Information Flow Analysis for Mobile Code in Dynamic Security Environments

Robert Grabowski

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig–Maximilians–Universität München

Erstgutachter:	Prof. Martin Hofmann, PhD *Ludwig-Maximilians-Universität München*
Zweitgutachter:	Prof. Dr. Heiko Mantel *Technische Universität Darmstadt*
Abgabedatum:	4. August 2011
Disputationsdatum:	23. März 2012

Summary

With the growing amount of data handled by Internet-enabled mobile devices, the task of preventing software from leaking confidential information is becoming increasingly important. At the same time, mobile applications are typically executed on different devices whose users have varying requirements for the privacy of their data. Users should be able to define their personal information security settings, and they should get a reliable assurance that the installed software respects these settings.

Language-based information flow security focuses on the analysis of programs to determine information flows among accessed data resources of different security levels, and to verify and formally certify that these flows follow a given policy. In the mobile code scenario, however, both the dynamic aspect of the security environment and the fact that mobile software is distributed as bytecode pose a challenge for existing static analysis approaches.

This thesis presents a language-based mechanism to certify information flow security in the presence of dynamic environments. An object-oriented high-level language as well as a bytecode language are equipped with facilities to inspect user-defined information flow security settings at runtime. This way, the software developer can create privacy-aware programs that can adapt their behaviour to arbitrary security environments, a property that is formalized as “universal noninterference”.

This property is statically verified by an information flow type system that uses restrictive forms of dependent types to judge abstractly on the concrete security policy that is effective at runtime. To verify compiled bytecode programs, a low-level version of the type system is presented that works on an intermediate code representation in which the original program structure is partially restored. Rigorous soundness proofs and a type-preserving compilation enable the generation of certified bytecode programs in the style of proof-carrying code.

To show the practical feasibility of the approach, the system is implemented and demonstrated on a concrete application scenario, where personal data are sent from a mobile device to a server on the Internet.

Zusammenfassung

Da internetfähige mobile Geräte eine zunehmende Menge an Daten verarbeiten, wird die Frage, wie die Veröffentlichung vertraulicher Informationen durch Software verhindert werden kann, immer wichtiger. Gleichzeitig werden mobile Applikationen typischerweise auf vielen unterschiedlichen Geräten ausgeführt, deren Benutzer verschiedene Anforderungen an die Vertraulichkeit ihrer Daten haben. Anwender sollten daher die Möglichkeit haben, ihre persönlichen Informationssicherheits-Einstellungen selbst zu setzen, und sie sollten eine zuverlässige Zusicherung bekommen, dass die installierte Software diese Einstellungen respektiert.

Das Gebiet der programmiersprachen-basierten Informationsflusssicherheit beschäftigt sich mit der Analyse von Programmen in Bezug auf Informationsflüsse zwischen Datenquellen mit verschiedenen Sicherheitsstufen, und mit der Verifikation und formellen Zertifizierung, dass diese Flüsse einer gegebenen Sicherheitspolitik entsprechen. Wenn es um mobilen Code geht, stellen aber sowohl der dynamische Charakter der Sicherheitsumgebung wie auch die Tatsache, dass mobile Software als Bytecode verteilt wird, eine Herausforderung für existierende statische Analyseansätze dar.

Diese Dissertation stellt einen programmiersprachen-basierten Mechanismus vor, um Informationsflusssicherheit in dynamischen Umgebungen zu zertifizieren. Eine objektorientierte Hochsprache und eine Bytecodesprache werden dazu mit der Möglichkeit erweitert, benutzerdefinierte Informationsfluss-Sicherheitseinstellungen zur Laufzeit zu untersuchen. Auf diese Weise kann der Softwareentwickler privatsphärenunterstützende Programme schreiben, die ihr Verhalten beliebigen Sicherheitsumgebungen anpassen können, eine Eigenschaft, die als "universelle Nichtinterferenz" formalisiert wird.

Diese Eigenschaft wird statisch durch ein Informationsfluss-Typsystem überprüft, das eine eingeschränkte Form von abhängigen Typen benutzt, um abstrakt über die konkrete, zur Laufzeit gültige Sicherheitspolitik zu urteilen. Um übersetzte Bytecode-Programme zu verifizieren, wird eine Low-Level-Version des Typsystems vorgestellt, welches auf einer Zwischendarstellung des Codes arbeitet, in der die ursprüngliche Programmstruktur teilweise wiederhergestellt worden ist. Ausführliche Korrektheitsbeweise und eine typerhaltende Übersetzung machen die Erzeugung von zertifizierten Bytecodeprogrammen im Stil von Proof-Carrying Code möglich.

Um die praktische Durchführbarkeit des Ansatzes zu zeigen, wird das System implementiert und anhand eines konkreten Anwendungsszenarios demonstriert, in dem persönliche Daten von einem mobilen Gerät zu einem Server im Internet gesendet werden.

Acknowledgements

The work on this dissertation, from the search for the topic to the final manuscript, would never have been possible without the support of a large number of people.

Martin Hofmann and Lennart Beringer got me interested in the field of information flow security. I thank them for countless discussions, for constant advice and suggestions, and for giving me the freedom to explore the topic in detail. I profited greatly from their profound knowledge in program analysis and type systems, and thank them for pointing out related and often helpful other work.

Alexander Knapp, Florian Lasinger, and David von Oheimb also provided vital inspiration to get my work started. Among others, they introduced me to a higher-level, more application-driven perspective on information flow security.

As I was developing the thesis, I received valuable feedback from many different people. In particular, Gilles Barthe and Daniel Hedin from IMDEA Madrid helped me clarify some aspects, Heiko Mantel and Henning Sudbrock from TU Darmstadt pointed out several ideas, and I got helpful remarks from Andrew Appel, David Walker, Rob Dockins, and others during my time at Princeton University. I also got the opportunity to discuss my work with Andrei Sabelfeld, David Sands, Sruthi Bandhakavi, Anindya Banerjee, Reiner Hähnle, Tarmo Uustalu, Florian Kammüller, Rebekah Leslie, Vladimir Klebanov, and many others.

I am profoundly indebted to Jan Hoffmann who shared a noisy office with me for several years. He and my colleagues Ulrich Schöpp and Jan Johannsen did a great job with proofreading the manuscript. I also thank Máté Kovács, Vivek Nigam, Dulma Rodriguez, Andreas Abel, and others from LMU and TU Munich for listening to my ideas, and Sigrid Roden and Max Jakob for helping me with administrative issues.

For my work, I received funding from the DFG projects InfoZert and PolyNI, from the EU project MOBIUS, and from the German Academic Exchange Service (DAAD). I also got various financial support to attend summer and winter schools, lectures of the doctorate programme PUMA, and meetings of the RS3 priority programme.

I am especially grateful to Marianne, who did a tremendous job of supporting me all these years, encouraging me to continue my work, and helping me to firmly pursue the goal of finishing the dissertation. I also thank my loving family for never asking me when the thesis will be ready, and my friends for often asking me this.

Contents

Summary i

Acknowledgements v

1 Introduction 1
- 1.1 Application Scenario . . . 3
- 1.2 Challenges . . . 5
- 1.3 Main Approach . . . 7
- 1.4 Privacy-Aware Programs . . . 9
- 1.5 The Verification Framework . . . 11
- 1.6 Synopsis . . . 13

2 Universal Noninterference for a Java-Like Language 15
- 2.1 The DSD Language . . . 15
- 2.2 An Example Program . . . 22
- 2.3 Universal Noninterference . . . 23

3 High-Level Type System 31
- 3.1 Labels: Symbolic Security Domains . . . 31
- 3.2 Ordering Labels . . . 33
- 3.3 Typing Expressions, and Meta-Label Monotonicity . . . 37
- 3.4 Typing Statements . . . 39
- 3.5 Soundness Results . . . 47
- 3.6 Typing the Example Program . . . 48

4 Universal Noninterference for a JVM-Like Language 51
- 4.1 The DSD Bytecode Language . . . 51
- 4.2 Compilation to Bytecode . . . 56
- 4.3 Universal Noninterference . . . 61

5 Low-Level Type System **63**
5.1 Intermediate Representation . . . 64
5.2 Universal Noninterference . . . 68
5.3 Transforming Bytecode to the Intermediate Representation . . . 70
5.4 The IR Type System . . . 75
5.5 Type Preservation Result . . . 80
5.6 Putting It All Together . . . 81

6 Automatic Type Inference and Implementation **83**
6.1 Solving Label Order Conditions . . . 83
6.2 Algorithmic Version of the High-Level Type System . . . 86
6.3 Type Inference for the Intermediate Representation . . . 90
6.4 Implementation . . . 91

7 Related Work **97**
7.1 Static Language-Based Information Flow Analysis . . . 97
7.2 Static Analysis of Privacy-Aware Software . . . 99
7.3 Bytecode Information Flow Analysis . . . 101
7.4 Other Previous Work . . . 103

8 Conclusion and Outlook **107**
8.1 The DSD Approach in the Real World . . . 108
8.2 Towards a More Flexible Framework . . . 109

A Correctness Proof for the High-Level Type System **113**
A.1 Expression Typing Soundness . . . 113
A.2 Statement Typing Soundness . . . 116

B Proof of the Semantics Preservation Result **123**
B.1 Call Depth Annotations . . . 123
B.2 Proof of Semantics Preservation . . . 126

C Correctness Proof for the IR Type System **131**
C.1 Properties of Single Executions . . . 131
C.2 Small-Step and Big-Step Universal Noninterference . . . 138

D Type-Preserving Compilation **147**
D.1 Properties of the IR Program . . . 147
D.2 Type Derivation for the IR Program . . . 152

Bibliography **161**

1 Introduction

Modern software systems manage and process an increasing amount of personal information. At the same time, the goal to keep this data confidential becomes more and more important. Smartphones in particular have recently become a significant application scenario: they typically carry a lot of personal information, they allow the installation of many additional applications that may access the stored information, and at the same time they provide a connection to the Internet. The task to prevent applications from publishing private information or leaking it to untrusted destinations is thus more relevant than ever.

Classically, the integrity and confidentiality of data is guaranteed by implementations based on *access control mechanisms* [Fen73; BL73; FLR77; SEL09]. They are intuitive to use for standard application scenarios, well-understood by security experts, and can be enforced strictly. The problem, however, is that they cannot enforce restrictions on the *propagation* of information, which is highly important in practice. For example, a smartphone application may legitimately need access to both personal data and the Internet. With access control, one may narrow down the set of accessed objects, define precise read and write permissions, or change these permissions during a program execution. Even so, all these measures can give at most an approximate approach to the goal of preventing information leakage, while causing complex security policies for even simple information flow requirements. For the normal smartphone user, as a result, the question of protecting data confidentiality while an application is running usually falls back to determining the trustworthiness of the application. Thus, the advantages of easy usability and reliable security guarantees are subverted when access control-based mechanisms are used to regulate the propagation of information.

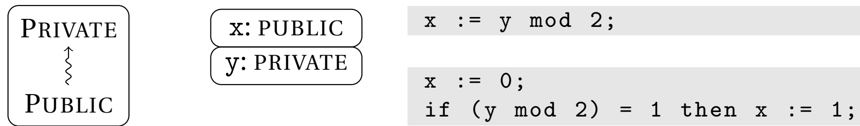

Figure 1.1: Simple information flow policy, and examples for insecure code

For these reasons, *information flow security* mechanisms have been proposed. Just as in access control, one considers data objects of a system, such as variables, files, or sockets, and assigns a *security domain* to each of them, such as PUBLIC and PRIVATE. However, instead of preventing access to data of specific domains altogether, the flow of information in a piece of software is analysed and verified against a given *information flow policy*, which defines the allowed flows among objects of these domains. For example, a flow policy may define that data may flow freely within the PUBLIC domain, or within the PRIVATE domain, or from PUBLIC to PRIVATE, but not from PRIVATE to PUBLIC, thereby preventing a leak of private information. A graphical representation of this policy is shown on the left of Figure 1.1; reflexive flow edges are left implicit.

The program property of adherence to an information flow policy has been semantically characterized as *noninterference*, a concept introduced by Goguen and Meseguer [GM82]. The property states that those parts of the program output that are visible to an observer must not depend on input parameters that are invisible to the same observer. To put it differently, private inputs must not affect the computation of public outputs. A noninterferent program thus transfers information among data objects in a manner that respects the flow policy. In the following, such programs are also simply called *secure*, as this thesis exclusively focuses on noninterference.

The first definitions of noninterference have been given for systems that are modelled as automata with multiple security levels, building directly on the access control model by Bell and LaPadula [BL73]. More recently, *language-based* formalizations of noninterference have been presented that consider programs on the level of code, and formalize the property in terms of the program semantics.

As examples for insecure programs, consider the two short WHILE language programs on the right in the figure above. The data objects are the program variables x and y, where we assume that x is assigned the security level PUBLIC, and y is assigned the level PRIVATE. According to the example policy on the left of the figure, no information may be transferred from the private variable y to the public variable x. The program on top thus clearly violates the policy, as the final value of the public variable x depends on the secret initial value of y, even if only the one-bit information about the parity of y is leaked. The problematic information leakage is directly caused by assigning secret data to a public variable. The program on the bottom is also insecure: Although there is no direct flow from y to x, information is leaked via a control flow structure; here,

a public variable depends on the value of a conditional expression involving private data. (In fact, the program is semantically equivalent to the first one.)

Classic noninterference is an *end-to-end* property that only gives requirements on the inputs and outputs of a program; any violations of the policy that occur temporarily during the execution of a program are not considered. The property is also *termination-insensitive*: it only talks about terminating programs, such that covert leaks of private information via the termination behaviour are excluded from the security examination.

To verify the security of information flows in software, Denning and Denning [DD77] have developed a static program analysis that can automatically check a given program for noninterference. More recently, Volpano, Smith, and Irvine [VSI96] have presented an information flow type system, and also formulated a proof of correctness for their analysis, which states that all typable programs are noninterferent. As with all correct program analyses, however, the information flow type system is not complete — that is, there are always secure programs that are rejected by the type system.

Since then, the popular baseline security notion of noninterference has been the basis of a larger research field. Over time, many formalizations of information flow security properties have been developed. The software system may be given as a transition system, or as code of a programming language; the flow policy may be formalized as a transitive order, or it may require that information passes security domains one after another intransitively; the security property may additionally include covert information leaks introduced by indirect channels such as execution time, heap space consumption, termination behaviour, and others; the property may be extended to data in program states reached during the execution, or to programs that do not terminate; and the security policy may be static or may change in certain situations, in particular by allowing the declassification of data. Likewise, different analysis techniques to check whether a program transfers data between those objects in a manner that respects the policy have been developed. A further explanation of these general categories and an overview of different approaches in the field of language-based information flow security can be found in survey papers [SM03; SS05].

Let us now consider a concrete application scenario, in which smartphone users are to be provided with privacy guarantees for software that operates on their personal data. The scenario provides a number of challenges that are, as it turns out, not easily addressable by existing techniques.

1.1 Application Scenario

A typical smartphone user stores various personal information on the mobile device, such as e-mails, calendars, contacts, or photos. A smartphone also allows the installation of additional software (apps), that can be downloaded from centralized distribution platforms (app stores), such as those provided by Google or Apple. Downloadable

software generally has access to data stored on the phone, and can communicate with Internet services. However, different users have different personal data stored on their devices, and the availability of external resources may vary. Also, users typically have different priorities with respect to the confidentiality of their data.

Consider, as a concrete use case, two users Dave and Sue, both having a smartphone with personal information, which we assume to be just a set of files for the sake of simplicity. Moreover, the phone also has access to external resources, such as data stored on some specific server, or simply “the Internet” in general. Both users wish to install an application that synchronizes local files with an online cloud storage service. The application thus has legitimate access to both local data and the Internet.

While Sue has a lot of files on her phone and would like to define intricate privacy settings on them, Dave just manages a few files and does not care about information flow security at all. Initially, the application copies files from the phone to the cloud service. During this process, Dave is content with backing up all the files, whereas Sue considers the cloud service as “somewhat public” and is therefore reluctant to send data there that she finds highly confidential. Also, she wants a guarantee that her data is not sent to any other place on the Internet.

In this application scenario, the users have the ability to classify their data and their view on the world (e.g. the Internet) by confidentiality levels, and to specify a flow policy that describes which flows among these security levels are allowed. The classification and the policy form the *security environment*; it may vary among users, and may be changed from time to time by the user.

To express her security intention, Sue thus assigns the security domains LOW, MED, and HIGH to her files to indicate their confidentiality level, where LOW stands for the most public classification, and HIGH represents the most private classification. She treats the cloud server as MED, and all other Internet connections as LOW. Her flow policy says that data must not flow from a higher to a lower security level. Dave, in contrast, assigns the same confidentiality level (security domain) DEF to all data resources, and defines a trivial flow policy that allows any flows within the domain DEF.

The goal is to ensure that when the software is executed on either Dave’s or Sue’s smartphone, it does not leak contents of files with a certain security level to servers that are marked as less confidential by the respective user. In other words, the application should always comply with the user-defined security environment at execution time, whatever it looks like at the given moment. In case the application needs to perform an action not endorsed by the security policy, the application detects this situation at runtime and presents an error message, or provides other options to circumvent the problematic action; the synchronization application could for example simply skip files whose contents shall not be copied.

The software is also able to synchronize files with storage devices on the local network. When Sue has a trusted local server with the security domain HIGH, the software should be able additionally to backup all her files to that server, including files marked

HIGH. Note that it is not possible to enforce precisely the desired security property with access control mechanisms: we do not want to prevent access to HIGH security files, but rather control the propagation of their contents to different destinations.

Both users should get a guarantee that the installed software always adheres to their respective security environments. They should be able to modify any aspect of their security settings without the need to download or verify the software again.

The guarantee is obtained by a formal analysis of the program code. However, the executable code is not checked on the phone directly, for two reasons. First, the computational capabilities of a device may be too limited to perform a complex software analysis without sacrificing performance. Second, many contemporary mobile distribution platforms have an interest in being able to verify the submitted apps before distributing them to the end user in the first place; today, the Apple app store, for example, already performs extensive verifications of submitted applications for various other security properties and functionality aspects.

In the scenario, distribution platform maintainers are equipped with an automatic information flow analysis tool that verifies that the cloud synchronization application respects arbitrary user-defined security policies. The tool should generate a formal proof certificate that is provided with the software download. The phone's operating system is equipped with a small trusted checker that only needs to verify the certificate. This gives both Dave and Sue a reliable assurance that the software may be safely installed and, when executed, always complies with their respective security environments.

1.2 Challenges

For information-flow–secure mobile software, a number of challenges come together.

- Different users have different personal data stored on their devices, and the availability of additional resources such as online services may vary. Users should be able to define their own security environments (information flow policies and security domains).

- The software developer does not simply want to make assumptions about the execution environment, which would render programs potentially insecure or useless on devices that do not meet these assumptions. Instead, she desires a simple method to create programs that respect arbitrary user-defined environments, and can thus be safely executed on any of the respective devices.

- The user would like to get a reliable security guarantee, therefore the execution model and the noninterference property have to be rigorously formalized and verified. As mobile software is deployed in an unstructured bytecode form, the

verification algorithm and the guarantee should be given for the low-level code, too.

- It should not be required to perform the analysis on the device where the code is executed. The analysis should be automatic and produce a proof certificate that can be easily verified by the mobile device, following the proof-carrying code (PCC) paradigm [Nec97].

In spite of the rich amount of previous work in the field of information flow security, I argue that none of them covers all these challenges exhaustively. In the following, I present a summary of approaches that are most relevant to the application scenario; a more in-depth account on related work is presented in Chapter 7.

The information flow type system by Volpano, Smith, and Irvine [VSI96] provides a sound analysis for a simple imperative WHILE language, but the information flow policy and the security levels of the variables are fixed. Banerjee and Naumann [BN05] extended this system to object-oriented languages with heaps, but also consider static security environments.

The Jif language [Mye99], a superset of Java, includes language facilities to model dynamic security environments, and to write programs that can adapt their behaviour to the security environment. A type system verifies that such programs act securely for any given environment. Although there is not yet a soundness result for the entire type system of Jif, the specific aspect of verifying programs that inspect dynamic security environments has been formalized by Lantian Zheng and Andrew Myers in a type system for a functional language called λ_{DSec} [ZM07]. However, neither Jif nor λ_{DSec} include the verification or certification of unstructured bytecode programs.

In the imperative language RTI [BWW08] and the functional language of the Paralocks framework [BS10], data are associated with roles (sets of users), which may be updated and queried programmatically to ensure that data flow securely. However, the focus of these works lies on the static tracking of updates to a previously known security environment. Also, these languages include rather substantial extensions to a standard syntax and semantics to account for role or locks management, and do not treat bytecode programs either.

The type-based analysis of information flows in unstructured bytecode program has been explored in the MOBIUS project [Bar+06]. In particular, Barthe et al. have developed a system for a certified compilation of an object-oriented language to bytecode that preserves security types [BPR07; BR05; BRN06]. A similar approach, with a somewhat different treatment of control dependence regions, has been developed by Medel, Compagnoni, and Bonelli [MCB05]. All these systems do not treat security environments that may vary between executions.

Relational bytecode correlations [Ber10] is a calculus to prove generalized noninterference properties that can be instantiated to a number of concrete security policies. The derived judgement essentially encodes for which security environments a given

program is information-flow–secure. In contrast to the requirements of the application scenario, it is not possible to write programs that can adapt their behaviour to the security environment. Also, the programmer lacks an analysis tool for the source code, as the calculus is defined only for the bytecode level.

As the security environment depends on the place of program execution, it seems natural to consider a dynamic form of enforcing the policy, for example by using a runtime monitor that detects malicious flows. Unfortunately, noninterference properties in general cannot be exhaustively covered in a purely dynamic fashion. The reason is that an indirect flow of information may occur already by *not* performing an action; an example provides the second program in Figure 1.1 on page 2, where information leaks even in the case that the “then” branch is not executed. Nevertheless, hybrid approaches have been developed, either by combining the runtime monitoring with a static analysis [SST07], or by first transforming the program into a form that is suitable for runtime monitoring [Vac+04]. As these partly-dynamic techniques deviate from the classic type-based analysis approach, they have not been followed here.

For the information flow analysis presented in this thesis, I build on many of these previous works, and extend and combine existing approaches to tackle all the challenges that arise in dynamic security environments.

1.3 Main Approach

The main idea presented in the thesis is the separation of the static code analysis from different instances of concrete security policies. Figure 1.2 on the next page outlines the overall approach. I extend a standard Java-like language with constructs to inspect the security environment at runtime. The developer can thus create *privacy-aware programs* that can adapt their behaviour to the security environment. A type-based analysis verifies that the code respects any particular security environment, a property called *universal noninterference*. The analysis uses an abstract environment model which represents arbitrary flow policies and security domains of the objects. The program is compiled to a privacy-aware bytecode program and submitted to the app store maintainer, who uses a bytecode analysis tool to verify similarly the universal noninterference property for the bytecode program, using the same abstract model of the security environment. If the verification is successful, the program is certified and distributed to the users. On each particular smartphone, after the certificate has been checked, the program can be executed with the confidence that it observes the effective security environment.

This thesis shows that privacy-aware software provides a basis for a flexible and reliable information flow analysis for mobile code which can address the challenges resulting from diverse and changing environments, unstructured bytecode, and limits imposed by computational capabilities.

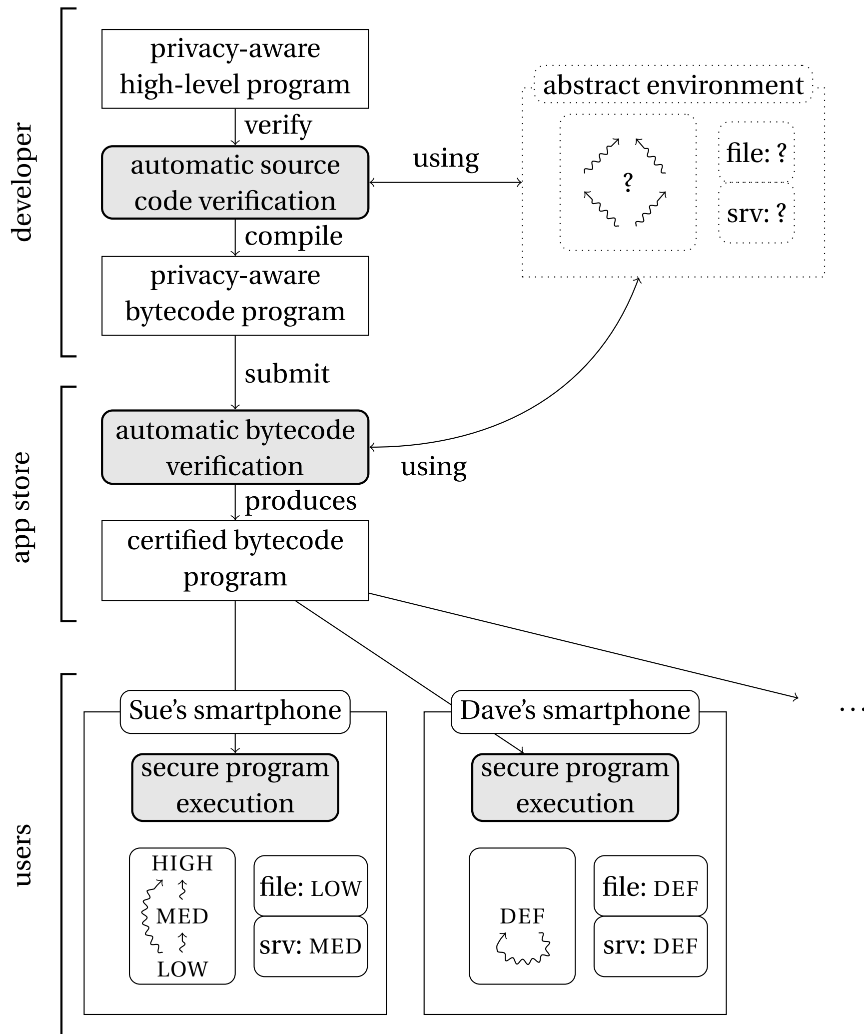

Figure 1.2: Outline of the approach

To concentrate on the security foundations of the approach, I deliberately focus on the sound formalization of the languages and the analysis techniques, leaving out more practice-oriented aspects such as the efficiency of the analysis, the authenticated communication between the involved parties, or the development of a security library to support the development of privacy-aware code. Moreover, to simplify the separation of abstract and concrete security environments, universal noninterference is defined as a generalization of standard end-to-end termination-insensitive noninterference for a transitive flow policy; other, more expressive definitions of security such as notions of declassification are not considered.

In the following, I concretize the approach depicted in this section by giving an overview of the language extension that enables privacy-aware programs, and of the static analysis framework to verify universal noninterference.

1.4 Privacy-Aware Programs

The core idea I propose in this thesis is the introduction of language support for querying or reflecting the domains of objects and the flow policy, such that potentially insecure operations can be guarded with an appropriate flow test. By inspecting the security environment, the programmer can write applications in a way such that they can be safely executed in any security environment.

For this purpose, I provide a standard Java-like object-oriented language called DSD (for "Dynamic Security Domains"). It features a light-weight extension in form of security domains as first-class values and a flow operator to test valid flows with respect to the effective security policy.

The language features a very restricted form of dependent types for objects: each object has a special field f_δ that contains a security domain value. In class type declarations, the field f_δ can be used as the *symbolic* security domain of other fields of the same class. This enables the modelling of data resources on the client in the language via a security API with dynamic domains.

