about the current $\underline{\mathsf{pc}}$ to their target, the rule for these statements simply requires that the restrictions in the current $\underline{\mathsf{pc}}$ be transferred to the target, which is expressed as $A[\underline{\mathsf{pc}}] \sqsubseteq A[\langle \mathsf{goto}\ \mathcal{L} \rangle]$. These two statements also generate path labels containing a mapping from the tuple $\langle \mathsf{goto}\ \mathcal{L} \rangle$ to the label $\top$. The reason for adding these mappings is to prevent the single-path rule from being erroneously used. The label $\top$ is used because the label binding is not used except that it must not be equal to $\underline{\emptyset}$.

$$\frac{A \vdash_T E : \mathsf{class}\ C\ \{\ldots\} \quad A \vdash E : X_E \quad X = \mathit{exc}(X_E, X_E[\underline{\mathsf{nv}}], C)[\underline{\mathsf{n}} := \underline{\emptyset}]}{A \vdash \mathsf{throw}\ E : X}$$

$$\frac{A \vdash \mathsf{try}\{\mathsf{try}\ \{S\}\ ..\mathsf{catch}(C_i\ v_i)\ \{S_i\}..\}\ \mathsf{finally}\{S'\} : X}{A \vdash \mathsf{try}\ \{S\}\ ..\mathsf{catch}(C_i\ v_i)\ \{S_i\}..\ \mathsf{finally}\{S'\} : X}$$

$$\frac{\begin{array}{c} A \vdash S : X_S \quad \underline{\mathsf{pc}}_i = \mathit{exc\text{-}label}(X_S, C_i) \\ A[\underline{\mathsf{pc}} := \underline{\mathsf{pc}}_i,\ v_i := \langle \mathsf{var\ final}\ C_i\{\underline{\mathsf{pc}}_i\}\ \mathit{fresh\text{-}uid}()\rangle] \vdash S_i : X_i \\ X = (\bigoplus_i X_i) \oplus \mathit{uncaught}(X_S, (.., C_i, ..)) \end{array}}{A \vdash \mathsf{try}\ \{S\}\ ..\mathsf{catch}(C_i\ v_i)\ \{S_i\}.. : X}$$

$$\frac{A \vdash S_1 : X_1 \qquad A \vdash S_2 : X_2 \quad X = X_1[\underline{\mathsf{n}} := \underline{\emptyset}] \oplus X_2}{A \vdash \mathsf{try}\ \{S_1\}\ \mathsf{finally}\ \{S_2\} : X}$$

$$\mathit{exc\text{-}label}(X, C) = \bigsqcup_{C' : (C' \leq C\ \vee\ C \leq C')} X[C']$$

$$(X' = \mathit{uncaught}(X, (.., C_i, ..))) \quad \equiv \quad \forall s\ X'[s] = (\mathsf{if}\ (\exists i\ (s \leq C_i))\ \mathsf{then}\ \underline{\emptyset}\ \mathsf{else}\ X[s])$$

Figure 4.22: try statements

The next rule ensures that appropriate environment entries are created for named goto targets. It introduces a binding from the name of a goto label that maps $\langle \mathsf{goto}\ \mathcal{L} \rangle$ to a label variable L. This binding is placed in the environment that is used to check S_1 and S_2. This rule exploits non-determinism for conciseness; because statement sequencing is associative, the rule does not make clear what sequences of statements should be considered to be S_1 and S_2. It is only necessary that S_1 contain all break statements naming $\mathcal{L}$, and that S_2 contain all continue statements naming it. If S_1 and S_2 cannot be chosen in this manner, the program is incorrect.

The JFlow compiler implementation does not precisely follow the approach described in this rule; instead, for each method it constructs a table targets that maps targets to label variables. This table is used to impose the condition $A[\underline{\mathsf{pc}}] \sqsubseteq \mathsf{targets}[\mathcal{L}]$ for each break or continue statement encountered, just as in the rule.

4.4.8 Exceptions

Exceptions can be thrown and caught safely in JFlow using the usual Java constructs. Figure 4.22 shows the rules for various exception-handling statements. The first rule, for throw statements, is straightforward.

The next rule shows how to desugar an arbitrary statement of the form try. . . catch. . . finally into a try. . . catch statement nested within a try. . . finally statement, which reduces the set of statements to be

```
y = true;
try {
   if (x) throw new E();
   y = false;
}
catch (E e) { }
```

Figure 4.23: An implicit flow using throw

checked statically.

The idea behind the try. . . catch rule is that each catch clause is executed with a $\underline{\mathsf{pc}}$ that includes all the paths that might cause the clause to be executed: all the paths that are exceptions where the exception class is *either* a subclass or a superclass of the class named in the catch clause. The function *exc-label* joins the labels of these paths in path labels X. The join is finite because only the exceptions paths of X that are not $\underline{\emptyset}$ need to be joined. The path labels of the whole statement merge all the path labels of the various catch clauses, plus the paths from X_S that might not be caught by some catch clause, which include the normal termination path of X_S if any.

The try. . . finally rule is similar to the rule for sequencing two statements. One interesting difference is that the statement S_2 is checked with exactly the same initial $\underline{\mathsf{pc}}$ that S_1 is, because S_2 is executed no matter how S_1 terminates.

To see how these exception rules work, consider the code in Figure 4.23. In this example, x and y are boolean variables. This code transfers the information in x to y by using an implicit flow resulting from an exception. In fact, the code is equivalent to the assignment $\mathsf{y} = \mathsf{x}$. Using the rule of Figure 4.22, the path labels of the throw statement are $\{\mathsf{E} \rightarrow \{\mathsf{x}\}\}$, so the path labels of the if statement are $X = \{\mathsf{E} \rightarrow \{\mathsf{x}\}, \underline{\mathsf{n}} \rightarrow \{\mathsf{x}\}\}$. The assignment $\mathsf{y} = \mathsf{false}$ is checked with $\underline{\mathsf{pc}} = X[\underline{\mathsf{n}}] = \{\mathsf{x}\}$, so the code is allowed only if $\{\mathsf{x}\} \sqsubseteq \{\mathsf{y}\}$. This restriction is correct because it is exactly what the equivalent assignment statement would have required. Finally, applying both the try-catch rule here and the single-path rule from Figure 4.15, the value of $\underline{\mathsf{pc}}$ after the code fragment is seen to be the same as at its start. Throwing and catching an exception does not necessarily taint subsequent computation.

4.4.9 Dynamic type discrimination

Java provides two mechanisms for dynamic type discrimination: checked run-time type casts and the instanceof operator. The rules for checking these constructs are shown in Figure 4.24. They are both straightforward. In each case, the result of the expression depends on the label of the value of the expression E. For instanceof, the path labels of the boolean result are the same as for E. For a run-time cast, the path labels are the same as for E, except that a ClassCastException is thrown if E has the wrong dynamic type; this exception is conditional on the value label of E, that is, $X_E[\underline{\mathsf{nv}}]$.

$$\frac{A \vdash E : X}{A \vdash E \ \mathsf{instanceof}\ t : X}$$

$$\frac{\begin{array}{c} A \vdash E : X_E \\ X = \mathit{exc}(X_E, X_E[\underline{\mathsf{nv}}], \mathsf{ClassCastException}) \end{array}}{A \vdash (t)E : X}$$

Figure 4.24: Dynamic type discrimination

$$\frac{\begin{array}{c} A \vdash p_1 : X_1 \\ A[\underline{\mathsf{pc}} := X_1[\underline{\mathsf{n}}]] \vdash p_2 : X_2 \\ p_1' = \mathit{interp\text{-}P}(p_1, A) \qquad p_2' = \mathit{interp\text{-}P}(p_2, A) \\ \nexists \mathit{uid}\ (p_1' = \langle \mathsf{pr\text{-}param}\ \mathit{uid} \rangle \vee p_2' = \langle \mathsf{pr\text{-}param}\ \mathit{uid} \rangle) \\ A' = A[\underline{\mathsf{pc}} := X_1[\underline{\mathsf{nv}}] \sqcup X_2[\underline{\mathsf{nv}}]] \\ A'[\underline{\mathsf{ph}} := A[\underline{\mathsf{ph}}] \cup \{(p_1', p_2')\}] \vdash S_1 : X_3 \\ \mathsf{if}\ \left[\mathsf{else}\ S_2\right]\ (A' \vdash S_2 : X_4)\ \mathsf{else}\ (X_4 = X_\emptyset) \\ X = X_1 \oplus X_2 \oplus X_3 \oplus X_4 \end{array}}{A \vdash \mathsf{actsFor}(p_1, p_2)\ S_1 \left[\mathsf{else}\ S_2\right] : X}$$

Figure 4.25: Checking the actsFor statement

4.5 Checking new statements and expressions

The previous section presented the rules for checking information flow in existing Java statements and expressions. This section shows how to statically check the JFlow statements and expressions that are not found in Java.

4.5.1 Testing the principal hierarchy

The actsFor statement is used to dynamically test the relationship between two principals in the current principal hierarchy. If the relationship exists between the two named principals, a statement is executed. Figure 4.25 shows how this statement is checked statically. The expressions p_1 and p_2 must be identifiers; this condition is enforced because the function *interp-P* is used to interpret them. They must name either external principals or run-time principals, because principals that are class parameters of type principal are not available at run time to be tested. Since the expressions p_1 and p_2 are identifiers, they cannot generate any exceptions when evaluated. However, if they name run-time principals, their values may carry information, which affects the result of the test; this information is in the labels $X_1[\underline{\mathsf{nv}}]$ and $X_2[\underline{\mathsf{nv}}]$. For this reason, the $\underline{\mathsf{pc}}$ for S is augmented to include these labels. The $\underline{\mathsf{ph}}$ component of the environment is also augmented to include the pair (p_1', p_2'), making the knowledge that $p_1 \succeq p_2$ available when statically checking S_1. Note that no extra knowledge is available when statically checking S_2; as discussed in Section 2.4.3, negative

$$\frac{\begin{array}{c}A \vdash E : X_E \\ L = \textit{interp-L}(l, A) \\ A \vdash X_E[\underline{\mathsf{nv}}] \sqsubseteq \textit{interp-L}(L, A) \sqcup \textit{auth-label}(A)\end{array}}{A \vdash \mathsf{declassify}\ (E, l) : X}$$

$$\frac{\begin{array}{c}L = \textit{interp-L}(l, A) \\ A \vdash A[\underline{\mathsf{pc}}] \sqsubseteq L \sqcup \textit{auth-label}(A) \\ A[\underline{\mathsf{pc}} := L] \vdash S : X_S \\ X = X_S[\underline{\mathsf{n}} := X_S[\underline{\mathsf{n}}] \sqcup A[\underline{\mathsf{pc}}]]\end{array}}{A \vdash \mathsf{declassify}\ (l)\ S : X}$$

$$\textit{auth-label}(L, A) = \bigsqcup(p \in A[\underline{\mathsf{auth}}])\langle \mathsf{policy}\ p : \ \rangle$$

Figure 4.26: Declassification statement and expression

information about the principal hierarchy is not useful during static checking.

4.5.2 Declassification

JFlow provides two mechanisms for declassifying information: the declassify expression and the declassify statement. Both of these constructs are checked statically, using the static authority of the code at the point of invocation, as shown in Figure 4.26. The static authority of the code is stored in the environment entry $A[\underline{\mathsf{auth}}]$ as a set of principals—principals for whom the code is currently known to have the authority to act. Principals for whom principals in $A[\underline{\mathsf{auth}}]$ can act also are implicitly in the static authority.

To check whether a label L_1 can be declassified to L_2, the equation $L_1 \sqsubseteq L_2 \sqcup \textit{auth-label}(A)$ must be satisfied, thus enforcing the constraint $L_1 \sqsubseteq L_2 \sqcup L_A$ from Section 2.4.4. The label $\textit{auth-label}(A)$, defined in the figure, contains policies of the form $\langle \mathsf{policy}\ p :\rangle$ for every principal p in $A[\underline{\mathsf{auth}}]$. This label is equivalent to L_A, a label in which policies of the form $\langle \mathsf{policy}\ p :\rangle$ are present for every principal p in the static authority, because the additional policies are redundant according to the redundancy rule of Section 2.4.4.

The first rule determines the path labels on the expression E and ensures that the label of the value of E ($X_E[\underline{\mathsf{nv}}]$) can be declassified to the label L. The second rule ensures that the current $\underline{\mathsf{pc}}$ can be declassified to the desired label L; this new declassified $\underline{\mathsf{pc}}$ is then used to check the statement S. The declassified $\underline{\mathsf{pc}}$ does not carry through to the statement following the declassify, because the fourth line rejoins $A[\underline{\mathsf{pc}}]$ to the normal termination label. However, any exceptions or return statements performed within S will be able to take advantage of the declassified $\underline{\mathsf{pc}}$, because these paths are not joined to $A[\underline{\mathsf{pc}}]$.

This statement could have been defined to modify the $\underline{\mathsf{pc}}$ of the subsequent statements by defining $X[\underline{\mathsf{n}}] = X_s[\underline{\mathsf{n}}]$, but that definition seems more likely to result in unintentional declassification. The semantics chosen are an engineering choice to avoid programming accidents.

$$\frac{\begin{array}{c} A \vdash E : X_E \\ L_i = \textit{interp-L}(l_i, A) \\ A \vdash X_E[\underline{\mathsf{nv}}] \sqsubseteq L_i \sqcup L_{RT} \\ A \vdash_T E : T \\ T_i = \textit{interp-T}(t_i, A) \\ A \vdash_T T \leq T_i \\ \underline{\mathsf{pc}}_0 = X_E[\underline{\mathsf{n}}] \\ \underline{\mathsf{pc}}_i = \underline{\mathsf{pc}}_{i-1} \sqcup \textit{label}(X_E[\underline{\mathsf{nv}}] \sqcup L_i) \\ A[\underline{\mathsf{pc}} := \underline{\mathsf{pc}}_i, v_i := \langle \mathsf{var\ final}\ T_i\{L_i\}\ \textit{fresh-uid}()\rangle] \vdash S_i : X_i \\ X = X_E \oplus (\bigoplus_i X_i) \end{array}}{A \vdash \mathsf{switch\ label}(E)\{..\mathsf{case}\ (t_i\{l_i\}\ v_i)\ S_i..\} : X}$$

Figure 4.27: Checking switch label

4.5.3 Run-time label tests

The most interesting aspect of checking JFlow is checking the switch label statement, which inspects a label value at run time. The inference rule for checking this statement is given in Figure 4.27. Intuitively, the switch label statement tests the equation $X_E[\underline{\mathsf{nv}}] \sqsubseteq L_i$ for every arm until it finds one for which the equation holds, and executes it. However, this test cannot be evaluated either statically or at run time. For this reason, the test is split into two stronger conditions: one that can be tested statically, and one that can be tested dynamically. This rule naturally contains the static part of the test.

Let L_{RT} be the join of all possible run-time-representable policies (that is, policies that do not mention label or principal parameters). The static test is that $X_E[\underline{\mathsf{nv}}] \sqcup L_{RT} \sqsubseteq L_i \sqcup L_{RT}$ (or the simpler but equivalent test $X_E[\underline{\mathsf{nv}}] \sqsubseteq L_i \sqcup L_{RT}$); the dynamic test is that $X_E[\underline{\mathsf{nv}}] \sqcap L_{RT} \sqsubseteq L_i \sqcap L_{RT}$. Together, these two tests imply the full condition $X_E[\underline{\mathsf{nv}}] \sqsubseteq L_i$.

The test itself may be used as an information channel, so after the check, the $\underline{\mathsf{pc}}$ must include the labels *of* $X_E[\underline{\mathsf{nv}}]$ and every L_i up to this point. The rule uses the *label* function, defined in Figure 4.28, to determine which labels to join together. When applied to a label L, the function *label* generates a new label that includes all the policies on variables that are mentioned in L. This function is complicated by the possibility of transferring information through dynamic principals, an information channel that is captured by the function *pr-label*.

Extracting the label from a dynamic component must account for the possible presence of recursive label references. Intuitively, the label of a component $\langle$dynamic *uid* $L\rangle$ is simply the label L. However, the label L might refer to the component that contains it. Recursive label references are not generated by any static checking rule seen so far; they are created by the constraint solver as it does its work. The definition of the function *subst*, which rewrites L to eliminate recursive references, accordingly is deferred until Chapter 5, where the constraint solver is discussed.

$$
\begin{array}{l}
\mathit{label}(\bot) = \bot \\
\mathit{label}(\top) = \top \\
\mathit{label}(\langle \textsf{label-param } \mathit{uid} \rangle) = \mathit{label}(\langle \textsf{covariant-label } \mathit{uid} \rangle) = \bot \\
\mathit{label}(\langle \textsf{dynamic } \mathit{uid}\ L \rangle) = \mathit{subst}(\mathit{uid}, L, L) \\
\mathit{label}(\langle \textsf{policy } o : .., r_i, .. \rangle) = \mathit{pr\text{-}label}(o) \sqcup \ldots \sqcup \mathit{pr\text{-}label}(r_i) \sqcup \ldots
\end{array}
$$

$$
\begin{array}{l}
\mathit{pr\text{-}label}(p) = \\
\quad \textsf{case } p \textsf{ of} \\
\qquad \langle \textsf{pr-external } \mathit{name} \rangle : \bot \\
\qquad \langle \textsf{pr-param } \mathit{uid} \rangle : \bot \\
\qquad \langle \textsf{pr-dynamic } \mathit{uid}\ L \rangle : L \\
\quad \textsf{end}
\end{array}
$$

Figure 4.28: Taking the label of a label

4.6 Method and constructor calls

Static checking in object-oriented languages is often complex, and the various features of JFlow only add to the complexity: covariant and invariant class parameters, implicit argument parameters, and method constraints. This section shows how, despite this complexity, method calls and constructor calls (via the operator new) can be checked statically.

4.6.1 Generic checking

The rules for checking method and constructor calls are shown in Figures 4.29 through 4.31. To avoid repetition, the checking of both static and non-static method calls, and also constructor calls, is expressed in terms of the predicate *call*, which is defined in Figure 4.29. This predicate is in turn expressed in terms of two predicates: *call-begin* and *call-end*.