An example of such an API is given by the pseudo-code in Figure 1.3 on the following page. The interface `System` provides facilities to access files on the client, and to open connections to servers. Both data resources are handled abstractly with the `Buffer` interface, which provides read and write operations. The interface is annotated with security domains. In particular, it features a special field f_δ, whose value encodes the runtime security domain of the buffer contents. Therefore, the read operation returns a string of the domain f_δ, that is, of the security domain that the concrete buffer has at runtime. Likewise, the write method accepts only strings that have the f_δ domain. (More precisely, the passed strings must be shown to be exactly as or less confidential than f_δ.)

```
interface System {
  method openFile(name: String⊥) : Buffer⊥;
  method openConn(url: String⊥) : Buffer⊥;
  ...
}

interface Buffer {
  field  fδ : Domain⊥;
  method read() : String_fδ;
  method write(s: String_fδ) : void;
}
```

Figure 1.3: Security API

```
method sendFile(sys: System, name: String) : void {
  file := sys.openFile(name);
  srv := sys.openConn("http://cloud.example.com");
  if (file.fδ ⊑ srv.fδ) then
    srv.write(file.read());
  else
    showErrorMessage();
}
```

Figure 1.4: Example program

Assume that the client has defined two security domains, LOW and HIGH, with a policy that permits flows from LOW to HIGH, but not vice versa. The user-defined confidentiality levels of the existing data resources on the client can be modelled by creating file and server buffers with the f_δ field set to the respecting security domain. For example, if the user assigns the domain LOW to the file "notes.txt", then `openFile("notes.txt")` would return a `Buffer` object with the value of f_δ set to LOW.

Now consider a (simplified) application with the method `sendFile` shown in Figure 1.4 that opens a file specified by the argument `name` and sends it to the server "http://cloud.example.com". The method uses the methods of the `System` class to obtain handles to the file and the server connection buffer. As transferring data from the file to the server induces a flow of information, the write statement is guarded by a test whether the domain of the file (stored in `file.f`$_\delta$) is lower or equal to the domain of the server (stored in `srv.f`$_\delta$) with respect to the effective security policy. If this is not

the case, the method may do something else; here, it presents an error message to the user.

However we label the file or server objects by means of the f_δ fields, the method is thus secure for *any* domains the objects might have, because the critical method call `srv.write` is only executed if the induced flow is permitted in the client environment.

In fact, the `sendFile` method is secure for any particular flow policy defined by the user, as the $\sqsubseteq$ operator is interpreted dynamically with respect to the policy that is in effect at runtime. This includes Sue's policy with a distinct third security domain called MED, and Dave's trivial policy that contains only one security level DEF. In summary, the method `sendFile` is secure for all security environments — that is, it is universally noninterferent.

1.5 The Verification Framework

To verify universal noninterference, I present a type-based analysis framework with the corresponding soundness proofs. A schematic overview is given in Figure 1.5 on the following page, where solid arrows represent operations, and dashed arrows represent implications shown by formal proofs.

For the high-level DSD language, I develop a type system given in the style of Volpano, Smith, and Irvine [VSI96] and Banerjee and Naumann [BN05]. As the domains are not statically available, the analysis is performed by collecting symbolic information about the domain fields and variables. For instance, the expression $\texttt{file.f}_\delta \sqsubseteq \texttt{srv.f}_\delta$ evaluates to true in the "then" branch of the `sendFile` method. The system aims to derive as much as possible from such information by employing a technique similar to Hoare logic [Hoa69], and verifies that the flow tests in the program are sufficient. A soundness proof shows that a well-typed program is indeed universally noninterferent, and thus acts securely in any particular client environment.

In the same way, a JVM-like bytecode language is extended with dynamic security domains. To facilitate the soundness proof, the type system is not given for the bytecode language directly. Instead, the code is translated into a *stackless intermediate representation* (IR) – a partially disassembled version of the program – and the type system and soundness result is given for the IR language. It is shown that the translation to IR preserves the semantics of the program, hence the noninterference property shown for IR code also holds for the original bytecode.

The high-level type system provides an analysis that can detect information flow violations already on the source code level, whereas a low-level type derivation can be used as a formal certificate for the deployed bytecode. To connect the high-level and the bytecode language, I define a formal compilation function, and show that the compilation preserves the typability relation: if the original high-level program is well-typed, then the compiled bytecode program is well-typed, too (or more precisely,

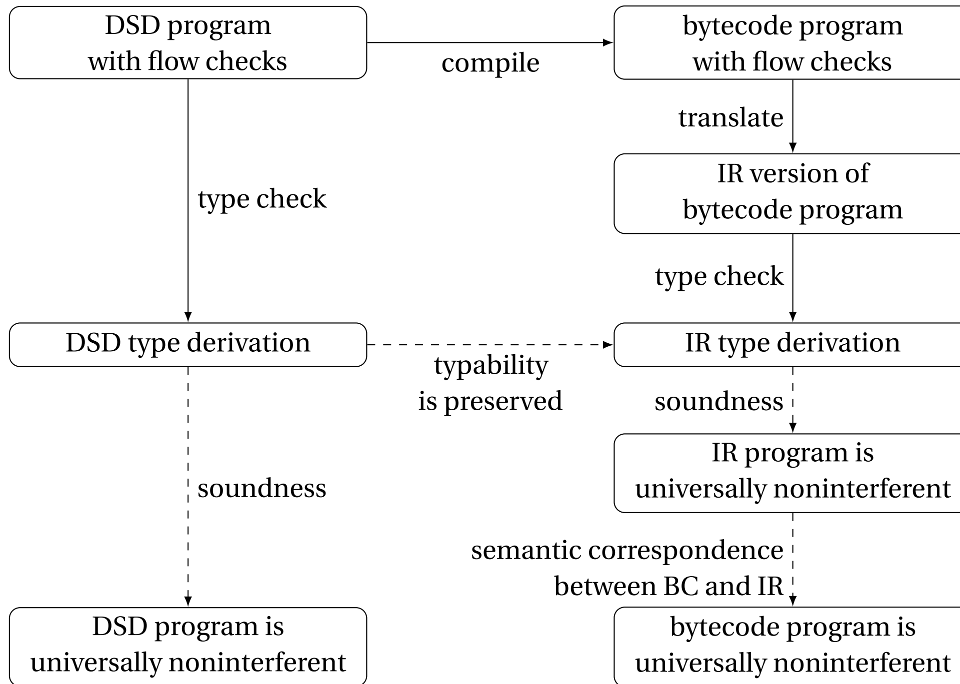

Figure 1.5: Overview of the verification framework

its IR version). Thus, if the software developer got a positive result from the source code type checker, then she can be sure that the program will be accepted by the bytecode analysis performed by the application distribution platform maintainer.

The rigorous formalization of the soundness proof establishes a reliable security guarantee for the end user. Additionally, it also enables the separation of the bytecode analysis from the execution of the program using the proof-carrying code technique [Nec97], which has been shown to be a practical way to shift the task of providing the proof to the code producer (here the app store), such that the code consumer (the smartphone) only needs to verify the proof. Once a static analysis has been performed by the distribution platform maintainer, the program can be certified, and the proof of noninterference can be distributed along with the code. On the device, the system only needs to verify that the certificate is correct. In the framework, the proof certificate is simply the type derivation, and the proof verifier is a type checker. As the soundness result shows that typability implies universal noninterference, it follows that the information flow property guaranteed by the type derivation is compatible with the flow policy set by the user on the client device.

1.6 Synopsis

The thesis proceeds as follows. Chapter 2 presents the high-level language DSD, and defines the universal noninterference property. In Chapter 3, the type system is presented, the concept of labels is introduced, and a soundness result is given. Chapter 4 presents the bytecode language and a corresponding universal noninterference property, and shows how DSD code is compiled to bytecode. The proof system for the bytecode level is more complicated due to the unstructured nature of bytecode; therefore, Chapter 5 introduces an intermediate representation IR, presents a transformation from bytecode that preserves the semantics, and gives an IR type system that proves noninterference for the corresponding bytecode. Also, it is shown that the bytecode compilation preserves the typability relation. Chapter 6 presents algorithmic versions of the type system, and provides details on the implementation of the verification framework. An overview of related work in the field is given in Chapter 7 before the thesis concludes with a discussion and an outlook. The formal proofs of correctness are given in Appendices A to D.

2 Universal Noninterference for a Java-Like Language

As outlined in the introduction, I present an object-oriented, imperative, Java-like high-level language called DSD (for "Dynamic Security Domains"). The language enables the creation of universally noninterferent programs that can be safely executed in arbitrary security environments at runtime. This chapter defines the language and its semantics, introduces the security types used in the language, and formally defines the universal noninterference property.

2.1 The DSD Language

The DSD language is based on and extends a proper subset of the Java language. This design decision aims to reach a middle ground between different languages: A functional core language such as Featherweight Java [IPW99] or variants of it [HJ06; BGH10] is conceptually further away from the imperative Java language, making it harder to transfer the DSD-specific extensions later to the full Java language. On the other hand, taking the full Java language right from the start would distract from the information flow security ideas presented here.

DSD extends the subset of Java with security domain values and operators. The design goal was to extend a well-known existing language as little as possible. Only a few extensions to the syntax and the semantics are required. This, again, makes it easier to extend the actual Java language later with dynamic security domains.

2.1.1 Syntax

The syntax of DSD relies on several disjoint sets of identifiers and corresponding meta-variables:

$$\begin{array}{rlll} \text{numbers:} & n & \in & \mathbb{N} \\ \text{variables:} & x & \in & \mathit{Var} \\ \text{fields:} & f & \in & \mathit{Fld} \\ \text{classes:} & C & \in & \mathit{Cls} \\ \text{methods:} & m & \in & \mathit{Mtd} \end{array}$$

The syntax is split into expressions and statements; expressions do not have side-effects.

$$\begin{array}{rcl} e \in \mathit{Exp} & ::= & n \mid x \mid e.f \mid e\,\mathbf{op}\,e \mid \top \mid \bot \mid e \sqcup e \mid e \sqsubseteq e \\ S \in \mathit{Stmt} & ::= & S\,;S \mid \mathbf{if}\ e\ \mathbf{then}\ S\ \mathbf{else}\ S \mid \mathbf{while}\ e\ \mathbf{do}\ S \mid \\ & & \mathbf{skip} \mid x := e \mid e.f := e \mid x := \mathbf{new}\ C(\overline{e}) \mid x := e.m(\overline{e}) \end{array}$$

DSD statements form an imperative WHILE language with objects and integers, extended with a few additional expressions to refer to security domains. The syntax does not depend on the concrete security domains and the policy, which are understood to be given by the environment in which the program is executed. Instead, the programmer can use the constants $\top$ and $\bot$ refer to the top and bottom element of the security policy, $\sqcup$ for the least upper bound of two security domains, and $\sqsubseteq$ for the domain order. (The security policy is modelled as a lattice over security domains, as will be explained in Section 2.1.2 on page 18.)

The language may include arbitrary side-effect free operators, such as $+$, $*$, $=$, $<$, etc. From an information flow point of view, there are no differences among these operations, hence they are treated in the syntax uniformly as e **op** e.

The notation $\overline{e}$ represents an ordered sequence of argument expressions for the called method m. In the following, I generally use the notation $\overline{x}$ for a sequence of items x. The set of sequences over elements of X is denoted as X^*. The power set (set of all subsets) of X is written $\mathscr{P}(X)$.

DSD programs To abstract away from a concrete class declaration syntax, we assume a DSD program to be given as follows:

Definition 2.1 *A DSD program P_{DSD} is a tuple* $(\prec, \mathsf{fields}, \mathsf{methods}, \mathsf{margs}, \mathsf{mbody})$ *where*

- $\prec \in \mathscr{P}(\mathit{Cls} \times \mathit{Cls})$ *is the subclass relation.*
- $\mathsf{fields} : \mathit{Cls} \to \mathit{Fld}^*$ *and* $\mathsf{methods} : \mathit{Cls} \to \mathit{Mtd}^*$ *assign to each class the identifiers of the fields and methods they contain.*

- margs : *$Mtd \to Var^*$ is a function that describes the names of the formal arguments of each method m.*

- mbody : *$Mtd \rightharpoonup Stmt$ is a partial function that assigns an* implementation *to method identifiers.*

DSD is a language with nominal subtyping: $D \prec C$ means D is an immediate subclass of C. I write $\preceq$ for the reflexive and transitive hull of $\prec$. The functions fields and methods describe the list of member fields and methods for each class C. The function margs gives the formal arguments of the method. A method implementation mbody gives for a method identifier its implementation, which is a DSD statement that forms the method's body. Any method that is not assigned a statement is assumed to be an external method (see below). I make the simplifying assumption that a method cannot be overridden in subclasses. As a consequence, all method calls in the language are essentially static. Also, by design, the sequence of formal arguments for each method identifier is fixed. Also, the signatures (method types) used later are fixed for each method identifier — in other words, I do not consider the full subclass system of Java. With all these simplifications, one can later define a type system that does not include class information at all; in other words, the class information is completely optional for the information flow domains covered here.

To prevent inconsistencies, a number of well-formedness conditions are imposed on the relations and functions:

- The relation $\prec$ is well-formed if it is a tree successor relation; multiple inheritance is not allowed.

- For all classes C, the first field in the list of fields fields(C) must be f_δ. This field will be available as the type of other fields.

- For all methods m, the first formal argument variable in the list of local variables margs(C) must be x_δ. This variable can be used as the type of other variables.

- Classes inherit fields and methods from their superclasses: For all classes C and D such that $D \preceq C$, it must be fields(C) $\subseteq$ fields(D) and methods(C) $\subseteq$ methods(D).

- For all methods m, the only variables that may occur freely in the body of the method must be the ones from margs(m), or the special variable *this* that holds the object reference, or the special variable *ret* that holds the return value.

In the following, I assume a fixed DSD program P_{DSD} whose components are all well-formed.

External methods The function mbody does not need to specify an implementation for each method. It is also possible to define *external methods*, which are methods whose semantics is provided directly. This is useful for methods that cannot be implemented in the DSD language, and thus provides a flexible interface to other parts of the software system. The security API shown in the introduction (see Figure 1.3 on page 10), for example, contains methods that could be implemented externally. These methods also have to be equipped with type signatures, though these signatures are trusted by the type system, as the implementation of external methods is not known.

Remarks on extending the program model A more realistic model of Java programs could be easily added: Overriding of methods can be treated by assigning different implementations depending on the class. To this end, dynamic class information needs to be included in the mbody function. This also enables two methods of unrelated classes to have the same name. The language semantics needs to be extended with dynamic method calls, such that at the site of the call, the actual class of an object is retrieved to determine which implementation is to be called. All implementations would need to be typable according to the given method type. To model the full Java subclass system, one could additionally allow the use of improved method types for subclass implementations, which requires the use of class types in the type system.

2.1.2 Security Policy and Domain Lattice

The information flow policy on the client is specified by a *domain lattice*, which is a join semi-lattice:

$$\text{domain lattice:} \quad \diamond \quad = \quad (Dom^\diamond, \leq^\diamond, \vee^\diamond, k_\top^\diamond, k_\perp^\diamond)$$

The set $Dom^\diamond$ is the set of security domains defined on the client, ranged over by the meta-variable k. The $\leq^\diamond$ relation is a partial order on $Dom^\diamond$, which means it is transitive, reflexive, and antisymmetric. Given two security domains $k_1, k_2 \in Dom^\diamond$, the order $k_1 \leq^\diamond k_2$ expresses that information may flow from k_1 to k_2. The $\vee^\diamond$ operator computes the least upper bound of two domains — that is, $k_1 \vee^\diamond k_2$ is the lowest domain that is at least as confidential as both k_1 and k_2. Finally $k_\top^\diamond, k_\perp^\diamond \in Dom^\diamond$ are the bounds of the lattice and specify the top-most and bottom-most domain, respectively.

For example, Sue's security policy, shown in the lower left of Figure 1.2 on page 8 in the introduction, would be modelled as a lattice $\diamond$ with

$$\begin{array}{rcl} Dom^\diamond & = & \{\text{LOW}, \text{MED}, \text{HIGH}\} \\ k_\top^\diamond & = & \text{HIGH} \\ k_\perp^\diamond & = & \text{LOW} \\ \leq^\diamond & = & \{(\text{LOW}, \text{LOW}), (\text{LOW}, \text{MED}), (\text{LOW}, \text{HIGH}), \\ & & \quad (\text{MED}, \text{MED}), (\text{MED}, \text{HIGH}), (\text{HIGH}, \text{HIGH})\} \end{array}$$

and a corresponding least upper bound operator $\vee^\diamond$.

$$
\begin{array}{rrcl}
\text{numbers:} & n & \in & \mathbb{N} \\
\text{domains:} & k & \in & \mathit{Dom}^{\diamond} \\
\text{references:} & r & \in & \mathit{Loc} \\
\text{values:} & v & \in & \mathit{Val} = \mathbb{N} \cup \mathit{Dom}^{\diamond} \cup \mathit{Loc} \cup \{\mathit{null}\} \\
\\
\text{states:} & \sigma \in \mathit{State} & = & \mathit{Store} \times \mathit{Heap} \\
\text{stores:} & s \in \mathit{Store} & = & \mathit{Var} \rightharpoonup \mathit{Val} \\
\text{heaps:} & h \in \mathit{Heap} & = & \mathit{Loc} \rightharpoonup \mathit{Obj} \\
\text{objects:} & (C, F) \in \mathit{Obj} & = & \mathit{Cls} \times (\mathit{Fld} \rightharpoonup \mathit{Val})
\end{array}
$$

Figure 2.1: State model of DSD

2.1.3 Semantics

I use a mostly standard object-oriented semantics for DSD programs, with program states that consist of a local variable store and a heap (memory) containing the objects.

The main extensions are the security domain values and operators specified by the execution-dependent domain lattice $\diamond$. The semantics of abstract domain expressions in the DSD language is defined with respect to this lattice.

Program states Figure 2.1 shows the definition of program states. $\mathbb{N}$ is the set of natural numbers. $\mathit{Dom}^{\diamond}$ is the set of security domains as defined by the domain lattice $\diamond$. *Loc* is an infinite set of arbitrary references (also called memory locations in the following). *Val* is the set of values, which consist of numbers, domains, references, and the special *null* constant. All other sets are finite sets of abstract identifiers.

A program state is a pair consisting of a store and a heap. A store maps local variables to values, whereas a heap maps locations to objects. An object contains its dynamic class information, and a mapping from its fields to values.

A state $\sigma = (s, h)$ is *well-formed* if the following conditions hold:

1. Every reference r that is the value of a field or a variable is in $\mathsf{dom}(h)$.
2. Every object contains the fields defined by the program syntax: for all objects (C, F) on the heap h, $\mathsf{dom}(F) = \mathsf{fields}(C)$.
3. The variable store contains the domain variable x_δ, that is, $x_\delta \in \mathsf{dom}(s)$.

Notational conventions To simplify the presentation, I use the following notation for extensions and updates of states and heaps: $h \cup [r \mapsto o]$ extends the function h by a

new mapping from r to o; it is undefined if $r \in \mathsf{dom}(h)$. $s[x \mapsto v]$ updates s such that $s(x) = v$; it is undefined if $x \not\in \mathsf{dom}(s)$. Moreover, I define a shorthand notation for field accesses that ignores the class information of the object: for an object $o = (C, F) \in \mathit{Obj}$, I write $o(f)$ for $F(f)$, and $o[f \mapsto v]$ for $(C, F[f \mapsto v])$. Also, I use a point-wise extension of pair mappings to sequences: $[\overline{x} \mapsto \overline{v}]$ is the function that maps each x_i to v_i; it is undefined if they do not have the same length.

Expression and statement semantics For expressions, the denotational semantics $[\![e]\!]^{\diamond}_{\sigma}$ defines the value an expression e in the program state σ given the domain lattice $\diamond$. The semantics is defined as shown in Figure 2.2 on the next page. The semantics of domain-related expressions (shown on the right in the figure) depends on a given domain lattice $\diamond$, which models the effective security policy at runtime. For example, the semantics of the $\sqcup$ operator in the language is defined in terms of the $\vee^{\diamond}$ operator.

For statements, a big-step operational semantics $\sigma_1 \xrightarrow{S}{}^{\diamond} \sigma_2$ expresses that if a statement S is executed in state σ_1 with the security policy given as the domain lattice $\diamond$, then execution terminates in state σ_2. The big-step relation is defined inductively by the rules shown in Figure 2.3 on the facing page.

Except for object creation and method invocation, the semantic is completely standard. In the absence of boolean values, branching statements test for integer values being zero or not.

The construct $x := \mathbf{new}\ C(\overline{e})$ creates an object of class C, and initializes its fields $\mathsf{fields}(C)$ with the values provided by the expressions $\overline{e}$. Because the first field of an object is always f_δ, the first argument of the constructor must evaluate to a domain, with which f_δ is initialized.

For method calls, a new local store is created for the execution of the method, where local variables are bound to the value of the actual arguments, and the special variable *this* is bound to the reference to the object itself. A special return variable *ret* is used in order to avoid special syntax and semantics for passing a return value back to the caller. It is initialized with a default value defval, which we assume to be defined as *null* here.[1] Apart from $\mathsf{margs}(m)$, *this* and *ret*, there are no other local variables in a method m, although they could be easily added to the semantics. For the actual execution of the method body, I define a shorthand notation:

$$(s_1, h_1) \overset{m}{\Longrightarrow}{}^{\diamond} (s_2, h_2) \quad \text{if and only if} \quad (s_1, h_1) \xrightarrow{\mathsf{mbody}(m)}{}^{\diamond} (s_2, h_2).$$

Note that the semantics of statements is undefined for certain initial states, for example if the conditional statement branches over a domain value, or if a *null* reference is dereferenced. Operationally, the execution is simply assumed to get stuck at these points.

[1] In a language with data type information such as `Integer` or `Domain`, in contrast, defval could stand for the default value of the data type of *ret* of the respective method.

$$\llbracket n \rrbracket^{\diamond}_{s,h} = n \qquad \llbracket \top \rrbracket^{\diamond}_{s,h} = k^{\diamond}_{\top}$$
$$\llbracket x \rrbracket^{\diamond}_{s,h} = s(x) \qquad \llbracket \bot \rrbracket^{\diamond}_{s,h} = k^{\diamond}_{\bot}$$
$$\llbracket e.f \rrbracket^{\diamond}_{s,h} = h(\llbracket e \rrbracket^{\diamond}_{s,h})(f) \qquad \llbracket e_1 \sqcup e_2 \rrbracket^{\diamond}_{s,h} = \llbracket e_1 \rrbracket^{\diamond}_{s,h} \vee^{\diamond} \llbracket e_2 \rrbracket^{\diamond}_{s,h}$$
$$\llbracket e_1 \,\mathbf{op}\, e_2 \rrbracket^{\diamond}_{s,h} = \llbracket e_1 \rrbracket^{\diamond}_{s,h} \,\mathsf{op}\, \llbracket e_2 \rrbracket^{\diamond}_{s,h} \qquad \llbracket e_1 \sqsubseteq e_2 \rrbracket^{\diamond}_{s,h} = \begin{cases} 1 & \text{if } \llbracket e_1 \rrbracket^{\diamond}_{s,h} \leq^{\diamond} \llbracket e_2 \rrbracket^{\diamond}_{s,h} \\ 0 & \text{else} \end{cases}$$

Figure 2.2: Semantics of expressions

$$\textsc{Skip}\ \frac{}{\sigma_1 \xrightarrow{\mathbf{skip}}{}^{\diamond} \sigma_1} \qquad \textsc{Seq}\ \frac{\sigma_1 \xrightarrow{S_1}{}^{\diamond} \sigma_2 \qquad \sigma_2 \xrightarrow{S_2}{}^{\diamond} \sigma_3}{\sigma_1 \xrightarrow{S_1\,;\,S_2}{}^{\diamond} \sigma_3} \qquad \textsc{If-T}\ \frac{\llbracket e \rrbracket^{\diamond}_{\sigma_1} > 0 \qquad \sigma_1 \xrightarrow{S_1}{}^{\diamond} \sigma_2}{\sigma_1 \xrightarrow{\mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2}{}^{\diamond} \sigma_2}$$

$$\textsc{If-F}\ \frac{\llbracket e \rrbracket^{\diamond}_{\sigma_1} = 0 \qquad \sigma_1 \xrightarrow{S_2}{}^{\diamond} \sigma_2}{\sigma_1 \xrightarrow{\mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2}{}^{\diamond} \sigma_2} \qquad \textsc{While-T}\ \frac{\begin{array}{c}\llbracket e \rrbracket^{\diamond}_{\sigma_1} > 0 \qquad \sigma_1 \xrightarrow{S}{}^{\diamond} \sigma_2 \\ \sigma_2 \xrightarrow{\mathbf{while}\ e\ \mathbf{do}\ S}{}^{\diamond} \sigma_3\end{array}}{\sigma_1 \xrightarrow{\mathbf{while}\ e\ \mathbf{do}\ S}{}^{\diamond} \sigma_3}$$

$$\textsc{While-F}\ \frac{\llbracket e \rrbracket^{\diamond}_{\sigma_1} = 0}{\sigma_1 \xrightarrow{\mathbf{while}\ e\ \mathbf{do}\ S}{}^{\diamond} \sigma_1} \qquad \textsc{Assign}\ \frac{s_2 = s_1[x \mapsto \llbracket e \rrbracket^{\diamond}_{s_1,h_1}]}{(s_1,h_1) \xrightarrow{x:=e}{}^{\diamond} (s_2,h_1)}$$

$$\textsc{PutField}\ \frac{\llbracket e_1 \rrbracket^{\diamond}_{s_1,h_1} = r \qquad h_1(r)[f \mapsto \llbracket e_2 \rrbracket^{\diamond}_{s_1,h_1}] = o \qquad h_2 = h_1[r \mapsto o]}{(s_1,h_1) \xrightarrow{e_1.f:=e_2}{}^{\diamond} (s_1,h_2)}$$

$$\textsc{New}\ \frac{r \notin \mathsf{dom}(h_1) \qquad h_2 = h_1 \cup [r \mapsto (C, [\mathsf{fields}(C) \mapsto \llbracket \overline{e} \rrbracket^{\diamond}_{s_1,h_1}])] \qquad s_2 = s_1[x \mapsto r]}{(s_1,h_1) \xrightarrow{x:=\mathbf{new}\ C(\overline{e})}{}^{\diamond} (s_2,h_2)}$$

$$\textsc{Call}\ \frac{\begin{array}{c}\llbracket e \rrbracket^{\diamond}_{s_1,h_1} = r \\ s'_1 = [\mathit{this} \mapsto r] \cup [\mathsf{margs}(m) \mapsto \llbracket \overline{e} \rrbracket^{\diamond}_{s_1,h_1}] \cup [\mathit{ret} \mapsto \mathsf{defval}] \\ (s'_1,h_1) \overset{m}{\Longrightarrow}{}^{\diamond} (s'_2,h_2) \qquad s_2 = s_1[x \mapsto s'_2(\mathit{ret})]\end{array}}{(s_1,h_1) \xrightarrow{x:=e.m(\overline{e})}{}^{\diamond} (s_2,h_2)}$$

Figure 2.3: Semantics of statements

$$
\begin{aligned}
\prec &= \emptyset \\
\mathsf{fields}(\texttt{Buffer}) &= [f_\delta] & \mathsf{fields}(\texttt{Main}) &= [f_\delta] \\
\mathsf{methods}(\texttt{Buffer}) &= [\texttt{read}, \texttt{write}] & \mathsf{methods}(\texttt{Main}) &= [\texttt{sendFile}] \\
\mathsf{margs}(\texttt{read}) &= [x_\delta] & \mathsf{margs}(\texttt{sendFile}) &= [x_\delta, \texttt{file}, \\
\mathsf{margs}(\texttt{write}) &= [x_\delta, \texttt{s}] & & \quad \texttt{srv}, \texttt{tmp}]
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{mbody}(\texttt{sendFile}) = \;& \textbf{if } \texttt{file}.f_\delta \sqsubseteq \texttt{srv}.f_\delta \textbf{ then} \\
& \quad \texttt{tmp} := \texttt{file.read}(\texttt{file}.f_\delta); \\
& \quad \texttt{ret} := \texttt{srv.write}(\texttt{file}.f_\delta, \texttt{tmp}) \\
& \textbf{else} \\
& \quad \texttt{ret} := 0;
\end{aligned}
$$

Figure 2.4: The example formalized as a DSD program

2.2 An Example Program

This section shows how the example program from the introduction chapter (see Figures 1.3 and 1.4 on page 10) is formally defined as a DSD program. I only specify the syntax here; the specification of the type information and a possible type derivation for the example program are given in the next chapter.