The predicate *call-begin* checks the argument expressions and checks whether the constraints for calling the method are satisfied. It produces the begin label L_I, the argument environment A^a, which binds all the method arguments to appropriately labeled types, and the default return label L_{RV}^{def}. Invoking a method requires evaluation of the arguments E_j, producing corresponding path labels X_j. The argument labels are bound in A^a to labels L_j, so the line $(X_j[\underline{\textsf{nv}}] \sqsubseteq L_j)$ ensures that the actual arguments can be assigned to the formals. If the begin-label is explicitly declared (as tested by if $\left[I\right]$), it is interpreted and is required to be more restrictive than the $\underline{\textsf{pc}}$ after evaluating all of the arguments, which is $X_{\max(j)}$. If the begin-label is not declared, it is an implicit parameter and is bound to $X_{\max(j)}$. It therefore passes the test against $X_{\max(j)}$ automatically.

The predicate *satisfies-constraints* is used by *call-begin* to establish that the constraints $\mathcal{K}_l$ for calling the method are satisfied. Only caller and actsFor constraints need to be satisfied, because authority constraints are tested when the class of the method is compiled, rather than when the method is used. The rule for this predicate, also in Figure 4.29, uses the function *interp-P-call*, which maps identifiers used in the method

$$\frac{A \vdash (A^a, L_I, L_{RV}^{def}) = \textit{call-begin}(C[\mathcal{Q}_i], (.., E_j, ..), \mathcal{S}) \quad A \vdash \textit{call-end}(C[\mathcal{Q}_i], \mathcal{S}, A^a, L_I, L_{RV}^{def}) : X}{A \vdash \textit{call}(C[\mathcal{Q}_i], (.., E_j, ..), \mathcal{S}) : X}$$

$$\frac{\begin{array}{c}\mathcal{S} = \langle \left[\mathsf{static}\right] \tau_r\ m \left[\{I\}\right] (..\tau_j\ a_j..) \left[:\{R\}\right] \mathsf{throws}(..\tau_k..) \mathsf{\ where\ } \mathcal{K}_l \rangle \\ X_0 = X_\emptyset[\underline{\mathsf{n}} := A[\underline{\mathsf{pc}}]] \\ A[\underline{\mathsf{pc}} := X_{j-1}[\underline{\mathsf{n}}]] \vdash E_j : X_j \\ L_j = \textit{fresh-variable}() \\ \textit{uid}_j = \textit{fresh-uid}() \\ A^c = \textit{class-env}(C[\mathcal{Q}_i]) \\ A^a = A^c[..a_j := \langle \mathsf{var\ final}\ \textit{type-part}(\tau_j, A^c)\{L_j\}\ \textit{uid}_j \rangle ..] \\ L_I = (\mathsf{if}\ \left[\{I\}\right]\ \mathsf{then}\ \textit{interp-L}(I, A^a)\ \mathsf{else}\ X_{\max(j)}[\underline{\mathsf{n}}]) \\ A \vdash L_j \approx (\mathsf{if}\ \textit{labeled}(\tau_j)\ \mathsf{then}\ \textit{label-part}(\tau_j, A^a) \sqcup L_I\ \mathsf{else}\ L_j) \\ A \vdash X_j[\underline{\mathsf{nv}}] \sqsubseteq L_j \\ A \vdash X_{\max(j)}[\underline{\mathsf{n}}] \sqsubseteq L_I \\ L_{RV}^{def} = (\mathsf{if}\ (\tau_r = \mathsf{void})\ \mathsf{then}\ \{\}\ \mathsf{else}\ \bigsqcup_j X_j[\underline{\mathsf{nv}}]) \\ \textit{satisfies-constraints}(A, A^a, A[..a_j := E_j..], (..\mathcal{K}_l..)) \end{array}}{A \vdash (A^a, L_I, L_{RV}^{def}) = \textit{call-begin}(C[\mathcal{Q}_i], (..E_j..), \mathcal{S})}$$

$$\frac{\begin{array}{l}\mathsf{let}\ \textit{interp}(p) = \textit{interp-P-call}(p, A, A^a, A^m)\ \mathsf{in} \\ \quad \forall i\ \mathsf{case}\ \mathcal{K}_i\ \mathsf{of} \\ \quad\quad \mathsf{authority}(\ldots) : \textit{true} \\ \quad\quad \mathsf{caller}(..p_j..) : \forall (p_j) \exists (p' \in A[\underline{\mathsf{auth}}])\ A \vdash p' \succeq \textit{interp}(p_j) \\ \quad\quad \mathsf{actsFor}(p_1, p_2) : A \vdash \textit{interp}(p_1) \succeq \textit{interp}(p_2) \\ \quad \mathsf{end} \\ \mathsf{end}\end{array}}{\textit{satisfies-constraints}(A, A^a, A^m, (..\mathcal{K}_i..))}$$

$$\frac{\begin{array}{c}\mathcal{S} = \langle \left[\mathsf{static}\right] \tau_r\ m \left[\{I\}\right] (..\tau_j\ a_j..) \left[:\{R\}\right] \mathsf{throws}(..\tau_k..) \mathsf{\ where\ } \mathcal{K}_l \rangle \\ L_R = L_I \sqcup (\mathsf{if}\ \left[:\{R\}\right]\ \mathsf{then}\ \textit{interp-L}(R, A^a)\ \mathsf{else}\ \{\}) \\ L_{RV} = L_R \sqcup (\mathsf{if}\ \textit{labeled}(\tau_r)\ \mathsf{then}\ \textit{label-part}(\tau_r, A^a)\ \mathsf{else}\ L_{RV}^{def}) \\ C_k[\,] = \textit{type-part}(\tau_k, \textit{class-env}(C[\mathcal{Q}_i])) \\ X' = (\bigoplus_j X_j)[\underline{\mathsf{n}} := L_R,\ \underline{\mathsf{nv}} := L_{RV}] \\ X = X' \oplus X_\emptyset[..C_k := \textit{label-part}(\tau_k, A^a) \sqcup L_R..] \end{array}}{A \vdash \textit{call-end}(C[\mathcal{Q}_i], \mathcal{S}, A^a, L_I, L_{RV}^{def}) : X}$$

Figure 4.29: Generic method-call checking

constraints to the corresponding principals. This function is defined in Figure 4.30. To perform this mapping, the function needs environments corresponding to the calling code (A), the called code (A^a), and a special environment that binds the actual arguments (A^m). The environment entry $A[\underline{\mathsf{auth}}]$ contains the set of principals that the code is known statically to act for.

```
interp-P-call(p, A, A^a, A^m) =
    let p' = interp-P(p, A^a) in
        case p' of
            ⟨pr-dynamic uid L⟩ : interp-P(A^m[p], A)
            else p'
        end
    end
```

Figure 4.30: Interpreting principals in a method call

Finally, the predicate *call-end* produces the path labels X of the method call by assuming that the method returns the path labels that its header claims. The label L_{RV}^{def} is used as the label of the return value in the case where the return type, τ_r, is not labeled. It joins together the labels of all of the arguments, because typically the return value of a function depends on all of its arguments. This rule also shows that the default end-label is the same as the begin-label, and that the end-label is included in the labels of all of the exception paths as well as in the label of the return value. The argument labels are not by default included in the end-label, because exceptions often do not depend on all of the arguments to a function; if argument labels were included by default, the programmer would be encouraged to write method specifications that were overly restrictive.

4.6.2 Specific rules for checking calls

The rules for the various kinds of method calls are built on top of this framework, as shown in Figure 4.31. The only subtlety that arises in these rules is that constructors are checked as though they were static methods with a similar signature. The function *signature* obtains the signature of the named method from the class.

Ordinary method calls are checked by using the *call* predicate in a straightforward manner. The pc for the *call* predicate is set from the normal termination path of the expression for the method receiver, E_s. Static method calls are checked even more simply, because there is no evaluation of a method receiver before the arguments are evaluated.

The final rule in Figure 4.31 covers calls to a constructor, which are handled similarly to a call to a static method. In fact, as the rule shows, a constructor call is checked as though it were a static method of the same class.

There is one additional check needed for constructor calls, however. Recall that the class declaration can have an authority clause that mentions principals that the objects of that class can act for. Two kinds of principals may be named in that clause: external principals, and parameters of the class of the type principal. The authority of an external principal derives from the user who installs the class in the system, but the authority of a principal parameter derives from the code that creates the object by calling a constructor. As the rule shows, the static authority of the caller must include any actual principal parameters passed in the position of formal parameters that happen to be listed in the authority clause of the class.

$$\frac{\begin{array}{c} A \vdash_T E_s : C[..\mathcal{Q}_i..] \\ A \vdash_T E_j : T_j \\ \mathcal{S} = \textit{signature}(C[..\mathcal{Q}_i..], m(..T_j..)) \\ A \vdash E_s : X_s \\ A[\underline{\mathsf{pc}} := X_s[\underline{\mathsf{nv}}]] \vdash \textit{call}(C[..\mathcal{Q}_i..], (..E_j..), \mathcal{S}) : X \end{array}}{A \vdash E_s \,.\, m(..E_j..) : X}$$

$$\frac{\begin{array}{c} T = \textit{interp-T}(t, A) \\ A \vdash_T E_j : T_j \\ \mathcal{S} = \textit{signature}(T, m(..T_j..)) \\ A \vdash \textit{call}(T, (..E_j..), \mathcal{S}) : X \end{array}}{A \vdash t \,.\, m(..E_j..) : X}$$

$$\frac{\begin{array}{c} T = C[..\mathcal{Q}_i..] = \textit{interp-T}(t, A) \\ A^g[C] = \langle \mathsf{class}\ C\ \Big[[..\mathcal{P}_i..]\Big]\ \ldots\ \Big[\mathsf{authority}(..p_k..)\Big]\ \ldots \rangle \\ A \vdash_T E_j : T_j \\ \mathcal{S} = \textit{signature}(T, C(..T_j..)) \\ \mathcal{S} = \langle C\Big[\{I\}\Big](..\tau_j\ a_j..)\Big[:\{R\}\Big]\ \mathsf{throws}(..\tau_k..)\ \mathsf{where}\ \mathcal{K}_l \rangle \\ \mathcal{S}' = \langle \mathsf{static}\ T\{\}\ \mathsf{dummy}\Big[\{I\}\Big](..\tau_j\ a_j..)\Big[:\{R\}\Big]\ \mathsf{throws}(..\tau_k..)\ \mathsf{where}\ \mathcal{K}_l \rangle \\ A \vdash \textit{call}(T, (..E_j..), \mathcal{S}') : X \\ \forall(\text{parameters } p_k)\ \exists(p \in A[\underline{\mathsf{auth}}])\ A \vdash p \succeq \textit{interp-P}(p_k, \textit{class-env}(T)) \end{array}}{A \vdash \mathsf{new}\ t(..E_j..) : X}$$

Figure 4.31: Method and constructor call checking

4.7 Checking classes and methods

The rules for checking virtually all of the statements and expressions of JFlow have now been defined. These rules have relied on the environment being properly set up with entries such as $A[\underline{\mathsf{auth}}]$ and $A[\underline{\mathsf{ph}}]$, and entries for method argument variables and class parameters. This section addresses static checking of information flow in entire class definitions, including the method and constructor declarations within them.

4.7.1 Checking classes

A class contains some number of methods and possibly extends a superclass and some interfaces. It may also be granted some authority by external principals or by principals that are its own parameters. The rule in Figure 4.32 describes how the various components of a class are checked in terms of a number of lower-level predicates that are discussed in the following sections.

In the figure, the function *inner-class-env* is used to create an environment in which the contents of the class C are checked. This function was defined earlier in Section 4.3.3. It adds a definition to the environment A for every formal parameter of the class. For example, label parameters of the class are bound to entries of the form $\langle \mathsf{param\ label}\ \textit{uid} \rangle$, which stand in for the actual parameters supplied in an instantiation

$$
\begin{array}{l}
A^g[C] = \mathsf{class}\ C\Big[[..\mathcal{P}_i..]\Big]\ \Big[\mathsf{extends}\ t_s\Big]\Big[\mathsf{implements}\ ..,t_j,..\ \mathsf{authority}(..p_k..)\Big]\{ \\
\qquad ..\mathcal{M}_m..\quad ..\Big[\mathsf{final}\Big]_n\ \tau_n\ v_n.. \\
\}
\end{array}
$$

$$
\frac{\begin{array}{c}
A = \textit{inner-class-env}(C) \\
A \vdash \textit{authority-ok}(C) \\
A \vdash \textit{match-method}(\textit{interp-T}(t_s, A), \mathcal{M}_m) \\
A \vdash \textit{match-method}(\textit{interp-T}(t_j, A), \mathcal{M}_m) \\
T = \textit{interp-T}(C[..\textit{param-id}(\mathcal{P}_i)..], A) \\
A \vdash \textit{check-method}(T, \mathcal{M}_m) \\
(\mathsf{if}\ \Big[\mathsf{final}\Big]_n\ \mathsf{then}\ \textit{true}\ \mathsf{else}\ \textit{invariant}(\textit{type-part}(\tau_n, A)) \wedge \textit{invariant}(\textit{label-part}(\tau_n, A)))
\end{array}}{\textit{check-class}(C)}
$$

Figure 4.32: Checking a class

of the class. The static checking rules are conservative with respect to these parameters, ensuring that the class would also statically check if any actual parameter were substituted for the corresponding formal parameter. The type expression t_s denotes the superclass of C, if any, and the type expressions t_j denote the interfaces that C implements, if any. These type expressions are interpreted in the environment A because they may mention the formal parameters of the class C.

Various aspects of the class declaration must be checked statically. The successive lines in the rule correspond to the following static tests, which are discussed in more detail in the remainder of the chapter.

- The authority declared in the authority clause of the class must actually have been granted to the class. This authority must also be at least as great as the authority of the superclass. These conditions are tested by the predicate *authority-ok*, described in Section 4.7.2.

- The signature of every method $\mathcal{M}_m$ must also be compatible with signatures that are inherited from the superclass or from interfaces that the class implements. The predicate *match-method*, defined in Section 4.7.3 verifies this compatibility.

- Each of the methods of the class also must provide an implementation that is safe with respect to information flow, and obeys the declared signature of the method. The predicate *check-method* ensures that the methods of the class have these properties, as described below in Section 4.7.4.

- Covariant label parameters may not be used to construct the labeled type of any instance variable (v_n) unless it is declared final. Instance variables that mention covariant label parameters cannot be mutable because they could be used to create information leaks.

$$\frac{\begin{array}{c} A^g[C] = \langle \mathsf{class}\ C\left[[..\mathcal{P}_i..]\right]\ \left[\mathsf{extends}\ C_s\ \ldots\right]\ \left[\mathsf{implements}\ ..,t_j,..\right]\ \mathsf{authority}(..p_k..)\rangle \\ p'_k = \textit{interp-P}(p_k, A) \\ \mathsf{case}\ p'_k\ \mathsf{of} \\ \quad \langle \mathsf{pr\text{-}external}\ \textit{uid}\rangle : \exists (p''_k \in A^g[\underline{\mathsf{auth}}])\ A^g \vdash p''_k \succeq p'_k \\ \quad \langle \mathsf{pr\text{-}param}\ \textit{uid}\rangle : \textit{true} \\ \mathsf{end} \\ \\ A^g[C_s] = \langle \mathsf{class}\ C_s\ \ldots\ \mathsf{authority}(..p_l..)\ldots\rangle \\ \forall l\ \exists (p'' \in \{..p'_k..\})\ A^g \vdash p'' \succeq \textit{interp-P}(p_l, A) \end{array}}{A \vdash \textit{authority-ok}(C)}$$

Figure 4.33: Checking the authority of a class

4.7.2 Class authority

The authority clause of a class declaration, if any, must be validated; any external principals listed in this clause must have granted their authority to the installation of this class. The authority clause may also name principals that are parameters of the class, but as discussed in Section 4.6.2, the authority for these principals is granted at the time of object creation. The predicate *authority-ok* checks that the claimed authority is present in the global environment, as shown in Figure 4.33.

The final two lines of this rule enforce another condition, that the authority declared in the authority clause of the class is at least as great as the authority declared in its superclass. Otherwise authority would be obtained by inheriting methods from the superclass.

4.7.3 Method signature compatibility

The methods of the class must have signatures compatible with the same methods in its superclass and interfaces it implements. JFlow follows Java in requiring exact matches in argument types for a method to be considered the same; overloaded methods are distinguished by their argument types. However, labels on argument and return types are not part of the method identity, and need not be the same in a class as in its superclass. As in the usual contravariance/covariance type rules [AC96], argument labels may be made more restrictive, whereas return labels and exception labels may be made less restrictive. In both cases, the subclass is able to accept more (or at least as many) values as method arguments, and may return fewer values. In addition, the constraints on the superclass method must be sufficiently strong to guarantee the satisfaction of the constraints on the subclass method.

All these conditions are enforced by the *match-method-one* test in Figure 4.34. In JFlow, as in most object-oriented languages, the essence of the test for method conformity is that the subclass method should be a valid implementation of the superclass method in the case that the object on which the method is invoked is actually of the subclass type. The rule in Figure 4.34 performs exactly this test, with one additional condition: the types of method arguments must be equal in the two classes—a Java rule. This strengthen-

$$\frac{\begin{array}{c} A^g[C_1] = \langle \mathsf{class}\ C_1 \Big[[..\mathcal{P}_i..]\Big] \ldots \{\ldots \mathcal{M}_1 \ldots\}\rangle \\ \mathcal{M}_1 = \tau_r^1\ m\Big[[\{I_1\}]\Big](..\tau_j^1\ a_j^1..)\Big[: \{R_1\}\Big]\ \mathsf{throws}(..\tau_k^1..)\ \mathsf{where}(\mathcal{K}_l^1)\{\ldots\} \\ \mathcal{M}_2 = \mathcal{S}_2\{\ldots\} \\ A_1 = \textit{class-env}(C_1[..\mathcal{Q}_i..]) \\ A_1 \vdash (L_I, A_1') = \textit{check-arguments}(\Big[[\{I_1\}]\Big], (..\tau_j^1..), (..a_j^1..), (..\mathcal{K}_l..)) \\ \textit{type-part}(\tau_j^1, A_1) = \textit{type-part}(\tau_j^2, A_2) \\ A_1'' = \textit{obj-env}(A_1', C_1[..\mathcal{Q}_i..]) \\ A_1'' \vdash \textit{call}(C_1[..\mathcal{Q}_i..], (..a_j^1..), \mathcal{S}_2) : X \\ A_1'' \vdash \textit{check-body}(L_I, X, ;, \Big[: \{R_1\}\Big], \tau_r^1, (..\tau_k^1..)) \end{array}}{A_2 \vdash \textit{match-method-one}(C_1[..\mathcal{Q}_i..], \mathcal{M}_2)}$$

$$\frac{\mathcal{M} = \langle \mathsf{static}\ \tau_r^2\ m[\{I_2\}](\tau_j^2\ a_j^2)\Big[: \{R_2\}\Big]_2\ \mathsf{throws}(..\tau_{k'}^2..)\ \mathsf{where}(\mathcal{K}_{l'}^2)\rangle}{A_2 \vdash \textit{match-method-one}(C[..\mathcal{Q}_i..], \mathcal{M})}$$

Figure 4.34: Superclass method conformance

$$\frac{\begin{array}{c} A \vdash \textit{match-method-one}(C[..\mathcal{Q}_i..], \mathcal{M}) \\ A^g[C] = \langle \mathsf{class}\ C\Big[[..\mathcal{P}_i..]\Big]\Big[\mathsf{extends}\ t_s\Big]\ \Big[\mathsf{implements}\ .., t_j, ..\Big] \ldots\rangle \\ A' = \textit{class-env}(C[..\mathcal{Q}_i..]) \\ \mathsf{if}\ \Big[\mathsf{extends}\ t_s\Big]\ \mathsf{then}\ (A \vdash \textit{match-method}(\textit{interp-T}(t_s, A'), \mathcal{M})) \\ \mathsf{if}\ \Big[\mathsf{implements}\ .., t_j, ..\Big]\ \mathsf{then}\ (A \vdash \textit{match-method}(\textit{interp-T}(t_j, A'), \mathcal{M})) \end{array}}{A \vdash \textit{match-method}(C[..\mathcal{Q}_i..], \mathcal{M})}$$

Figure 4.35: Recursively checking method compatibility

ing condition is needed because the subclass method is a valid implementation of the superclass method even when the types of the method arguments in the subclass are supertypes of the corresponding method argument types in the superclass. Java enforces this rule because it supports overloading, not because it is needed for type soundness. In the rule, the subscript 1 indicates superclass components, and the subscript 2 indicates subclass components. The goal of the rule is to check the signatures of the methods $\mathcal{M}_1$ and $\mathcal{M}_2$ against each other. The signature $\mathcal{S}_2$ is the signature of the method $\mathcal{M}_2$; the body of the method is irrelevant to this test. The rule works by simulating the checking of a call to method $\mathcal{M}_2$ from within a method with the same signature as $\mathcal{M}_1$.

The second rule in Figure 4.34 shows that checking for method signature conformance is not needed for static methods. It is also unnecessary for constructors. Finally, the *match-method-one* test is satisfied not only through the rule of Figure 4.34, but also if the superclass $C[..\mathcal{Q}_i..]$ has no method with a matching name and argument types, a condition that is more easily described in words than in an inference rule.

Method compatibility must be insured not only with the direct superclass, but also with indirect super-

$$\frac{\begin{array}{c}\mathcal{M} = \left[\mathsf{static}\right]\ \tau_r\ m\left[\{I\}\right](..\tau_j\ a_j..)\left[:\{R\}\right]\ \mathsf{throws}(..\tau_k..)\ \mathsf{where}\ \mathcal{K}_l\ \{S\} \\ A \vdash (L_I, A') = \textit{check-arguments}(\left[\{I\}\right], (..\tau_j..), (..a_j..), (..\mathcal{K}_l..)) \\ \mathsf{if}\ \left[\mathsf{static}\right]\ \mathsf{then}\ A'' = A'\ \mathsf{else}\ A'' = \textit{obj-env}(A', C[..\mathcal{Q}_i..]) \\ A \vdash \textit{check-body}(L_I, X_\emptyset, S, \left[\ :\{R\}\right], \tau_r, (..\tau_k..))\end{array}}{A \vdash \textit{check-method}(C[..\mathcal{Q}_i..], \mathcal{M})}$$

Figure 4.36: Checking method declarations

$$\frac{\begin{array}{c}L_j = \textit{fresh-variable}() \\ \textit{uid}_j = \textit{fresh-uid}() \\ A' = [..a_j := \langle \mathsf{var\ final}\ \textit{type-part}(\tau_j, A)\{L_j\}\ \textit{uid}_j\rangle..] \\ L_I = (\mathsf{if}\ \left[\{I\}\right]\ \mathsf{then}\ \textit{interp-L}(I, A')\ \mathsf{else}\ \langle\mathsf{covariant\text{-}label}\ \textit{fresh-uid}()\rangle) \\ A' \vdash L_j \approx \textit{arg-label}(\tau_j, A') \sqcup L_I \\ A'' = A'[\underline{\mathsf{pc}} := L_I, \underline{\mathsf{auth}} := \textit{constraint-authority}((..\mathcal{K}_l..), A'), \underline{\mathsf{ph}} := \textit{constraint-ph}((..\mathcal{K}_l..), A')] \\ \forall (p \in A''[\underline{\mathsf{auth}}])\exists (p' \in A[\underline{\mathsf{auth}}])\ A'' \vdash p' \succeq p\end{array}}{A \vdash (L_I, A'') = \textit{check-arguments}(\left[\{I\}\right], (..\tau_j..), (..a_j..))}$$

$$\textit{arg-label}(\tau, A) = (\mathsf{if}\ \textit{labeled}(\tau)\ \mathsf{then}\ \textit{label-part}(\tau, A)\ \mathsf{else}\ \langle\mathsf{covariant\text{-}label}\ \textit{fresh-uid}()\rangle)$$

Figure 4.37: Checking a method header

classes and interfaces. The *match-method* test, used in the rule for *check-class* above, applies the function *match-method-one* to all of the supertypes of the class, as shown in Figure 4.35.

4.7.4 Method declarations

There are several kinds of methods: object methods, static methods, and different kinds of constructors. Object methods and static methods are treated similarly. The predicate *check-method* is defined for these methods as shown in Figure 4.36. There are three parts to this rule: first, the method arguments (a_j) and constraints ($\mathcal{K}_l$) are used to create an environment A' in which the body of the method (the statement S) can be checked. If the method is non-static, the environment A' is effectively extended to include definitions for the identifier this and the non-final instance variables.

We saw earlier that checking calls to these different kinds of methods had much in common, and general predicates *call-begin* and *call-end* were defined to capture this common checking. Similarly, there is much common in checking the declarations of different kinds of methods. In particular, checking the method arguments and the paths at method termination involve common work. These common checks are defined by the *check-arguments* and *check-body* predicates, defined in Figures 4.37 and 4.40.

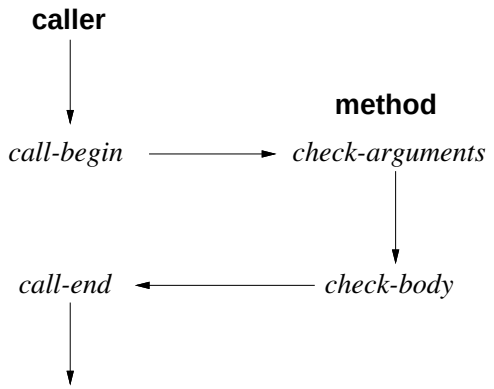

Figure 4.38: Structure of method checking

Checking method arguments. The *check-arguments* predicate is similar in form to the *call-begin* predicate defined earlier in Figure 4.29. This is not surprising, because these two predicates are the caller-side and callee-side tests for method arguments, respectively, as indicated intuitively in Figure 4.38. The *check-arguments* predicate establishes the begin-label L_I, which is also the label of the object this in a non-static method. This label is defined as the interpretation of the label $\{I\}$ if it is provided, or as a label parameter otherwise. In either case, the initial pc for checking the method body is defined by L_I. If $\{I\}$ is omitted, L_I is defined to be a fresh label parameter that cannot be mentioned anywhere outside the method. No results of computations performed by the method can be stored externally, because no external label can be provably as restrictive as L_I. For this reason, methods lacking an explicit begin-label are side-effect free.

The predicate *check-arguments* also establishes the environment A'', which is used for statically checking the body of the method. It contains definitions for the arguments of the method. The arguments are automatically final variables of the declared type. The method arguments are all in scope for use in label expressions in the method header, so a level of indirection is required to define their labels. To allow the variables to refer to one another, the arguments a_j are bound to label variables L_j, in the third line. Equations are then constructed that require these L_j to be equivalent to the interpretation of the label part of τ_j, in the environment A', which contains bindings for a_j. This indirection allows the label parts of τ_j to refer to each other's variables. Note that the begin-label, L_I, is automatically a part of every argument label. The sixth line establishes the environment A'' that is used to check the body of the method. This environment extends the argument environment A' to add definitions for the method body pc, its authority (auth), and static principal hierarchy (ph). The functions *constraint-authority* and *constraint-ph*, defined in Figure 4.39, are used to construct these definitions. The seventh line ensures that the authority claimed by the method (in its authority clause) is a subset of the authority possessed by the class. The environment A, which was defined by the *inner-class-env* function, contains the class authority; the seventh line requires that each principal in the method authority is authorized by some principal in the class authority (which may be a principal parameter).

$$
\begin{array}{l}
\textit{constraint-authority}((..\mathcal{K}_l..), A) = \\
\quad \mathsf{let}\ (\text{for all } l)\ \textit{auth}_l = \\
\quad\quad \mathsf{case}\ \langle \mathcal{K}_l \rangle\ \mathsf{of} \\
\quad\quad\quad \langle \mathsf{authority}\ (..p_i^l..) \rangle : \{..\textit{interp-P}(p_i^l, A)..\} \\
\quad\quad\quad \langle \mathsf{caller}\ (..p_j^l..) \rangle : \{..\textit{interp-P}(p_j^l, A)..\} \\
\quad\quad\quad \mathsf{else}\ \{\} \\
\quad\quad \mathsf{end} \\
\quad \mathsf{in} \\
\quad\quad \bigcup_l \textit{auth}_l \\
\quad \mathsf{end} \\
\\
\textit{constraint-ph}((..\mathcal{K}_l..), A) = \\
\quad \mathsf{let}\ (\text{for all } l)\ \textit{ph}_l = \\
\quad\quad \mathsf{case}\ \langle \mathcal{K}_l \rangle\ \mathsf{of} \\
\quad\quad\quad \langle \mathsf{actsFor}\ (p_1^l, p_2^l) \rangle : \{(\textit{interp-P}(p_1^l, A), \textit{interp-P}(p_2^l, A))\} \\
\quad\quad\quad \mathsf{else}\ \{\} \\
\quad\quad \mathsf{end} \\
\quad \mathsf{in} \\
\quad\quad \bigcup_l \textit{ph}_l \\
\quad \mathsf{end}
\end{array}
$$

Figure 4.39: Building environment entries from constraints

$$
\frac{\begin{array}{c}
A \vdash S : X_s X = X_0 \oplus X_s \\
L_R = (\mathsf{if}\ \left[: \{R\} \right] \mathsf{then}\ \textit{interp-L}(R, A') \sqcup L_I\ \mathsf{else}\ L_I) \\
A \vdash X[\underline{\mathsf{n}}] \sqcup X[\underline{\mathsf{r}}] \sqsubseteq L_R \\
L_{RV} = (\mathsf{if}\ \left[\tau_r\right] \wedge \textit{labeled}(\tau_r)\ \mathsf{then}\ \textit{label-part}(\tau_r, A) \sqcup L_R\ \mathsf{else}\ \underline{\emptyset}) \\
A \vdash X[\underline{\mathsf{nv}}] \sqcup X[\underline{\mathsf{rv}}] \sqsubseteq L_{RV} \\
\forall (C' : X[C'] \neq \underline{\emptyset})\ \forall (k : C' \leq \textit{type-part}(\tau_k, A))\ A \vdash X[C'] \sqsubseteq \textit{label-part}(\tau_k, A) \sqcup L_R
\end{array}}{A \vdash \textit{check-body}(L_I, X_0, S, \left[: \{R\} \right], \left[\tau_r\right], (..\tau_k..))}
$$

Figure 4.40: Checking a method body

Checking method bodies. Using the environment established by *check-arguments*, checking of a method body is completed by using the *check-body* predicate, shown in Figure 4.40. This rule determines the path labels of S in the environment A and then requires that the result path labels declared in the method header are at least as restrictive as the path labels of S. The need for the second argument, X_0, will not be clear at this point; it is used for checking constructors. It effectively allows the insertion of an arbitrary statement to be executed in the method body before S. For ordinary methods, $X_0 = X_\emptyset$.

Checking constructor bodies. Constructors are checked similarly to ordinary methods, but there is added complexity because of the need to initialize instance variables and invoke superclass constructors. A con-

$$\frac{\begin{array}{c}\mathcal{M} = \langle C\Big[\{I\}\Big](..\tau_j\ a_j..)\Big[:\{R\}\Big]\ \mathsf{throws}(..\tau_k..)\ \mathsf{where}\ \mathcal{K}_l\ \{S\}\rangle \\ \textit{final-vars}(C) = \{\} \\ A \vdash \textit{check-arguments}(\Big[\{I\}\Big], (..\tau_j..), (..a_j..), (..p_k..), L_I, A') \\ A'' = \textit{obj-env}(A', C[..\mathcal{Q}_i..]) \\ A'' \vdash \textit{check-body}(L_I, X_\emptyset, S, \Big[: \{R\}\Big], \Big[\ \Big], (..\tau_k..))\end{array}}{A \vdash \textit{check-method}(C[..\mathcal{Q}_i..], \mathcal{M})}$$

Figure 4.41: A simple constructor

$$\frac{\begin{array}{c}\mathcal{M} = \langle C\Big[\{I\}\Big](..\tau_j\ a_j..)\Big[:\{R\}\Big]\ \mathsf{throws}(..\tau_k..)\ \mathsf{where}\ \mathcal{K}_l\ \{C(E_m);\ S\}\rangle \\ A \vdash \textit{check-arguments}(\Big[\{I\}\Big], (..\tau_j..), (..a_j..), (..p_k..), L_I, A') \\ A^g[C] = \langle \mathsf{class}\ C\Big[[..P_i..]\Big] \ldots\rangle \\ q_i = \textit{param-id}(P_i) \\ A' \vdash \mathsf{new}\ C[..q_i..](E_m) : X \\ A'' = \textit{obj-env}(A', C[..\mathcal{Q}_i..]) \\ A'' \vdash \textit{check-body}(L_I, X, S, \Big[: \{R\}\Big], \Big[\ \Big], (..\tau_k..))\end{array}}{A \vdash \textit{check-method}(C[..\mathcal{Q}_i..], \mathcal{M})}$$

Figure 4.42: A constructor with a superclass constructor invocation

structor for a class with no final instance variables and no superclass is checked simply, as shown in Figure 4.41. The condition $\textit{final-vars}(C) = \{\}$ prevents C from having any final instance variables.

A constructor may also defer initialization to another constructor of the same class, as shown in Figure 4.42. It is checked as though the constructor body is executed after another object of class C is created.

The final form of a constructor is one that invokes a superclass constructor, as shown in Figure 4.43. All final instance variables must to be initialized before the call to the superclass constructor. The object (this) and its instance variables are not in scope in this prologue to the constructor, nor in the call to the superclass constructor. This scoping rule is shown by the use of the environment A' in these contexts.

Checking instance variable initialization. A constructor prologue must be checked while keeping track of which final instance variables have been initialized. The *check-inits* predicate, in Figure 4.44, describes this checking. The predicate builds a new environment into which final instance variables of type label are placed for use in label checking.

Figure 4.45 contains one final rule that improves static reasoning about dynamic labels in constructors, by keeping track of what expression final instance variables of type label are initialized with. This rule is used preferentially to the more general rule for an initial statement $v = E$. Its effect is that if an instance variable is initialized from another final variable of type label, the two variables will share the same *uid* and will be treated as containing the same label. Without this rule, we would expect that v_1 would obtain a fresh

$$\frac{\begin{array}{c}\mathcal{M} = \langle C\Big[\{I\}\Big](..\tau_j\ a_j..)\Big[:\{R\}\Big]\ \mathsf{throws}(..\tau_k..)\ \mathsf{where}\ \mathcal{K}_l\ \{S_1;\ \mathsf{super}(E_m);\ S_2\}\rangle \\ A \vdash \textit{check-arguments}(\Big[\{I\}\Big], (..\tau_j..), (..a_j..), (..p_k..), L_I, A') \\ A' \vdash A'' = \textit{check-inits}(C, S_1, \textit{final-vars}(C), X_0) \\ A^g[C] = \langle \mathsf{class}\ C\Big[[..P_i..]\Big]\ \mathsf{extends}\ t_s \ldots\rangle \\ A''[\underline{\mathsf{pc}} := X_0[\underline{\mathsf{n}}]] \vdash \mathsf{new}\ t_s(E_m) : X_1 \\ A''' = A''[\mathsf{this} := \langle \mathsf{var\ final}\ C[\mathcal{Q}_i]\{A[\underline{\mathsf{pc}}]\}\ \textit{fresh-uid}()\rangle, \underline{\mathsf{pc}} := X_1[\underline{\mathsf{n}}]] \\ A''' \vdash \textit{check-body}(L_I, X_0 \oplus X_1, S_2, \Big[:\{R\}\Big], \Big[\Big], (..\tau_k..)) \end{array}}{A \vdash \textit{check-method}(C[..\mathcal{Q}_i..], \mathcal{M})}$$

Figure 4.43: A constructor with final instance variables

$$\frac{\textit{true}}{A \vdash A = \textit{check-inits}(C, ;, \{\}, X_\emptyset[\underline{\mathsf{n}} := A[\underline{\mathsf{pc}}]])}$$

$$\frac{\begin{array}{c}\langle S\rangle = \langle v = E;\ S_2\rangle \\ A[C] = \langle \mathsf{class}\ C \ldots \{\ldots \mathsf{final}\ \tau\ v \ldots\}\rangle \\ A \vdash E : X_E \\ A \vdash X_E[\underline{\mathsf{nv}}] \sqsubseteq \textit{label-part}(\tau, A) \\ A[\underline{\mathsf{pc}} := X_E[\underline{\mathsf{n}}]] \vdash A' = \textit{check-inits}(C, S_2, V - \{v\}, X_2) \\ X = X_E \oplus X_2 \end{array}}{A \vdash A' = \textit{check-inits}(C, S, V, X)}$$

$$\frac{\begin{array}{c}\langle S\rangle = \langle S_1; S_2\rangle \\ A \vdash S_1 : X_1 \\ A[\underline{\mathsf{pc}} := X_1[\underline{\mathsf{n}}]] \vdash A' = \textit{check-inits}(C, S_2, V, X) \end{array}}{A \vdash A' = \textit{check-inits}(C, S, V, X)}$$

Figure 4.44: Checking instance variable initialization

uid and would be treated statically as containing a different label. This optimization avoids unnecessary dynamic testing of the labels in some situations where they can be determined to be identical statically. One

$$\frac{\begin{array}{c}\langle S\rangle = \langle v_1 = v_2;\ S_2\rangle \\ A[C] = \langle \mathsf{class}\ C \ldots \{\ldots \mathsf{final}\ \tau\ v_1 \ldots\}\rangle \\ A[v_2] = \langle \mathsf{var\ final\ label}\{L\}\ \textit{uid}\rangle \\ A \vdash L \sqcup A[\underline{\mathsf{pc}}] \sqsubseteq \textit{label-part}(\tau, A) \\ A' = A[v_1 := \langle \mathsf{var\ final\ label}\{\textit{label-part}(\tau, A)\}\ \textit{uid}\rangle] \\ A' \vdash A'' = \textit{check-inits}(C, S_2, V - \{v\}, X) \end{array}}{A \vdash A'' = \textit{check-inits}(C, S, V, X)}$$

Figure 4.45: Improving static reasoning about dynamic labels

example of this situation is in the implementation of the class Protected, in Figure 3.15. The assignment context $=$ x can be checked statically because lb and LL are bound to the same dynamic label variable using the rule of Figure 4.45. The key step in this rule is the fifth line, which creates the environment A', setting the label of the instance variable v_1 to be *uid*, which is the same as the label of the assigned variable, v_2, as seen in the second line.