Let us assume that the method `sendFile` is part of a class called `Main`. To point out the essential aspects, I make some simplifications to the original example program. I omit the `System` class, and assume that the `file` and `srv` objects are already passed initialized to the `sendFile` method. Thus, the arguments `sys` and `name` for `sendFile` are not needed. Also, if the label test fails, this is communicated via the *ret* variable.

In the DSD language, the example program can be defined as a quintuple of the form $P_{\mathrm{DSD}} = (\prec, \mathsf{fields}, \mathsf{methods}, \mathsf{margs}, \mathsf{mbody})$, where the components are defined as shown in Figure 2.4. Only `sendFile` is implemented; the methods `read` and `write` are external — that is, we assume a given semantics for them.

Due to restrictions in the DSD syntax, the implementation looks a bit different here:

- Each class contains an f_δ field, and each method has an x_δ argument.
- Method calls cannot be used as expressions, thus we need to store the result of `read` in a temporary variable `tmp` first before passing it to `write`. As DSD does not allow the declaration of local variables in the method body, `tmp` has to be an argument of the method `sendFile`. It is a "dummy" argument, since any information passed in `tmp` to the method is ignored. Also, changes to `tmp` (and all other local variables apart from *ret*) have no effect on the caller.

Finally, the methods `read` and `write` are called with the expected domain of the read data. This is required due to restrictions in the type system, and shall be discussed in Section 3.6 on page 48, where a type derivation for the example program is presented.

2.3 Universal Noninterference

I now show how the security types for variables and fields can make use of dynamic security domains. Based on this, I define universal noninterference for DSD programs, which expresses information flow security for arbitrary security environments.

2.3.1 Type Environments

As I want to concentrate on information flow aspects in the type system, I only consider the confidentiality of values in the program, and ignore their data types completely. The type system does not prevent access to fields that do not exist, or the assignment of a string to an integer, or similar errors, because they are orthogonal to information flow security. It is safe to consider only programs that are well-typed with respect to data types: Should the execution get stuck because of a data type mismatch, the program would still be secure according to the termination-insensitive noninterference property presented later.

Type environments assign symbolic security domains to variables and fields:

$$\Gamma : \mathit{Var} \rightharpoonup \{\top, \bot, x_\delta\}$$
$$\Phi : \mathit{Fld} \rightarrow \{\top, \bot, f_\delta\}$$

A variable typing Γ associates a symbolic security domain to each local variable of the active method body. The meaning of the symbolic domain depends on the given variable store and the lattice. The types $\top$ or $\bot$ refer abstractly to the top-most and bottom-most domain of $\mathit{Dom}^\diamond$, that is, $k^\diamond_\top$ and $k^\diamond_\bot$, respectively. A variable typed with the special symbol x_δ has the domain that is stored in the variable x_δ at runtime. A variable typing Γ is well-formed if $x_\delta \in \mathsf{dom}(\Gamma)$ and $\Gamma(x_\delta) = \bot$.

A field typing Φ associates a type with each field. Again, the field types $\top$ and $\bot$ refer abstractly to $k^\diamond_\top$ and $k^\diamond_\bot$. A field typed with f_δ has the domain that is stored in the field f_δ *of the same object*. In contrast to the variable typing that only applies to the local variables in a specific variable store, the field typing Φ is a total function that defines the types globally for all fields, including f_δ. A field typing Φ is well-formed if $\Phi(f_\delta) = \bot$. From now on, I assume a fixed well-formed field typing Φ and leave it implicit.

Note that the well-formedness conditions require that the symbolically referenced domain variable x_δ and domain field f_δ themselves get the type $\bot$. Other types for $\Gamma(x_\delta)$ and $\Phi(f_\delta)$ shall be discussed later.

Remarks on extending the type information A type is assigned to each field *identifier*, independent of the class it occurs in. In other words, if two objects that are unrelated in the class hierarchy both contain a field named f, then both fields get the same security type. As mentioned in the remarks for the DSD syntax, a more realistic extension is to have Φ define a type per field *per class*. This involves additional well-formedness conditions to soundly handle subtyping and inheritance, and also requires the tracking of class information in the type system. Going even further, one could also distinguish “get” and “set” types for each field. All this can be achieved by *refining* class types in the standard Java type system with the security types presented here. Examples for refinement type systems for Java-like languages can be found in our previous work on region and string type systems [BGH10; GHL11]. To keep the type system simple, I refrain from including class information here.

2.3.2 Type Interpretation

The meaning of types depends on the concrete program state, and the effective domain lattice $\diamond$. More precisely, types of variables and of fields are interpreted with respect to given stores s and field valuations F, respectively:

$$\begin{aligned} \langle\top\rangle_s^\diamond &= k_\top^\diamond & \qquad \langle\top\rangle_F^\diamond &= k_\top^\diamond \\ \langle\bot\rangle_s^\diamond &= k_\bot^\diamond & \langle\bot\rangle_F^\diamond &= k_\bot^\diamond \\ \langle x_\delta\rangle_s^\diamond &= s(x_\delta) & \langle f_\delta\rangle_F^\diamond &= F(f_\delta) \end{aligned}$$

The interpretation is well-defined for well-formed states, as every store includes x_δ and every object has a field f_δ. As x_δ and f_δ may contain arbitrary domains k from the domain lattice $Dom^\diamond$, it is thus possible to assign these arbitrary security domains to variables and fields by using x_δ or f_δ as the symbolic type. Variables and fields can thus be given a dynamic security domain. Note that dynamic security domains are completely encapsulated, as the field type f_δ refers to the value of the f_δ field in the same object, and the variable x_δ refers to the value of the x_δ variable in the same local variable store.

The definition of type interpretations provides the basis for the visibility of variables and fields in a domain lattice $\diamond$ at a security domain k.

Definition 2.2 *Let $\diamond$ be a domain lattice, and let $k \in Dom^\diamond$ be a security domain. If $\langle\Gamma(x)\rangle_s^\diamond \leq^\diamond k$, the variable x of the store s is* visible at k in $\diamond$. *Similarly, if $\langle\Phi(f)\rangle_F^\diamond \leq^\diamond k$, the field f of the field valuation F is* visible at k in $\diamond$.

In other words, a variable or field is visible at k in $\diamond$ if its dynamic security domain is lower than or equal to k with respect to $\diamond$. As the lattice $\diamond$ models the security policy, this means that information may flow from that field or variable to the security domain k.

2.3.3 Equivalence of States

In preparation for the definition of noninterference, I now define when two states are equivalent with respect to a domain lattice $\diamond$ and a security domain $k \in Dom^{\diamond}$.

Indistinguishable values To capture related allocations of different fresh locations in two parallel runs, equivalence is parametrized by a partial bijection β, following the approach by Banerjee and Naumann [BN05]. Informally, two states are equivalent if the following holds. Locations in public variables must be related by the bijection, and objects with β-related locations must be indistinguishable, which means that locations in their public fields must again be related by the bijection.

As such, a partial bijection is a special form of a heap typing [Pie02], a standard technique used only in the soundness formulation to separate the well-typedness definitions of locations from the actual heap, thereby avoiding the need for a co-inductive definition for well-typed heaps in the presence of cyclic structures. Partial bijections differ in that they "type" pairs of locations, but only the public ones.

Formally, I use partial bijections $\beta, \gamma \subseteq Loc \times Loc$, and write $\beta(r) = r'$ for $(r, r') \in \beta$. *Indistinguishable values* $v \sim_\beta v'$ are defined as follows:

$$\frac{}{n \sim_\beta n} \qquad \frac{}{k \sim_\beta k} \qquad \frac{}{null \sim_\beta null} \qquad \frac{\beta(r) = r'}{r \sim_\beta r'}$$

Two locations r, r' are indistinguishable if $\beta(r) = r'$. Two domains, numbers, or null values are indistinguishable if they are equal.

Equivalent program states For $k \in Dom$, two stores are $\diamond, k$-*equivalent* with respect to a variable typing Γ and a bijection β if all variables of the store that are visible at k in $\diamond$ contain β-indistinguishable values in both stores:

$$\vdash^{\diamond} s \sim_{\beta}^{\Gamma,k} s' \iff \forall x \in \mathsf{dom}(\Gamma).\ \langle \Gamma(x) \rangle_s^{\diamond} \leq^{\diamond} k \wedge \langle \Gamma(x) \rangle_{s'}^{\diamond} \leq^{\diamond} k \ \Rightarrow\ s(x) \sim_\beta s'(x)$$

Two heaps are $\diamond, k$-equivalent if β only relates objects from the heaps, and all objects related by the bijection β are $\diamond, k$-equivalent. Two objects are $\diamond, k$-equivalent if they are of the same class and if all fields of the class visible at k in $\diamond$ contain β-indistinguishable values in both objects.

$$\begin{aligned}
\vdash^{\diamond} h \sim_\beta^k h' \iff\ & \mathsf{dom}(\beta) \subseteq \mathsf{dom}(h) \wedge \mathsf{rng}(\beta) \subseteq \mathsf{dom}(h') \wedge \\
& \forall r \in \mathsf{dom}(\beta).\ \ \vdash^{\diamond} h(r) \sim_\beta^k h(\beta(r)) \\
\vdash^{\diamond} (C, F) \sim_\beta^k (C', F') \iff\ & C = C' \wedge \forall f \in \mathsf{fields}(C). \\
& \quad \left\langle \Phi(f) \right\rangle_F^{\diamond} \leq^{\diamond} k \wedge \left\langle \Phi(f) \right\rangle_{F'}^{\diamond} \leq^{\diamond} k \Rightarrow F(f) \sim_\beta F'(f)
\end{aligned}$$

The bijection β thus links the visible references to objects: all references in visible variables and fields must be related by β, and all objects whose references are related by β must contain indistinguishable values in their visible fields.

Finally, $\diamond, k$-equivalence is extended to program states:

$$\vdash^{\diamond} (s,h) \sim_{\beta}^{\Gamma,k} (s',h') \quad \Longleftrightarrow \quad \vdash^{\diamond} s \sim_{\beta}^{\Gamma,k} s' \wedge \vdash^{\diamond} h \sim_{\beta}^{k} h'$$

Since $\Gamma(x_\delta) = \bot$, the variable x_δ is always visible in two β-equivalent related stores s and s', regardless of the domain k. By definition, it follows $s(x_\delta) \sim_\beta s'(x_\delta)$, and as x_δ contains a domain, the indistinguishability relation by definition stands for equality: $s(x_\delta) = s'(x_\delta)$. From this, it follows that for each variable x, the interpretation of its type $\Gamma(x)$ is the same in both stores: it is either $k_\top^\diamond$, or $k_\bot^\diamond$, or $s(x_\delta) = s'(x_\delta)$. Likewise, we have defined $\Phi(f_\delta) = \bot$, and thus get with a similar argument that all fields of related objects are either visible in both heaps, or invisible in both heaps. This observation is formalized by the following lemma.

Lemma 2.3 *Let Γ be a type environment, $\diamond$ be a domain lattice, $k \in Dom^\diamond$ be a domain, and β be a bijection.*

- *If $\vdash^\diamond s \sim_\beta^{\Gamma,k} s'$, then for all $x \in \mathsf{dom}(\Gamma)$, $\langle\Gamma(x)\rangle_s^\diamond = \langle\Gamma(x)\rangle_{s'}^\diamond$.*
- *If $\vdash^\diamond h \sim_\beta^{k} h'$ and $\beta(r) = r'$, then for all $f \in \mathsf{dom}(h(r))$, $\langle\Phi(f)\rangle_{h(r)}^\diamond = \langle\Phi(f)\rangle_{h'(r')}^\diamond$.*

This lemma is just a special case of a more fundamental property called *meta-label monotonicity* discussed in Chapter 3.

Properties of the state equivalence relation Because of the parametrization with a partial bijection, state equivalence is not an equivalence relation. However, reflexivity, symmetry and transitivity hold by choosing the appropriate bijections; these properties are extensively used in the soundness proofs.

1. Reflexivity: $\vdash^\diamond (s,h) \sim_{\mathsf{id}(h)}^{\Gamma,k} (s,h)$, where $\mathsf{id}(h) = \{(r,r) \mid r \in \mathsf{dom}(h)\}$ is the identity relation on locations in h.
2. Symmetry: $\vdash^\diamond \sigma \sim_\beta^{\Gamma,k} \sigma'$ implies $\vdash^\diamond \sigma' \sim_{\beta^{-1}}^{\Gamma,k} \sigma$, where β^{-1} is the inverse relation — that is, $\beta^{-1} = \{(r',r) \mid (r,r') \in \beta\}$.
3. Transitivity: $\vdash^\diamond \sigma \sim_\beta^{\Gamma,k} \sigma'$ and $\vdash^\diamond \sigma' \sim_{\beta'}^{\Gamma,k} \sigma''$ implies $\vdash^\diamond \sigma \sim_{\beta\circ\beta'}^{\Gamma,k} \sigma''$, where $\beta\circ\beta' = \{(r,r'') \mid (r,r') \in \beta \wedge (r',r'') \in \beta'\}$ is the composition of two bijections.

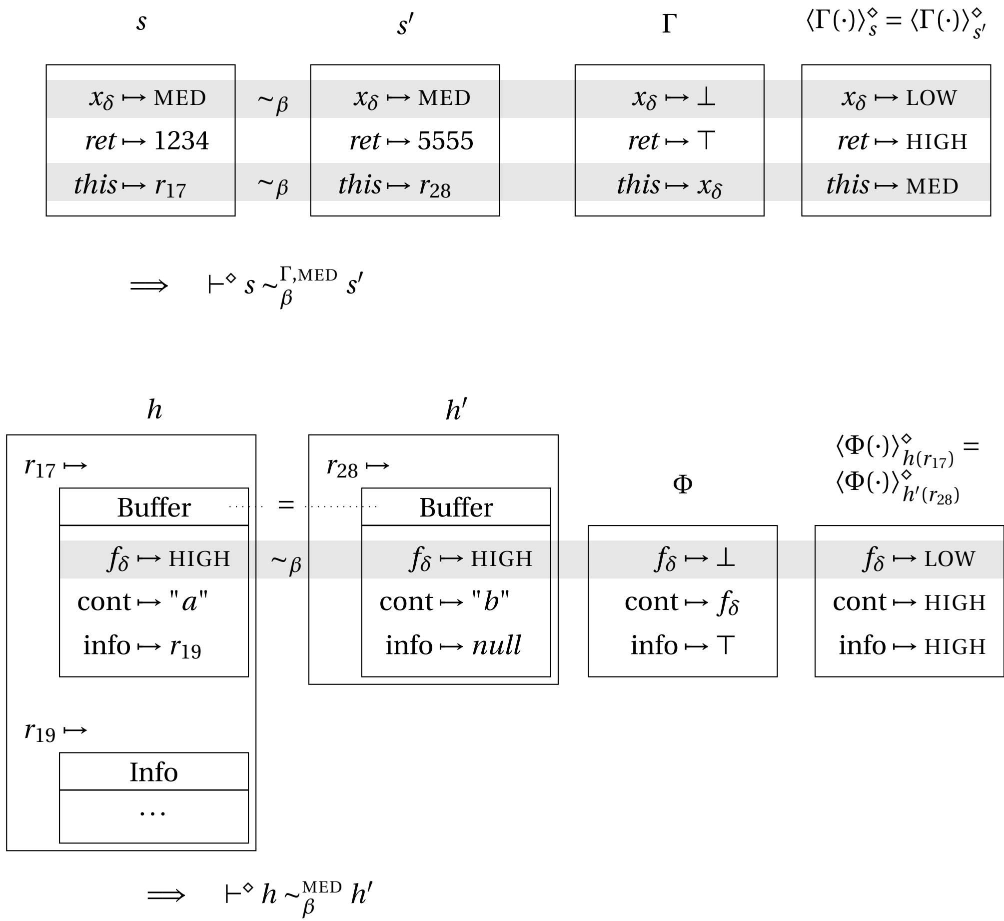

Figure 2.5: Examples for $\diamond$, MED-equivalent stores and heaps

Example Figure 2.5 on the preceding page shows an example for $\diamond, k$-equivalent stores and heaps, where $\diamond$ models Sue's policy as shown on page 18, and with $k = \text{MED}$. For the bijection, we assume $\beta = \{(r_{17}, r_{28})\}$; the type environments Γ and Φ are as shown in the figure.

The upper part displays two equivalent stores s and s'. The variables that are visible at MED in $\diamond$ are marked in grey. As $\perp$ evaluates to the bottom-most domain in the lattice (i.e., LOW), the $\perp$-typed variable x_δ is visible at MED in $\diamond$. Likewise, the interpretation of the x_δ type is the value of x_δ in both states, that is, MED. Therefore, the x_δ-typed variable *this* is visible at MED in $\diamond$. Since the values of all visible variables are related by β, we have by definition $\vdash^\diamond s \sim_\beta^{\Gamma,\text{MED}} s'$.

In the lower part of the figure, two heaps h and h' are shown. We have the equivalence $\vdash^\diamond h \sim_\beta^{\text{MED}} h'$ for the following reasons. All β-related objects (in this case, the objects at $h(r_{17})$ and $h'(r_{28})$) are of the same class, and all their fields that are visible at MED in $\diamond$ have β-related values. In this case, this only applies to f_δ, which is marked in grey. As the interpretation of the types of the other fields is not lower than MED in $\diamond$, there is no requirement on their values. Also, heaps may include objects not related by β, as illustrated by $h(r_{19})$ in the figure. Moreover, we get by definition from the store and heap equivalences that $\vdash^\diamond (s,h) \sim_\beta^{\Gamma,\text{MED}} (s',h')$.

2.3.4 Universal Noninterference

Using the definition of state equivalence, I now define information flow security for a method body as a termination-insensitive noninterference property.

Definition 2.4 *Let P_{DSD} be a DSD program. A method m is* universally noninterferent *with respect to a variable typing Γ if for all domain lattices $\diamond$, for all security domains $k \in Dom^\diamond$, for all states $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$ and for all partial bijections β, if*

- $\vdash^\diamond \sigma_1 \sim_\beta^{\Gamma,k} \sigma_1'$ *and*
- $\sigma_1 \overset{m}{\Longrightarrow}{}^\diamond \sigma_2$ *and*
- $\sigma_1' \overset{m}{\Longrightarrow}{}^\diamond \sigma_2'$,

then there exists a partial bijection $\gamma \supseteq \beta$ such that

- $\vdash^\diamond \sigma_2 \sim_\gamma^{\Gamma,k} \sigma_2'$

All objects reachable by visible variables and fields must be related before and after the execution. The extended bijection γ captures the possibility that new locations may have been allocated and related in both executions. The property actually states a little more: all objects that are related before the execution stay related after the execution, even if they are not reachable by visible variables or fields.

To show universal noninterference, one may need to make certain assumptions about the values of the domain fields and variables that occur in the program states. For this purpose, I will now parametrize the preceding definition with certain state predicates Q and Q' that hold before and after the execution. The notation $\sigma \models^{\diamond} Q$ means σ satisfies Q given the domain lattice $\diamond$. These predicates and the satisfiability relation shall be instantiated in the next section.

Definition 2.5 *Let P_{DSD} be a DSD program. A method m is* (Q, Q')-universally noninterferent *with respect to a variable typing Γ if for all domain lattices $\diamond$, for all security domains $k \in Dom^{\diamond}$, for all states $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$ and for all partial bijections β, if*

- $\sigma_1 \models^{\diamond} Q$ *and* $\sigma_1' \models^{\diamond} Q$ *and*
- $\vdash^{\diamond} \sigma_1 \sim_{\beta}^{\Gamma,k} \sigma_1'$ *and*
- $\sigma_1 \overset{m}{\Longrightarrow}{}^{\diamond} \sigma_2$ *and*
- $\sigma_1' \overset{m}{\Longrightarrow}{}^{\diamond} \sigma_2'$,

then there exists a partial bijection $\gamma \supseteq \beta$ such that

- $\vdash^{\diamond} \sigma_2 \sim_{\gamma}^{\Gamma,k} \sigma_2'$ *and*
- $\sigma_2 \models^{\diamond} Q'$ *and* $\sigma_2' \models^{\diamond} Q'$.

The difference to the preceding definition is that a noninterferent behaviour is only required for initial states that satisfy Q, and additionally the final states must then satisfy Q'. If $\star$ is the universal state predicate with $\sigma \models^{\diamond} \star$ for all states σ, then Definition 2.4 exactly defines $(\star, \star)$-universal noninterference.

The main contributions of the universal noninterference property are the universal quantification of both the lattice $\diamond$ and the dynamic security domains that occur as domain values in the program states. To the best of my knowledge, such a generalization has not been given before.

3
High-Level Type System

The high-level type system is a static analysis which verifies that a given DSD program is universally noninterferent. As the domain lattice that describes effective runtime security policy is not known statically, the type system for the DSD language uses symbolic security domains to reason abstractly over concrete domains in the lattice.

In this chapter, I define the types of the type system as a restricted set of expressions, and explore the meaning and implications of this mild form of dependent types. Afterwards, I present the typing rules for DSD programs, and give a soundness theorem that states that well-typed programs are universally noninterferent. The chapter concludes with a demonstration of a type derivation for the introductory example program.

3.1 Labels: Symbolic Security Domains

The type environments allow the use of domain variables x_δ and domain fields f_δ as security types. Since their values are not known statically, any references to x_δ and f_δ can only appear symbolically in the types of DSD expressions. While a classic information flow type system assigns to each expression a concrete domain $k \in Dom$, the DSD type system instead assigns symbolic security domains which I call *labels*. In the following, I formalize the set of labels, and define an abstract order on them.

3.1.1 Access Paths

To make the type system tractable, references to f_δ fields in the labels are permitted in a normalized form only, denoted by access path expressions:

$$\pi ::= x \mid \pi.f$$

Access path expressions are a subset of expressions as defined in the DSD syntax. The paths exclude pointer operations or other arbitrary expressions that denote memory locations.[1]

3.1.2 Labels

Building on the definition of access paths, one can now define the set of labels that are used as types in the type system. Labels are a subset of expressions:

$$Lab \ni \ell ::= \bot \mid \top \mid \pi.f_\delta \mid x_\delta \mid \ell_1 \sqcup \ell_2$$

The symbols $\top$ and $\bot$ refer to the top-most and bottom-most domain of the effective security policy, that is, to $k^\diamond_\top$ and $k^\diamond_\bot$. The label $\pi.f_\delta$ refers to the f_δ field of the object reachable by an access path $\pi = x.f_1.f_2.\ldots.f_n$, whereas x_δ refers to the x_δ variable. Finally, we define the symbolic least upper bound (join) of two labels $\ell_1 \sqcup \ell_2$, as the values of the labels are not known statically.

The type system assigns to each expression a label. For example, if $\Gamma(y) = x_\delta$ and $\Gamma(z) = \top$, the expression $y + z$ is assigned the label $x_\delta \sqcup \top$. This is in contrast to classic information flow type systems, where the security domains of variables are statically known, such that the least upper bound can be directly computed in the type system.

Here, instead, a label is itself a special expression. When applied to the expression evaluation function $[\![\cdot]\!]^\diamond_\sigma$ in a given program state σ and lattice $\diamond$, it evaluates to a security domain from $Dom^\diamond$. This defines the meaning of labels: If an expression e is typed with a label ℓ, then e depends in a program state σ only on data that is visible at $[\![\ell]\!]^\diamond_\sigma$ in $\diamond$. As labels are special expressions, DSD features a strictly constrained form of dependent types.

3.1.3 Types as Labels

The variable and field types defined by Γ and Φ can be used as labels in the type system as follows:

- The possible variable types $\top$, $\bot$, and x_δ are all valid labels, and can be used as labels.

[1] Admittedly, the DSD language does not include such expressions anyway, so that access paths may as well be made explicit in the syntax of the language. As they are merely needed for the analysis, however, I opted to keep the original syntax flexible for future enhancements.

- The possible field types $\top$, $\bot$, and f_δ are not all valid labels, as f_δ lacks an access path. One can turn a field type into a valid label by employing a *qualified field type* $\Phi^\pi(f)$, which is defined as follows:

$$\Phi^\pi(f) = \begin{cases} \top & \text{if } \Phi(f) = \top \\ \bot & \text{if } \Phi(f) = \bot \\ \pi.f_\delta & \text{if } \Phi(f) = f_\delta \end{cases}$$

There is a correspondence between the type and label evaluation functions.

Lemma 3.1 *Let (s,h) be a state, and $\diamond$ be a domain lattice. It holds $[\![\Gamma(x)]\!]^\diamond_{(s,h)} = \langle\Gamma(x)\rangle^\diamond_s$ and, if $[\![\pi]\!]^\diamond_{(s,h)} = (C,F)$, then $[\![\Phi^\pi(f)]\!]^\diamond_{(s,h)} = \langle\Phi(f)\rangle^\diamond_F$.*

PROOF This follows directly from the definitions of the evaluation functions for types and for labels, and from the definition of $\Phi^\pi(f)$. □

3.2 Ordering Labels

Classic information flow type systems with static domains contain a number of side conditions $k_1 \leq^\diamond k_2$ which require that information may flow from the domain k_1 to the domain k_2 according to the security policy defined by a fixed lattice $\diamond$. The type system presented here, in contrast, has no static information about the values of labels at runtime. It is nevertheless possible to define an abstract order on labels, based on the information from label test conditionals in the program, and by exploiting the lattice structure of the domain lattice $\diamond$.

3.2.1 Information from Label Test Conditionals

Consider a conditional statement of the form **if** $\ell_1 \sqsubseteq \ell_2$ **then** S_1 **else** S_2. For the sub-statement S_1 in the **then** branch, we can assume that an information flow from $[\![\ell_1]\!]^\diamond_\sigma$ to $[\![\ell_2]\!]^\diamond_\sigma$ is permitted by the lattice $\diamond$; otherwise, the branch is not taken during the execution. The typing judgements for statements are therefore parametrized over a set $Q \subseteq Lab \times Lab$ containing label pairs. A pair $(\ell_1, \ell_2) \in Q$ expresses the assumption that a flow from ℓ_1 to ℓ_2 is allowed.