Chapter 5

Constraint Solving and Translation

This chapter covers some aspects of implementing JFlow that were not described in Chapter 4. Figure 5.1 depicts the top-level structure of the JFlow compiler. In this figure, the dark ovals indicate two parts of this implementation that have yet to be described. Chapter 4 described the first phase of static checking: application of the inference rules by the rule checker. The second phase of static checking is constraint solving, which is described in Section 5.1. Constraint solving is used to assign labels automatically to local variables and to the program counter (pc). If a satisfying assignment is constructed by the constraint solver, the JFlow program is translated into an equivalent Java program, a process that is described in Section 5.2.

5.1 Constraint solving

As the rules for static checking are applied, they generate a constraint system of labels for each method. For example, the assignment rule of Figure 4.15 generates a constraint $X[\underline{\mathsf{nv}}] \sqsubseteq L$. In this constraint system, some of the labels are unknowns and are called *label variables*. The job of the constraint solver is to

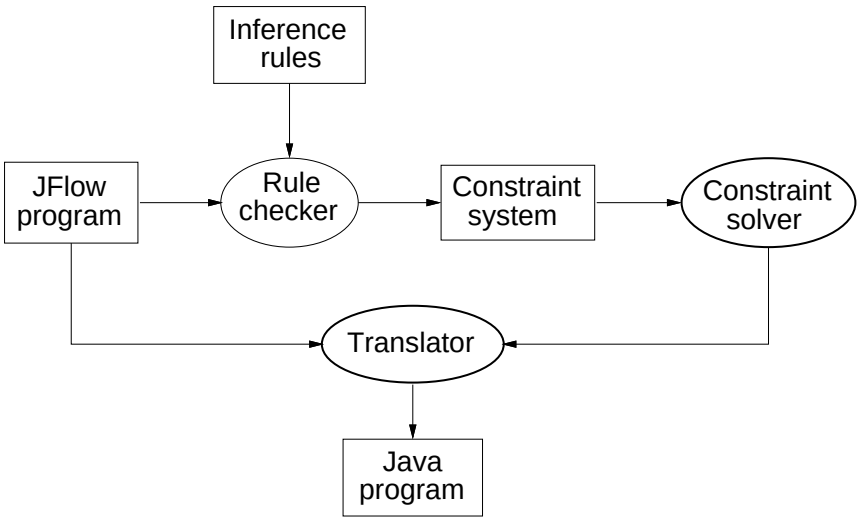

Figure 5.1: Structure of the JFlow compiler

find assignments for these label variables that satisfy all of the constraints. The inference rules generate label variables whenever the function *fresh-variable*() is used, as described in Section 4.2.7. This section describes the final step in statically checking JFlow code: solving the system of constraints generated during the application of the inference rules, and producing satisfying assignments for all label variables. By producing these satisfying assignments, the constraint solver automatically infers labels for local variables and the program counter.

5.1.1 Integrating static checking and constraint solving

As the inference rules in Chapter 4 are used to check the program, antecedents in the form of label constraints are encountered. In general, these constraints contain label variables and cannot be tested when the constraints are first encountered. The static checker records these constraints for later consideration.

Each constraint takes the form $A \vdash L_1 \sqsubseteq L_2$, where A is an environment and L_1 and L_2 are labels. Constraints may also take the form $A \vdash L_1 \approx L_2$, but this constraint is equivalent to the pair of constraints $A \vdash L_1 \sqsubseteq L_2$ and $A \vdash L_2 \sqsubseteq L_1$.

Deferring the checking of label constraints is safe because no searching is necessary to apply the inference rules of the previous chapter, despite the apparent non-determinism of the rules. The selection of which rule to apply at each step is based on syntactic considerations, not whether a particular label constraint can be satisfied. In other words, removing all the antecedents from the inference rules that are label constraints would have no effect on which rules would need to be applied to show a program correct.

Solving constraints is also practical because it is done on a method-by-method basis rather than on an entire program. Although the rules of the previous chapter do not make it explicit, the constraints generated by statically checking one method do not affect the constraints of any other method, so the constraint systems of the various methods can be solved in isolation without loss of expressive power. This property holds because every label variable (for which the constraint solver is to find a value) is associated with only one method, and each constraint mentions label variables from only one method. Constraint systems tend to be small because the constraint system generated by each method can be solved in isolation.

5.1.2 Constraint equations

The first step in solving a set of constraint equations is to put them in canonical form. The constraints generated by application of the inference rules are all of the form $A \vdash L_1 \sqsubseteq L_2$, where L_1 and L_2 may be the join of other labels. The first step in creating the canonical constraint equations is to break up the labels L_1 and L_2 into their individual components. The letter P will be used here to denote a label containing a single component, so the labels L_1 and L_2 can be written as a join of their components $.. \sqcup P_i^1 \sqcup ..$ and $.. \sqcup P_j^2 \sqcup ..$. Because of the properties of the join operator ($\sqcup$), the constraint $L_1 \sqsubseteq L_2$ is equivalent to a set of individual constraints $P_i^1 \sqsubseteq .. \sqcup P_j^2 \sqcup ..$ for each left-hand-side component P_i^1. Therefore, in the canonical form of the constraints, the left-hand side of each equation is a single component.

Constraint:
  LHS $\sqsubseteq$ *RHS*

LHS:
  SimpleComponent
  LabelVariable
  label(LabelVariable)

RHS:
  $\bot$
  RHS $\sqcup$ *RHSComponent*

SimpleComponent:
  $\langle$policy o : .., r_i, ..$\rangle$
  $\langle$label-param *uid*$\rangle$
  $\langle$covariant-label *uid*$\rangle$
  $\langle$dynamic *uid* $DynamicLabel\rangle$

LabelVariable:
  $\langle$variable *uid*$\rangle$

RHSComponent:
  LHS
  L_{inv}
  L_{RT}

DynamicLabel:
  $\bot$
  DynamicLabel $\sqcup$ *SimpleComponent*
  DynamicLabel $\sqcup$ *LabelVariable*

Figure 5.2: Grammar of canonical constraints

The canonical form of a constraint is expressed by the grammar in Figure 5.2. The terminals in this grammar are all expressions that appear in the static checking rules of the previous chapter. The four simple component types (policy, label-param, covariant-label, dynamic) are the only components that may appear in the constraint solver solution. The job of the constraint solver is to replace each label variable with a join of these simple components, with the result that all the constraints are satisfied. These components and the other components are summarized here:

$\langle$policy o : ..r_i..$\rangle$	a policy
$\langle$label-param *uid*$\rangle$	an invariant label parameter
$\langle$covariant-label *uid*$\rangle$	a covariant label parameter
$\langle$dynamic *uid* $L\rangle$	a dynamic label contained in a final variable of type label
$\langle$variable *uid*$\rangle$	a label variable: a label to be solved for
L_{inv}	the join of all components that are not invariant label parameters
L_{RT}	the join of all run-time representable components
label(L)	the label of a label L, which may contain only simple components or variable components

Certain terms may appear on the right-hand side of an equation but not on the left: the two special labels L_{RT} and L_{inv}, which are used when checking the switch label statement and the *invariant* predicate, respectively. These labels are infinite but are never expanded during static checking.

A constraint term also may take the form *label*(L) for some label L, using the function *label* that was defined earlier in Figure 4.28. Applying *label* to a join of several components is defined as the join of *label* applied to the individual components. The result of applying *label* to all label components is well-defined, except for label variables (of type ⟨variable *uid*⟩). Therefore, the function *label* shows up in the canonical constraint equations only in terms of the form *label*(⟨variable *uid*⟩).

Dynamic labels have the unique property that they contain another label L. In the canonical form of the constraint system, this internal label L is also reduced to canonical form, as a join of simple components and label variables, as shown in the grammar.

A constraint equation contains more than just a pair of labels; it also contains an environment A, which records the static checking environment in which the label constraint occurred. However, only one part of the static checking environment is relevant for label constraints: the static principal hierarchy, which is stored in $A[\underline{\text{ph}}]$. The static principal hierarchy affects judgements about the $\sqsubseteq$ relation between two policies, as seen earlier in Figure 4.10.

5.1.3 Solving constraints

A simple iterative work-list algorithm can be used to solve constraints in the canonical form just described. Ignoring dynamic components and terms involving the function *label*, the constraint equations form a simple system of lattice constraints that can be solved using a generalization of the linear-time algorithm for satisfying boolean Horn clauses [DG84, RM96]. The Horn-clause algorithm works because only the *join* operator appears in the constraint equations; if the *meet* operator were allowed, the SAT problem would be reducible to this form, and the constraint-solving problem would become NP-complete [RM96].

The algorithm works by keeping track of conservative upper bounds for each label variable, and iteratively refining that upper bound downward in the label lattice. Initially, all the upper bounds are set to $\top$, the top of the label lattice. The algorithm then iteratively refines the upper bounds, until either all constraints are satisfied or a contradiction is observed. The upper bound of a variable always is either $\top$ or a join of simple components. At each step, the algorithm picks a constraint that might not be satisfied when all label variables are substituted by their upper bounds and *applies* the constraint, forcing it to become satisfied.

A possibly unsatisfied constraint is applied as follows: If the constraint has a label variable on its left-hand side, the upper bound estimate for the variable is lowered to be the *meet* ($\sqcap$) of its current upper bound and the value of the right-hand side. The upper bound of a variable is denoted here by $U(V)$. In evaluating the right-hand side, all variables are replaced with their current upper bound estimates. In other words, a constraint of the form $V \sqsubseteq L$, where V is a label variable and L is a join of some components is satisfied by the assignment $U(V) := U(V) \sqcap U(L)$. This assignment ensures that the constraint in question is satisfied by the current assignments of all variables, even if V appears in L. If the assignment has no effect, the

constraint was already satisfied by the existing $U(V)$. Since the meet operator produces the most restrictive label that is at most as restrictive as its operands, the new $U(V)$ is the most restrictive label that V can have while still managing to satisfy both the constraint and the old upper bound. Inductively, the new upper bound remains conservative.

At every step during constraint solving, the upper bound of each variable is either $\top$ or a join of components of the sorts that are allowed in the final solution: policy, param-label, covariant-label, or dynamic. Therefore, once all constraints are satisfied, the upper bounds of each variable are legal satisfying assignments. If at some step the component on the left-hand-side of an unsatisfied constraint is not a variable (that is, one of the constant policies named above), the constraint system is not solvable: a contradiction has been observed. The reason that the constraints are not solvable is that all variable assignments are conservative upper bounds, so no set of refinements of variable assignments can cause the unsatisfied constraint to become satisfied.

The labels found by this simple algorithm are the most restrictive labels that satisfy the constraints. However, the actual values that the inference algorithm finds are irrelevant, because they are never converted to first-class values of type label. What is important is that there *is* a satisfying assignment to all the labels, proving that the code is safe.

The special labels L_{RT} and L_{inv} are added to the constraint system by checking the switch label statements and the invariance of labels, respectively. In principle, these labels are each a join of a potentially infinite set of components. In practice, they can be integrated into the algorithm just described in a straightforward manner. Recall that L_{RT} is the join of all run-time-representable label components, as defined in Section 4.5.3. The label L_{RT} appears in constraints of the form $V \sqsubseteq L \sqcup L_{RT}$, where V is a variable component and L is a join of arbitrary terms. If this constraint is selected to be satisfied, $U(V)$ is updated just as in the simple algorithm. The new $U(V)$ is $U(V) \sqcap U(L \sqcup L_{RT})$, which is equivalent to $(U(V) \sqcap U(L)) \sqcup (U(V) \sqcap L_{RT})$ because of the distribution properties of $\sqcap$ and $\sqcup$. The term $U(V) \sqcap L_{RT}$ is the *intersection* of $U(V)$ and L_{RT}, which is a join of all run-time-representable components in $U(V)$. In other words, the infinitely large label L_{RT} can be manipulated without expansion into its full form.

The label L_{inv} is treated similarly. This label, defined in Section 4.2.9, arises only from occurrences of the *invariant* predicate. This predicate results in constraints of the form $V \sqsubseteq L_{inv}$. If this constraint is selected to be satisfied, the upper bound for V is changed to $U(V) \sqcap L_{inv}$; in other words, any components of the form $\langle$covariant-label *uid*$\rangle$ are dropped from the upper bound of V.

5.1.4 Determining the meet of two components

In Section 2.4.4, the rule for the meet of two labels was defined. However, in the model of Chapter 2, labels only contained policy components. The rule for meet extends to labels containing the four simple kinds of components, while preserving the necessary label lattice properties. The rule follows directly from the rule for the ordering operator $\sqsubseteq$ presented earlier in Section 4.3.2. As in Chapter 2, the meet of two components that have no relabeling relationship is the bottom label, $\bot$. If the two components have a

$$\langle \textsf{label-param } \textit{uid} \rangle \sqcap_P \langle \textsf{label-param } \textit{uid} \rangle = \langle \textsf{label-param } \textit{uid} \rangle$$

$$\langle \textsf{covariant-label } \textit{uid} \rangle \sqcap_P \langle \textsf{covariant-label } \textit{uid} \rangle = \langle \textsf{covariant-label } \textit{uid} \rangle$$

$$\langle \textsf{dynamic } \textit{uid}\ L_1 \rangle \sqcap_P \langle \textsf{dynamic } \textit{uid}\ L_2 \rangle = \langle \textsf{dynamic } \textit{uid}\ (L_1 \sqcap L_2) \rangle$$

$$\frac{o = o' \vee (P \vdash o' \succ o)}{\langle \textsf{policy } o : .., r_i, .. \rangle \sqcap_P \langle \textsf{policy } o' : .., r'_j, .. \rangle = \langle \textsf{policy } o : .., r_i, .., .., r'_j, .. \rangle}$$

$$\frac{\begin{array}{c} P \vdash o \approx o' \wedge (o \neq o') \\ L = \langle \textsf{policy } o : .., r_i, .., .., r'_j, .. \rangle \sqcup \langle \textsf{policy } o' : .., r_i, .., .., r'_j, .. \rangle \end{array}}{\langle \textsf{policy } o : .., r_i, .. \rangle \sqcap_P \langle \textsf{policy } o' : .., r'_j, .. \rangle = L}$$

Figure 5.3: The meet of two related components

relabeling relationship according to the rules of Figure 4.10, their meet is defined by the rules in Figure 5.3. Note that the meet of two components is defined with respect to a static principal hierarchy P; this is indicated in the rules by writing the static principal hierarchy as a subscript: $\sqcap_P$. Note that the last two rules in the figure correspond to the definitions of Section 4.3.2. The notation $P \vdash o' \succ o$ is used to indicate that o' acts for o in P, but not vice-versa.

5.1.5 Handling dynamic constraints

The algorithm described in the previous section does not handle terms in constraint equations of the form *label*($\langle$variable *uid*$\rangle$). These terms may be generated by uses of the switch label construct, as seen in the rule of Figure 4.27.

A term of this form may occur on either the left- or right-hand side of a constraint equation. Let us first consider how to handle terms of this form that occur on the right-hand side.

An important property of the constraint systems considered in the previous section is that as the upper bounds are refined downward in the label lattice, the values of the right-hand sides of constraint equations also change monotonically downward in the lattice. That is, if the upper bound for a variable $U(V)$ iteratively takes the values $V_1, \ldots, V_n$ during constraint solving, it is always the case that $V_n \sqsubseteq \ldots \sqsubseteq V_1$. In addition, if the right-hand side of a constraint is the label L, then $U(L)$ also decreases monotonically during solving. This property is important for ensuring that $U(V)$ is always a conservative upper bound on V, so application of constraints with a non-variable on the left-hand side can be delayed until all constraints with a variable on the left-hand side are satisfied.

Because of the structure of the function *label*, this important property can be preserved even with the introduction of terms that use *label*. The definition of *label*, which was presented earlier in Figure 4.28, is reproduced here in Figure 5.4. This definition allows the function *label* to be applied to the current upper bound of any variable, since it is defined for all components that can occur in an upper bound.

$\mathit{label}(\bot) = \bot$
$\mathit{label}(\top) = \top$
$\mathit{label}(\langle \mathsf{label\text{-}param}\ \mathit{uid} \rangle) = \mathit{label}(\langle \mathsf{covariant\text{-}label}\ \mathit{uid} \rangle) = \bot$
$\mathit{label}(\langle \mathsf{dynamic}\ \mathit{uid}\ L \rangle) = \mathit{subst}(\mathit{uid}, L, L)$
$\mathit{label}(\langle \mathsf{policy}\ o : .., r_i, .. \rangle) = \mathit{pr\text{-}label}(o) \sqcup \ldots \sqcup \mathit{pr\text{-}label}(r_i) \sqcup \ldots$

$\mathit{pr\text{-}label}(p) =$
    case p of
        $\langle \mathsf{pr\text{-}external}\ \mathit{name} \rangle : \bot$
        $\langle \mathsf{pr\text{-}param}\ \mathit{uid} \rangle : \bot$
        $\langle \mathsf{pr\text{-}dynamic}\ \mathit{uid}\ L \rangle : L$
    end

Figure 5.4: Taking the label of a label

When *label* is applied to a dynamic component, the result is the contained label L. Some substitution (applied by the function *subst*) may be necessary to handle recursive references; this effect is described shortly. As the constraint solver refines variables downward, the current upper bound of the contained label L also changes downward monotonically, and therefore so does the result of applying *label* to the dynamic component. As the constraint solver iteratively refines the upper bounds of variables, the set of dynamic components in the current upper bound of a variable only can decrease in size, because the upper bound is refined by using the meet operator.

When the function *label* is applied to a label L, the result may contain components that derive from policy components in L where the principals in the policy are variables of type principal. The function *pr-label* in Figure 5.4 extracts the label of such policies. Just as with dynamic components, the set of policies in an upper bound only can decrease in size.

Since the set of dynamic components and policy components only can decrease as constraints are applied, and the result of applying *label* to either kind of component can move only downward in the label lattice, the result of applying *label* to a label can move only downward in the label lattice as well.

This argument shows that terms of the form $\mathit{label}(V)$ are well-behaved during constraint solving, and so the constraint-solving algorithm needs little modification to support terms of this form on the *right-hand* side of a constraint equation. When a constraint is used to refine the upper bound of a variable, any terms of this form are evaluated using the current upper bound for the variable V and the definition of *label* in Figure 5.4.

Terms of the form $\mathit{label}(V)$ may also appear on the left-hand side of a constraint. A constraint of the form $\mathit{label}(V) \sqsubseteq L$ is called a *dynamic constraint* here. A dynamic constraint is applied differently from other constraints. If it is not satisfied, at least one component P' in $\mathit{label}(U(V))$ is not covered by any component in $U(L)$. This component must come from the contained label L' of some dynamic component or policy P in $U(V)$.

In general, there are two ways to refine the upper bounds of variables in the constraint system to ensure that P is not part of *label*$(U(V))$. In general, neither refinement is guaranteed to preserve the upper-bound property. One refinement is to drop the component P from $U(V)$, lowering the upper bound of V. It is also possible that $U(L')$ contains P' because L' includes a variable V', and the component P' is part of $U(V')$. If $U(V')$ is the only source of P', then dropping P' from $U(V')$ also will ensure the constraint *label*$(V) \sqsubseteq L$. If both refinements (dropping P from $U(V)$ or P' from some $U(V')$) can be used to ensure the constraint, then neither refinement is in general safe, in the sense that neither $U(V)$ nor $U(V')$ are guaranteed to be upper bounds for their respective variables. The two refinements are not guaranteed to be confluent.

If there is ambiguity about which refinement to apply to eliminate a particular component P', the dynamic constraint is deferred, and another unsatisfied constraint is applied instead. If all unsatisfied constraints are dynamic constraints with this ambiguity, the JFlow constraint solver always selects the refinement of dropping P from $U(V)$. If this arbitrary choice results in a contradiction, the constraint solver reports that it is unable to prove that the method is correct, rather than reporting that the method is provably incorrect. In this case, the programmer must add label annotations to the code to help the constraint solver. Adding these label annotations is usually straightforward. It is only necessary for code that contains the relatively infrequent switch label construct, and only when the label of either the expression whose label is being tested, or of the case labels, must be at least partly inferred automatically. However, in this case the programmer can annotate the code with explicit labels in order to avoid the need to infer them. Thus, the label inference algorithm is not complete for code containing switch label statements, but it is sound. It would be possible to provide a complete constraint solver by adding searching (allowing both refinements to be tried). However, the worst case solving time then would be exponential in the size of the program.

5.1.6 Recursion in dynamic components

A problem that is unique to dynamic components is recursion. When the dynamic component is evaluated using the current upper bounds of the label variables, these upper bounds may mention the dynamic component that is being evaluated, creating infinite recursion. This situation can arise when label variables refer to each other, as in the following function definition:

```
void f(label{*b} a, label{*a} b) {
  ...
}
```

This function has two arguments of type label, each of which dynamically labels the other. This function will result in constraints of the following form, where a, b, la, and lb are the unique identifiers for the various components:

$$\langle \text{variable } a \rangle \quad \sqsubseteq \quad \langle \text{dynamic } lb \ \langle \text{variable } b \rangle \rangle$$
$$\langle \text{variable } b \rangle \quad \sqsubseteq \quad \langle \text{dynamic } la \ \langle \text{variable } a \rangle \rangle$$

Assuming the first constraint is applied first, the algorithm as described so far will refine the upper bounds in the following infinite sequence:

$$\begin{aligned}
\langle \mathsf{variable}\ a\rangle &:= \langle \mathsf{variable}\ b\rangle := \top \\
\langle \mathsf{variable}\ a\rangle &:= \langle \mathsf{dynamic}\ lb\ \top\rangle \\
\langle \mathsf{variable}\ b\rangle &:= \langle \mathsf{dynamic}\ la\ \langle \mathsf{dynamic}\ lb\ \top\rangle\rangle \\
\langle \mathsf{variable}\ a\rangle &:= \langle \mathsf{dynamic}\ lb\ \langle \mathsf{dynamic}\ la\ \langle \mathsf{dynamic}\ lb\ \top\rangle\rangle\rangle \\
\langle \mathsf{variable}\ b\rangle &:= \langle \mathsf{dynamic}\ la\ \langle \mathsf{dynamic}\ lb\ \langle \mathsf{dynamic}\ la\ \langle \mathsf{dynamic}\ lb\ \top\rangle\rangle\rangle\rangle \\
&\ldots
\end{aligned}$$

To avoid this recursion, an additional kind of component is needed when the label contained in a dynamic component refers to its containing label. This kind of recursive reference cannot occur in the initial set of constraints, even when reduced to canonical form, but as the previous example demonstrates, it can arise during constraint applications. A component of the form $\langle \mathsf{dynrec}\ \textit{uid}\rangle$ is used to support recursive dynamic components: components of the form $\langle \mathsf{dynamic}\ \textit{uid}\ L\rangle$ where the label L contains a reference to the enclosing component. To prevent infinite recursion, any such reference is replaced by a component of the form $\langle \mathsf{dynrec}\ \textit{uid}\rangle$, with a matching *uid*. The previous example is solved as follows:

$$\begin{aligned}
\langle \mathsf{variable}\ a\rangle &:= \langle \mathsf{variable}\ b\rangle := \top \\
\langle \mathsf{variable}\ a\rangle &:= \langle \mathsf{dynamic}\ lb\ \top\rangle \\
\langle \mathsf{variable}\ b\rangle &:= \langle \mathsf{dynamic}\ la\ \langle \mathsf{dynamic}\ lb\ \top\rangle\rangle \\
\langle \mathsf{variable}\ a\rangle &:= \langle \mathsf{dynamic}\ lb\ \langle \mathsf{dynamic}\ la\ \langle \mathsf{dynrec}\ lb\ \rangle\rangle\rangle \\
\langle \mathsf{variable}\ b\rangle &:= \langle \mathsf{dynamic}\ la\ \langle \mathsf{dynamic}\ lb\ \langle \mathsf{dynrec}\ la\ \rangle\rangle\rangle \\
&\ldots
\end{aligned}$$

At this point, both constraints are satisfied by the upper bounds of the two label variables.

Components of this new form can occur only within a dynamic component that refers to the same variable. Therefore, the definition of *label* for dynamic components must take into consideration the possible presence of dynrec components by replacing them with the containing component. This substitution is performed by the function *subst*, defined in Figure 5.5. It rewrites the label that is its third argument, substituting any occurrences of $\langle \mathsf{dynrec}\ \textit{uid}\rangle$ for its second argument. The function *subst* only needs to be defined on simple components, plus dynrec components.

5.1.7 Ordering the relaxation steps

The algorithm as described may require $O(nh)$ constraint applications, where n is the number of variables in the constraint system, and h is the maximum height of the label lattice. The height of the lattice that can be

$$\mathit{subst}(\mathit{uid}, L, .. \sqcup P_i \sqcup ..) = .. \sqcup \mathit{subst}(\mathit{uid}, L, P_i) \sqcup ..$$

$$\mathit{subst}(\mathit{uid}, L, \langle \text{label-param } \mathit{uid} \rangle) = \langle \text{label-param } \mathit{uid} \rangle$$
$$\mathit{subst}(\mathit{uid}, L, \langle \text{covariant-label } \mathit{uid} \rangle) = \langle \text{covariant-label } \mathit{uid} \rangle$$

$$\mathit{subst}(\mathit{uid}, L, \langle \text{policy } o : ..r_i.. \rangle) = \langle \text{policy } \mathit{pr\text{-}subst}(\mathit{uid}, L, o) : ..\mathit{pr\text{-}subst}(\mathit{uid}, L, r_i).. \rangle$$

$$\mathit{subst}(\mathit{uid}, L, \langle \text{dynrec } \mathit{uid}' \rangle) = (\text{if } (\mathit{uid} = \mathit{uid}') \text{ then } L \text{ else } \langle \text{dynrec } \mathit{uid} \rangle)$$

$\mathit{pr\text{-}subst}(\mathit{uid}, L, p) =$
  case p in
    $\langle$pr-dynamic *uid* L')$\rangle$: $\langle$pr-dynamic *uid* *subst*$(\mathit{uid}, L, L')\rangle$
    else : p
  end

Figure 5.5: Substituting away recursive label references

observed during an execution of this algorithm is at most equal to the number of non-variable components present in the constraint system. Therefore, the number of lowerings is at most $O(n^2)$ in the size of the method being checked, even when constraints are selected for application in the worst possible order. The performance of the algorithm usually can be improved by more intelligently selecting constraints to be applied. This section discusses how to select and apply constraints so that a satisfying assignment (or a contradiction) is arrived at as rapidly as possible.

The constraint systems solved by the JFlow static checker are similar in form to a *dataflow analysis framework* [Kil73, KU76], and techniques used to accelerate iterative dataflow analysis also can be used to accelerate their solution.

The key observation for accelerating the constraint solver is that there are *dependencies* between different constraints in the constraint system. We are now concerned only with constraints in which the left-hand side is a variable; constraints in which the left-hand side is not a variable are only used to determine whether the constraints are satisfiable once all the former constraints have been satisfied. If one constraint E_1 has a variable v_1 on its left-hand side, applying this constraint will result in v being updated so that E_1 is satisfied. If v_1 appears on the right-hand side of another constraint E_2, then E_2 can be said to *depend* on E_1. It makes sense to apply E_1 before E_2 so that the constraint enforced by E_1 affects E_2's variable.

The dependencies among the constraints can be envisioned as a *dependency graph*, with nodes for each of the constraints in the constraint system. The dependency graph is a directed graph; nodes in the graph are connected if there is a dependency between the corresponding constraints. In the simplest case, the dependency graph is acyclic, and the constraint system can be solved with only one application of each constraint. In this case, the constraints are topologically sorted and then applied sequentially in the order generated. The time required to perform the topological sort is linear in the number of constraints.

In general, the dependency graph will contain cycles. For example, loops in the program will generate

```
ordered = 0;
visited = new boolean[n];
ordering = new int[n];
position = new int[n];
for (int i = 0; i < n; i++) visit(i);
...
void visit(int i) {
   if (visited[i]) return;
   visited[i] = true;
   Iterator[int] e = dependencies(i);
   while (e.hasMore()) visit(e.next());
   ordered++;
   ordering[n - ordered] = i;
   position[i] = n - ordered;
}
```

Figure 5.6: Ordering the constraint equations

cycles in the label dependency graph. In the rule for the while statement (Section 4.4.6), a label variable L is introduced and explicitly made part of a constraint cycle. Cycles in the dependency graph result in *strongly connected components*: sets of constraints in which each constraint is transitively dependent on every other constraint. A strongly connected component can be handled by simply looping on each of the constraints in the component in turn until every constraint is satisfied.

The JFlow constraint solver selects constraints by first topologically sorting the constraints using the standard algorithm based on the depth-first traversal of the constraints [CLR90]. This algorithm is shown in the PolyJ code of Figure 5.6. This code places the indices of the constraints $0 \ldots n-1$ in the array ordering, and assumes that dependencies(i) produces an Iterator that yields the indices of constraints dependent on constraint i. The inverse of ordering is placed in the array position.

When applied to a directed acyclic graph, this algorithm produces an ordering of the nodes in which a node never occurs before any node that it depends on. Strongly connected components within the ordering then can be identified by a depth-first traversal of the *transposed* dependency graph—also a linear-time algorithm [CLR90].

The algorithm using strongly connected components effectively constructs a schedule for solving the constraint system. Once they are identified, the constraint solver applies the strongly connected components in topological order. Each strongly connected component is looped over sequentially in the order in which its node occurred within the original topological sort, until every constraint in the component is satisfied. Once an entire component is satisfied, its constraints need no further consideration. A subtle benefit of applying strongly connected components using the topological ordering is that constraints tend to be propagated very effectively *within* a strongly connected component. For example, a strongly connected component comprising a single cycle needs to repeated only once in order to ensure that all the constraints in the

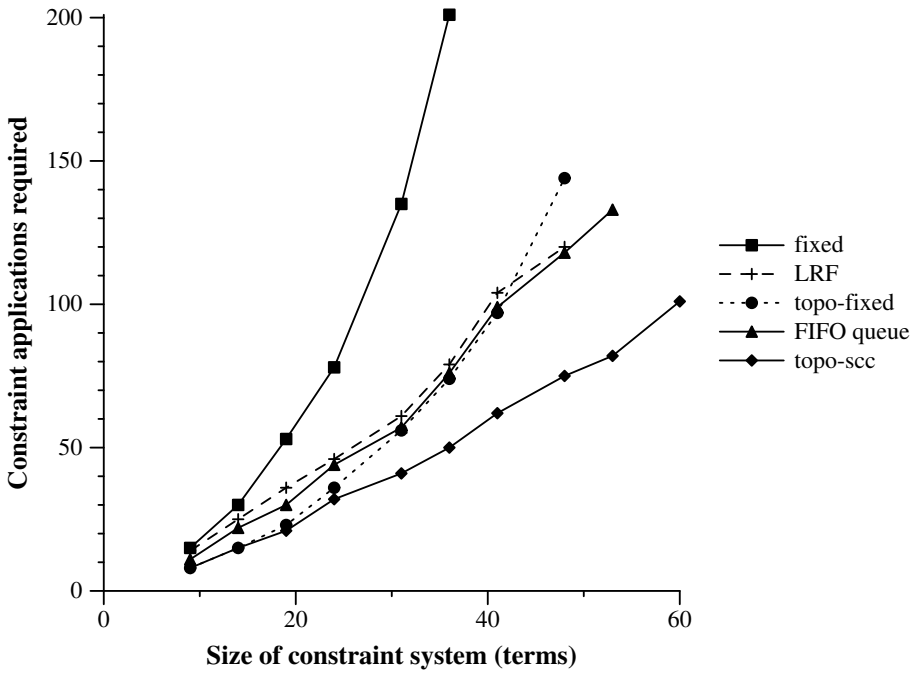

Figure 5.7: Performance of various heuristics for ordering constraints

component are satisfied.

This algorithm is similar in its use of topological sorting and identification of strongly connected components to the *Priority-Scc* algorithm used to optimize iterative dataflow analysis [HDT87]. Apart from the difference in the form of the constraint equations, one difference between the algorithms is that the dataflow analysis algorithm orders *variables* rather than *constraints* as in this algorithm. Ordering on the basis of individual constraints appears always to offer better performance in empirical measurements. The number of iterations required by the dataflow analysis algorithm has been shown to be $O(nd)$ where d is the maximum number of back edges in depth-first traversal of the constraint dependency graph. For dataflow analysis it has been observed that the number of back edges d is bounded for reasonable programs; this property seems to hold for label constraints as well. Even when the number of back edges is linear in the size of the graph, it proves very difficult to observe the $O(n^2)$ behavior that this asymptotic bound predicts; for example, the results in the next section do not suggest $O(n^2)$ behavior. However, a tighter bound on the run time of the algorithm has not been shown.

5.1.8 Empirical comparisons

The observed behavior of the JFlow compiler is that constraint solving is a negligible part of run time when compiling methods of a few tens of lines in length. However, an empirical analysis of performance is useful for understanding how the performance of the constraint-solving technique scales with the size of the constraint system.

The algorithm based on identification of strongly connected components and several other algorithms

for solving dataflow systems were empirically compared for label constraint systems. Many of the same ordering algorithms have been empirically compared earlier for use in dataflow analysis [KW94]. The results observed for dataflow analysis largely agree with the results for label constraints, which are shown in Figure 5.7. The Y axis is the maximum number of iterations required to solve a complex system of constraints containing a number of back edges linear in the number of constraints, using various techniques for choosing constraints. The size of the constraint systems tested is about the same as or somewhat larger than the constraint systems generated by typical method definitions.

The constraints in these systems are all of the form $v_1 \sqsubseteq v_2 \sqcup L_i$, where v_1 and v_2 are variables and L_i is a non-variable constraint. Empirically, constraints of this form require a relatively large number of iterations to arrive at a fixed point assignment to all of the upper bounds. The maximum number of iterations for a constraint system is determined by introducing components L_i such that the meet of every possible subset of the L_i resulted in a different label. Programs with this behavior are extremely unlikely, but the resulting constraint system is useful in gaining some understanding of the behavior of the algorithms. The constraint systems used for the comparison are related to each other in a simple fashion; each consecutive constraint system is the same as the next smaller constraint system, but with one or two additional constraints.

The performance of several heuristics for ordering was compared for these constraint systems. In this comparison, all of the ordering heuristics are used within a common constraint-solving framework. This framework uses information about the dependencies between constraints, to keep track of which constraints might be unsatisfied at any given step. Often a constraint is known to be satisfied because it was previously known to be satisfied, and no variable on its right-hand side has been modified since that point. With all of the constraint-ordering heuristics, a constraint was not applied if it was known to be satisfied based on this reasoning.