The set Q thus stores abstract information about f_δ fields and x_δ variables at a point of execution. Since Q gives requirements for suitable program states, I also call it the *constraint set.* (Constraint sets are the state predicates that were used in Definition 2.5 on page 29.)

Definition 3.2 *A program state σ* satisfies *a constraint set Q in a domain lattice $\diamond$, written $\sigma \models^\diamond Q$, if for all pairs $(\ell_1, \ell_2) \in Q$ it holds $[\![\ell_1]\!]^\diamond_\sigma \leq^\diamond [\![\ell_2]\!]^\diamond_\sigma$.*

3.2.2 Information from Lattice Properties

Labels evaluate to domains from $Dom^\diamond$, which are ordered as a lattice. For example, it can be inferred that data may always flow from an expression labelled with $\bot$ to an expression labelled with $x_\delta \sqcup y.f_\delta$ for any variable y, as the evaluation of $\bot$ is always the lowest element in $Dom^\diamond$.

One can thus order labels by computing the *lattice closure* of Q. This is done syntactically by the rules in Figure 3.1 on the next page, which define an order and an equality judgement on labels. A judgement of the form $\ell_1 \sqsubseteq_Q \ell_2$ states that information may flow from ℓ_1 to ℓ_2, while $\ell_1 \equiv_Q \ell_2$ states that the two labels, though possibly syntactically different, must refer to the same security domain. Thus, they abstractly describe the operator $\leq^\diamond$ and equality for the domain set $Dom^\diamond$.

The first row of rules translates the label pair information from Q into a partial order that is reflexive, transitive, and antisymmetric. The second row of rules describes how the order relates to least upper bounds, exploiting basic lattice properties: every label is equal or larger than the lowest element $\bot$ and equal or smaller than the highest element $\top$. A least upper bound (join) of two labels is larger than each of the two labels. Also, the least upper bound operation is monotone with respect to the order. The third row of rules mirrors idempotence, commutativity, and associativity of the join operator. Finally, the last two rules link label equality back to the order.

The following theorem states that the rules for label order and equality are sound with respect to their interpretation in satisfying program states.

Theorem 3.3 *Given a constraint set Q, two labels ℓ_1 and ℓ_2 and a state σ satisfying Q.*

1. *If $\ell_1 \sqsubseteq_Q \ell_2$, then $[\![\ell_1]\!]^\diamond_\sigma \leq^\diamond [\![\ell_2]\!]^\diamond_\sigma$.*
2. *If $\ell_1 \equiv_Q \ell_2$, then $[\![\ell_1]\!]^\diamond_\sigma = [\![\ell_2]\!]^\diamond_\sigma$.*

PROOF By induction over the derivation of the label order and equality. □

3.2.3 Constraint Sets as Program Predicates

In the type system, constraint sets are regarded as simple predicates over program states. I now define a syntactic implication relation for constraint sets, and show that this implication is indeed sound with respect to constraint set satisfiability.

Definition 3.4 *A constraint set Q implies another set Q', written $Q \Rightarrow Q'$, if and only if $\forall(\ell_1, \ell_2) \in Q'.\ \ell_1 \sqsubseteq_Q \ell_2$.*

Lemma 3.5 *If $\sigma \models^\diamond Q$ and $Q \Rightarrow Q'$, then $\sigma \models^\diamond Q'$.*

PROOF The lemma follows directly from the definitions. □

$$\frac{(\ell_1, \ell_2) \in Q}{\ell_1 \sqsubseteq_Q \ell_2} \qquad \frac{}{\ell \sqsubseteq_Q \ell} \qquad \frac{\ell_1 \sqsubseteq_Q \ell_2 \qquad \ell_2 \sqsubseteq_Q \ell_3}{\ell_1 \sqsubseteq_Q \ell_3} \qquad \frac{\ell_1 \sqsubseteq_Q \ell_2 \qquad \ell_2 \sqsubseteq_Q \ell_1}{\ell_1 \equiv_Q \ell_2}$$

$$\frac{}{\bot \sqsubseteq_Q \ell} \qquad \frac{}{\ell \sqsubseteq_Q \top} \qquad \frac{}{\ell \sqsubseteq_Q \ell \sqcup \ell'} \qquad \frac{\ell_1 \sqsubseteq_Q \ell_3 \qquad \ell_2 \sqsubseteq_Q \ell_4}{\ell_1 \sqcup \ell_2 \sqsubseteq_Q \ell_3 \sqcup \ell_4}$$

$$\frac{}{\ell \equiv_Q \ell \sqcup \ell} \qquad \frac{}{\ell_1 \sqcup \ell_2 \equiv_Q \ell_2 \sqcup \ell_1} \qquad \frac{}{\ell_1 \sqcup (\ell_2 \sqcup \ell_3) \equiv_Q (\ell_1 \sqcup \ell_2) \sqcup \ell_3}$$

$$\frac{\ell_1 \equiv_Q \ell_2}{\ell_1 \sqsubseteq_Q \ell_2} \qquad \frac{\ell_1 \equiv_Q \ell_2}{\ell_2 \sqsubseteq_Q \ell_1}$$

Figure 3.1: Rules for label order and label equality with respect to a constraint set Q

3.2.4 Remarks on the Lattice Structure of Labels

The proof of Theorem 3.3 on the facing page justifies the soundness of the label order rules with respect to the domain lattice. While this is sufficient to show the correctness of the type system, I will now elaborate on the relation between the statically inferred label order, the denotational semantics of labels and the domain lattice $\diamond$ in order to clarify what is actually described by a constraint set Q.

We observe that

$$L_Q = (Lab, \sqsubseteq_Q, \sqcup, \top, \bot)$$

forms a bounded join-semilattice over the set of labels, where labels related by $\equiv_Q$ denote the same point in the lattice. This is the case because $\sqsubseteq_Q$ is a partial order (it is reflexive, transitive, and antisymmetric), $\top$ and $\bot$ are the top and bottom element of *Lab*, and for any two label classes ℓ_1 and ℓ_2, there exists a unique join (least upper bound), namely the label $\ell_1 \sqcup \ell_2$, which is indeed larger than both ℓ_1 and ℓ_2 with respect to $\sqsubseteq_Q$.[2] Note that there are labels, such as $\ell_1 \sqcup \ell_2$ and $\ell_2 \sqcup \ell_1$, which are distinct elements in *Lab* but equal with respect to the lattice order; this does not contradict the definition of a lattice.

The evaluation function $[\![\cdot]\!]^{\diamond}_{\sigma}$ is a homomorphism that embeds the label lattice L_Q into the domain lattice $\diamond$, because it adheres to the definition of a lattice homomorphism: the evaluation function preserves least upper bounds (by definition of

[2] As arbitrary sets of labels can be joined together, including the empty set which yields $\bot$ and *Lab* which yields $\top$, it is even a complete join semi-lattice.

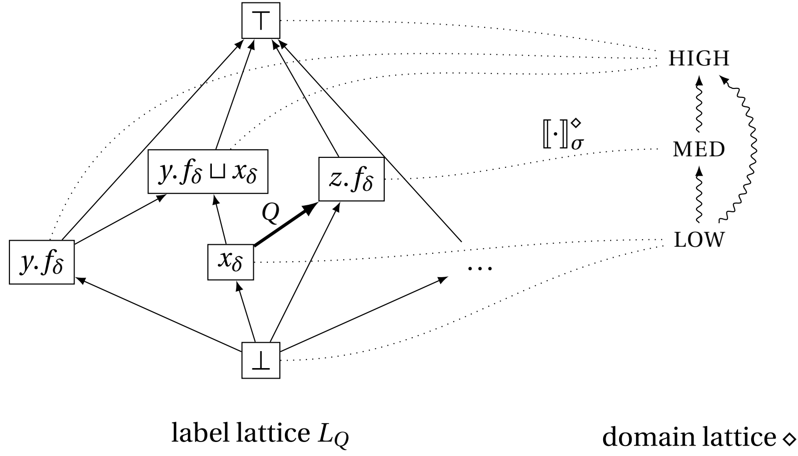

Figure 3.2: Example for embedding label lattice into domain lattice

$[\![\ell_1 \sqcup \ell_2]\!]^{\diamond}_{\sigma}$), the greatest and least element (by definition of $[\![\top]\!]^{\diamond}_{\sigma}$ and $[\![\bot]\!]^{\diamond}_{\sigma}$), as well as the order (by the soundness property stated in Theorem 3.3). The interpretation of the label lattice L_Q is thus a sublattice of ⋄.

Figure 3.2 illustrates the embedding, when one takes as an example the constraint set $Q = \{(x_\delta, z.f_\delta)\}$ and Sue's domain lattice ⋄ as presented in Section 2.1.2 on page 18. The label lattice over Q, L_Q, is partly shown on the left, with the arrows corresponding to the order $\sqsubseteq_Q$ (not all transitive edges are shown). For any state σ such that $\sigma \models^{\diamond} Q$, the label evaluation function $[\![\cdot]\!]^{\diamond}_{\sigma}$ assigns to each label a domain from $Dom^{\diamond}$ on the right such that the order is preserved. The dotted lines in the diagram represent the evaluation in an assumed example state σ that satisfies Q. In particular, the thick edge from x_δ to $z.f_\delta$, induced by the constraint set Q, corresponds to an edge in the domain lattice (from LOW to MED).

A constraint set Q spans a label lattice L_Q. Via the satisfiability relation $\sigma \models^{\diamond} Q$, the set Q describes those program states σ for which L_Q can be embedded into the domain lattice ⋄ by $[\![\cdot]\!]^{\diamond}_{\sigma}$. From a different perspective, if one fixes the state σ, then Q can as well be interpreted as a description of those domain lattices ⋄ whose structure includes the pairwise domain positionings which are abstractly described by label pairs in Q.

Therefore, the set Q and thus the entire label lattice L_Q collect information about both the state σ (namely the values of domain variables and fields) and the structure of the domain lattice ⋄ which describes the effective security policy.

$$\text{T-VAL}\ \frac{c \in \{n, \top, \bot\}}{\Gamma \vdash c : \bot} \qquad \text{T-VAR}\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \text{T-GETF}\ \frac{\Gamma \vdash \pi : \ell}{\Gamma \vdash \pi.f : \Phi^{\pi}(f) \sqcup \ell}$$

$$\text{T-OP}\ \frac{\circ \in \{\mathbf{op}, \sqcup, \sqsubseteq\} \quad \Gamma \vdash e_1 : \ell_1 \quad \Gamma \vdash e_2 : \ell_2}{\Gamma \vdash e_1 \circ e_2 : \ell_1 \sqcup \ell_2}$$

Figure 3.3: Expression type system

3.3 Typing Expressions, and Meta-Label Monotonicity

This section presents the information flow type system for the DSD language, and provides a soundness theorem.

3.3.1 Typing Expressions

The typing judgement $\Gamma \vdash e : \ell$ assigns to an expression e a label ℓ, given a variable type environment Γ. The typing rules are shown in Figure 3.3. They correspond to the ones by Banerjee and Naumann [BN05], but with labels instead of security domains. The rule T-VAL assigns to values (constants) the bottom-most label $\bot$, because they never depend on any private information. For variables, the rule T-VAR looks up the type of the variable in the environment Γ and simply treats it as a label. To compute the label of the field access, the rule T-GETF needs to turn the field type into a label; this is done using the qualified field type Φ^{π} defined earlier in this chapter. Additionally, the rule takes the label of the reference π into account, because the act of dereferencing leaks information about the reference itself. The T-OP rule, finally, is used to assign a label to an operator expression. The label of an operation is the least upper bound of the labels of the operands.

As labels are just a specific form of expressions, the expression typing rules are also used to assign a label to a label: T-VAL assigns a label to the constant labels $\top$ and $\bot$, T-VAR is used to type the label x_{δ}, T-GETF can also type accesses to f_{δ} fields, and T-OP is used to type the label operations $\ell_1 \sqcup \ell_2$ and $\ell_1 \sqsubseteq \ell_2$.

3.3.2 Meta-Label Monotonicity

Universal noninterference is defined in terms of related program states that correspond on the values of visible variables and fields. In the presence of dynamic security domains, however, this visibility may depend on other variables and fields, which again may or may not be visible.

Without further mechanisms, a dependent typing scheme for information security types, such as the one presented here, causes a problem for the definition of equivalence of two program states. It may happen that a variable x (or a field f) is visible in one state, but invisible in the other state, such that it is not clear whether the value of x (or f) in both states should be required to be β-related or not.

An obvious approach is to just compare the values in those cases where x (or f) is visible in both states. If the visibility is different in the two states, the variable (or field) is effectively treated as invisible in both states. However, I argue that this simplified interpretation of dependent security types contradicts the intuition that they are meant to express. I illustrate this by two cases that use more flexible typing schemes than the ones presented so far.

First, let us look at a `Buffer` object with an f_δ-typed field `contents`, and let us assume that the f_δ field has the type $\top$. If we take a simple 2-point lattice $\diamond$ with the domains LOW and HIGH and assume the value of f_δ is LOW, we have a paradoxical situation: according to the value of f_δ, the `contents` field is public — that is, some LOW observer can see the `contents` field. However, because f_δ is itself typed $\top$, which is interpreted as HIGH, its value should remain secret. Just by being able to see the `contents` field, the observer can deduce that f_δ must contain the value LOW. In other words, the very fact that something is public cannot reasonably be a secret.

As a second case, let us now assume instead that $\Phi(f_\delta) = f_\delta$, which means f_δ contains its very own security domain and thus defines its own visibility. What does this mean? If the value of f_δ is LOW, then a LOW observer may see that it is LOW, but if it is HIGH, then a LOW observer may not see its contents; though it is (again) easy to deduce that f_δ must be HIGH in this case. Here, the circular dependency causes the domain field to leak its own contents.

For these reasons, I have introduced the restrictions $\Gamma(x_\delta) = \bot$ and $\Phi(f_\delta) = \bot$, which excludes both circularities and public types expressed by secret type variables. With these restrictions, the semantic problem disappears: as I have shown in Lemma 2.3 on page 26, in two $\diamond, k$-equivalent states, each variable or field is either visible in both states, or invisible in both states.

The problems that arise with flexible typing schemes for variables and fields are mirrored when one uses typing rules that assign expressions as security types to other expressions. This is the case here: the judgement $\Gamma \vdash e : \ell$ states that the security domain of e is the evaluation of the expression ℓ. Thus, we can use the same rules to derive $\Gamma \vdash \ell : \ell'$ such that the meta-label ℓ' describes the security level of ℓ. Without further care, it may happen that ℓ contains the information that e is public, but this information itself is not public (as indicated by ℓ'). Furthermore, a judgement may label a label with itself, causing a possible leak as described above.

To prevent these phenomena, a system that assigns security labels not only to expressions, but in particular to security labels themselves should be *monotone* and

well-founded. I will now define these properties precisely, and show that the expression typing rules have these properties.

Definition 3.6 *Let $f : Exp \mapsto Lab$ be a labelling function.*

- *The function is* monotone *if for all expressions e, $f(f(e)) \sqsubseteq_\emptyset f(e)$.*
- *The function is* well-founded *if for all expressions e, there exists an $n \in \mathbb{N}$ such that for all $m \geq n$, $f^m(e) \equiv_\emptyset \bot$.*[3]

The expression typing judgement, as defined by rules in Figure 3.3 on page 37, indeed has these properties intrinsically when it is interpreted as a labelling function:

Lemma 3.7 *Let $\mathsf{labelof}_\Gamma$ be a labelling function such that for all DSD expressions e, $\mathsf{labelof}_\Gamma(e) = \ell \iff \Gamma \vdash e : \ell$. For any well-formed type environment Γ, the function $\mathsf{labelof}_\Gamma$ is both monotone and well-founded.*

The proof of this lemma can be found in Appendix A. The main implication that arises from this syntactic characterization is that in two $\diamond, k$-equivalent states, a label ℓ assigned to an expression e always evaluates in both states to domains below k or not below k, which means that e is either visible or not visible in both states.

3.4 Typing Statements

In this section, I present and explain the typing rules for DSD statements, and define what it means for a program to be well-typed.

3.4.1 Syntactic Definitions

The typing rules for DSD statements rely on a number of syntactic definitions.

Subexpressions The functions $\mathsf{vars}(e)$ and $\mathsf{flds}(e)$ collect all variable and field identifiers that occur syntactically in an expression e:

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{vars}(c) & = & \emptyset & \mathsf{flds}(c) & = & \emptyset \\
\mathsf{vars}(x) & = & \{x\} & \mathsf{flds}(x) & = & \emptyset \\
\mathsf{vars}(e.f) & = & \mathsf{vars}(e) & \mathsf{flds}(e.f) & = & \mathsf{flds}(e) \cup \{f\} \\
\mathsf{vars}(e_1 \circ e_2) & = & \mathsf{vars}(e_1) \cup \mathsf{vars}(e_2) & \mathsf{flds}(e_1 \circ e_2) & = & \mathsf{flds}(e_1) \cup \mathsf{flds}(e_2)
\end{array}$$

$$\text{where} \quad c ::= \top \mid \bot \mid n \text{ and } \circ \in \{\mathbf{op}, \sqcup, \sqsubseteq\}$$

[3] The notation $f^m(e)$ means f applied m times to e — that is, $f^0(e) = e$ and $f^{n+1}(e) = f^n(f(e))$.

The definition is lifted to constraint sets:

$$\mathsf{vars}(Q) = \bigcup_{(\ell_1,\ell_2)\in Q} \mathsf{vars}(\ell_1) \cup \mathsf{vars}(\ell_2) \qquad \mathsf{flds}(Q) = \bigcup_{(\ell_1,\ell_2)\in Q} \mathsf{flds}(\ell_1) \cup \mathsf{flds}(\ell_2)$$

I write $x \in \ell$ and $x \in Q$ for $x \in \mathsf{vars}(\ell)$ and $x \in \mathsf{vars}(Q)$, respectively. Likewise, I write $f \in \ell$ and $f \in Q$ for $f \in \mathsf{flds}(\ell)$ and $f \in \mathsf{flds}(Q)$. Most often, this is used to express that an identifier does *not* appear within a label or a constraint set, by using side conditions like $x \not\in \ell$ or $f \not\in Q$.

Substitutions Two forms of substitutions are defined:

- The term $\ell[\pi_{new}/\pi_{old}]$ describes the label ℓ where all syntactic occurrences of the access path π_{old} are replaced by the access path π_{new}.
- The term $\ell[\ell'/\ell'']$ replaces every occurrence of the label ℓ'' (as a subexpression) in ℓ with ℓ'.

The substitutions in labels are point-wise lifted to substitutions in constraint sets Q by applying the substitution to each label in Q. Similarly, a substitution can be applied to a function that returns labels.

$$Q[u/v] = \{(\ell[u/v], \ell'[u/v]) \mid (\ell, \ell') \in Q\}$$
$$f[u/v] = \lambda t.\, f(t)[u/v]$$

Sequences In the typing rules, I write $\overline{x}.i$ for the i-th element of the sequence $\overline{x}$, where the first element has the index 1. The length of a sequence $\overline{x}$ is written $|\overline{x}|$. To construct a sequence, I use the ML style notations $[a, b, c]$ and $a :: \overline{x}$. The term ϵ denotes the empty sequence. I also write X, a to denote the set $X \cup \{a\}$.

Point-wise extensions to sequences In the typing rules, judgements are lifted point-wise to sequences. More precisely,

- $\overline{\ell} \sqsubseteq_Q \overline{\ell'}$ if and only if $|\overline{\ell}| = |\overline{\ell'}|$ and for all $i \in \{1, \ldots, |\overline{\ell}|\}$, $\overline{\ell}.i \sqsubseteq_Q \overline{\ell'}.i$.
- Similarly, $\Gamma \vdash \overline{e} : \overline{\ell}$ if and only if $|\overline{e}| = |\overline{\ell}|$ and for all $i \in \{1, \ldots, |\overline{e}|\}$, $\Gamma \vdash \overline{e}.i : \overline{\ell}.i$.

3.4.2 The Typing Rules

For statements, the system defines a typing judgement $\Gamma, pc \vdash \{Q\}\ S\ \{Q'\}$, which means that the program is (Q, Q')-universally noninterferent with respect to the variable typing Γ. Furthermore, $pc \in \mathit{Lab}$ is a *program counter label* whose meaning shall

$$\textsc{T-Weak}\ \frac{\Gamma, pc \vdash \{Q_0\}\ S\ \{Q_0'\} \quad Q \Rightarrow Q_0 \quad Q_0' \Rightarrow Q'}{\Gamma, pc \vdash \{Q\}\ S\ \{Q'\}} \qquad \textsc{T-Skip}\ \frac{}{\Gamma, pc \vdash \{Q\}\ \textbf{skip}\ \{Q\}}$$

$$\textsc{T-Seq}\ \frac{\Gamma, pc \vdash \{Q\}\ S_1\ \{Q'\} \quad \Gamma, pc \vdash \{Q'\}\ S_2\ \{Q''\}}{\Gamma, pc \vdash \{Q\}\ S_1 ; S_2\ \{Q''\}} \qquad \textsc{T-While}\ \frac{\Gamma \vdash e : \ell \quad \Gamma, pc \sqcup \ell \vdash \{Q\}\ S\ \{Q\}}{\Gamma, pc \vdash \{Q\}\ \textbf{while}\ e\ \textbf{do}\ S\ \{Q\}}$$

$$\textsc{T-If}\ \frac{\Gamma \vdash e : \ell \quad \Gamma, pc \sqcup \ell \vdash \{Q\}\ S_1\ \{Q'\} \quad \Gamma, pc \sqcup \ell \vdash \{Q\}\ S_2\ \{Q'\}}{\Gamma, pc \vdash \{Q\}\ \textbf{if}\ e\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q'\}}$$

$$\textsc{T-IfLabel}\ \frac{\Gamma \vdash \ell_1 \sqsubseteq \ell_2 : \ell \quad \Gamma, pc \sqcup \ell \vdash \{Q, (\ell_1, \ell_2)\}\ S_1\ \{Q'\} \quad \Gamma, pc \sqcup \ell \vdash \{Q\}\ S_2\ \{Q'\}}{\Gamma, pc \vdash \{Q\}\ \textbf{if}\ \ell_1 \sqsubseteq \ell_2\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q'\}}$$

$$\textsc{T-Assign}\ \frac{\Gamma \vdash e : \ell \quad \ell \sqcup pc \sqsubseteq_{Q'} \Gamma(x) \quad x \neq x_\delta \quad x \notin pc}{\Gamma, pc \vdash \{Q[e/x] \cup Q'\}\ x := e\ \{Q\}}$$

$$\textsc{T-PutF}\ \frac{\Gamma \vdash \pi : \ell_1 \quad \Gamma \vdash e : \ell_2 \quad \ell_1 \sqcup \ell_2 \sqcup pc \sqsubseteq_{Q'} \Phi^{\pi}(f) \quad f \neq f_\delta \quad f \notin pc \quad f \notin Q[e/\pi.f]}{\Gamma, pc \vdash \{Q[e/\pi.f] \cup Q'\}\ \pi.f := e\ \{Q\}}$$

$$\textsc{T-New}\ \frac{\Gamma \vdash \overline{e} : \overline{\ell} \quad \mathsf{fields}(C) = \overline{f} \quad \Phi^* = \Phi^x[\overline{e}.1/x.f_\delta] \quad \overline{\ell} \sqsubseteq_{Q'} \Phi^*(\overline{f}) \quad pc \sqsubseteq_{Q'} \Gamma(x) \quad x \notin pc \quad x \neq x_\delta \quad x \notin Q[\overline{e}/x.\overline{f}]}{\Gamma, pc \vdash \{Q[\overline{e}/x.\overline{f}] \cup Q'\}\ x := \textbf{new}\ C(\overline{e})\ \{Q\}}$$

$$\textsc{T-Call}\ \frac{\begin{array}{c} \mathsf{msig}(m) = [\Gamma_m, pc_m, Q_m, Q_m'] \quad \mathsf{margs}(m) = \overline{x} \\ \Gamma \vdash \pi : \ell_r \quad \Gamma \vdash \overline{e} : \overline{\ell} \quad \Gamma^* = \Gamma_m[\overline{e}.1/x_\delta] \\ \ell_r \sqsubseteq_{Q'} \Gamma^*(\mathit{this}) \quad \overline{\ell} \sqsubseteq_{Q'} \Gamma^*(\overline{x}) \quad pc \sqsubseteq_{Q'} pc_m[\overline{e}/\overline{x}][\pi/\mathit{this}] \\ \Gamma^*(\mathit{ret}) \sqcup pc \sqsubseteq_Q \Gamma(x) \quad x \notin pc, Q \quad \forall f.\, f \neq f_\delta \Rightarrow f \notin pc, Q \quad x \neq x_\delta \end{array}}{\Gamma, pc \vdash \{Q_m[\overline{e}/\overline{x}][\pi/\mathit{this}] \cup Q' \cup Q\}\ x := \pi.m(\overline{e})\ \{Q_m'[x/\mathit{ret}] \cup Q\}}$$

Figure 3.4: Type system for DSD statements

be explained below. The rules for statement typing are shown in Figure 3.4 on the preceding page. In the following, the constraint set Q is called the *precondition*, whereas the constraint set Q' is called the *postcondition*.

Classic information flow type systems contain order requirements for concrete domains. For example, an assignment $x := y$ is typable if (among others) the domain of y is lower than or equal to the domain of x with respect to the domain order. These requirements are usually specified as side conditions that follow directly from the fixed domain lattice. In contrast, the DSD type system specifies these order requirements at the level of labels; in the typing rules, these label order requirements appear as predicates in the precondition Q.

The sets Q and Q' are derived in a similar fashion as predicates are derived in Hoare logic [Hoa69]. The rules for the constraint sets are mostly syntax-directed and are suitable for backward reasoning to compute the weakest liberal precondition. The difference to Hoare logic is that constraint sets contain only label order requirements $\ell_1 \sqsubseteq \ell_2$ instead of arbitrary formulae.

The control flow structure of a program may cause indirect information leaks of branching conditions. When different execution paths can be taken, the observable effect of each branch unveils information about the condition on which the execution branched. An information flow analysis has to make sure that if the branching condition is to be kept confidential, the observable effects of all execution branches should be identical. In DSD, an observable effect is the change of a visible variable or field. The type system by Volpano et al. [VSI96] and many other type systems that verify end-to-end noninterference properties approach this requirement by conservatively ensuring that subbranches do not have any observable effects at all. To this end, all written variables and fields in a branch must be more confidential than the labels of all enclosing branching conditions, which are stored in the program counter label pc. Here, I follow this standard: the labels of the branching conditions are collected in the pc label, and all written variables and fields must be at least confidential as this pc label.

The type system makes sure that x_δ and f_δ are immutable. The x_δ variable can only be initialized with a passed argument at a method call, and an f_δ field can only be initialized when an object is created.

Each rule is now described in detail:

T-WEAK: This rule, also called the weakening or subsumption rule, derives a stronger precondition and a weaker postcondition, using the syntactic constraint set implication as defined in the previous chapter.

T-SKIP, T-SEQ, T-WHILE, and T-IF: These rules resemble those in the type system by Volpano et al., where the constraint sets are defined as in Hoare logic: the constraint set remains unchanged for **skip** operations; in a sequence, the postcondition of the first statement must be the precondition of the following statement;

for while loops, the constraint set must be invariant; and for conditionals, the subbranches must have the same pre- and postconditions. As explained above, the label of a conditional expression in an **if** or **while** statement is added to the *pc* label of the subprogram in order to prevent indirect flows of information.