The ordering heuristics tested were the following:

- *fixed*: the constraints are placed in a fixed order; the first potentially unsatisfied constraint is applied at each step.
- *topo-fixed*: the constraints are topologically sorted using the algorithm of Figure 5.6, and this ordering is used as in the *fixed* ordering.
- *LRF*: the *least-recently-fired* ordering of Kanamori and Weise [KW94]; the least-recently-applied unsatisfied constraint is selected at each step.
- *FIFO queue*: a FIFO queue of potentially unsatisfied constraints is maintained. This is the standard technique for iterative dataflow analysis [KW94].
- *topo-scc*: this is the approach implemented in the JFlow constraint solver; as described in Section 5.1.7, it loops on strongly connected components.

$$\mathbf{T}[\![\,\mathsf{actsFor}(p_1, p_2)\ S_1 \left[\mathsf{else}\ S_2\right]\,]\!] =$$
$$\quad \mathsf{if}\ (\mathsf{Principal.actsFor}(\mathbf{T}[\![\,p_1\,]\!], \mathbf{T}[\![\,p_2\,]\!]))\ \mathbf{T}[\![\,S_1\,]\!]\ \left[\mathsf{else}\ \mathbf{T}[\![\,S_2\,]\!]\right]$$

$\mathbf{T}[\![\,p\,]\!] = \mathsf{case}\ A[p]\ \mathsf{of}$
$\quad \langle \mathsf{param\ principal}\ \textit{uid} \rangle : \textit{error}$
$\quad \langle \mathsf{constant\ principal} \rangle : \mathsf{jflow.principal}.p.\mathsf{ThePrincipal}$
$\quad \langle \mathsf{var\ final\ principal}\{L\}\ \textit{uid} \rangle : p$
end

Figure 5.8: Translating principals and actsFor

In the particular example for which results are presented, almost the entire constraint system was a single strongly connected component. This situation is a worst case for the topo-scc ordering for comparison to the other orderings. However, the topo-scc ordering still results in substantially better performance than the other ordering techniques. The results shown in Figure 5.7 are in fact typical for a variety of different kinds of constraint systems containing strongly connected components.

Interestingly, the best ordering techniques appear to be the FIFO queue ordering and the topological sort with strongly connected components. The number of iterations required with a simple fixed ordering grows as $O(n^2)$ for this sequence of constraint systems, and even for simpler constraint systems that do not contain strongly connected components.

5.2 Translation

The JFlow compiler is a static checker and source-to-source translator. Its output is a standard Java program. Most of the annotations in JFlow have no run-time representation; translation erases them, leaving a Java program. For example, all type labels are erased to produce the corresponding unlabeled Java type. Class parameters and authority clauses are erased, including the label parameter of array types. Method begin- and end-labels and constraints are erased. The declassify expression and statement are replaced by their contained expression or statement.

Variables of the built-in types label and principal are translated to the Java types jflow.lang.Label and jflow.lang.Principal, respectively. Variables declared to have these types remain in the translated program. Only two statements translate to interesting code: the actsFor and switch label statements. The translated code for each is simple and efficient, as shown in Figures 5.8 and 5.9. In these figures, $\mathbf{T}[\![\,E\,]\!]$ is the translation of a JFlow expression E into a Java expression, and $\mathbf{T}[\![\,S\,]\!]$ is the translation of a statement S.

5.2.1 Principal values and the actsFor statement

The actsFor statement translates to an if statement that tests the current principal hierarchy and executes either the statement S_1 or S_2, depending on whether the relation between the two principals exists. The

$\mathbf{T}[\![t]\!] = t$
$\mathbf{T}[\![t\{l\}]\!] = t$
$\mathbf{T}[\![t\{l\}[\,]]\!] = t[\,]$

$\mathbf{T}[\![\,\mathsf{new\ label}\{P_1; P_2; \ldots; P_n\}]\!] = \mathbf{TL}[\![\,\{P_1; P_2; \ldots; P_n\}]\!]$
$\mathbf{TL}[\![\,\{P_1; P_2; \ldots; P_n\}]\!] =$
$\mathbf{TL}[\![\,\mathsf{new\ label}\{P_1; P_2; \ldots; P_n\}]\!] =$
    $\mathsf{new\ Label}(\mathbf{TL}[\![\,P_1]\!]).\mathsf{join}(\mathsf{new\ Label}(\mathbf{TL}[\![\,P_2]\!]).\ldots.\mathsf{join}(\mathsf{new\ Label}(\mathbf{TL}[\![\,P_n]\!]))\ldots))$

$\mathbf{TL}[\![\,v]\!] = \mathsf{case}\ A[v]\ \mathsf{of}$
    $\langle \mathsf{var}\ \left[\mathsf{final}\right]\ T\{L\}\ \textit{uid}\rangle : \mathbf{TL}[\![\,L]\!]$
    $\langle \mathsf{constant\ principal}\rangle : \mathsf{Label.bottom}()$
    $\langle \mathsf{param\ principal}\ \textit{uid}\rangle : \mathsf{Label.bottom}()$
end

$\mathbf{TL}[\![\,o : \ ..,r_i,..]\!] = \mathsf{new\ Label}(\mathbf{T}[\![\,o]\!], .., \mathbf{T}[\![\,r_i]\!], ..)$

$\mathbf{TL}[\![\,*v]\!] =$
    $\mathsf{case}\ A[v]\ \mathsf{of}$
        $\langle \mathsf{var\ final\ label}\{L\}\ \textit{uid}\rangle : v$
    end

$\mathbf{T}[\![\,\mathsf{switch\ label}(E)\{..\mathsf{case}(\ t_i\{l_i\})\ S_i..\ \mathsf{else}\ S_e\}]\!] =$
    $T\ v\ =\ \mathbf{T}[\![\,E]\!];$
    $\mathsf{if}\ (\mathbf{TL}[\![\,X_E[\underline{\mathsf{nv}}] \sqcap L_{RT}]\!].\mathsf{relabelsTo}(\mathbf{TL}[\![\,L_1 \sqcap L_{RT}]\!]))\{$
    $\mathbf{T}[\![\,S_1]\!]$
    $\}\mathsf{else}\ldots$
    $\mathsf{if}\ (\mathbf{TL}[\![\,X_E[\underline{\mathsf{nv}}] \sqcap L_{RT}]\!].\mathsf{relabelsTo}(\mathbf{TL}[\![\,L_i \sqcap L_{RT}]\!])\{$
    $\mathbf{T}[\![\,S_i]\!]$
    $\}\ldots\mathsf{else}\{\mathbf{T}[\![\,S_e]\!]\}$

Figure 5.9: Translating labels and switch label

class jflow.lang.Principal provides a static method actsFor that can be used to test whether one principal may act for another.

Principals in JFlow are represented both by classes that are subclasses of jflow.lang.Principal, and by instances of these classes. Having a class for each principal in the system simplifies the management of the principal hierarchy in a Java run-time system. Each Principal object contains a list of other Principal objects that can act for it directly: its immediate *superiors* in the principal hierarchy. The object also contains a hash table that maps Principal objects to booleans; this hash table is used to *memoize* actsFor tests so that they can be performed more quickly the second and following times. Every subclass of Principal contains a static initializer that sets up its ThePrincipal object with the initial list of superiors and an empty hash table.

Every subclass of the class Principal is located in the package jflow.principal, and contains a static

variable ThePrincipal of type Principal. Thus, references in JFlow code to an external principal p are translated to expressions of the form jflow.principal.p.ThePrincipal. New principals may be added freely to the package jflow.principal, since a principal is only responsible for identifying the principals that may act for *it*; adding a new principal cannot grant new privileges to that principal, or give power to any principal over any other principal but the new principal. However, the right to modify the class of a principal in order to add new superiors must be controlled, since adding superiors to or removing superiors from an existing principal can affect the principal hierarchy in potentially unsafe ways. The current implementation does not model this aspect of the system, although it appears to be straightforward.

5.2.2 Label values and the switch label statement

As indicated by Figure 5.9, most labels are simply erased from the JFlow program as it is translated into Java. Labels that must be represented at run time are represented as values of type jflow.lang.Label. The translation function $\mathbf{TL}[\![L]\!]$ translates a label expression into a Java expression that generates the appropriate run-time representation. It is undefined for components that are not representable at run time, such as label parameters. Note that policies within a label are translated by translating the principals mentioned in the policies; a policy is only representable at run time if all of the principals it mentions are also representable at run time.

The translation rule for switch label uses definitions from the static checking rule for switch label in Figure 4.27. As discussed earlier, the run-time check to be performed is $X_E[\underline{\mathsf{nv}}] \sqcap L_{RT} \sqsubseteq L_i \sqcap L_{RT}$, a test that mentions only labels that are representable at run time. The relabelsTo method is used to check whether this label relationship exists. Like actsFor, the relabelsTo method is accelerated by a hash table lookup into a cache of memoized results.

Chapter 6

Related Work

Most of this thesis has been concerned with the problem of protecting the secrecy of data. This problem has been recognized for at least 25 years, and also has been referred to as *confinement* [Lam73] of data, or *confidentiality*. In this thesis, it has been referred to as protecting *privacy*, since the goal is to protect data owned by mutually distrusting principals, rather than the secret data of a single entity such as the government. A great deal of work has gone into addressing the problem of secrecy, and it is not feasible to enumerate all of it. This chapter summarizes previous work done on various kinds of security techniques that relate to this work, particularly focusing on information flow control.

6.1 Access control

Most systems protect privacy and integrity through *discretionary access control*, or what is usually called simply *access control*. The idea of access control is that before a potentially dangerous action may be taken by a computer program, a run-time test is made to ensure that the program has been granted the necessary authority for the action. Many access control mechanisms have been designed, such as *capabilities* [DV66, WCC+74], *access control lists* [Lam71], and various hybrid schemes (*e.g.*, [RSC92]). Actions that do not conform to stated policies are not permitted, whether they are reads, writes, or higher-level operations. Unix file permissions are an example of a simple, well-known access-control mechanism.

Since JFlow provides a simple mechanism for controlling the privileges of a program, in the form of static authority, it is interesting to compare it to existing Java access control models, based on *stack inspection* [WBF97, WF98]. Current versions of the Java run-time environment provided by Netscape, Microsoft, and Sun implement variants of this model [Net97, Mic97, GS98]. In Java, privileges are needed to perform various unsafe operations, such as accesses to the local filesystem. In the stack inspection approach, these privileges are known as *targets*. Each class can be authorized to claim one or more privileges, but by default, the class code does not possess these privileges. Explicit operations are provided for enabling and disabling privileges. When a privilege is needed in order to perform an unsafe operation, the stack leading up from the point of invocation is inspected at run time. Every class whose code is on the stack, up to the point where the needed privilege was enabled, must be authorized to claim that privilege. This model allows a class to

grant a privilege, but only if it has itself enabled the privilege explicitly. The privilege can be granted only to the code of another trusted class that could have claimed the privilege for itself. Thus, privilege is enabled *explicitly*, but granted *implicitly*, by the act of calling another method while the privilege has been enabled.

This set of design choices differs in several respects from those in JFlow. In JFlow, principals may be used to represent targets as well as users. The authority clause of a class gives a class the power to act for the named principals, but individual methods do not possess the corresponding privilege unless they explicitly declare it. Thus, the models are similar in that privileges are not available unless explicitly declared. In JFlow, authority is granted to a called method *explicitly*: it is passed as an argument of type principal that is present in a caller clause of the called method. Unlike in Java, the called method need not have the potential authority of that principal (*i.e.*, target). The stated reason for preventing this in the Java models is that it defeats *luring attacks* in which the authority granted is misused by the called method. Luring attacks are a greater concern in the Java model, since authority is granted implicitly. In JFlow, it is clear what authority is granted to the called method (although it may be a run-time parameter). JFlow also allows authority to be bound into an object in a parametric fashion; a class can require that its constructors be called from a site possessing the authority of its principal parameters; this authority is bound into the object. An obvious difference between the models is the manner in which they are enforced. The JFlow authority mechanisms are largely statically checked (though there is support for dynamic checking), whereas the Java model is checked entirely dynamically, with consequent run-time overhead. Static checking is possible in JFlow because authority transfers are completely explicit. Since the Java model of access control is largely a subset of that in JFlow, it seems likely that it could be enforced at load time by an extended Java Virtual Machine if class files were extended with explicit annotations about granted authority.

6.2 Limitations of discretionary access control

Discretionary access control does not support privacy well, because although it prevents information release, it does not control information propagation. For example, consider the tax preparer example of Section 1.1, reproduced here in Figure 6.1. In this example, Bob is preparing his tax form using a piece of software called "WebTax". Bob would like to be able to prepare his final tax form using WebTax, but without trusting WebTax to protect his privacy. Bob can impose an access check that determines whether Preparer can see his tax data. However, once the access is allowed, Bob cannot control how Preparer distributes the information it has read. He is forced to trust that the WebTax program will respect his privacy correctly. Thus, discretionary access protects the privacy of data against others, but it is vulnerable to Trojan horse programs.

Everything that has just been said about privacy applies to integrity as well. If program *A* allows program *B* to modify *A*'s data, then *A* has controlled who may write the data, but cannot control how *B* obtains the data to write there. With only discretionary access control, *A* must trust not only *B* but every program that might have affected the data *B* is providing. Discretionary access control is a point-of-sale mechanism that cannot

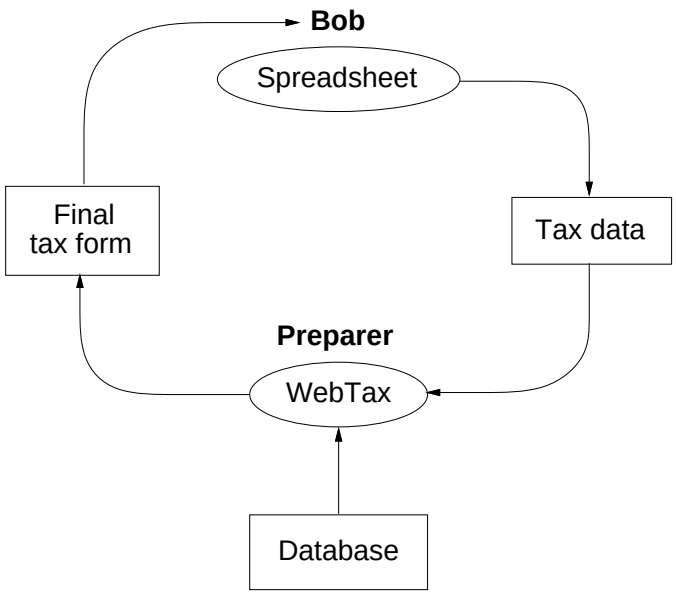

Figure 6.1: Tax preparer example

control either the propagation of information after its release or the propagation of information leading to an update.

6.3 Information flow control

In the case of both privacy and integrity, what is wanted is a way to extend access restrictions transitively, arbitrarily far from the point where data is released or updated. This transitive extension is not possible in a conventional discretionary access control system, because the decision about whether to transfer information from program *A* to program *B* is made based upon the authority and privileges possessed by *A* and *B*; restrictions that the data's ultimate source or destination might like to apply cannot be enforced reliably because information about these restrictions in general has been lost. This insight leads to *information flow control* and *mandatory access control* models, which apply sensitivity labels to data. These labels propagate with the data and are used to mediate information transfers within and between programs. Restrictions on the use of data propagate with the data and apply to any data derived from it. Privacy restrictions prevent data from being seen by untrusted users; integrity restrictions prevent untrusted data from affecting storage locations. A good overview of information flow control is presented by Denning [Den82].

The original model of information flow for secrecy comes from the early work of Bell and LaPadula [BL75]. In this work, objects in the system are assigned to security classes from a small ordered set (e.g., unclassified, classified, secret). Information can flow between the partitions only by moving upward in security class. A subject, or process, in the system is assigned a security class, and the data it manipulates is assigned the same security class. It can read data from a subject of the same or lower security class. The Bell-LaPadula model supports privacy through information flow control; it also controls writes through

access control. Non-destructive writes are permitted to an object of a higher security class, but destructive writes are permitted only to objects of the same security class. This rule prevents low-level subjects from overwriting high-level data, even though this overwriting would not cause an information leak.

The most common information flow enforcement mechanism is dynamic. Fenton's Data Mark Machine (DMM), an early abstract model for information flow enforcement [Fen73, Fen74], is a good example of the dynamic approach to fine-grained information flow. As a program computes, sensitivity labels (security classes) are associated with all data values. The sensitivity label of a computed value must be at least as restrictive as the sensitivity labels of the values it was computed from. In the DMM model, the program-counter label pc is maintained at run time. One weakness of the DMM model is its inability to deal with implicit flows precisely. After an if statement, pc does not revert to its former value, unlike in JFlow. Data computed after a conditional becomes excessively restrictively labeled. The DMM model is made workable because the pc is unaffected by a function call, but at the cost that exceptions are not supported. JFlow allows the program-counter label to revert if the method can terminate only normally, but also allows fine-grained tracking of information communicated through exceptions.

The DoD Orange book requires a dynamic mechanism for enforcing *mandatory access control* (MAC) for secure systems of class B1 and higher [DOD85]. In this approach, a fixed label is associated with the currently running process. As in the Bell-LaPadula model, a process may read only from objects with a label that is of the same or lower level than its own. However, it may write to an object with an equal or higher-security label. The Orange Book specifies that in systems with mandatory access control, information can leak only by leaving the system through *channels*. There are two kinds of channels: single-level channels, which have a single fixed label against which all data is dynamically tested before transmission; and multi-level channels, which allow arbitrarily labeled data to be transmitted, but also dynamically transmit the label of the data along with it.

The JFlow language provides both static and dynamic enforcement of information flow, with an emphasis on making static enforcement as expressive as possible. However, the dynamic enforcement features of mandatory access control can be simulated in JFlow by using run-time labels and run-time principals. Channels in the decentralized label model are single-level channels; however, multi-level channels can be simulated by transmitting values of the type Protected, which encapsulates a value with its label. JFlow also provides fine-grained tracking of information labels. With mandatory access control, a process is irrevocably tainted by the label of data it has observed, and therefore passes the label on to all data it touches afterward, making that data unnecessarily restrictive. This approach is necessary with purely dynamic enforcement in order to prevent implicit flows. The fine-grained static analysis in JFlow allows implicit flows to be prevented while avoiding many unnecessary restrictions.

There has been considerable work on developing richer and more expressive models for labeling data. Denning extended and clarified the Bell-LaPadula label model with the notion of a *lattice* of security classes [Den75, Den76]. As in the model defined in this thesis, information may be relabeled upward in the lattice, and information derived from multiple sources acquires a label (security class) that is the join of

the labels of the sources. The decentralized label model does not quite fit into Denning's lattice structure, although it retains the essential properties. One obvious difference is that the decentralized label model supports a limited form of declassification. The label system looks different to each principal; every principal shares a common set of safe relabelings, but has access to its own *declassification relabelings*. Relabeling in the decentralized label model defines an ordering relation ($\sqsubseteq$), as in Denning's model, but it is not a partial order, since two labels may be equivalent without being equal. However, it does support the lattice operations of join ($\sqcup$) and meet ($\sqcap$) on equivalence classes of labels, and these operations distribute over each other.

Denning's lattice framework was instantiated by Feiertag et al. [FLR77] in *multilevel security policies*. A multilevel security policy is a pair (A, C), where A is a *hierarchical security class*, and C is a set of *categories*. Hierarchical security classes form a totally ordered set like that of the Bell-LaPadula model; categories are arbitrary symbols. One multilevel security policy (A_1, C_1) can be relabeled to another, (A_2, C_2), as long as $A_1 \sqsubseteq A_2$ and $C_1 \subseteq C_2$. Categories operate in the reverse direction one might expect: it is acceptable to increase the set of categories but not to decrease them. They provide a notion of the owners of the data rather than of potential readers of the data.

Multilevel security policies are a common underlying model used with mandatory access control systems. However, they can be modeled straightforwardly within the decentralized label model by introducing principals to represent each of the hierarchical security classes and each of the possible categories. The principals representing security classes have the corresponding acts-for relations: the principal representing top secret can act for the principal representing secret, and so on. A multilevel policy $(A, \{c_1 \ldots c_n\})$ is translated to a decentralized label $\{A : \ ; \ c_1 : \ ; \ldots ; \ c_n : \ \}$; the complete relabeling rule then enforces exactly the relabeling rule for multilevel policies. Users are given security classifications by introducing acts-for relations between their principals and the appropriate A and c_i principals; the output channel to a user p can be labeled root : p (where root is a highly trusted principal) and the relabeling rule will enforce the appropriate restriction. One weakness of this translation is that it allows the user p to declassify all the data he can read; this flaw can be fixed using the approach of Section 2.6.3.

Biba showed that information flow control can be used to enforce *integrity* as well as secrecy, and that integrity is a dual of secrecy [Bib77]; this insight has been employed in several subsequent systems, and also applies to the decentralized integrity policies described in Section 2.6.1. IX [MR92] is a good example of a real-world information flow control system that implements MAC and supports both secrecy and integrity policies simultaneously.

More recent work on label models has not been as widely adopted. One popular theme has been models for commercial applications that capture conflicts of interest and allow non-transitive flow policies [CW87, BN89, TW89, Fol91]. The Chinese Wall policy of Brewer and Nash [BN89] has been the subject of some study. The idea behind this policy is that information labels should be able to enforce separation of duties. For example, a bank might maintain a separation between its accounts and investments departments. An employee who is supposed to handle the investments of the bank should not have access

to information about customer accounts, and vice versa. However, Sandhu has argued that the Chinese Wall policy can be implemented according to a standard lattice-based labeling policy by properly distinguishing users and programs [San92]. In the decentralized label model, this separation of duties can be enforced through restrictions on the principal hierarchy rather than through labels. The group principals accounts and investments are introduced, and employee principals are prohibited from belonging to both groups. This structure is arguably more intuitive, since the separation of duties is built into the principals themselves, rather than into the labels of individual pieces of data. More recent work on modeling separation of duties has taken a similar approach of mapping user and duties into a role hierarchy [GGF98].

The decentralized label model has several similarities to the ORAC model of McCollum et al. [MMN90]: both models provide some approximation of the "originator-controlled release" labeling used by the U.S. DoD/Intelligence community. The ORAC model was developed because of the observation that conventional MAC and DAC policies do not adequately support this kind of security policy. Both ORAC and the decentralized label model have the key concept of *ownership* of policies. Both models also support the joining of labels as computation occurs, though the ORAC model lacks some important lattice properties since it attempts to merge policies with common owners. In the ORAC model, as in some mandatory access control models, both process labels and object labels can float upward in the label lattice arbitrarily, a phenomenon called *label creep* that leads to excessively restrictive labels. The absence of lattice properties and the dynamic binding of labels to objects and processes makes any static analysis of the ORAC model rather difficult. Interestingly, ORAC does allow owners to be replaced in label components (based on ACL checks that are analogous to acts-for checks), but it does not support extension of the reader set. The ORAC model also does not support any form of declassification.

All practical information flow control systems provide the ability to *declassify* or *downgrade* data because strict information flow control is too restrictive for writing real applications. More complex mechanisms such as *inference controls* [Den82, SS98] often are used to decide when declassification is appropriate. Declassification in these systems lies outside the label model, so declassification is performed by a *trusted subject*: code with the authority of a highly trusted principal. A recent variant of this approach by Ferrari et. al [FSBJ97] introduces a form of dynamically-checked declassification through special *waivers* to strict flow checking. Some of the need for declassification in their framework would be avoided with fine-grained static analysis. Because waivers are applied dynamically and mention specific data objects, they seem likely to have administrative and run-time overheads. One key advantage of the new label structure is that it is *decentralized*: unlike in the trusted subject approach, other principals in the system need not trust the declassification decision of a principal p, since p cannot weaken the policies of principals that it does not act for.

Previous information flow techniques do not deal well with situations of mutual distrust. These techniques were originally designed to protect the privacy and integrity of data owned by a single principal—typically, the government. If one considers privacy and integrity in a more decentralized setting, such as the community of Web users, it is clear that no universal notion of secret sensitivity can be established.

No label including a hierarchical security class can be acceptable in a decentralized environment. Even schemes containing a generalized lattice of labels do not solve the problem of mutual distrust. Consider the tax preparation example in a lattice-based MAC system. Unless Bob can act for Preparer or vice-versa, the final tax form in this example will be labeled so that neither Bob nor Preparer are able to read it—a result that is safe but not very useful.

JFlow provides a programming model that integrates information flow control and a simple model of access control. Stoughton [Sto81] developed a purely dynamic model integrating both access control and information flow control, defined formally using denotational semantics. This model does not seem to have been implemented. In the model, objects have both a *current access level* and a *potential access level*. The potential access level is used to enforce information flow constraints as in mandatory access control systems. The current access level is used to enforce discretionary access control; it can be relaxed by an appropriately trusted principal, but only to the point where it is as restrictive as the potential access level. To relax it further would violate information flow control. Thus, this model does not support declassification. Because this model is purely dynamic, it also does not treat implicit flows securely. The model of access control is particularly simple; it mediates accesses at the level of reads and writes to objects, and does not provide the ability to control higher-level operations.

6.4 Static enforcement of security policies

JFlow is unusual not only in integrating information flow control and access control, but also in providing both static and dynamic enforcement of these mechanisms. Most prior security work has focused on dynamic enforcement, but there has been some earlier work on static enforcement of access control.

Jones and Liskov defined a system for statically enforcing discretionary access control through a scheme of restricted types, in which some methods were marked as inaccessible [JL78]. Their rules define a form of subtyping, with security guaranteed by the inability to cast downward in the type hierarchy dynamically. However, the lack of any capability for dynamically enforcing access control checks makes this scheme impractical.

The CACL model of access control [RSC92] has a model of mixed static and dynamic enforcement of access control that is more practical. As in the Jones and Liskov model, references to objects may have a type in which certain methods are inaccessible. However, when objects cross protection domains, new copies of the references are constructed for which method accessibility is recomputed lazily. In JFlow, methods can be called only if all of their caller constraints are satisfied. When objects are passed between different trust domains, method accessibility changes automatically based on static reasoning about authority; no rewriting is needed.

Static analysis was applied to information flow control early on by Denning and Denning [DD77], but has not been adopted widely since because of its limitations. Static checking allows the fine-grained tracking of sensitivity and integrity labels through program computations, without the run-time overhead of dynamic

security classes. Because this approach inspects entire programs, it has a significant advantage over simple dynamic checking: a program can be checked to determine that no possible execution results in a security policy violation. However, dynamic checking is needed for some programming examples, and previous static checking techniques did not integrate dynamic checking, making them impractical. Earlier static checking techniques did not handle exceptions, either.

Another approach to checking programs for information flows statically has been automatic or semi-automatic theorem proving. Researchers at MITRE [Mil76, Mil81] and SRI [Fei80] developed techniques for information flow checking using formal specifications. Feiertag [Fei80] developed a tool for automatically checking these specifications using a Boyer-Moore theorem prover.

Recently, there has been more interest in provably-secure programming languages, treating information flow checks in the domain of type checking, which does not require a theorem prover. Palsberg and Ørbæk have developed a simple type system for checking integrity [PO95]. Volpano, Smith and Irvine have taken a similar approach to static analysis of secrecy, encoding Denning's rules in a functional type system and showing them to be sound using standard programming language techniques [VSI96, Vol97]. Also, Abadi [Aba97] has examined the problem of achieving secrecy in security protocols, also using typing rules, and has shown that encryption can be treated as a form of safe declassification through a primitive encryption operator.

Heintze and Riecke [HR98] have shown that information-flow-like labels can be applied to a simple language with reference types (the SLam calculus). They show how to statically check an integrated model that provides access control, information flow control, and integrity. Their model is similar to Stoughton's earlier, dynamic model; labels include two components: one that enforces conventional access control, and another that enforces information flow control. Their model inherits some limitations of Stoughton's model.

The models of Smith, Volpano, and Irvine and of Heintze and Riecke have the limitation that they are entirely static: unlike JFlow, they have no run-time access control, no declassification, and no run-time flow checking. These models also do not provide label polymorphism or support for objects. Addition of these features is important for supporting a realistic programming model, though it does make the programming language more difficult to treat with the conventional tools of programming language theory. Heintze and Riecke do prove some useful soundness theorems for their model. This step would be desirable for JFlow, but the various language extensions make formal proofs of correctness difficult at this point.

6.5 Modeling principals and roles

The notion of a principal hierarchy, used in the decentralized label model, is similar to several other models for modeling roles. The acts-for relation is similar to the *speaks-for* relation that is introduced by Lampson et al. [LABW91] for describing authentication in a distributed system. In that model, a notion of *compound principals* is introduced; a compound principal is an expression such as `Bob as manager`, where `Bob` is an ordinary principal, and `manager` is a role. The decentralized label model does not provide this much

structure; however, a compound principal can be modeled as a third principal for which Bob acts, and which acts for manager.

Some work on *role-based access control* has also introduced notions of a *role hierarchy* based on various kinds of dominance relations among principals and roles [FK92, SCFY96]. This structure is used to model the assignment of users to groups and to roles, similarly to the decentralized label model. Roles have also been used as security classes in an information flow model [San96]. However, because this model does not distinguish between roles and information flow labels, information can flow only upward in the role hierarchy.

6.6 Cryptography

In the minds of many people, computer security is associated with encryption. It is reasonable to ask how cryptographic techniques are related to this work. Encryption can be used to achieve some important security goals that are subsidiary to protecting privacy and integrity, and much recent computer security research has focused on this use. One such goal is *authentication*: the reliable identification of *who* is requesting that an action be performed [Lam71, LABW91, ABLP93]. Many computer systems use password checking to authenticate their users. However, in a distributed system, some form of encryption is generally needed to perform authentication securely. Reliable authentication is a prerequisite for protecting privacy and integrity. For example, any access control mechanism requires an underlying authentication mechanism so that one can be sure that a process does possess the granted authority that claims to.

Another important feature of a secure system is reliable information channels that cannot be subverted by unrelated third parties. Encryption protects privacy by preventing these channels from having their information extracted; digital signatures protect integrity by preventing new material from being inserted onto the channel by a third party to fool the receiver.

The encryption technology for reliable authentication and secure channels has been researched heavily and also is widely available, in systems like Kerberos [SNS88] and ssh [Ylo96]. Encryption provides a rather elemental protection for privacy and integrity. The work presented herein makes the assumption that these technologies are available as a standard component, and builds on them.

6.7 Covert channels

This work has ignored covert channels arising from time measurement and thread communication. These channels have long been recognized as very difficult to control [Lam73]. A scheme for statically analyzing thread communication has been proposed [Rei79, AR80]; essentially, a second pc is added with different propagation rules. A local pc handles information flow within a thread; the global pc restricts operations that communicate with other threads. Stoughton's model [Sto81] also uses this local/global approach. The same technique can be used to control timing channels. This approach could be applied to JFlow and even

checked statically, similarly to static side-effect and region analysis [JG91], which aims to infer all possible side-effects caused by a piece of code. However, it is not clear how well this scheme works in practice; it seems likely to restrict timing and communication quite severely, particularly if applied directly to a programming model in which objects are shared between threads. In such a programming model, all object modifications are potentially asynchronous communications with other threads, and will be highly restricted if limited by a pc that is shared across all threads. Smith and Volpano have developed rules recently for checking information flow in a multithreaded functional language [SV98]. As might be expected, the rules they define prevent the run time of a program from depending in any way on non-public data, which is arguably impractical.

Chapter 7

Conclusions

Protecting privacy and secrecy of data has long been known to be a very difficult problem. The increasing use of untrusted programs in decentralized environments with mutual distrust makes a solution to this problem both more important and more difficult to solve. Existing security techniques do not provide satisfactory solutions to this problem.

The goal of this work is to make information flow control a viable technique for providing privacy in a complex, decentralized world with mutually distrusting principals. Information flow control is an attractive approach to protecting the privacy (and integrity) of data because it allows security requirements to be extended transitively towards or away from the principals whose security is being protected. However, it has not been a widely accepted technique because of the excessive restrictiveness it imposes and the computational overhead.

To address these limitations of conventional information flow techniques, this work focuses on two areas. First, a new model of decentralized information flow labels provides the ability to express privacy policies for multiple, mutually distrusting principals, and to enforce all of their security requirements simultaneously. Second, the new language JFlow permits static checking of decentralized information flow annotations. JFlow seems to be the most practical programming language yet that allows this checking.

7.1 Decentralized label model

The decentralized label model described in Chapter 2 makes information flow more practical by removing some of the unnecessary restrictiveness of earlier models. It provides considerable flexibility by allowing individual principals to attach flow policies to individual values manipulated by a program. It also incorporates a notion of principal hierarchy that allows these policies to be expressed in terms of and on behalf of more complex authority entities such as groups and roles.

Practical information flow systems require some ability to declassify or downgrade data. Since the policies in decentralized labels have a notion of ownership, the owner can be allowed to declassify policies that it owns. This declassification is safe because it does not affect the secrecy guarantees to other principals who have an interest in the secrecy of the data. The owner may use reasoning processes such as information

theory techniques or inference controls to determine that the information leaked through declassification is acceptably small, but other principals in the system do not need to trust these reasoning processes. This support for decentralized declassification makes the label model ideal for a system containing mutually distrusting principals.

An important feature of the decentralized label model is the formal semantics that are defined for the model, and the relabeling rule that was shown to be both sound and complete with respect to this formal semantics. The relabeling rule precisely captures all the legal relabelings that are allowed when knowledge about the principal hierarchy is available statically, and has the necessary lattice properties to support static checking and automatic label inference. Because the complete relabeling rule is as permissive as possible without being unsafe, it is easier to model common security paradigms, allowing control of information flow in a system with group or role principals. Examples in Chapter 2 showed that the expressive power of the complete relabeling rule was helpful in modeling reasonable application scenarios without resorting to declassification.

Extensions to the basic model discussed in Chapter 2 also show that integrity [Bib77] constraints have a natural lattice structure, and decentralized integrity policies can also be expressed conveniently in the same framework, with rules precisely dual to those of decentralized privacy policies. In addition, labels that combine integrity and privacy constraints can be expressed, with straightforward rules. Finally, extensions to the principal hierarchy model allow more expressive modeling of group and role principals.

7.2 Static analysis of information flow

Information flow control is usually enforced dynamically, causing substantial loss of performance and also difficulty in handling implicit information flows. Static program checking appears to be the only enforcement technique that can control information flows with reasonable efficiency and precision, although it cannot identify certain covert channels. However, previous static analysis techniques have not been shown to be practical.

Chapters 3–5 describe the new language JFlow, which extends the Java language to permit simple static checking of flow annotations. The goal of this work is to add enough power to the static checking framework to allow reasonable programs to be written in a natural manner. JFlow addresses many of the limitations of previous work in this area. It supports many language features that previously have not been integrated with static flow checking, including mutable objects (which subsume function values), subclassing, dynamic type tests, dynamic access control, and exceptions.

Avoiding unnecessary restrictiveness while supporting a complex language has required the addition of sophisticated language mechanisms: implicit and explicit polymorphism, so that code can be written in a generic fashion; dependent types, to allow dynamic label checking when static label checking would be too restrictive; static reasoning about access control; statically-checked declassification. Making the programming language convenient has also involved automatic label inference, as described in Chapter 5.

This list of mechanisms suggests that one reason why static flow checking has not been accepted widely as a security technique, despite having been invented over two decades ago, is that programming language techniques and type theory were not then sophisticated enough to support a sound, practical programming model. By adapting these techniques, JFlow makes a useful step towards usable static flow checking.

7.3 Future work

There are several directions for extending this work. One obviously important direction is to continue to make it a more practical system for writing applications. JFlow addresses many of the limitations of earlier information flow systems that have prevented their use for the development of reasonable applications; however, more experience is needed to better understand the practical applications of this approach.

One direction for exploration is the development of secure run-time libraries written in JFlow that support JFlow applications. Features of JFlow such as polymorphism and hybrid static/dynamic checking should make it possible to write such libraries in a generic and reusable fashion. One interesting possibility is the development of a secure user interface library that provides event distribution and rendering capabilities available in user interface toolkits. This library should include user interface widgets that support information flow control directly; for example, a type-in that reliably notifies the user of what security policy is applied to data entered into it.

It should also be possible to augment the Java Virtual Machine [LY96] with annotations similar to those used in JFlow source code. The bytecode verifier would check both types and labels at the time that code is downloaded into the system. Other recent work [LY96, Nec97, MWCG98] has shown that type checking performed at compile time can be transformed into machine-code or bytecode annotations. The code can then be transmitted along with the annotations, and the two checked by their receiver to ensure that the machine code obeys the constraints established at compile time. This approach also should be applicable to information flow annotations that are expressible as a kind of type system.