T-IFLABEL: The rule for label tests behaves exactly like the T-IF rule, with one difference: we can assume the tested label flow as a new label order information in the precondition of the **then** branch.

T-ASSIGN: The rule for variable assignments regards an assignment secure if there is a precondition Q' from which one can derive that the label of the assigned expression (ℓ) and the lower bound on the store side effects (pc) are both lower than the label of the updated variable ($\Gamma(x)$).

The set Q is used for Hoare-like backward reasoning to be able to make assertions about the value of x after the assignment: whatever is known about x after the execution of the assignment should hold for e before the assignment; thus, the first part of the precondition is $Q[e/x]$. The other part (the set Q') is used independently for all side conditions that are required to make the statement typable. The reason for separating $Q[e/x]$ and Q' is that the conditions in Q' may depend on the value of x before the assignment takes place. Nevertheless, the sets $Q[e/x]$ and Q' do not need to be disjoint. Moreover, the entire precondition of the rule can be derived in classic Hoare logic by combining the rules for assignment and for consequence of the logic.

In the T-ASSIGN rule and all other rules for assignments, it is not allowed to update a variable or field if it is mentioned anywhere in the pc, because the pc label must stay invariant. Also, it is not possible to update x_δ or f_δ.

T-PUTF: The rule for updating fields shares the concepts of the rule for variable assignments. It must be checked whether information may flow from the assigned expression e to the field f, and whether the label pc is a lower bound of the label of the affected field.

The object reference must be an access path π. If a field update statement references an object by an expression that is not an access path, the typing rule cannot be applied, thus the program is untypable.

The reference π may be accessible and observable by another reference, such that the field update may indirectly reveal information about the reference π itself; thus there is a flow from π to the field, which has to be verified too. Again, we use a dedicated constraint set Q' in the precondition to derive all the required flows, namely that the label of the reference (ℓ_1), the label of the assigned expression (ℓ_2), and the pc label are all lower than or equal to the label of the updated field expressed by the qualified field type $\Phi^\pi(f)$.

When updating a field, one also has to prevent aliasing issues. While the set $Q[e/\pi.f]$ ensures that postconditions about $\pi.f$ can be derived if the respective information is present for e in the precondition, the same field f could be mentioned in $Q[e/\pi.f]$ via a different access path, such that changing the field f might invalidate that information. To prevent this situation, the rule conservatively disallows any appearance of f in the precondition $Q[e/\pi.f]$, therefore eliminating any information about the field f of any object, including the object whose field is updated by the statement.

T-NEW: The premises of this rule take into account that an object creation can be seen as a sequence of assignments:

$$\begin{aligned} x &:= (\text{new object}); \\ x.(\overline{f}.1) &:= \overline{e}.1; \\ x.(\overline{f}.2) &:= \overline{e}.2; \\ x.(\overline{f}.3) &:= \overline{e}.3; \\ &\ldots \end{aligned}$$

where $\overline{f} = \mathsf{fields}(C)$. Similar to the rule for field updates, everything that holds for $x.(\overline{f}.i)$ in the postcondition also holds for $\overline{e}.i$ in the precondition.

Similar to field updates, there is the precondition that the labels of the expressions are lower than the types of the field. A field type f_δ is interpreted as the label $x.f_\delta$. Since $x.f_\delta$ is the first field of the object, it is initialized with the first argument $\overline{e}.1$, hence a corresponding substitution takes place in the definition of the labelling function Φ^*.

If the field initializations were ordinary field updates, one would additionally need to check for each field f that a flow from the pc label and from the reference x to $\Phi^*(f)$, is allowed. The first flow, however, is subsumed by the condition $pc \sqsubseteq_{Q'} \Gamma(x)$, and the second check is not needed because the object cannot be aliased by another reference with a lower label, as it is new. Another difference to ordinary field updates is that one does not need to require that the fields do not occur in the precondition or in the pc label, because the object is fresh and the field updates do not have an effect on the evaluation of the precondition and pc.

Finally, the assignment to x is treated in a similar fashion as variable assignments in the T-ASSIGN rule, with the same prerequisites. Here, however, the assigned value is the fresh object and can be seen as public: although the choice of the memory location is non-deterministic, no secret data leaks from the location itself. Thus, the assigned expression **new** $C(\overline{e})$ can be considered as having the label $\bot$, and the flow requirement from the expression to x can be omitted.

Also, as **new** $C(\overline{e})$ is not a valid access path, there is no substitution, hence no information about x may exist in the precondition $Q[\overline{e}/x.\overline{f}]$.

T-CALL: The rule for method calls relies on a given method signature

$$\mathsf{msig}(m) = [\Gamma_m, pc_m, Q_m, Q'_m]$$

where Γ_m specifies the types of the local variables of m, pc_m is the lower bound on side effects that occur in m, and Q_m and Q'_m are the pre- and post-conditions of the method execution, respectively. Signatures are explained in detail below.

The typing rule treats the initialization of the variables of the called methods as a sequence of assignments, where $\overline{x} = \mathsf{margs}(m)$:

$$\begin{aligned} &this := \pi; \\ &\overline{x}.1 := \overline{e}.1; \\ &\overline{x}.2 := \overline{e}.2; \\ &\overline{x}.3 := \overline{e}.3; \\ &\ldots \end{aligned}$$

Just as in the T-ASSIGN rule, it is checked whether the passed arguments and the object reference have labels lower than the formal types declared in Γ_m. To this end, one transforms the types in Γ_m. The type x_δ is replaced by the first argument $\overline{e}.1$, because it is the value with which the variable x_δ is initialized. The other possible variable types in Γ_m ($\top$ and $\bot$) are just treated as labels. Thus, Γ^* assigns to each variable of the method an "outside label" that has a meaning at the level of the method caller.

Just as with field update statements, the object reference must be an access path π. The rule checks that the label of the object reference π is lower than the outside label of *this*, and that the labels of the passed expressions have labels lower than the outside labels of the formal arguments $\overline{x}$, all with respect to the set Q' in the precondition of the call. In contrast to ordinary variable assignments, one does not need to take the pc label into account, because the variables assigned in the called method m are local to that method and not visible from the caller's side.

The precondition of the rule also includes the outside view of the method's precondition Q_m, which means local variables of the called method m are replaced with the parameter expressions of the call. Likewise, if the variable *ret* in the method's postcondition Q'_m (the only variable that may occur there) is replaced with x, one gets the outside view of Q'_m, which is in the postcondition of the call. Also, method calls cannot be used to update x_δ.

As with other assignment rules, it is required that x is not in pc. Since the method may have any effect on the heap, one cannot allow any field f to occur syntactically in pc either, because changing a field may change the evaluation of pc.

One may use a constraint set Q in the pre- and in the post-condition provided that its evaluation does not change during the method call, similar to a frame rule in separation logic [Rey02]. "Separation" is broadly enforced by requiring that no item that can possibly be updated (i.e., the variable x or any field that is not f_δ) occurs syntactically in Q. The set Q is not only used to preserve information outside of the method call, but also to verify the actual assignment to x: The outside label of the return variable *ret* as well as the pc of the caller must be lower than the type of the assigned variable x with respect to Q.

3.4.3 Method Signatures and Well-Typed Programs

As mentioned above, the type system depends on a given typing information for methods in form of *method signatures*, which are quadruples of the form

$$\mathsf{msig}(m) = [\Gamma, pc, Q, Q']$$

such that each typing judgement provides the type for the respective method body. It intuitively means: In method m, the local variables shall have the types assigned by Γ. Also, if Q holds when the method m is called, Q' holds on its return. Finally, no data below pc are updated during the execution of the method.

A method signature is *well-formed* if the following conditions hold:

- The variable type environment Γ must be well-formed, and it must specify a type for all local variables including *this* and *ret*: $\mathsf{dom}(\Gamma) = \mathsf{margs}(m) \cup \{\mathit{this}, \mathit{ret}\}$.
- The pc label should originate from a monotone and well-founded labelling function — that is, there needs to exist an expression e such that $\Gamma \vdash e : pc$.
- The pre- and post-conditions may only refer to local variables of the method in a restricted way: $\mathsf{vars}(Q) \subseteq \mathsf{margs}(m) \cup \{\mathit{this}\}$, and $\mathsf{vars}(Q') \subseteq \{\mathit{ret}\}$. It is not useful to include *ret* in Q, as the variable is always initialized with the defval constant. Likewise, information about variables other than *ret* are not useful in Q', as these variables are not returned.

The following definition connects the signature of a method to the type derivations of its implementation.

Definition 3.8 *Let P_{DSD} be a DSD program, and let m be a method with a well-formed signature* $\mathsf{msig}(m) = [\Gamma, pc, Q, Q']$. *The method m is* well-typed *if the typing judgement* $\Gamma, pc \vdash \{Q\}\ \mathsf{mbody}(m)\ \{Q'\}$ *can be derived. A DSD program P_{DSD} is* well-typed *if all its methods are well-typed.*

A method signature has to be provided for all methods of the program, including external methods. As no DSD implementation is specified for external methods, external methods are automatically considered well-typed, which means that the signatures of those methods are trusted by the type system.

3.5 Soundness Results

Before I come to the main theorem, I adjust the preceding definitions of universal noninterference from Section 2.3.4 to the method signatures.

Definition 3.9 *A method m with* $\mathsf{msig}(m) = [\Gamma, pc, Q, Q']$ *is called* universally noninterferent with respect to its signature *if it is* (Q, Q')*-universally noninterferent with respect to* Γ. *A program* P_{DSD} *is* universally noninterferent *if each (non-external) method is universally noninterferent with respect to its respective signature.*

The following is the main soundness theorem.

Theorem 3.10 *If* P_{DSD} *is a well-typed DSD program, then it is universally noninterferent.*

The theorem is proved by induction over the operational semantics. The full proof can be found in Appendix A; here, I give a number of lemmas that also follow from the type derivation. First, if the pc label is invariant, and if Q holds before the execution of statement S, then Q' holds afterwards:

Lemma 3.11 *Let* $\Gamma, pc \vdash \{Q\}\ S\ \{Q'\}$ *and* $\sigma_1 \xrightarrow{S}{}^{\diamond} \sigma_2$, *and let all methods be well-typed.*

1. $[\![pc]\!]^{\diamond}_{\sigma_1} = [\![pc]\!]^{\diamond}_{\sigma_2}$.
2. *If* $\sigma_1 \models^{\diamond} Q$, *then* $\sigma_2 \models^{\diamond} Q'$.

Next, a pc label that is below k in $\diamond$ ensures that no variables or fields that are visible at k in $\diamond$ are written, thereby preventing indirect flows.

Lemma 3.12 *Let* P_{DSD} *be a well-typed program. Suppose* $\Gamma, pc \vdash \{Q\}\ S\ \{Q'\}$ *and* $(s_1, h_1) \xrightarrow{S}{}^{\diamond} (s_2, h_2)$ *and* $(s_1, h_1) \models^{\diamond} Q$. *Let* $\mathsf{id} = \{(r, r) \mid r \in \mathsf{dom}(h_1) \cap \mathsf{dom}(h_2)\}$.
If

- $[\![pc]\!]^{\diamond}_{s_1, h_1} \not\leq^{\diamond} k$,

then

1. $\vdash^{\diamond} s_1 \sim^{\Gamma,k}_{\mathsf{id}} s_2$, *and*
2. $\vdash^{\diamond} h_1 \sim^{k}_{\mathsf{id}} h_2$.

The expression typing judgement can be interpreted as a visibility judgement for expressions, which extends the visibility definition for variables and fields from Section 2.3.2 on page 24.

Definition 3.13 *An expression is* visible at k in $\diamond$ and a state σ *if* $\Gamma \vdash e : \ell$ *and* $[\![\ell]\!]^{\diamond}_{\sigma} \leq^{\diamond} k$.

That is, an expression is visible at k if its label evaluates to a domain at or below k. The soundness statement then says that an expression e that is visible at k in $\diamond$ and two $\diamond, k$-equivalent program states evaluates to indistinguishable values, so that e does not depend on data that is not visible at k.

Theorem 3.14 *If* $\Gamma \vdash e : \ell$*, then for all states* σ *and* σ' *and all partial bijections* β *such that* $\vdash^{\diamond} \sigma \sim^{\Gamma,k}_{\beta} \sigma'$*, if* $[\![\ell]\!]^{\diamond}_{\sigma} \leq^{\diamond} k$ *and* $[\![\ell]\!]^{\diamond}_{\sigma'} \leq^{\diamond} k$*, then* $[\![e]\!]^{\diamond}_{\sigma} \sim_{\beta} [\![e]\!]^{\diamond}_{\sigma'}$.

The most important result that follows from monotonicity and well-foundedness is that an expression is always either visible in two states, or invisible in two equivalent states:

Theorem 3.15 *If* $\Gamma \vdash e : \ell$*, then for all states* σ *and* σ'*, for all partial bijections* β *and for all domains* k *such that* $\vdash^{\diamond} \sigma \sim^{\Gamma,k}_{\beta} \sigma'$*,* $[\![\ell]\!]^{\diamond}_{\sigma} \leq^{\diamond} k$ *if and only if* $[\![\ell]\!]^{\diamond}_{\sigma'} \leq^{\diamond} k$.

Using the expression typing soundness theorem 3.14, one can even show that if $[\![\ell]\!]^{\diamond}_{\sigma} \leq^{\diamond} k$, then $[\![\ell]\!]^{\diamond}_{\sigma} = [\![\ell]\!]^{\diamond}_{\sigma'}$. A specialization of this theorem is Lemma 2.3 on page 26, which stated a similar result for types of variables and fields.

Note that the requirements on the pc label in the well-formedness conditions for signatures in the previous section are directly related to Theorem 3.15. The requirements are needed to ensure that the pc label evaluates to a domain at or below k in both states or in none of the two states.

3.6 Typing the Example Program

In the final section of the chapter, I illustrate an application of the type system to the example program defined in Figure 2.4 on page 22. I define the type environments for the example program, and show why the body of the `sendFile` method indeed has a type derivation in these environments. For better clarity, I write $(\ell_1 \sqsubseteq \ell_2)$ instead of (ℓ_1, ℓ_2) for a label pair in a constraint set Q to underline that it is a label flow predicate.

Figure 3.5 on the facing page defines the types of the fields and the signatures of the methods that occur in the program. In fact, the only field identifier in the program is f_δ. The method signatures are a bit more complicated than those presented in the introduction. The `read` method of the `Buffer` class returns a value that has the dynamic domain of the buffer, stored in the f_δ field of the object. It is, however, not possible to annotate the return variable *ret* of the `read` method with $this.f_\delta$, because this is not a valid variable type. Instead, the *ret* variable gets the type x_δ. The method

$$
\begin{array}{rcl}
\Phi(f_\delta) & = & \bot \\
\mathsf{msig}(\texttt{read}) & = & [\quad \Gamma = \{x_\delta \mapsto \bot, \mathit{this} \mapsto \bot, \mathit{ret} \mapsto x_\delta\}, \quad pc = \bot, \\
 & & \quad Q = \{\mathit{this}.f_\delta \sqsubseteq x_\delta\}, \quad Q' = \emptyset \quad] \\
\mathsf{msig}(\texttt{write}) & = & [\quad \Gamma = \{x_\delta \mapsto \bot, s \mapsto x_\delta, \mathit{this} \mapsto \bot, \mathit{ret} \mapsto \bot\}, \quad pc = \bot, \\
 & & \quad Q = \{x_\delta \sqsubseteq \mathit{this}.f_\delta\}, \quad Q' = \emptyset \quad] \\
\mathsf{msig}(\texttt{sendFile}) & = & [\quad \Gamma = \{x_\delta \mapsto \bot, \texttt{file} \mapsto \bot, \texttt{srv} \mapsto \bot, \texttt{tmp} \mapsto x_\delta, \\
 & & \qquad \mathit{this} \mapsto \bot, \mathit{ret} \mapsto \bot\}, \quad pc = \bot, \\
 & & \quad Q = \{x_\delta \sqsubseteq \texttt{file}.f_\delta, \texttt{file}.f_\delta \sqsubseteq x_\delta\}, \quad Q' = \emptyset \quad]
\end{array}
$$

Figure 3.5: Type environments for the example program

expects the caller to provide this x_δ domain, and gives the precondition in Q that data must be allowed to flow from $\mathit{this}.f_\delta$ (the domain of the buffer) to x_δ (the provided domain of the return variable). In an analogous way, the `write` method expects the caller to provide the domain of the argument s in the argument x_δ. Since s is written into the buffer, there is the precondition that x_δ must be lower than the domain of the buffer object, that is, $\mathit{this}.f_\delta$. In the signature of the `sendFile` method, finally, the variable `tmp` gets the type x_δ. The variable `tmp` is supposed to hold the contents read from the file. Thus, there are constraints that the type of `tmp` (x_δ) is equal to the dynamic domain of `file` object ($\texttt{file}.f_\delta$).

Let Q_m be the precondition of the `sendFile` method as specified by its signature, and the statement S be the **then** branch of the method body, that is, the sequential composition of the two method calls. The body of the `sendFile` method is well-typed, because we can give a type derivation as follows:

$$
\text{T-IfLabel}\;
\dfrac{
\dfrac{
\text{T-Op}\;
\dfrac{
\text{T-GetF}\;\dfrac{\dfrac{\Gamma(\texttt{file}) = \bot}{\Gamma \vdash \texttt{file} : \bot}\,\text{T-Var} \quad \vdots \quad \Phi^{\texttt{file}}(f_\delta) = \bot}{\Gamma \vdash \texttt{file}.f_\delta : \bot \sqcup \bot}
\qquad
\text{T-GetF}\;\dfrac{\dfrac{\Gamma(\texttt{srv}) = \bot}{\Gamma \vdash \texttt{srv} : \bot}\,\text{T-Var} \quad \vdots \quad \Phi^{\texttt{srv}}(f_\delta) = \bot}{\Gamma \vdash \texttt{srv}.f_\delta : \bot \sqcup \bot}
}{\Gamma \vdash \texttt{file}.f_\delta \sqsubseteq \texttt{srv}.f_\delta : \bot \sqcup \bot \sqcup \bot \sqcup \bot}
}{
\vdots
}
\quad
\Gamma, \bot \vdash \{Q_m, (\texttt{file}.f_\delta \sqsubseteq \texttt{srv}.f_\delta)\}\ S\ \{\emptyset\}
\quad
\text{T-Weak}\;\dfrac{\text{T-Assign}\;\dfrac{\Gamma(\texttt{ret}) = \bot \qquad \Gamma \vdash 0 : \bot}{\Gamma, \bot \vdash \{\emptyset\}\ \texttt{ret:=0}\ \{\emptyset\}}}{\Gamma, \bot \vdash \{Q_m\}\ \texttt{ret:=0}\ \{\emptyset\}}
}{
\Gamma, \bot \vdash \{Q_m\}\ \textbf{if}\, \texttt{file}.f_\delta \sqsubseteq \texttt{srv}.f_\delta\ \textbf{then}\ S\ \textbf{else}\ \texttt{ret:=0}\ \{\emptyset\}
}
$$

In the following, let Q be the precondition of the statement S, that is, $Q = \{\texttt{file}.f_\delta \sqsubseteq \texttt{srv}.f_\delta, \texttt{file}.f_\delta \sqsubseteq x_\delta, x_\delta \sqsubseteq \texttt{file}.f_\delta\}$. To show the typing judgement for S, we need to give a judgement for each of the two method calls:

$$\text{T-WEAK}\ \dfrac{\text{T-SEQ}\ \dfrac{\begin{array}{c}\Gamma,\bot \vdash \{Q\}\ \texttt{tmp:=file.read(file.}f_\delta\texttt{)}\ \{Q\}\\ \Gamma,\bot \vdash \{Q\}\ \texttt{ret:=srv.write(file.}f_\delta\texttt{,tmp)}\ \{Q\}\end{array}}{\Gamma,\bot \vdash \{Q\}\ \begin{array}{l}\texttt{tmp:=file.read(file.}f_\delta\texttt{);}\\ \texttt{ret:=srv.write(file.}f_\delta\texttt{,tmp)}\end{array}\ \{Q\}}}{\Gamma,\bot \vdash \{Q\}\ S\ \{\emptyset\}}$$

The first method call can be typed by instantiating the rule T-CALL appropriately:

$$\text{T-WEAK}\ \dfrac{\text{T-CALL}\ \dfrac{\begin{array}{c}\Gamma \vdash \texttt{file} : \bot \qquad \Gamma \vdash \texttt{file.}f_\delta : \bot\\ Q_m[\texttt{file.}f_\delta / x_\delta][\texttt{file}/\mathit{this}] = \{\texttt{file.}f_\delta \sqsubseteq \texttt{file.}f_\delta\}\\ Q'_m[\texttt{tmp}/\mathit{ret}] = \emptyset \qquad \bot \sqsubseteq_\emptyset \Gamma^*(\mathit{this}) = \bot\\ \bot \sqsubseteq_\emptyset \Gamma^*(x_\delta) = \bot \qquad \bot \sqsubseteq_\emptyset pc_m[\texttt{file.}f_\delta / x_\delta][\texttt{file}/\mathit{this}] = \bot\\ \Gamma^*(\mathit{ret}) \sqcup \bot = \Gamma_m(\mathit{ret})[\texttt{file.}f_\delta / x_\delta] \sqcup \bot = \texttt{file.}f_\delta \sqsubseteq_Q \Gamma(\texttt{tmp}) = x_\delta\\ \texttt{tmp} \not\sqsubseteq \bot, Q \qquad \forall f.\ f \neq f_\delta \Rightarrow f \not\sqsubseteq \bot, Q \qquad \texttt{tmp} \neq x_\delta\end{array}}{\Gamma,\bot \vdash \{(\texttt{file.}f_\delta \sqsubseteq \texttt{file.}f_\delta) \cup \emptyset \cup Q\}\ \texttt{tmp:=file.read(file.}f_\delta\texttt{)}\ \{\emptyset \cup Q\}}}{\Gamma,\bot \vdash \{Q\}\ \texttt{tmp:=file.read(file.}f_\delta\texttt{)}\ \{Q\}}$$

The typing rule for the second method call is very similar and thus not given here. Its most important aspect is that the precondition induced by the signature of the `write` method, $Q_m[\texttt{file.}f_\delta / x_\delta][\texttt{srv}/\mathit{this}] = \{\texttt{file.}f_\delta \sqsubseteq \texttt{srv.}f_\delta\}$, is indeed implied by Q. This is the point where the collected flow information from the label test statement is used.

4 Universal Noninterference for a JVM-Like Language

Typically, mobile code is distributed in some bytecode format that is interpreted on the device. To examine policy-aware mobile software realistically, it is therefore necessary to examine a bytecode language that features the runtime inspection of security domains and policies.

For this purpose, this chapter defines the syntax and semantics of a bytecode version of the DSD language. I present a compiler that compiles high-level DSD programs to bytecode programs, and define the universal noninterference property for bytecode programs. The type-based analysis of the security property is then presented in the next chapter.

4.1 The DSD Bytecode Language

Just as the high-level DSD language is similar to a subset of Java, the presented custom bytecode language is similar to a subset of JVM code. This section presents the syntax and semantics of the bytecode language, and defines the compilation from high-level programs to bytecode programs.

4.1.1 Syntax

Bytecode programs are sequences of unstructured instructions that operate on a store (variable valuation), a heap, and an operand stack. The syntax of bytecode instructions is defined as follows:

$$
\begin{array}{rrcl}
\text{numbers:} & n & \in & \mathbb{N} \\
\text{variables:} & x & \in & \mathit{Var} \\
\text{fields:} & f & \in & \mathit{Fld} \\
\text{classes:} & C & \in & \mathit{Cls} \\
\text{methods:} & m & \in & \mathit{Mtd} \\
\text{instruction addresses:} & i, j & \in & \mathit{Adr} \\
\\
\text{constants:} & c & ::= & \top \mid \bot \mid n \\
\text{binary operators:} & op & ::= & \mathbf{op} \mid \sqcup \mid \sqsubseteq \\
\text{instruction syntax:} & \mathit{Instr}_{\mathrm{BC}} \ni \mathtt{B} & ::= & \mathtt{nop} \mid \mathtt{push}\ c \mid \mathtt{pop} \mid \mathtt{prim}\ op \mid \\
 & & & \mathtt{load}\ x \mid \mathtt{store}\ x \mid \\
 & & & \mathtt{new}\ C \mid \mathtt{putf}\ f \mid \mathtt{getf}\ f \mid \mathtt{call}\ m \mid \\
 & & & \mathtt{bnz}\ i \mid \mathtt{jmp}\ i \mid \mathtt{cpush}\ j \mid \mathtt{cjmp}\ j
\end{array}
$$

The instruction set consists of basic instructions for stack, heap, and store manipulations; method calls; as well as conditional and unconditional jumps to other addresses. The two pseudo-instructions `cpush` j and `cjmp` j have the same semantics as `nop` and `jmp` j, respectively; they are used only for the analysis to indicate control dependence regions, and will be described later.

Definition 4.1 *A* DSD bytecode program P_{BC} *is a tuple*

$$(\prec, \mathsf{fields}, \mathsf{methods}, \mathsf{margs}, BC, \mathsf{mentry}, \mathsf{mexit})$$

where

- $\prec \in \mathscr{P}(\mathit{Cls} \times \mathit{Cls})$ *is the subclass relation.*
- $\mathsf{fields} : \mathit{Cls} \to \mathit{Fld}^*$ *and* $\mathsf{methods} : \mathit{Cls} \to \mathit{Mtd}^*$ *assign to each class the identifiers of the fields and methods they contain.*
- $\mathsf{margs} : \mathit{Mtd} \to \mathit{Var}^*$ *is a function that describes the names of the formal arguments of each method* m*.*
- $BC : \mathit{Mtd} \times \mathit{Adr} \rightharpoonup \mathit{Instr}_{\mathrm{BC}}$ *assigns to methods and instruction addresses a bytecode instruction, thereby specifying the method implementations.*
- $\mathsf{mentry}, \mathsf{mexit} : \mathit{Mtd} \to \mathit{Adr}$ *specify the entry point and the exit point of a method.*

A bytecode program thus consists of a number of method bodies (implementations) specified by $BC(m,i)$, which maps instruction addresses to instructions. I sometimes assume the function to be in curried style, and write $BC(m)$ to refer to all instructions that belong to a specific method. To simplify reasoning, each method has its own instruction address space, and may in principle have an arbitrary program layout. The entry point is indicated by $\mathsf{mentry}(m)$. To simplify the semantics, the language does not have a special “return” instruction. Instead, just like in the high-level language, there is a special variable *ret* that holds the return value of a method. The method terminates whenever the execution reaches the (unique) instruction address $\mathsf{mexit}(m)$; therefore, `jmp` $\mathsf{mexit}(m)$ simulates a return instruction. The return value of a method is the value of the *ret* variable at that point.

A bytecode program is well-formed if the following conditions hold:

- All variables that occur free in any bytecode instruction $BC(m,i)$ of a method m are in $\mathsf{margs}(m) \cup \{this, ret\}$.
- For all methods $m \in Mtd$, $\mathsf{mexit}(m) \notin \mathsf{dom}(BC(m))$, that is, there is no instruction at the exit point.
- The other components $\prec$, fields, methods, and margs have the same meaning and the same well-formedness requirements as defined for (high-level) DSD programs.

In the following, we assume a fixed bytecode program P_{BC} whose components are all well-formed.

4.1.2 Semantics

The bytecode language relies on the same kinds of values and identifiers as the high-level language. A bytecode state σ_{BC} is a triple (s,h,ρ) consisting of a store s and a heap h (defined just as for the high-level DSD language), as well as an operand stack ρ, which is a sequence or stack of values. Figure 4.1 on the next page shows a formal definition of the state model.

The operational semantics is defined by a mostly small-step transition relation

$$(m,i_1,s_1,h_1,\rho_1) \xrightarrow{\mathrm{B}}{}^{\diamond} (m,i_2,s_2,h_2,\rho_2)$$

which means that given a domain lattice $\diamond$ and starting from state (s_1,h_1,ρ_1) at instruction address (m,i_1), the instruction B leads to the instruction address (m,i_2) and the new state (s_2,h_2,ρ_2). The quintuple (m,i,s,h,ρ) consisting of a method, an instruction address, and a bytecode state is also called a *bytecode configuration*. While the relation is defined for arbitrary instructions B, it will be linked to the instruction $BC(m,i)$ below.

numbers:	n	$\in$	$\mathbb{N}$
domains:	k	$\in$	$Dom^\diamond$
memory locations:	r	$\in$	Loc
values:	v	$\in$	$Val = \mathbb{N} \cup Dom^\diamond \cup Loc \cup \{null\}$
state:	$\sigma_{\mathrm{BC}} \in State_{\mathrm{BC}}$	:	$Store \times Heap \times Stack$
store:	$s \in Store$	:	$Var \to Val$
heap:	$h \in Heap$	:	$Loc \rightharpoonup (Cls \times (Fld \rightharpoonup Val))$
operand stack:	$\rho \in Stack$	:	Val^*

Figure 4.1: Bytecode program state

The operational bytecode semantics is defined according to the rules in Figure 4.2 on the facing page. Except for jump instructions, all instructions increase the instruction address by 1. The instructions `nop` and `cpush` j have no effect on the state. The instruction `push` c pushes a constant value onto the operand stack, where constants c are interpreted state-independently as follows:

$$[\![\top]\!]^\diamond = k^\diamond_\top \qquad [\![\bot]\!]^\diamond = k^\diamond_\bot \qquad [\![n]\!]^\diamond = n$$

The instruction `pop` removes the top element from the stack, while `prim` op removes the top two values from the stack, performs the operation op on them, and puts the result back onto the stack. `load` x puts the contents of variable x onto the stack, while `store` x stores the top element of the stack into x and removes it from the stack. `bnz` j removes the top element v and branches to address j in case v is not zero. `jmp` j and `cjmp` j always jump to j. The instructions `getf` f and `putf` f read and update the field of an object, respectively, where the object reference is popped from the stack. The instruction `new` C creates a new object in the heap, initializes the field with values removed from the stack, and pushes the reference to the new object onto the stack. Finally, the `call` m instruction invokes a method m. As in the high-level language, the constant defval = $null$ specifies a default value for the return variable ret.

The rule for method calls relies on a big-step execution of the method. More formally, there is a big-step execution

$$(m, i_0, s_0, h_0, \rho_0) \underset{BC}{\Longrightarrow}^\diamond (m, i_k, s_k, h_k, \rho_k)$$

if and only if there is a (possibly empty) sequence of small execution steps such that $(m, i_0, s_0, h_0, \rho_0) \xrightarrow{BC(m,i_0)}{}^\diamond (m, i_1, s_1, h_1, \rho_1) \xrightarrow{BC(m,i_1)}{}^\diamond \ldots \xrightarrow{BC(m,i_{k-1})}{}^\diamond (m, i_k, s_k, h_k, \rho_k)$.

$$\frac{\mathtt{B} = \mathtt{nop}}{(m,i,s,h,\rho) \xrightarrow{\mathtt{B}}{}^{\diamond} (m,i+1,s,h,\rho)} \qquad \frac{\mathtt{B} = \mathtt{push}\ c}{(m,i,s,h,\rho) \xrightarrow{\mathtt{B}}{}^{\diamond} (m,i+1,s,h,[\![c]\!]^{\diamond} :: \rho)}$$

$$\frac{\mathtt{B} = \mathtt{pop}}{(m,i,s,h,v :: \rho) \xrightarrow{\mathtt{B}}{}^{\diamond} (m,i+1,s,h,\rho)}$$

$$\frac{\mathtt{B} = \mathtt{prim}\ op}{(m,i,s,h,v_2 :: v_1 :: \rho) \xrightarrow{\mathtt{B}}{}^{\diamond} (m,i+1,s,h,(v_1\ \mathsf{op}\ v_2) :: \rho)}$$

$$\frac{\mathtt{B} = \mathtt{load}\ x}{(m,i,s,h,\rho) \xrightarrow{\mathtt{B}}{}^{\diamond} (m,i+1,s,h,s(x) :: \rho)}$$

$$\frac{\mathtt{B} = \mathtt{store}\ x}{(m,i,s,h,v :: \rho) \xrightarrow{\mathtt{B}}{}^{\diamond} (m,i+1,s[x \mapsto v],h,\rho)} \qquad \frac{\mathtt{B} = \mathtt{bnz}\ j \qquad v = 0}{(m,i,s,h,v :: \rho) \xrightarrow{\mathtt{B}}{}^{\diamond} (m,i+1,s,h,\rho)}$$

$$\frac{\mathtt{B} = \mathtt{bnz}\ j \qquad v \neq 0}{(m,i,s,h,v :: \rho) \xrightarrow{\mathtt{B}}{}^{\diamond} (m,j,s,h,\rho)} \qquad \frac{\mathtt{B} \in \{\mathtt{jmp}\ j, \mathtt{cjmp}\ j\}}{(m,i,s,h,\rho) \xrightarrow{\mathtt{B}}{}^{\diamond} (m,j,s,h,\rho)}$$

$$\frac{\mathtt{B} = \mathtt{cpush}\ j}{(m,i,s,h,\rho) \xrightarrow{\mathtt{B}}{}^{\diamond} (m,i+1,s,h,\rho)} \qquad \frac{\mathtt{B} = \mathtt{putf}\ f \qquad h' = h[r \mapsto h(r)[f \mapsto v]]}{(m,i,s,h,v :: r :: \rho) \xrightarrow{\mathtt{B}}{}^{\diamond} (m,i+1,s,h',\rho)}$$

$$\frac{\mathtt{B} = \mathtt{getf}\ f}{(m,i,s,h,r :: \rho) \xrightarrow{\mathtt{B}}{}^{\diamond} (m,i+1,s,h,h(r)(f) :: \rho)}$$

$$\frac{\begin{array}{c}\mathtt{B} = \mathtt{new}\ C \qquad r \notin \mathsf{dom}(h) \\ h' = h \cup [r \mapsto (C, [\mathsf{fields}(C) \mapsto \overline{v}])]\end{array}}{(m,i,s,h,\overline{v} :: \rho) \xrightarrow{\mathtt{B}}{}^{\diamond} (m,i+1,s,h',r :: \rho)}$$

$$\frac{\begin{array}{c}\mathtt{B} = \mathtt{call}\ m' \\ s' = [\mathit{this} \mapsto r] \cup [\mathsf{margs}(m') \mapsto \overline{v}] \cup [\mathit{ret} \mapsto \mathsf{defval}] \\ (s',h) \overset{m'}{\underset{BC}{\Longrightarrow}}{}^{\diamond} (s'',h')\end{array}}{(m,i,s,h,\overline{v} :: r :: \rho) \xrightarrow{\mathtt{B}}{}^{\diamond} (m,i+1,s,h',s''(\mathit{ret}) :: \rho)}$$

Figure 4.2: Mostly small-step semantics for bytecode

The following notation is a shorthand for the execution of entire method bodies:

$$(s_1, h_1) \overset{m}{\underset{BC}{\Longrightarrow}}^{\diamond} (s_2, h_2) \iff \exists \rho.\, (m, \mathsf{mentry}(m), s_1, h_1, \epsilon) \underset{BC}{\Longrightarrow}^{\diamond} (m, \mathsf{mexit}(m), s_2, h_2, \rho)$$

Thus, methods are called with an empty operand stack, and the shape of the stack at the return point is unconstrained. Note that the transition essentially works on *high-level* program states (store-heap pairs).

I write $(m, i_1) \overset{\mathrm{B}}{\longmapsto} (m, i_2)$ if there are stores, heaps, and stacks, and a lattice domain $\diamond$ such that $(m, i_1, s_1, h_1, \rho_1) \overset{\mathrm{B}}{\longrightarrow}^{\diamond} (m, i_2, s_2, h_2, \rho_2)$. The set of successors of an instruction is defined as $\mathsf{succ}(m, i) = \{j \mid (m, i) \overset{\mathrm{B}}{\longmapsto} (m, j)\}$. Jump targets of a method are those instruction addresses that can be reached by a conditional or unconditional jump instruction.

Definition 4.2 (Jump targets) *The set of* jump targets *within a method m of a bytecode program BC, written* jmpTgt_m^{BC}*, is defined as:*

$$\mathsf{jmpTgt}_m^{BC} = \{j \mid \exists i.\, BC(m, i) \in \{\mathtt{jmp}\ j, \mathtt{bnz}\ j, \mathtt{cjmp}\ j\}\}$$

The operational semantics is mostly small-step, because method calls are defined with respect to the big-step execution of the method body, which in turn is defined in terms of small steps. The execution gets stuck if a successor instruction is not defined in $BC(m)$. This does not affect the security property, as universal noninterference is termination-insensitive.

4.2 Compilation to Bytecode

I define a compilation function $\mathsf{compile}(P_{\mathrm{DSD}})$ that compiles a high-level DSD program P_{DSD} to the corresponding bytecode program P_{BC} as follows:

$$\mathsf{compile}(\prec, \mathsf{fields}, \mathsf{methods}, \mathsf{margs}, \mathsf{mbody}) = (\prec, \mathsf{fields}, \mathsf{methods}, \mathsf{margs}, BC, \mathsf{mentry}, \mathsf{mexit})$$

such that for all methods $m \in \mathsf{dom}(\mathsf{mbody})$,

$$\begin{aligned} \mathsf{mentry}(m) &= 0 \\ (BC(m), \mathsf{mexit}(m)) &= \mathsf{compile}_{\mathsf{stmt}}(m, \mathsf{mbody}(m), 0) \end{aligned}$$

In other words, every high-level method body $\mathsf{mbody}(m)$ is transformed into the corresponding list of instructions, such that each method body starts at address 0. The argument list $\mathsf{margs}(m)$ for each method, as well as the structural information $\prec$, fields, and methods, are unchanged from the high-level program.

Statement compilation Statements of the high-level DSD language are compiled to bytecode sequences using a function

$$(BC, i_1) = \text{compile}_{\text{stmt}}(m, S, i_0)$$

as defined in Figure 4.3 on the next page. It takes a statement S with an address i_0, and returns a bytecode program BC (mappings of addresses to instructions) as well as an address i_1, such that BC contains the compiled instructions corresponding to the statement, starting from i_0 and up to (but excluding) i_1. The function proceeds recursively on substatements.

For the compilation of conditionals, the aforementioned pseudo-instructions are inserted. A conditional statement is compiled to a sequence of bytecode instructions that starts with an instruction `cpush` $i_1 + 1$. Here, $i_1 + 1$ is the first address that follows the compiled sequence – that is, the confluence point that the two branches eventually reach. At the end of each branch, there is an instruction `cjmp` $i_1 + 1$ that causes a jump to this confluence point. The pseudo-instructions thus mark where the entire conditional statement starts in the bytecode, and where the subbranches end. Though having no special meaning in the semantics, this information is required by the type system to correctly determine the extent of control dependence regions where an indirect flow of information may occur, such that the *pc* label can be constrained accordingly. This approach follows the work by Medel, Compagnoni, and Bonelli [MCB05]; see Section 5.4 on page 75 in this section for a detailed motivation for the use of these pseudo-instructions.

The compilation of a loop statement **while** e **do** S corresponds to the compilation of an (imaginary) conditional statement **if** e **then** (**do** S **while** e) **else skip**, with the pseudo-instructions inserted as described above for conditionals. The reason for this unusual form lies in the type system: a more direct compilation of **while** e **do** S would not be typable. The compilation presented here produces a somewhat larger bytecode program, but in a way such that the execution time is not affected.

Expression compilation The compilation function $\text{compile}_{\text{stmt}}(m, S, i_0)$ relies on another function to compile expressions,

$$(BC, i_1) = \text{compile}_{\text{exp}}(m, e, i_0)$$

as defined in Figure 4.4 on page 59. The expression compilation function returns a bytecode program BC with instructions starting at i_0, and ending at i_1. The instructions compute the value of the expression e, such that the value can be found on top of the stack after the execution of the instructions. For method calls, the function is extended point-wise to sequences $\text{compile}_{\text{exp}}(m, \overline{e}, i_0)$, such that every expression in $\overline{e}$ is compiled in the reverse order in which values are removed from the operand stack in the small-step semantics for method calls.

$$
\begin{array}{l}
\mathsf{compile}_{\mathrm{stmt}}(m, \mathbf{skip}, i) \quad = \\
\quad [\,], i \\[1ex]
\mathsf{compile}_{\mathrm{stmt}}(m, S_1\,;\,S_2, i) \\
\quad \mathsf{let}\ BC_1, i_1 = \mathsf{compile}_{\mathrm{stmt}}(m, S_1, i)\ \mathsf{in} \\
\quad \mathsf{let}\ BC_2, i_2 = \mathsf{compile}_{\mathrm{stmt}}(m, S_2, i_1)\ \mathsf{in} \\
\quad BC_1 \cup BC_2, i_2 \\[1ex]
\mathsf{compile}_{\mathrm{stmt}}(m, x := e, i) \quad = \\
\quad \mathsf{let}\ BC, i' = \mathsf{compile}_{\mathrm{exp}}(m, e, i)\ \mathsf{in} \\
\quad BC \cup [(m, i') \mapsto \mathtt{store}\ x], i' + 1 \\[1ex]
\mathsf{compile}_{\mathrm{stmt}}(m, e_r.f := e, i) \quad = \\
\quad \mathsf{let}\ BC_r, i_r = \mathsf{compile}_{\mathrm{exp}}(m, e_r, i)\ \mathsf{in} \\
\quad \mathsf{let}\ BC, i' = \mathsf{compile}_{\mathrm{exp}}(m, e, i_r)\ \mathsf{in} \\
\quad BC_r \cup BC \cup [(m, i') \mapsto \mathtt{putf}\ f], i' + 1 \\[1ex]
\mathsf{compile}_{\mathrm{stmt}}(m, x := \mathbf{new}\ C(\overline{e}), i) \quad = \\
\quad \mathsf{let}\ BC, i' = \mathsf{compile}_{\mathrm{exp}}(m, \overline{e}, i)\ \mathsf{in} \\
\quad [(m, i') \mapsto \mathtt{new}\ C] \cup [(m, i'+1) \mapsto \mathtt{store}\ x], i + 2 \\[1ex]
\mathsf{compile}_{\mathrm{stmt}}(m, x := e_r.m(\overline{e}), i) \quad = \\
\quad \mathsf{let}\ BC_r, i_r = \mathsf{compile}_{\mathrm{exp}}(m, e_r, i)\ \mathsf{in} \\
\quad \mathsf{let}\ BC, i' = \mathsf{compile}_{\mathrm{exp}}(m, \overline{e}, i_r)\ \mathsf{in} \\
\quad BC_r \cup BC \cup [(m, i') \mapsto \mathtt{call}\ m] \cup [(m, i'+1) \mapsto \mathtt{store}\ x], i' + 2 \\[1ex]
\mathsf{compile}_{\mathrm{stmt}}(m, \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2, i) \quad = \\
\quad \mathsf{let}\ BC, i' = \mathsf{compile}_{\mathrm{exp}}(m, e, i+1)\ \mathsf{in} \\
\quad \mathsf{let}\ BC_2, i_2 = \mathsf{compile}_{\mathrm{stmt}}(m, S_2, i'+1)\ \mathsf{in} \\
\quad \mathsf{let}\ BC_1, i_1 = \mathsf{compile}_{\mathrm{stmt}}(m, S_1, i_2+1)\ \mathsf{in} \\
\quad [(m, i) \mapsto \mathtt{cpush}\ i_1 + 1] \cup BC \cup [(m, i') \mapsto \mathtt{bnz}\ i_2 + 1] \cup \\
\quad BC_2 \cup [(m, i_2) \mapsto \mathtt{cjmp}\ i_1 + 1] \cup \\
\quad BC_1 \cup [(m, i_1) \mapsto \mathtt{cjmp}\ i_1 + 1], i_1 + 1 \\[1ex]
\mathsf{compile}_{\mathrm{stmt}}(m, \mathbf{while}\ e\ \mathbf{do}\ S, i) \quad = \\
\quad \mathsf{let}\ BC_1, i_1 = \mathsf{compile}_{\mathrm{exp}}(m, e, i+1)\ \mathsf{in} \\
\quad \mathsf{let}\ BC, i' = \mathsf{compile}_{\mathrm{stmt}}(m, S, i_1+2)\ \mathsf{in} \\
\quad \mathsf{let}\ BC_2, i_2 = \mathsf{compile}_{\mathrm{exp}}(m, e, i')\ \mathsf{in} \\
\quad [(m, i) \mapsto \mathtt{cpush}\ i_2 + 2] \cup BC_1 \cup [(m, i_1) \mapsto \mathtt{bnz}\ i_1 + 2] \cup \\
\quad [(m, i_1 + 1) \mapsto \mathtt{cjmp}\ i_2 + 2] \cup \\
\quad BC \cup BC_2 \cup [(m, i_2) \mapsto \mathtt{bnz}\ i_1 + 2] \cup [(m, i_2 + 1) \mapsto \mathtt{cjmp}\ i_2 + 2], i_2 + 2
\end{array}
$$

Figure 4.3: Compilation of statements

$$
\begin{aligned}
\mathsf{compile}_{\mathsf{exp}}(m, c, i) &= [(m, i) \mapsto \texttt{push}\ c], i+1 \\
\mathsf{compile}_{\mathsf{exp}}(m, x, i) &= [(m, i) \mapsto \texttt{load}\ x], i+1 \\
\mathsf{compile}_{\mathsf{exp}}(m, e.f, i) &= \mathsf{let}\ BC, i' = \mathsf{compile}_{\mathsf{exp}}(m, e, i)\ \mathsf{in} \\
&\quad\ BC \cup [(m, i') \mapsto \texttt{getf}\ f], i'+1 \\
\mathsf{compile}_{\mathsf{exp}}(m, e_1\ \mathbf{op}\ e_2, i) &= \mathsf{let}\ BC_1, i_1 = \mathsf{compile}_{\mathsf{exp}}(m, e_1, i)\ \mathsf{in} \\
&\quad\ \mathsf{let}\ BC_2, i_2 = \mathsf{compile}_{\mathsf{exp}}(m, e_2, i_1)\ \mathsf{in} \\
&\quad\ BC_1 \cup BC_2 \cup [(m, i_2) \mapsto \texttt{prim}\ op], i_2+1
\end{aligned}
$$

Figure 4.4: Compilation of expressions

Finally, we observe two properties of the bytecode compilation. First, expressions and statements always compile to contiguous bytecode instruction sequences. Also, compiled expressions do not contain any jump targets, whereas compiled statements contain only jump targets within their compiled bytecode sequence, or at the address immediately following the sequence.

Proposition 4.3 *Compiled expressions and statements have the following properties:*

- *If* $(BC, i_1) = \mathsf{compile}_{\mathsf{exp}}(m, e, i_0)$*, then* $\mathsf{dom}(BC(m)) = [i_0, i_1[$ *and* $\mathsf{jmpTgt}_m^{BC} = \emptyset$.
- *If* $(BC, i_1) = \mathsf{compile}_{\mathsf{stmt}}(m, S, i_0)$*, then* $\mathsf{dom}(BC(m)) = [i_0, i_1[$*, and* $\mathsf{jmpTgt}_m^{BC} \subseteq [i_0, i_1]$.

PROOF By induction on the structure of e or S. This follows directly from the definition of $\mathsf{compile}_{\mathsf{exp}}(m, e, i)$ or $\mathsf{compile}_{\mathsf{stmt}}(m, S, i)$. □

Compilation of the example program The example program, which I have defined as a DSD program $P_{\mathrm{DSD}} = (\prec, \mathsf{fields}, \mathsf{methods}, \mathsf{margs}, \mathsf{mbody})$ in Section 2.2 on page 22, is compiled into a bytecode program

$$P_{\mathrm{BC}} = (\prec, \mathsf{fields}, \mathsf{methods}, \mathsf{margs}, BC, \mathsf{mentry}, \mathsf{mexit})$$

where most of the program structure is retained. The only method that has been given a DSD implementation, `sendFile`, is compiled to a corresponding bytecode implementation such that mentry(`sendFile`) = 0, mexit(`sendFile`) = 22, and BC(`sendFile`) is defined as shown in Figure 4.5 on the next page. In the figure, the highlighted regions show how particular expressions and statements correspond to sequences of instructions in the compilation. Note that there is no instruction at the exit point of the method (address 22).

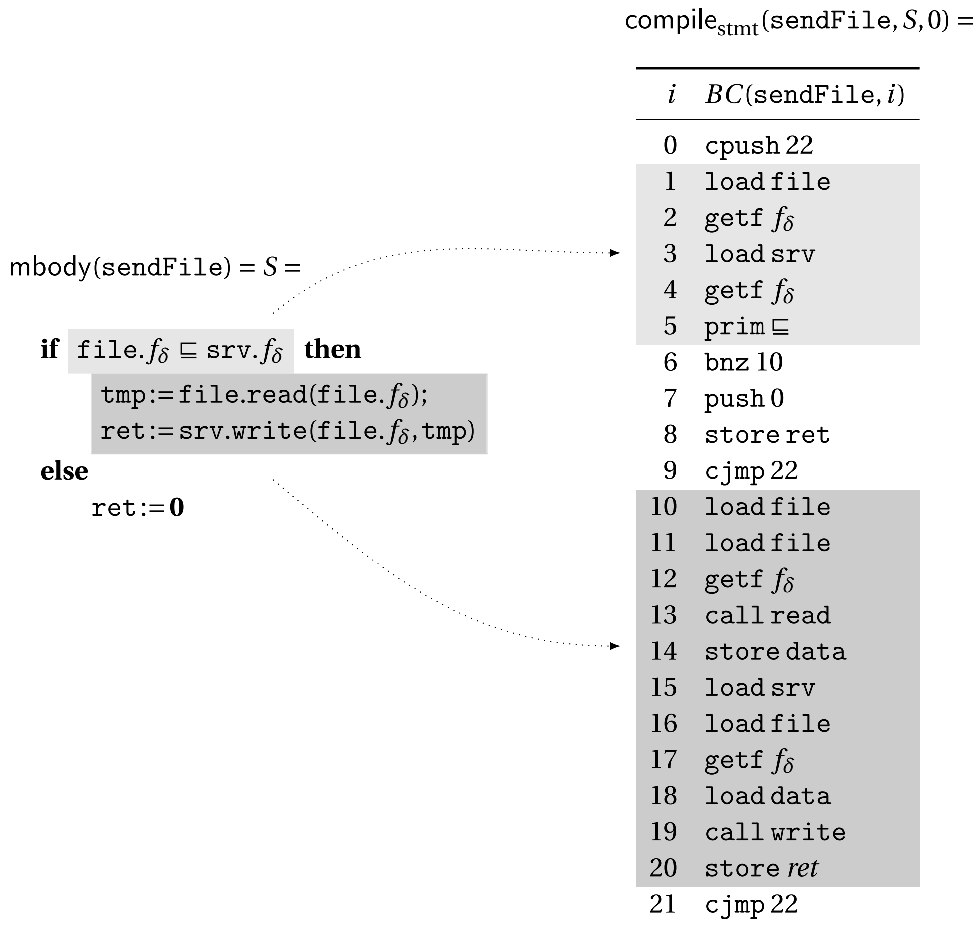

Figure 4.5: Compilation of the `sendFile` method of the example program

4.3 Universal Noninterference

In this section, I formally define universal noninterference for bytecode programs.

For each method m, I assume there is a given method signature

$$\mathsf{msig}(m) = [\Gamma, pc, Q, Q']$$

exactly as defined in the high-level language. In particular, we retain the definitions for type environments Γ and Φ, for access paths π, for labels $pc, \ell \in Lab$, and for constraint sets Q. As before, we assume the field typing Φ to be fixed.

I reuse the state satisfiability relation $\sigma \models^{\diamond} Q$ as well as store, heap, and (high-level) state equivalences

$$\vdash^{\diamond} s \sim_{\beta}^{\Gamma,k} s' \quad \text{and} \quad \vdash^{\diamond} h \sim_{\beta}^{k} h' \quad \text{and} \quad \vdash^{\diamond} (s,h) \sim_{\beta}^{\Gamma,k} (s',h')$$

defined as in the high-level language (see Section 2.3 on page 23).

As noted before, the operand stack is neither passed to methods nor returned, and thus folded away in the big-step transition for method body executions. Due to this fact and thanks to the close relationship between high-level and bytecode program states, universal noninterference for bytecode programs can be formulated identically to universal noninterference for high-level programs. The only difference is that I now refer to the bytecode program execution, of course.

Definition 4.4 *Let P_{BC} be a bytecode program. A method m is* universally noninterferent *with respect to its signature* $\mathsf{msig}(m) = [\Gamma, pc, Q_1, Q_2]$ *if for all domain lattices* $\diamond$*, for all domains* $k \in Dom^{\diamond}$*, for all store-heap pairs* $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$ *and for all bijections* β *such that*

- $\sigma_1 \models^{\diamond} Q_1$ *and* $\sigma_1' \models^{\diamond} Q_1$,
- $\vdash^{\diamond} \sigma_1 \sim_{\beta}^{\Gamma,k} \sigma_1'$,
- $\sigma_1 \overset{m}{\underset{BC}{\Longrightarrow}}{}^{\diamond} \sigma_2$ *and* $\sigma_1' \overset{m}{\underset{BC}{\Longrightarrow}}{}^{\diamond} \sigma_2'$,

there exists a bijection $\beta' \supseteq \beta$ *such that*

- $\sigma_2 \models^{\diamond} Q_2$ *and* $\sigma_2' \models^{\diamond} Q_2$, *and*
- $\vdash^{\diamond} \sigma_2 \sim_{\beta'}^{\Gamma,k} \sigma_2'$.

A bytecode program P_{BC} *is* universally noninterferent *if all its methods are universally noninterferent with respect to their signatures.*

5
Low-Level Type System

In this chapter, I present an analysis for the compiled low-level code. However, I do not give a type system for bytecode programs directly. As already mentioned in Section 1.5 on page 11, I first transform the bytecode program to an *intermediate representation* (IR) that does not require an operand stack, but instead relies on additional temporary variables. The transformation is a partial decompilation which disassembles stack-manipulating instructions back into high-level expressions.

I then define universal noninterference for IR programs, and provide a type system along with a soundness proof. Thanks to the translation to IR code, the proof is relatively short, because it can rely on existing soundness results from the high-level language for decompiled expressions. Also, the intermediate representation makes reasoning on information flow properties much easier.