The JFlow language contains relatively complex features such as objects, inheritance and dependent types, and these features have made it difficult thus far to use theoretical programming-language techniques to show that the static checking rules of Chapter 4 are sound. However, this demonstration is important for widespread acceptance of a language for secure computation.

This work has assumed an entirely trusted execution environment. The model described here does not work well in large, networked systems in which different principals may have different levels of trust in the various hosts in the network. One simple technique for dealing with distrusted nodes is to transmit opaque receipts or tokens for the data. Another approach is for a third party to provide a trusted host to get around the impasse of mutually distrusted hosts. It would be interesting to investigate a distributed computational environment in which secure computation is made transparent through the automatic application of these techniques.

This work shows how to control several kinds of information flow channels better, including channels

through storage, implicit flows, and run-time security checks. However, covert channels that arise from timing channels and from the timing of asynchronous communication between threads are not treated in this thesis, by ruling out timing and multi-threaded code. Supporting multi-threaded applications would make this work more widely applicable. Although there has been work on analyzing these channels through static analysis [SV98, HR98], the current techniques are restrictive. One central difficulty is the need to distinguish between locally and globally visible operations within a multi-threaded program. Current multi-threaded programming environments have tended to minimize this distinction, but without it, static analysis will not be a reasonably precise tool for controlling information flow. An altered programming model may be possible in which enough information is available about inter-thread communication to permit precise analysis.

This thesis has provided new models and techniques for protecting privacy. Providing better protection of privacy is a challenging and important problem for future computing environments. These environments are likely to be large and distributed, and to contain distrusted users, programs, and hosts. This problem has not received as much attention recently as it merits, and I hope that the contributions of this thesis will serve as a fresh impetus to its further consideration.

Bibliography

[Aba97] Martín Abadi. Secrecy by typing in security protocols. In *Proc. Theoretical Aspects of Computer Software: Third International Conference*, September 1997.

[ABLP93] Martín Abadi, Michael Burrows, Butler W. Lampson, and Gordon D. Plotkin. A calculus for access control in distributed systems. *TOPLAS*, 15(4):706–734, 1993.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, 1996.

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):237–268, April 1991. Also appeared as SRC Research Report 47.

[AR80] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, 1980.

[Bib77] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977.

[BL75] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975. Available as NTIS AD-A023 588.

[BN89] D. F. Brewer and J. Nash. The Chinese Wall security policy. In *Proc. IEEE Symposium on Security and Privacy*, pages 206–258, May 1989.

[Car91] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*. Springer-Verlag, 1991. An earlier version appeared as DEC Systems Research Center Research Report #45, February 1989.

[CLR90] Thomas A. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CW87] David Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proc. IEEE Symposium on Security and Privacy*, pages 184–194, 1987.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[DE97] Sophia Drossopoulou and Susan Eisenbach. Java is type safe – probably. In *Proceedings of Object-Oriented Programming, 11th European Conference (ECOOP 1997)*, pages 389–418, Jyväskylä, Finland, June 1997. Lecture Notes in Computer Science, Vol. 1241, Springer, 1997.

[Den75] Dorothy E. Denning. *Secure Information Flow in Computer Systems*. PhD thesis, Purdue University, W. Lafayette, Indiana, USA, May 1975.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, 1976.

[Den82] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[DG84] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulæ. *Journal of Logic Programming*, 1(3):267–284, October 1984.

[DGLM95] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. Where Clauses: Constraining Parametric Polymorphism. In *Proc. OOPSLA '95*, pages 156–168, Austin TX, October 1995. ACM SIGPLAN Notices 30(10).

[DOD85] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD (The Orange Book) edition, December 1985.

[DV66] J. B. Dennis and E. C. VanHorn. Programming semantics for multiprogrammed computations. *Comm. of the ACM*, 9(3):143–155, March 1966.

[Fei80] Richard J. Feiertag. A technique for proving specifications are multilevel secure. Technical Report CSL-109, SRI International Computer Science Lab, Menlo Park, California, January 1980.

[Fen73] J. S. Fenton. *Information Protection Systems*. PhD thesis, University of Cambridge, Cambridge, England, 1973.

[Fen74] J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.

[FK92] David Ferraiolo and Richard Kuhn. Role-based access controls. In *15th National Computer Security Conference*, 1992.

[FLR77] R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a system design. *Proc. 6th ACM Symp. on Operating System Principles (SOSP), ACM Operating Systems Review*, 11(5):57–66, November 1977.

[FM96] J. Steven Fritzinger and Marianne Mueller. Java security. Technical report, Sun Microsystems, Inc., 1996.

[Fol91] Simon N. Foley. A taxonomy for information flow policies and models. In *Proc. IEEE Symposium on Security and Privacy*, pages 98–108, 1991.

[FSBJ97] Elena Ferrari, Pierangela Samarati, Elisa Bertino, and Sushil Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 130–140, Oakland, CA, USA, May 1997.

[GGF98] Virgil D. Gligor, Serban I. Gavrila, and David Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *Proc. IEEE Symposium on Security and Privacy*, pages 172–183, Oakland, California, USA, May 1998.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996. ISBN 0-201-63451-1.

[GM84] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20, April 1984.

[GS98] Li Gong and Roland Schemers. Implementing protection domains in the Java Development Kit 1.2. In *The Internet Society Symposium on Network and Distributed System Security*, San Diego, California, USA, March 1998. Internet Society.

[HDT87] Susan Horwitz, Alan Demers, and Tim Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24:679–694, 1987.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, California, January 1998.

[JD96] Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, July 1996.

[JG91] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *ACM Symposium on Principles of Programming Languages*, pages 303–310, January 1991.

[JL75] A. K. Jones and R. J. Lipton. The enforcement of security policies for computation. In *Proc. 5th ACM Symp. on Operating System Principles (SOSP), ACM Operating Systems Review*, pages 197–206, November 1975.

[JL78] Anita K. Jones and Barbara Liskov. A language extension for expressing constraints on data access. *Comm. of the ACM*, 21(5):358–367, May 1978.

[Kil73] G. Kildall. A unified approach to global program optimization. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, 1973.

[KU76] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, 1976.

[KW94] Atsushi Kanamori and Daniel Weise. Worklist management strategies for dataflow analysis. Technical Report MSR–TR–94–12, Microsoft Research, May 1994.

[LAB+84] Barbara Liskov, Russell Atkinson, Toby Bloom, J. Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*. Springer-Verlag, 1984. Also published as Lecture Notes in Computer Science 114, G. Goos and J. Hartmanis, Eds., Springer-Verlag, 1981.

[LABW91] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. In *Proc. 13th ACM Symp. on Operating System Principles (SOSP)*, pages 165–182, October 1991. *Operating System Review*, 253(5).

[Lam71] Butler W. Lampson. Protection. In *Proc. Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, March 1971. Reprinted in Operating Systems Review, 8(1), January 1974, pp. 18–24.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Comm. of the ACM*, 16(10):613–615, October 1973.

[LCD+94] Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Andrew C. Myers. *Theta Reference Manual*. Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, Cambridge, MA, February 1994. Available at http://www.pmg.lcs.mit.edu/papers/thetaref/.

[LMM98] Barbara Liskov, Nicholas Mathewson, and Andrew C. Myers. PolyJ: Parameterized types for Java. Software release. Located at http://www.pmg.lcs.mit.edu/polyj, July 1998.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine*. Addison-Wesley, Englewood Cliffs, NJ, May 1996.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 132–145, Paris, France, January 1997.

[McL88] John McLean. Reasoning about security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 123–131, Oakland, CA, 1988. IEEE.

[McL90] John McLean. Security models and information flow. In *Proc. IEEE Symposium on Security and Privacy*, pages 180–187, 1990.

[MF96] Gary McGraw and Edward Felten. *Java Security: Hostile Applets, Holes and Antidotes*. John Wiley and Sons, New York, 1996.

[Mic97] Microsoft Corporation. *Trust-Based Security for Java*, April 1997.

[Mil76] Jonathan K. Millen. Security kernel validation in practice. *Comm. of the ACM*, 19(5):243–250, May 1976.

[Mil81] Jonathan K. Millen. Information flow analysis of formal specifications. In *Proc. IEEE Symposium on Security and Privacy*, pages 3–8, April 1981.

[Mil87] Jonathan K. Millen. Covert channel capacity. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, 1987.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, Saint-Malo, France, 1997.

[ML98] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 1998.

[MMN90] Catherine J. McCollum, Judith R. Messing, and LouAnna Notargiacomo. Beyond the pale of MAC and DAC—defining new forms of access control. In *Proc. IEEE Symposium on Security and Privacy*, pages 190–200, 1990.

[MR92] M. D. McIlroy and J. A. Reeds. Multilevel security in the UNIX tradition. *Software—Practice and Experience*, 22(8):673–694, August 1992.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, California, January 1998.

[Mye99] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, San Antonio, TX, USA, January 1999.

[Nec97] George C. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 106–119, January 1997.

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

[Net97] Netscape Communications Corporation. *Introduction to the Capabilities Classes*, 1997.

[Nv98] Tobias Nipkow and David von Oheimb. Java$_{light}$ is type-safe—definitely. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 161–170. ACM, New York, January 1998.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 146–159, Paris, France, January 1997.

[PO95] Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. In *Proc. 2nd International Symposium on Static Analysis*, number 983 in Lecture Notes in Computer Science, pages 314–329. Springer, September 1995.

[Rei79] Richard P. Reitman. A mechanism for information control in parallel programs. In *Proc. 7th ACM Symp. on Operating System Principles (SOSP), ACM Operating Systems Review*, pages 55–62, December 1979.

[RM96] Jakob Rehof and Torben Æ. Mogensen. Tractable constraints in finite semilattices. In *Proc. 3rd International Symposium on Static Analysis*, number 1145 in Lecture Notes in Computer Science, pages 285–300. Springer-Verlag, September 1996.

[RSC92] Joel Richardson, Peter Schwarz, and Luis-Felipe Cabrera. CACL: Efficient fine-grained protection for objects. In *Proceedings of the 1992 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 154–165, Vancouver, BC, Canada, October 1992.

[Sal74] J. H. Saltzer. Protection and the control of information sharing in Multics. *Comm. of the ACM*, 17(7):388–402, July 1974.

[San92] Ravi S. Sandhu. A lattice interpretation of the Chinese Wall policy. In *Proc. of the 15th NIST-NCSC National Computer Security Conference*, pages 221–235, Baltimore, Maryland, USA, October 1992.

[San96] Ravi S. Sandhu. Role hierarchies and constraints for lattice-based access controls. In *Proc. Fourth European Symposium on Research in Computer Security*, Rome, Italy, September 25-27 1996.

[SCFY96] Ravi S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *IEEE Computer*, 29:38–47, February 1996.

[SNS88] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. Technical report, Project Athena, MIT, Cambridge, MA, March 1988.

[SS98] Pierangela Samarati and Latanya Sweeney. Generalizing data to provide anonymity when disclosing information. In *ACM Principles of Database Systems*, Seattle, Washington, USA, June 1998.

[Sto81] Allen Stoughton. Access flow: A protection model which integrates access control and information flow. In *IEEE Symposium on Security and Privacy*, pages 9–18. IEEE Computer Society Press, 1981.

[Sto87] B. Stoustrup. *The C++ Programming Language*. Addison-Wesley, 1987.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, California, January 1998.

[Swe96] Latanya Sweeney. Replacing personally-identifying information in medical records, the Scrub System. *Proceedings, Journal of the American Medical Informatics Association*, pages 333–337, 1996.

[Sym97] Don Syme. Proving Java type soundness. Technical Report 427, University of Cambridge Computer Laboratory, June 1997.

[TW89] Phil Terry and Simon Wiseman. A 'new' security model. In *Proc. IEEE Symposium on Security and Privacy*, pages 215–228, 1989.

[Vol97] Dennis Volpano. Provably-secure programming languages for remote evaluation. *ACM SIGPLAN Notices*, 32(1):117–119, January 1997.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[WBF97] Dan S. Wallach, Dirk Balfanz, and Edward W. Felten. Extensible security architectures for Java. In *Proc. 16th ACM Symp. on Operating System Principles (SOSP)*, pages 116–128, Saint-Malo, France, October 1997.

[WCC+74] W. A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor system. *Comm. of the ACM*, 17(6):337–345, June 1974.

[WF98] Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 1998.

[Ylo96] Tatu Ylonen. Ssh – secure login connections over the Internet. In *The Sixth USENIX Security Symposium Proceedings*, pages 37–42, San Jose, California, 1996.

Index

access control, 14, 28, 58
  discretionary, 68, 149, 151
  integrating with, 155, 156
  mandatory, 151
acts-for relation, 30, 33
  extending, 55
actsFor statement, 68, 117, 146
annotations, 38
arrays, 66, 78, 112
  access rule, 110
  assignment rule, 112
assignment rules, 112
authority, 25, 68, 73, 98, 124, 125, 146

bank example, 70
basic blocks, 64
boolean satisfaction problem, 136
break statement, 114

capabilities, 68, 80, 149
channels, 26, 29, 50
Chinese Wall policy, 153
classes, 71, 80, 98, 103, 120, 123
code labels, 54
compatibility with Java, 85
completeness, 39
  proof, 42
constraints, 73, 120, 125, 133
  dynamic, 138
constructors, 120, 129
continue statement, 114
correctness condition
  dynamic, 36
  static, 39
covariant parameters, 77, 100, 124
covert channels, 60, 62, 65, 84, 157
cryptography, 157

dataflow analysis, 144
declassification, 13, 21, 25, 54, 69
declassify statement and expression, 118, 146
dependent types, 77
devices, 55
discretionary access control, 14, 28, 149, 151
distribution properties, 45, 49
dynamic cast, 116
dynamic checking, 62
dynamic constraints, 138
dynamic labels, 66, 148
dynamic principals, 70, 146

empty statement, 108
encryption, 157
environments, 92, 97, 103, 123
example
  bad relabeling, 39
  bank, 70
  complex number, 81
  hospital, 28
  password file, 82
  protected, 84
  tax preparer, 12, 27
  vector, 71, 76
exceptions, 93
  unchecked, 75, 85, 94

fields, *see* instance variables
final variables, 77, 105
finalizers, 84
flows, 32

greatest lower bound, 25

hashCode, 84
Horn clauses, 136
hospital example, 28

if statement, 113
implicit flows, 62, 64
incremental relabeling, 30, 41, 52
inference rules, 92
input channels, 26

instance variables, 64, 112, 130
    access rule, 110
    assignment rule, 112
instanceof, 116
integrity, 51–153
    combining with privacy, 54
    declassification for, 54
    policies, 51
interpretation functions, 35
interpreting labels, 32

JFlow, 60
join ($\sqcup$), 25, 46
judgements, 93

labeled statement, 114
labeled types, 63
labels, 22, 98, 102
    code, 54
    components, 98, 103
    creep, 24
    defaults, 72
    dynamic checking, 62
    generalizing, 51
    inference, 61
    interpretation function, 33, 35
    label expressions, 99, 146
    paths, 94
    polymorphism, 61, 75
    program-counter, 64, 69
    restriction, 30
    run-time, 66, 119, 148
    semantics, 32
    static checking, 62, 92
    variables, 23, 103
lattices, 49, 137
least upper bound, 25
literal, 108

mandatory access control, 151
meet ($\sqcap$), 25, 47
meet operator, 136, 137
member functions, *see* methods
member variables, *see* instance variables
methods, 71, 123, 127
    arguments, 127
    bodies, 128
    calls, 120
    constraints, 73, 128
    signatures, 106, 124, 125
monotonicity, 36, 39
multilevel security, 62, 153
mutual distrust, 13, 154

notation, 95, 96, 98

originator-controlled release, 154
output channels, 26, 29, 50
owners, 22

parameterized types, 75
parameters, 100, 103
    covariant, 77
password file example, 82
path labels, 94
policies
    integrity, 51
    privacy, 22
    redundant, 46
PolyJ, 77
polymorphism, 75
    labels, 61
principals, 22, 98, 102, 146
    hierarchy, 22, 68, 74, 98, 117
    owners, 22
    readers, 22
    run-time, 70, 146
    writers, 51
principle of least privilege, 32, 56
privacy, 22–26
    combining with integrity, 54
process authority, 25
program annotations, 38
program-counter label, 64, 69
proof
    of relabeling completeness, 42
    of relabeling soundness, 42
protected example, 84

readers, 22
    constraint, 34
redundant policies, 46
relabeling
    complete rule, 40
    examples, 24, 39
    incremental, 30, 41, 52
    proof of completeness, 42
    proof of soundness, 42

relation $\sqsubseteq$, 41
subset rule, 23
relations
$\succeq$, 22, 55
$\sqsubseteq$, 23, 41, 53
restriction, 30
revocation, 68
role-based access control, 156
rules
actsFor statement, 117
arithmetic, 109
break, 114
constructors, 129
continue, 114
declassify statement and expression, 118
dynamic cast, 116
empty statement, 108
exceptions, 115
if, 113
instanceof, 116
literal, 108
local variables, 110
method arguments, 127
method bodies, 128
method declarations, 127
sequence, 113
signature compatibility, 125
single-path, 108
subtyping, 106
switch label, 119
variable access, 110
variable assignment, 112
while, 113
run-time labels, 66, 119, 148
run-time principals, 70, 146

separation of duties, 153
signatures of methods, 106, 124, 125
single-path rule, 108
soundness, 24
proof, 42
stack inspection, 149
static checking, 62
static correctness condition, 39
static security enforcement, 155
strongly connected components, 143
subset relabeling rule, 23, 30, 39
subtyping, 64, 106
switch label statement, 119, 146, 148

tax preparer example, 12, 27
termination channels, 65
theorem proving, 156
threads, 84
throw statement, 115
timing channels, 65, 84, 85
topological sort, 143
transactions, 69
translating JFlow code, 146
trusted subject, 15, 154
try/catch/finally statement, 115
types
dependent, 77
discrimination, 78
interpretation, 100
labeled, 63
parameters, 78
static checking, 92
subtype rules, 106
subtyping, 64

uids, 102

variables, 23, 112
access rules, 110
assignment rule, 112
extend function, 110
final, 105
static, 84
Vector, 76
visibility, 82, 92

waivers, 154
while statement, 113