Another proof shows that the translation to the intermediate representation preserves the semantics of the bytecode. It follows that whenever a bytecode program can be translated to an IR program that is typable, then the original bytecode program is universally noninterferent. This approach of indirectly showing bytecode properties by using a stackless intermediate representation builds on the work by Demange, Jensen, and Pichardie [DJP10].

An important aspect of the approach is that the parallels between the high-level language and the intermediate representation are only exploited to simplify the low-level analysis and the corresponding correctness proofs. There is no dependency on the result of the high-level program analysis, and almost no requirement for the specific compiler that is used by the programmer.

This suits the application scenario outlined in the introduction: The verification of the source code by the software developer is independent of the verification of the bytecode by the app store provider. The software developer may switch to a different high-level code analysis or to a different compiler at any time without affecting the ability of the distribution platform maintainer to verify the bytecode program. Also, there are no requirements for the format of the submitted bytecode code, as the translation to IR code happens only within the bytecode analysis.

Nevertheless, I give a type preservation result for the specific DSD compiler presented in the previous chapter. It shows that all bytecode programs produced by the compiler can be transformed into an IR program, such that if the original high-level program was typable with the high-level type system, then the corresponding IR program is typable with the IR type system. This gives the programmer the confidence that typable DSD programs will always be compiled to bytecode programs that are universally noninterferent. Note, however, that neither the type preservation result nor the specific compiler are required to certify bytecode for universal noninterference.

The Grail intermediate language [BMS03] serves a similar purpose as the intermediate representation: it is used to transfer results from a high-level resource analysis to simplify the analysis of low-level code. In contrast to the presented approach, however, the compiler of the Grail framework generates the Grail code directly from the high-level language, and additionally translates the high-level proof to a proof for the corresponding Grail code. To make this approach work in the application scenario presented here, one has to make considerable changes to existing code distribution environments and the execution platform.

5.1 Intermediate Representation

The intermediate language is located conceptually between the high-level DSD language and the bytecode language. While the language is – like bytecode – an unstructured language with instructions that are executed in a small-step fashion, the instructions may become rather complex and resemble high-level DSD statements. Consequently, the syntax and semantics of the IR language are a mixture of the high-level and bytecode syntax and semantics presented in the preceding chapters.

5.1.1 Syntax

Apart from ordinary program variables from a set *Var*, IR programs may also contain *temporary variables* t from a set *TVar* that is disjoint from *Var*. These temporary variables are used to store the results of computations that would be put as values on the stack on the bytecode level.

The syntax of IR instructions is defined as follows:

temporary variables:	t	$\in$	$TVar$
variables:	x	$\in$	$Var \cup TVar$
fields:	f	$\in$	Fld
classes:	C	$\in$	Cls
methods:	m	$\in$	Mtd
instruction addresses:	i, j	$\in$	Adr

instruction syntax:	$Instr_{\mathrm{IR}} \ni \mathtt{I}$	::=	$\mathtt{if}\ e\ j \mid \mathtt{jmp}\ j \mid \mathtt{cpush}\ j \mid \mathtt{cjmp}\ j \mid \mathtt{block}\ \overline{a}$
assignments:	$Assn \ni a$	::=	$x := e \mid e.f := e \mid x := \mathbf{new}\ C(\overline{e}) \mid x := e.m(\overline{e})$
expressions:	$Exp \ni e$	::=	(as defined in the high-level language)

The instruction set consists of:

- blocks, which are sequences of assignment statements, similar to those from the high-level language;
- conditional and unconditional jumps, similar to those on the bytecode level, but with the conditional expression restored; and
- pseudo-instructions that indicate control dependence regions, with the same meaning as on the bytecode level.

The set of IR instructions is much smaller than the bytecode instruction set; most functionality is provided by the assignment block instruction $\mathtt{block}\ \overline{a}$. An occurrence of x in assignments a and high-level expressions e may refer to either an ordinary variable or a temporary variable.

IR programs are defined very similarly to bytecode programs.

Definition 5.1 *An* IR program P_{IR} *is a tuple*

$$(\prec, \mathsf{fields}, \mathsf{methods}, \mathsf{margs}, \mathsf{tvars}, IR, \mathsf{mentry}, \mathsf{mexit})$$

where

- $\prec \in \mathscr{P}(Cls \times Cls)$ *is the subclass relation.*
- $\mathsf{fields} : Cls \rightarrow Fld^*$ *and* $\mathsf{methods} : Cls \rightarrow Mtd^*$ *assign to each class the identifiers of the fields and methods they contain.*
- $\mathsf{margs} : Mtd \rightarrow Var^*$ *is a function that describes the names of the formal arguments of each method* m.

- $\mathsf{tvars} : Mtd \to TVar^*$ *specifies for each method the temporary variables that may occur in the body of the method.*
- $IR : Mtd \times Adr \rightharpoonup Instr_{\mathrm{IR}}$ *assigns to methods and instruction addresses an IR instruction, thereby specifying the method implementations.*
- $\mathsf{mentry}, \mathsf{mexit} : Mtd \to Adr$ *specify the entry point and the exit point of methods.*

The difference to bytecode programs is that IR method bodies contain IR instructions, of course, and that IR methods may also contain temporary variables specified by tvars. Apart from that, the meaning and well-formedness requirements are the same as for bytecode programs. In the following, let us assume a given fixed IR program P_{IR} whose components are all well-formed.

5.1.2 Semantics

An IR state σ_{IR} is a triple (s, s_t, h) where h is a heap, and s and s_t are two variable stores, one for ordinary and one for temporary variables. The definition of stores, heaps, and values are exactly as for bytecode and high-level programs.

$$\begin{array}{rrcl}
\text{values:} & v \in Val & : & \text{(as before)} \\
\text{state:} & \sigma_{\mathrm{IR}} \in State_{\mathrm{IR}} & : & Store \times TStore \times Heap \\
\text{store:} & s \in Store & : & Var \to Val \\
\text{temporary store:} & s_t \in TStore & : & TVar \to Val \\
\text{heap:} & h \in Heap & : & Loc \rightharpoonup (Cls \times (Fld \rightharpoonup Val))
\end{array}$$

Note that $s \cup s_t$ always forms a valid store, because $TVar$ is disjoint from Var. For the same reason, it is always possible to split a combined store $s \cup s_t$ back into s and s_t. All expressions e that occur in statements are evaluated using an evaluation function that is defined in terms of the high-level expression evaluation:

$$[\![e]\!]^{\diamond}_{s,s_t,h} = [\![e]\!]^{\diamond}_{s \cup s_t,h}$$

In the following, I define five mutually recursive transition relations, listed in the following table:

small-step transition for IR instructions:	(m, i, s, s_t, h)	$\xrightarrow{\mathrm{I}}{}^{\diamond}$	(m, i', s', s'_t, h')
big-step transition for IR instructions:	(m, i, s, s_t, h)	$\underset{IR}{\Longrightarrow}{}^{\diamond}$	(m, i', s', s'_t, h')
big-step transition for IR methods:	(s, h)	$\overset{m}{\underset{IR}{\Longrightarrow}}{}^{\diamond}$	(s', h')
small-step transition for IR assignments:	(s, s_t, h)	$\xrightarrow{a}{}^{\diamond}$	(s', s'_t, h')
big-step transition for IR assignments:	(s, s_t, h)	$\overset{\overline{a}}{\Longrightarrow}{}^{\diamond}$	(s', s'_t, h')

$$\frac{\text{I} = \texttt{if}\ e\ j \qquad [\![e]\!]^{\diamond}_{s,s_t,h} = 0}{(m,i,s,s_t,h) \xrightarrow{\text{I}}{}^{\diamond} (m,i+1,s,s_t,h)} \qquad \frac{\text{I} = \texttt{if}\ e\ j \qquad [\![e]\!]^{\diamond}_{s,s_t,h} \neq 0}{(m,i,s,s_t,h) \xrightarrow{\text{I}}{}^{\diamond} (m,j,s,s_t,h)}$$

$$\frac{\text{I} \in \{\texttt{jmp}\ j, \texttt{cjmp}\ j\}}{(m,i,s,s_t,h) \xrightarrow{\text{I}}{}^{\diamond} (m,j,s,s_t,h)} \qquad \frac{\text{I} = \texttt{cpush}\ j}{(m,i,s,s_t,h) \xrightarrow{\text{I}}{}^{\diamond} (m,i+1,s,s_t,h)}$$

$$\frac{\text{I} = \texttt{block}\ \overline{a} \qquad (s,s_t,h) \overset{\overline{a}}{\Rightarrow}{}^{\diamond} (s',s'_t,h)}{(m,i,s,s_t,h) \xrightarrow{\text{I}}{}^{\diamond} (m,i+1,s',s'_t,h')}$$

Figure 5.1: Mostly small-step semantics for IR instructions

$$\frac{\begin{array}{c} S = a \in \{x := e, e.f := e, x := \textbf{new}\ C(\overline{e})\} \\ (s \cup s_t, h) \xrightarrow{S}{}^{\diamond} (s' \cup s'_t, h') \end{array}}{(s,s_t,h) \xrightarrow{a}{}^{\diamond} (s',s'_t,h')}$$

$$\frac{\begin{array}{c} a = x := e.m(\overline{e}) \qquad [\![e]\!]^{\diamond}_{s,s_t,h} = r \qquad [\![\overline{e}]\!]^{\diamond}_{s,s_t,h} = \overline{v} \\ s_m = [\mathit{this} \mapsto r] \cup [\mathsf{margs}(m) \mapsto \overline{v}] \cup [\mathit{ret} \mapsto \mathsf{defval}] \\ (s_m,h) \underset{IR}{\overset{m}{\Longrightarrow}}{}^{\diamond} (s'_m,h') \qquad s' \cup s'_t = (s \cup s_t)[x \mapsto s'_m(\mathit{ret})] \end{array}}{(s,s_t,h) \xrightarrow{a}{}^{\diamond} (s',s'_t,h')}$$

Figure 5.2: Semantics for IR assignments

The rules in Figure 5.1 on the preceding page define a (mostly) small-step operational semantics for single IR instructions

$$(m, i, s, s_t, h) \xrightarrow{\text{I}}{}^{\diamond} (m, i', s', s'_t, h')$$

which means that given a domain lattice $\diamond$ and starting from state (s, s_t, h) at instruction address (m, i), the instruction I leads to instruction address (m, i') and new state (s', s'_t, h'). A special relation for assignment blocks is defined below.

While the relation is defined for arbitrary instructions I, it is linked to the instruction $IR[m, i]$ via a big-step transition, which is defined similar to the bytecode big-step semantics:

$$(m, i^0, s^0, s^0_t, h^0) \underset{IR}{\Longrightarrow}{}^{\diamond} (m, i^k, s^k, s^k_t, h^k)$$

if and only if there is a (possibly empty) sequence of small execution steps such that $(m, i^0, s^0, s^0_t, h^0) \xrightarrow{IR[m,i^0]}{}^{\diamond} (m, i^1, s^1, s^1_t, h^1) \xrightarrow{IR[m,i^1]}{}^{\diamond} \dots \xrightarrow{IR[m,i^{k-1}]}{}^{\diamond} (m, i^k, s^k, s^k_t, h^k)$. I use the following notation is a shorthand for the execution of entire method bodies:

$$(s, h) \underset{IR}{\overset{m}{\Longrightarrow}}{}^{\diamond} (s', h') \quad \text{if and only if}$$

$$\exists s_t.\ (m, \mathsf{mentry}(m), s, [\mathsf{tvars}(m) \mapsto \mathsf{defval}], h) \underset{IR}{\Longrightarrow}{}^{\diamond} (m, \mathsf{mexit}(m), s', s_t, h')$$

The small-step rule for assignment blocks relies on a transition

$$(s^0, s^0_t, h^0) \overset{\overline{a}}{\Rightarrow}{}^{\diamond} (s^k, s^k_t, h^k)$$

that executes the assignments in $\overline{a}$ in order — that is, there must exist a sequence of execution steps $(s^0, s^0_t, h^0) \xrightarrow{\overline{a}.1}{}^{\diamond} (s^1, s^1_t, h^1) \xrightarrow{\overline{a}.2}{}^{\diamond} \dots \xrightarrow{\overline{a}.k}{}^{\diamond} (s^k, s^k_t, h^k)$ such that $|\overline{a}| = k$. For each individual assignment, in turn, a transition $(s, s_t, h) \xrightarrow{a}{}^{\diamond} (s', s'_t, h')$ is used, defined by the rules in Figure 5.2 on the previous page. IR assignments have exactly the semantics as assignment statements in the DSD language; in fact, I directly use the operational semantics of DSD for variable updates, field updates, and object creation assignments. The only difference is that IR method calls initialize all temporary variables of a method with the default value defval, and use

Following the notation for bytecode execution successors, I write $(m, i) \overset{\text{I}}{\longmapsto} (m, i')$ if there are stores, temporary stores, and heaps such that $(m, i, s, s_t, h) \xrightarrow{\text{I}}{}^{\diamond} (m, i', s', s'_t, h')$. The set of successors of an instruction is defined as $\mathsf{succ}(m, i) = \{i' \mid (m, i) \overset{\text{I}}{\longmapsto} (m, i')\}$.

5.2 Universal Noninterference

This section defines universal noninterference for IR programs, which is very similar to the preceding definitions for bytecode and for high-level programs.

Type Environments and Method Signatures For the IR language, variable type environments Γ and the fixed field type environment Φ are defined exactly as for the bytecode and the high-level languages. For the intermediate representation, however, there is another variable typing Γ_t for temporary variables:

$$\text{temporary variable typing:} \quad \Gamma_t \quad : \quad TVar \rightharpoonup \{\top, \bot, x_\delta\}$$

I assume that the types of the temporary variables in each method are given by a temporary variable typing Γ_t that belongs to the method. This type environment, however, is only used for method-internal type derivations. As no data is passed via temporary variables at method calls, Γ_t is not included in the signature of a method.

Instead, I assume each method m has a well-formed signature

$$\mathsf{msig}(m) = [\Gamma, pc, Q, Q']$$

that is defined exactly as in the high-level language and bytecode languages. In particular, labels $pc, \ell \in Lab$ and constraint sets Q are defined as before. Moreover, I reuse the state satisfiability relation $\sigma \models^\diamond Q$ as well as store and heap equivalences:

$$\vdash^\diamond s \sim_\beta^{\Gamma,k} s' \quad \text{and} \quad \vdash^\diamond h \sim_\beta^{k} h' \quad \text{and} \quad \vdash^\diamond (s,h) \sim_\beta^{\Gamma,k} (s',h').$$

Universal Noninterference As noted above, no data is passed in temporary variables to methods, hence the stores s_t can be folded away in the big-step transition for method body executions. Thanks to the close correspondence of the definitions and notations across the languages, the definition of universal noninterference for IR programs is identical to the one for bytecode and high-level programs; the only difference is that it refers to IR method executions.

Definition 5.2 *A method m is* universally noninterferent *with respect to its signature* $\mathsf{msig}(m) = [\Gamma, pc, Q_1, Q_2]$ *if for all domain lattices $\diamond$, for all domains $k \in Dom^\diamond$, for all store-heap pairs $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$ and for all bijections β such that*

- $\sigma_1 \models^\diamond Q_1$ *and* $\sigma_1' \models^\diamond Q_1$,
- $\vdash^\diamond \sigma_1 \sim_\beta^{\Gamma,k} \sigma_1'$,
- $\sigma_1 \overset{m}{\underset{IR}{\Longrightarrow}}{}^\diamond \sigma_2$ *and* $\sigma_1' \overset{m}{\underset{IR}{\Longrightarrow}}{}^\diamond \sigma_2'$,

there exists a bijection $\beta' \supseteq \beta$ such that

- $\sigma_2 \models^\diamond Q_2$ *and* $\sigma_2' \models^\diamond Q_2$, *and*
- $\vdash^\diamond \sigma_2 \sim_{\beta'}^{\Gamma,k} \sigma_2'$

An IR program P_{IR} is universally noninterferent *if all its methods are universally noninterferent with respect to their signatures.*

5.3 Transforming Bytecode to the Intermediate Representation

The translation from bytecode to the intermediate representation is essentially a simplified version of the algorithm presented by Demange, Jensen, and Pichardie [DJP10]. Their algorithm considers general JVM bytecode, and is generic in the program property that shall be shown. Since I adapt the approach specifically for the DSD verification framework, a number simplifications can be made:

- Neither error states nor the order of errors are treated — noninterference is defined on successful executions only.
- There is no return instruction, hence we do not need to consider special return states.
- One does not have to consider event traces, therefore there are no IR pseudo-instructions that write such a trace.
- There are no constructor methods. The allocation and initialization of objects are both performed by a single instruction. Therefore, we do not need to handle uninitialized objects and the order of their initialization with care.
- The original transformation by Demange et al. may create, for a single bytecode instruction, a sequence of IR assignment instructions, which then all have to be linked back to the same original instruction for the proof. The IR language, in contrast, has these assignment sequences built into the IR syntax as single `block` $\overline{a}$ instructions.
- Only special bytecode programs are considered by the translation: the operand stack must be empty during a jump.

The last condition may appear to be a major restriction; however, the DSD compiler produces only such bytecode. In fact, empty-stack requirements have been given for previous bytecode verification techniques [Ler02], as it has been noticed that current Java compilers rarely take advantage of elaborate uses of stacks and local variables. Nevertheless, the extended version of the paper by Demange et al. shows how to deal with arbitrary bytecode, which requires a proper extension of the concepts presented here. Besides these simplifications, I also need to consider the pseudo-instructions `cjmp` j and `cpush` j in the translation, which are not present in JVM bytecode.

5.3.1 The Bytecode Transformation Algorithm

The transformation operates on four levels: single bytecode instructions, multiple bytecode instructions, method bodies, and full bytecode programs. I will now present the parts in this order.

bytecode	input stack	IR	output stack	conditions
`nop`	as	`block` ϵ	as	
`push` c	as	`block` ϵ	$c :: as$	
`pop`	$e :: as$	`block` ϵ	as	
`prim` op	$e_2 :: e_1 :: as$	`block` ϵ	$(e_1\ \mathbf{op}\ e_2) :: as$	
`load` x	as	`block` ϵ	$x :: as$	
`getf` x	$e :: as$	`block` ϵ	$e.f :: as$	
`new` C	$\overline{e} :: as$	`block` $[t_i^0 := \mathbf{new}\ C(\overline{e})]$	$t_i^0 :: as$	$t_i^0 \notin as$
`bnz` j	$e :: as$	`if` $e\ j$	as	
`jmp` j	as	`jmp` j	as	
`cpush` j	as	`cpush` j	as	
`cjmp` j	as	`cjmp` j	as	
`store` x	$e :: as$	`block` $[t_i^0 := x ; x := e]$	$as[t_i^0/x]$	$t_i^0 \notin as$
`putf` f	$e' :: e :: as$	`block` $[\overline{t_i} := as ; e.f := e']$	$\overline{t_i}$	$\overline{t_i} \notin as$
`call` m	$\overline{e} :: e :: as$	`block` $[\overline{t_i} := as ; t_i^0 := e.m(\overline{e})]$	$t_i^0 :: \overline{t_i}$	$t_i^0, \overline{t_i} \notin as$

Table 5.1: $\mathsf{BC2IR}_{\text{instr}}$: Transformation of a single bytecode instruction

Algorithm 1 $\mathsf{BC2IR}_{\text{rng}}(BC,\text{m},\text{I})$

for all $i \in \mathsf{sort}(I)$ **do**
  if $i \in \mathsf{jmpTgt}_m^{BC}$ **then**
    $AS_{in}[m,i] := \epsilon$
  end if
  $(AS_{out}[m,i], IR[m,i]) := \mathsf{BC2IR}_{\text{instr}}(i, BC(m,i), AS_{in}[m,i])$

  if $AS_{out}[m,i] \neq \epsilon$ **and** $\exists j \in \mathsf{succ}(m,i) \cap \mathsf{jmpTgt}_m^{BC}$ **then**
    fail
  end if
  if $i+1 \in \mathsf{succ}(m,i)$ **then**
    $AS_{in}[m,i+1] := AS_{out}[m,i]$
  end if
end for
return IR

Algorithm 2 $\mathsf{BC2IR}_{\text{mtd}}(BC,\text{m})$

$AS_{in}[m, \mathsf{mentry}(m)] := \epsilon$
$\mathsf{BC2IR}_{\text{rng}}(BC, m, \mathsf{dom}\, BC(m))$

Transformation of single bytecode instructions The transformation is based on abstract stacks, which are stacks of high-level expressions from *Exp*:

$$\text{abstract stack:} \quad as \quad \in \quad Exp^*$$

Table 5.1 on the preceding page defines the function $\mathsf{BC2IR}_{\text{instr}}$ which takes a bytecode address, a bytecode instruction and an abstract input stack, and produces an IR instruction and an abstract output stack. The function reconstructs the high-level expressions from operand-stack manipulating bytecode instructions. The bytecode instructions `store` x, `putf` f, `new` C, and `call` m are translated to assignments blocks, where the result of the operation is immediately written to a temporary variable. There is a one-to-one correspondence between the BC instructions and the IR instructions.

The function assumes for each method m and for each instruction address i the existence of arbitrarily many temporary variables t_i^0, t_i^1, t_i^2, ... that are available for the instruction $IR[m,i]$. This way, each temporary variable t_i^k is assigned at exactly one point in the IR program, namely at address i. The side conditions require that an assigned temporary variable t_i^k must not have occurred in the input stack of the instruction at address i.

Special care has to be taken for instructions that possibly invalidate the contents of the abstract stack. For `store` x operations, the contents of the variable x is first stored in a temporary variable t_i^0 before x is overwritten. In the output stack, all occurrences of x are replaced by t_i^0, which holds the saved old value of x. For `putf` f and `call` m, the entire abstract input stack as is saved in a sequence of temporary variables $\overline{t_i}$, and this sequence is then the output stack.

Transformation of multiple instructions Ranges of bytecode instructions are transformed with $\mathsf{BC2IR}_{\text{rng}}(BC,m,I)$, shown as Algorithm 1 on the previous page. The function traverses a set of instruction addresses I sorted in ascending order, and writes the compiled instructions into the array $IR[m,i]$. At the same time, it defines the arrays $AS_{in}[m,i]$ (input stack for instruction i) and $AS_{out}[m,i]$ (output stack for instruction i). The function $\mathsf{BC2IR}_{\text{rng}}(BC,m,I)$ relies on $\mathsf{BC2IR}_{\text{instr}}$, and chains the abstract stacks, such that the output stack of an instruction becomes the input stack of its immediate successor. Additionally, it ensures that the abstract input stacks at jump targets are empty, and fails if a jump instruction produces an output stack that is not empty.

Note that it cannot happen that a side condition of $\mathsf{BC2IR}_{\text{instr}}$ fails when used within $\mathsf{BC2IR}_{\text{rng}}$. This would only be possible if any of the variables t_i^k occurs in the input stack $AS_{in}[m,i]$, but no instruction preceding the one at address i generates the variable t_i^k.

Transformation of method bodies The function $\mathsf{BC2IR}_{\text{mtd}}(BC,m)$, shown as Algorithm 2 on the preceding page, transforms methods by setting the input stack of the entry point to ϵ, and then using $\mathsf{BC2IR}_{\text{rng}}$ for the method body.

Transformation of bytecode programs Finally, the full program translation function $\mathsf{BC2IR}(P_{\mathrm{BC}}) = P_{\mathrm{IR}}$ is defined as

$$\mathsf{BC2IR}(\prec, \mathsf{fields}, \mathsf{methods}, \mathsf{margs}, BC, \mathsf{mentry}, \mathsf{mexit}) = (\prec, \mathsf{fields}, \mathsf{methods}, \mathsf{margs}, \mathsf{tvars}, IR, \mathsf{mentry}, \mathsf{mexit})$$

such that for all methods $m \in \mathsf{dom}(BC)$,

$$\begin{array}{rcl} IR(m) & = & \mathsf{BC2IR}_{\mathrm{mtd}}(BC, m) \\ \mathsf{tvars}(m) & = & \{t \mid t \text{ occurs in } IR(m)\} \end{array}$$

In particular, all method bodies are translated using the algorithm $\mathsf{BC2IR}_{\mathrm{mtd}}$.

Properties of the algorithm We observe that a translated IR program covers the same range of instructions addresses as the original bytecode program; that the abstract stack is empty at jump targets; and that abstract stacks are passed through instructions.

Proposition 5.3 *Let* $IR = \mathsf{BC2IR}_{rng}(BC, m, [i_0, i_1[)$. *Then* $\mathsf{dom}(IR) = [i_0, i_1[$.

PROOF By definition of $\mathsf{BC2IR}_{\mathrm{rng}}$. □

Proposition 5.4 *For all* $i \in \mathsf{jmpTgt}_m^{BC}$, $AS_{in}[m, i] = \epsilon$. *Also,* $AS_{in}[m, \mathsf{mentry}(m)] = \epsilon$.

PROOF By definition of the BC2IR algorithm. □

Proposition 5.5 *If* $(m, i) \xrightarrow{IR[m,i]} (m, i')$, *then* $AS_{out}[m, i] = AS_{in}[m, i']$.

PROOF If $i' \in \mathsf{jmpTgt}_m^{BC}$, then $i' \in \mathsf{succ}(m, i) \cap \mathsf{jmpTgt}_m^{BC}$. As the algorithm did not fail, it must be $AS_{out}[[, m], i] = \epsilon$ by definition of BC2IR. With Proposition 5.4, we have $AS_{in}[m, i'] = \epsilon = AS_{out}[m, i]$.

If $i' \notin \mathsf{jmpTgt}_m^{BC}$, then since $i' \in \mathsf{succ}(m, i)$, it must be $i' = i + 1$ by definition of jmpTgt_m^{BC} and by definition of IR semantics. With $i + 1 \in \mathsf{succ}(m, i)$ and $i' = i + 1 \notin \mathsf{jmpTgt}_m^{BC}$, we get $AS_{in}[m, i'] = AS_{out}[m, i]$ by definition of BC2IR. □

5.3.2 Translation of the Example Program

Our example program in bytecode form, as presented at the end of Section 4.2 on page 56, is translated to an IR program

$$P_{\mathrm{IR}} = (\prec, \mathsf{fields}, \mathsf{methods}, \mathsf{margs}, \mathsf{tvars}, IR, \mathsf{mentry}, \mathsf{mexit})$$

such that $\prec$, fields, methods, margs, mentry, and mexit are as in the bytecode program. The only method that needs to be translated to the bytecode program is `sendFile`. Table 5.3.2 shows the method in bytecode form and the corresponding IR form, including the abstract input and output stacks for each instruction. Note how the IR instructions are a partial reconstruction of the original high-level method body from Section 2.2 on page 22. The translation also defines tvars(`sendFile`) $= [t_8^0, t_{13}^0, t_{14}^0, t_{19}^0, t_{20}^0]$, because these are the temporary variables that occur in IR(`sendFile`).

i	BC(sendFile, i)	AS_{in}[sendFile, i]	IR[sendFile, i]	AS_{out}[sendFile, i]
0	cpush 22	ϵ	cpush 22	ϵ
1	load file	ϵ	block ϵ	[file]
2	getf f_δ	[file]	block ϵ	[file.f_δ]
3	load srv	[file.f_δ]	block ϵ	[srv, file.f_δ]
4	getf f_δ	[srv, file.f_δ]	block ϵ	[srv.f_δ, file.f_δ]
5	prim $\sqsubseteq$	[srv.f_δ, file.f_δ]	block ϵ	[file.$f_\delta \sqsubseteq$ srv.f_δ]
6	bnz 10	[file.$f_\delta \sqsubseteq$ srv.f_δ]	if (file.$f_\delta \sqsubseteq$ srv.f_δ) 8	ϵ
7	push 0	ϵ	block ϵ	[0]
8	store ret	[0]	block [t^0_8 := 0 ; ret := 0]	ϵ
9	cjmp 22	ϵ	cjmp 22	ϵ
10	load file	ϵ	block ϵ	[file]
11	load file	[file]	block ϵ	[file, file]
12	getf f_δ	[file, file]	block ϵ	[file.f_δ, file]
13	call read	[file.f_δ, file]	block [t^0_{13} := file.read(file.f_δ)]	[t^0_{13}]
14	store data	[t^0_{13}]	block [t^0_{14} := t^0_{13} ; data := t^0_{13}]	ϵ
15	load srv	ϵ	block ϵ	[srv]
16	load file	[srv]	block ϵ	[file, srv]
17	getf f_δ	[file, srv]	block ϵ	[file.f_δ, srv]
18	load data	[file.f_δ, srv]	block ϵ	[data, file.f_δ, srv]
19	call write	[data, file.f_δ, srv]	block [t^0_{19} := srv.write(file.f_δ, data)]	[t^0_{19}]
20	store *ret*	[t^0_{19}]	block [t^0_{20} := t^0_{19} ; *ret* := t^0_{19}]	ϵ
21	cjmp 22	ϵ	cjmp 22	ϵ

Table 5.2: Translation of the example program from bytecode to IR

5.3.3 Semantics Preservation

The transformation algorithm preserves the semantics: if a bytecode method execution starting in an initial store s and heap h results in a final store s' and final heap h' (with an empty operand stack), then the execution of the corresponding IR method in the same initial store s and heap h results in the same final store s' and heap h' (with a default temporary variable store).

Theorem 5.6 *Let P_{BC} be a bytecode program, and P_{IR} be an IR program such that $P_{\mathrm{IR}} = \mathsf{BC2IR}(P_{\mathrm{BC}})$. Let m be a method, $\diamond$ be a domain lattice, s, s' be stores, and h, h' be heaps. If*

$$(s, h) \underset{BC}{\overset{m}{\Longrightarrow}}{}^{\diamond} (s', h')$$

then

$$(s, h) \underset{IR}{\overset{m}{\Longrightarrow}}{}^{\diamond} (s', h')$$

The full proof of this theorem is given in Appendix B. It relies on the correctness of the $\mathsf{BC2IR}_{\mathrm{instr}}$ function: if the input stack as describes the initial operand stack, then the output stack as' describes the final operand stack of the execution of the bytecode instruction:

Proposition 5.7 *Suppose $(m, i, s, h, \rho) \overset{\mathrm{B}}{\longrightarrow}{}^{\diamond} (m, i', s', h', \rho')$. Let s_t be a temporary variable store, and as, as' be abstract stacks and I be an IR instruction such that*

$$[\![as]\!]^{\diamond}_{s,s_t,h} = \rho \quad \text{and} \quad \mathsf{BC2IR}_{instr}(i, \mathrm{B}, as) = (\mathrm{I}, as').$$

Then there exists a temporary variable store s'_t such that $(m, i, s, s_t, h) \overset{\mathrm{I}}{\longrightarrow}{}^{\diamond} (m, i', s', s'_t, h')$ and

$$[\![as']\!]^{\diamond}_{s',s'_t,h'} = \rho'.$$

5.4 The IR Type System

This section presents an information flow type system for the intermediate language. With the translation from bytecode and the semantics preservation result, one obtains a technique to prove noninterference property for bytecode programs.

The type system follows the spirit of the system for the (high-level) DSD language. In fact, as IR code contains some constructs of DSD, the type system reuses high-level DSD typing judgements for these constructs. Moreover, we shall see later that the compilation to bytecode and the translation to IR code are type preserving — that is, if the original DSD program is typable, then the corresponding IR program is typable, too.

The IR type system follows the small-step semantics of the IR language. For example, if there is an instruction I such that $(m,i) \overset{\mathrm{I}}{\longmapsto} (m,i')$, then it assigns to i a precondition Q and an initial label pc, and to i' a postcondition Q' and a final label pc' that hold before and after the execution of the instruction, respectively. A program is well-typed if it is possible to assign to each address a unique typing information in this way.

5.4.1 The Purpose of the Pseudo-Instructions

A common problem for information flow type systems for unstructured code is that the instructions on which a subbranch ranges are not obvious from the syntax. This information, however, is needed to update the pc label in order to prevent indirect leaks. While it is easy to "raise" the pc label at a branching point, an expressive type system also needs to safely "lower" the pc label back to the previous level at the corresponding *confluence point*, which is the first instruction address that all execution paths that start at the branching point eventually reach.

A common approach is to provide the structural information in form of externally computed *control dependence regions*, an approach taken for the bytecode type system by Barthe et al [BBR04]. The type system then uses this precomputed information for the manipulation of the pc label. The authors point out that the information may be easily provided by a compiler, because it has access to the control flow structure from the program's high-level syntax. The problem, however, is that this delegation to external computation mechanisms introduces a new structure which must be trusted or verified: the computed regions must satisfy a set of safe over-approximation properties to ensure that they indeed describe the subbranches.

To simplify the proofs, I therefore follow the approach by Medel, Compagnoni, and Bonelli [MCB05] by using pseudo-instructions that indicate the start and the end of a subbranch. These instructions need to be available in the IR code. In contrast to control dependence regions, they are directly verified by the type system itself. Furthermore, a bytecode compiler can be easily extended to additionally produce these pseudo-instructions. Alternatively, if arbitrary bytecode without pseudo-instructions shall be analysed, one may compute control dependence regions and use this information to determine the points where the pseudo-instructions should be inserted.

5.4.2 Confluence Point Stacks

The typing judgement maintains *confluence point stacks*, which contain information before and after each instruction. These are stacks of pairs of program addresses and labels:

$$\Delta \in (Adr \times Lab)^*$$

The top element on the stack indicates the address of the confluence point where the current subbranch ends, and the pc label which shall be restored at that point. The

element below the top indicates the address and the *pc* label of confluence point of the subbranch surrounding the current one, and so on. The stacks are manipulated by the IR pseudo-instructions `cpush` j and `cjmp` j in the following fashion:

- `cpush` j pushes the address j and the current pc label on the stack Δ in the type system. In the semantics, the instruction has no effect.
- `cjmp` j pops the address j and the label pc from the stack Δ, jumps to j, and restores the label pc. In the semantics, `cjmp` j jumps to the address j.

The compiler translates a high-level conditional statement **if** e **then** S_1 **else** S_2 as follows: first, a `cpush` j instruction is generated, where j is the address of the first instruction that follows the compiled statement (and also the confluence point). Then, the actual statement is compiled, such that the final instruction of both compiled branches S_1 and S_2 is `cjmp` j. This causes the execution to jump to j. In the type system, the `cpush` j instruction causes the initial pc label and j to be pushed on the stack. At the branching instruction, the pc label is possibly raised to a different label pc'. At the final `cjmp` j instructions, the type system restores the original pc label from the stack.

5.4.3 The Type System

In the following, let us fix a method m. I define a typing judgement

$$\Gamma, \Gamma_t \vdash i, pc, \Delta, Q \xrightarrow{\text{I}} i', pc', \Delta', Q'$$

which means that with instruction I, it is possible to get from program point i to i' such that if pc, Δ, Q is valid in the initial state, then pc', Δ', Q' is valid in the final state. (I will give the exact notion of validity later.) The judgement is defined by the rules shown in Figure 5.3 on the following page.

The type system relies on the judgement $\Gamma \cup \Gamma_t \vdash e : \ell$ from the high-level type system, where the type environments for ordinary and temporary variables are combined. Also, I use on $\ell \sqsubseteq_Q \ell'$ as defined as in the high-level type system.

Assignment blocks are typed with another judgement

$$\Gamma, \Gamma_t, \Delta, pc \vdash \{Q\}\ \overline{a}\ \{Q'\}$$

as given by the rules in Figure 5.4 on the next page. Assignment blocks are typed just like sequential compositions of statements in the high-level type system. The typing rules for individual assignments make use of the corresponding high-level typing rules, with the variable type environments combined: $\Gamma \cup \Gamma_t, pc \vdash \{Q\}\ a\ \{Q'\}$. As in the high-level type system, I require that the object reference expressions for field updates and method calls are access paths π. The only extension to the original typing rules, apart from taking Γ_t into account, is that assigned variables and fields may not occur in Δ (in addition to not occurring in the pc label).

$$\frac{\begin{array}{c}\Gamma, \Gamma_t \vdash i, pc, \Delta, Q_0 \xrightarrow{\textsc{i}} i', pc', \Delta', Q_0' \\ Q \Rightarrow Q_0 \qquad Q_0' \Rightarrow Q'\end{array}}{\Gamma, \Gamma_t \vdash i, pc, \Delta, Q \xrightarrow{\textsc{i}} i', pc', \Delta', Q'}$$

$$\frac{\textsc{i} = \texttt{if}\ e\ j \qquad \Gamma \cup \Gamma_t \vdash e : \ell \qquad i' \in \{i+1, j\}}{\Gamma, \Gamma_t \vdash i, pc, \Delta, Q \xrightarrow{\textsc{i}} i', pc \sqcup \ell, \Delta, Q}$$

$$\frac{\textsc{i} = \texttt{if}\ e\ j \qquad \Gamma \cup \Gamma_t \vdash e : \ell \qquad e = \ell_1 \sqsubseteq \ell_2}{\Gamma, \Gamma_t \vdash i, pc, \Delta, Q \xrightarrow{\textsc{i}} j, pc \sqcup \ell, \Delta, Q \cup \{\ell_1 \sqsubseteq \ell_2\}}$$

$$\frac{\textsc{i} = \texttt{jmp}\ j}{\Gamma, \Gamma_t \vdash i, pc, \Delta, Q \xrightarrow{\textsc{i}} j, pc, \Delta, Q}$$

$$\frac{\textsc{i} = \texttt{cpush}\ j}{\Gamma, \Gamma_t \vdash i, pc, \Delta, Q \xrightarrow{\textsc{i}} i+1, pc, (j, pc) :: \Delta, Q}$$

$$\frac{\textsc{i} = \texttt{cjmp}\ j}{\Gamma, \Gamma_t \vdash i, pc, (j, pc') :: \Delta, Q \xrightarrow{\textsc{i}} j, pc', \Delta, Q}$$

$$\frac{\textsc{i} = \texttt{block}\ \overline{a} \qquad \Gamma, \Gamma_t, \Delta, pc \vdash \{Q\}\ \overline{a}\ \{Q'\}}{\Gamma, \Gamma_t \vdash i, pc, \Delta, Q \xrightarrow{\textsc{i}} i+1, pc, \Delta, Q'}$$

Figure 5.3: IR type system

$$\frac{}{\Gamma, \Gamma_t, \Delta, pc \vdash \{Q\}\ \epsilon\ \{Q\}}$$

$$\frac{\begin{array}{c}\Gamma, \Gamma_t, \Delta, pc \vdash \{Q\}\ a\ \{Q'\} \\ \Gamma, \Gamma_t, \Delta, pc \vdash \{Q'\}\ as\ \{Q''\}\end{array}}{\Gamma, \Gamma_t, \Delta, pc \vdash \{Q\}\ a :: as\ \{Q''\}}$$

$$\frac{\begin{array}{c}a \in \{x := e, x := \textbf{new}\ C(\overline{e})\} \\ x \notin \Delta \\ \Gamma \cup \Gamma_t, pc \vdash \{Q\}\ a\ \{Q'\}\end{array}}{\Gamma, \Gamma_t, \Delta, pc \vdash \{Q\}\ a\ \{Q'\}}$$

$$\frac{\begin{array}{c}a = \pi.f := e \\ f \notin \Delta \\ \Gamma \cup \Gamma_t, pc \vdash \{Q\}\ a\ \{Q'\}\end{array}}{\Gamma, \Gamma_t, \Delta, pc \vdash \{Q\}\ a\ \{Q'\}}$$

$$\frac{\begin{array}{c}a = x := \pi.m(\overline{e}) \\ x \notin \Delta \\ f \neq f_\delta \Rightarrow f \notin \Delta \\ \Gamma \cup \Gamma_t, pc \vdash \{Q\}\ a\ \{Q'\}\end{array}}{\Gamma, \Gamma_t, \Delta, pc \vdash \{Q\}\ a\ \{Q'\}}$$

Figure 5.4: Typing rules for assignment blocks

5.4.4 Well-Typed Programs

A *type mapping* Λ for a method m associates each instruction address $i \in \mathsf{dom}(IR(m))$ with a program counter label pc, a confluence point stack Δ, and a constraint set Q. To access the components of $\Lambda(i)$, I also write $\Lambda_{pc}(i)$, $\Lambda_\Delta(i)$, and $\Lambda_Q(i)$ in the following. We are interested in type mappings that are well-formed with respect to the small-step typing rules:

Definition 5.8 *A type mapping Λ is* derivable *for a method body $IR(m)$ and type environments Γ and Γ_t, if for all $i, i' \in \mathsf{dom}(IR(m))$, whenever $(m,i) \overset{\mathrm{I}}{\hookrightarrow} (m,i')$, then $i, i' \in \mathsf{dom}(\Lambda)$ and $\Gamma, \Gamma_t \vdash i, \Lambda_{pc}(i), \Lambda_\Delta(i), \Lambda_Q(i) \overset{\mathrm{I}}{\longrightarrow} i', \Lambda_{pc}(i'), \Lambda_\Delta(i'), \Lambda_Q(i')$.*

An IR method is well-typed if the method signature $\mathsf{msig}(m)$ can be extended by a type environment Γ_t such that a type mapping Λ can be derived for its instructions, where the type information for $\mathsf{mentry}(m)$ and $\mathsf{mexit}(m)$ matches the one given by the signature.

Definition 5.9 *A method m is* well-typed *with respect to a signature*

$$\mathsf{msig}(m) = [\Gamma, pc, Q, Q']$$

if there exists a type environment Γ_t and a type mapping Λ such that

1. *Λ is derivable for $IR(m)$, Γ and Γ_t, and*
2. *$\Lambda(\mathsf{mentry}(m)) = (pc, \epsilon, Q)$, and*
3. *$\Lambda(\mathsf{mexit}(m)) = (pc, \epsilon, Q')$.*

An IR program P_{IR} is well-typed *if all its methods are well-typed.*

5.4.5 Soundness

The following theorem states the main soundness result for IR programs.

Theorem 5.10 *If an IR program P_{IR} is well-typed, then it is universally noninterferent.*

The proof of the theorem can be found in Appendix C. With the semantics preservation result, we immediately get a soundness result for the underlying bytecode program.

Theorem 5.11 *Let P_{BC} be a bytecode program. Let P_{IR} be an IR program such that $P_{\mathrm{IR}} = \mathsf{BC2IR}(P_{\mathrm{BC}})$ and P_{IR} is well-typed. Then P_{BC} is universally noninterferent.*

PROOF Let m be a method such that $\mathsf{msig}(m) = [\Gamma, pc, Q, Q']$. Let (s_1, h_1), (s_2, h_2), (s_1', h_1'), (s_2', h_2') be store-heap pairs, β be bijection, $\diamond$ be a domain lattice, and $k \in Dom^\diamond$ be a domain. Furthermore, let

- $(s_1, h_1) \models^\diamond Q$ and $(s'_1, h'_1) \models^\diamond Q$, and
- $\vdash^\diamond (s_1, h_1) \sim^{\Gamma,k}_{\beta} (s'_1, h'_1)$, and
- $(s_1, h_1) \overset{m}{\underset{BC}{\Longrightarrow}}{}^\diamond (s_2, h_2)$ and $(s'_1, h'_1) \overset{m}{\underset{BC}{\Longrightarrow}}{}^\diamond (s'_2, h'_2)$.

Applying the semantics preservation result (Theorem 5.6 on page 75) to both executions, we know there are corresponding IR executions $(s_1, h_1) \overset{m}{\underset{IR}{\Longrightarrow}}{}^\diamond (s_2, h_2)$ and $(s'_1, h'_1) \overset{m}{\underset{IR}{\Longrightarrow}}{}^\diamond (s'_2, h'_2)$. As P_{IR} is well-typed, we can apply the soundness theorem 5.10 on the previous page and get that m is universally noninterferent; that is, there exists a bijection $\beta' \supseteq \beta$ such that

- $(s_2, h_2) \models^\diamond Q'$ and $(s'_2, h'_2) \models^\diamond Q'$ and
- $\vdash^\diamond (s_2, h_2) \sim^{\Gamma,k}_{\beta'} (s'_2, h'_2)$.

It follows by definition that the bytecode version of m is universally noninterferent. As this can be shown for each method, the entire program P_{BC} is universally noninterferent. □

5.5 Type Preservation Result

It can be shown that the compilation of a high-level DSD program to the bytecode level and the subsequent transformation to an IR program preserves the types. That is, if P_{DSD} is a well-typed DSD program with respect to the high-level type system, then $\mathsf{BC2IR}(\mathsf{compile}(P_{\mathrm{DSD}}))$ exists and is well-typed with respect to the IR type system.

Theorem 5.12 *Let P_{DSD} be well-typed with respect to given method signatures. Then $\mathsf{BC2IR}(\mathsf{compile}(P_{\mathrm{DSD}}))$ exists and is well-typed with respect to these signatures.*

The complete proof of this theorem can be found in Appendix D on page 147; here, I only outline the basic ideas.

The proof is split into two parts. First, one shows that the compiled and translated program $\mathsf{BC2IR}(\mathsf{compile}(P_{\mathrm{DSD}}))$ indeed exists, and has the following properties: stacks are empty between translated statements, and high-level expressions e are completely recovered by the BC2IR algorithm in the abstract stacks.

Lemma 5.13 *If $(BC, i_1) = \mathsf{compile}_{\mathrm{exp}}(m, e, i_0)$ and $AS_{in}[m, i_0] = as$, then*

1. *$IR = \mathsf{BC2IR}_{rng}(BC, m, [i_0, i_1[)$ exists, and*
2. *$AS_{in}[m, i_1] = e :: as$.*

If $(BC, i_1) = \mathsf{compile}_{\mathsf{stmt}}(m, S, i_0)$*, and* $AS_{in}[m, i_0] = \epsilon$*, then*

1. $IR = \mathsf{BC2IR}_{rng}(BC, m, [i_0, i_1[)$ *exists, and*
2. $AS_{in}[m, i_1] = \epsilon$.

The main lemma of the second part of the proof shows that for every high-level typing judgement, there is a corresponding IR type mapping.

Lemma 5.14 *Let* $(BC, i_1) = \mathsf{compile}_{\mathsf{stmt}}(m, S, i_0)$*, and* $IR = \mathsf{BC2IR}_{rng}(BC, m, [i_0, i_1[$*. Let* Δ *be a confluence point stack that does not contain any variable or field modified by* S*. If* $\Gamma, pc \vdash \{Q\}\ S\ \{Q'\}$*, then there exists a temporary variable type environment* Γ_t *and a type mapping* Λ *derivable for IR,* Γ*, and* Γ_t *such that*

1. $\Lambda(i_0) = (pc, \Delta, Q)$*, and*
2. $\Lambda(i_1) = (pc, \Delta, Q')$.

The main type preservation theorem follows as a corollary by unfolding the definitions of well-typedness.

5.6 Putting It All Together

I have now presented almost all parts that constitute the DSD verification framework. Figure 5.5 on the next page is a repetition of the overview diagram from the introduction (see page 12), with all parts annotated with the numbers of the sections where I have presented the respective definitions (boxes in the figure), algorithms (solid arrows), or theorems (dotted arrows).

For the actual implementation, there are still two parts missing — namely the type check algorithms that fully automatically determine whether a given high-level or IR program is well-typed. So far, I have only declaratively described what constitutes a valid type derivation, leaving open the task of finding such derivations.

In the next chapter, I therefore show how to transform the presented type systems into a type check algorithm, before I present the implementation of the verification framework.

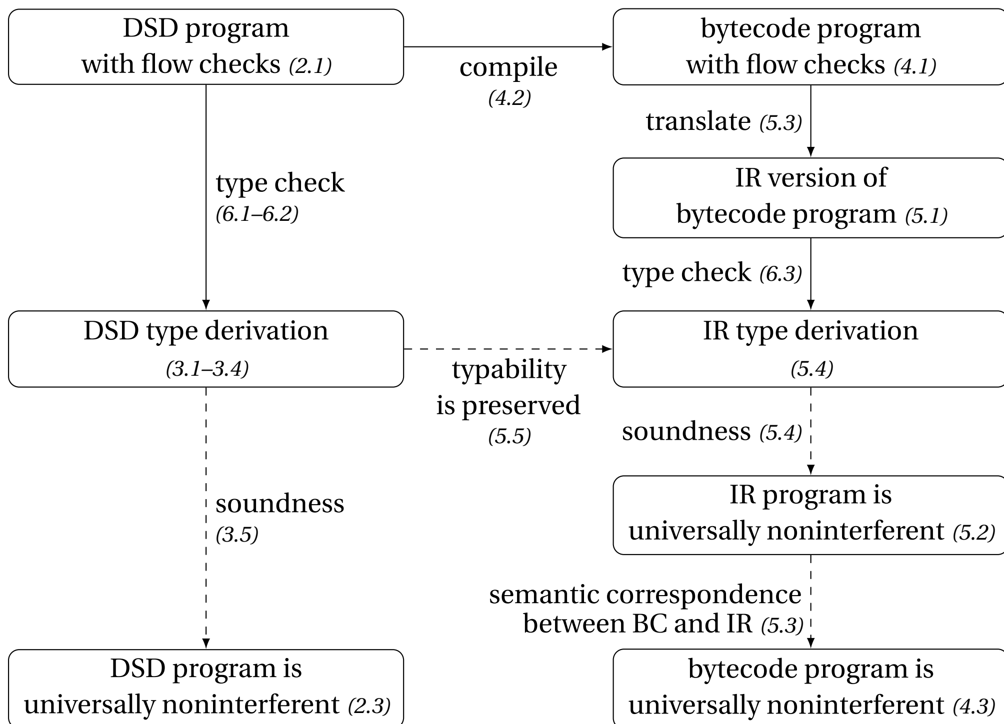

Figure 5.5: The verfication framework, and the corresponding sections

6 Automatic Type Inference and Implementation

The type systems presented in the preceding chapters declaratively describe which typing judgements are valid. For an actual implementation, I have not yet explained *how* to automatically determine whether a given program has a valid typing judgement. This chapter presents an automatic *type inference* for high-level DSD and for IR programs, and presents several aspects of the implementation I have developed.

The task that usually constitutes a type inference – determining the types of expressions – is comparatively easy. Moreover, as all method signatures have to be provided by the programmer, the type inference for DSD is a purely intra-procedural mechanism. The main difficulties lie in solving the label order side conditions (Section 6.1), in computing the pre- and post-conditions for each statement (Section 6.2), and, for IR code, in finding a suitable, consistent type mapping (Section 6.3). These aspects come together in the prototypical implementation of the entire framework, described in Section 6.4.

6.1 Solving Label Order Conditions

The type systems for the high-level DSD language and the intermediate representation contain numerous side conditions on the order of labels. Therefore, any implementation of an automatic type inference repeatedly has to solve side conditions on the order of labels. More precisely, given a constraint set Q and two labels ℓ_1 and ℓ_2, the algorithm has to decide whether $\ell_1 \sqsubseteq_Q \ell_2$ can be derived or not.

$$\frac{}{\ell \sqsubseteq_Q \ell} \qquad \frac{}{\bot \sqsubseteq_Q \ell} \qquad \frac{}{\ell \sqsubseteq_Q \top} \qquad \frac{}{\ell \sqsubseteq_Q \ell \sqcup \ell'}$$

$$\frac{(\ell_1, \ell_2) \in Q}{\ell_1 \sqsubseteq_Q \ell_2} \qquad \frac{\ell_1 \sqsubseteq_Q \ell_3 \qquad \ell_2 \sqsubseteq_Q \ell_4}{\ell_1 \sqcup \ell_2 \sqsubseteq_Q \ell_3 \sqcup \ell_4} \qquad \frac{\ell_1 \sqsubseteq_Q \ell_2 \qquad \ell_2 \sqsubseteq_Q \ell_3}{\ell_1 \sqsubseteq_Q \ell_3}$$

$$\frac{}{\ell \equiv_Q \ell \sqcup \ell} \qquad \frac{}{\ell_1 \sqcup \ell_2 \equiv_Q \ell_2 \sqcup \ell_1} \qquad \frac{}{\ell_1 \sqcup (\ell_2 \sqcup \ell_3) \equiv_Q (\ell_1 \sqcup \ell_2) \sqcup \ell_3}$$

$$\frac{\ell_1 \equiv_Q \ell_2}{\ell_1 \sqsubseteq_Q \ell_2} \qquad \frac{\ell_1 \equiv_Q \ell_2}{\ell_2 \sqsubseteq_Q \ell_1} \qquad \frac{\ell_1 \sqsubseteq_Q \ell_2 \qquad \ell_2 \sqsubseteq_Q \ell_1}{\ell_1 \equiv_Q \ell_2}$$

Figure 6.1: Rules for label order equality (repeated)

The solution is not obvious from the rules for label order and label equalities defined in Section 3.2 on page 33. For reference, they are presented here again in Figure 6.1. It is difficult to follow the rules in a bottom-up fashion to determine whether a given label order is justified, because the rules are not syntax-directed: for one thing, it is not always clear which rule to apply; for another thing, the instantiations may be not unique, for example when the correct label ℓ_2 has to be chosen in the transitivity rule. On the other hand, one cannot generate all possible label order pairs from a given set Q and check whether the given flow is contained, because the first four rules generate an infinite set of such pairs for arbitrary syntactic labels $\ell \in Lab$.

Fortunately, many of the rules are in fact independent of the constraint set Q. For example, the first four rules describe basic lattice properties, and the label equality rules in the third row describe the properties of the least upper bound operator $\sqcup$. The rules which do depend on Q, in contrast, can be used to generate a hull of Q that remains finite. The solution is thus to split the inference into two parts.

To simplify the algorithm, all labels that occur during the inference are first transformed into a normalized, canonical form. For this, I take the algebraic definition of a complete semi-lattice, which is given by the following equations:

$$\begin{array}{rcll}
\ell_1 \sqcup \ell_2 & = & \ell_2 \sqcup \ell_1 & \text{(commutativity)} \\
(\ell_1 \sqcup \ell_2) \sqcup \ell_3 & = & \ell_1 \sqcup (\ell_2 \sqcup \ell_3) & \text{(associativity)} \\
\ell \sqcup \ell & = & \ell & \text{(idempotence)} \\
\ell \sqcup \bot & = & \ell & \text{(neutral element)} \\
\ell \sqcup \top & = & \top & \text{(top element)}
\end{array}$$

The label normalization algorithm essentially understands these rules as left-to-right rewrite rules. Each label is treated as a list of operator-free atomic labels that are connected by the least upper bound operator $\sqcup$. A label is normalized by ordering all its

atomic labels with respect to some fixed and previously defined order on atomic labels, by removing duplicate atomic labels, by removing the atomic label $\bot$, by removing all other atomic labels if the entire label contains $\top$, and by structuring the least upper bound list right-associatively.

The following rule expresses the correspondence between the algebraic definition of a lattice to its definition as a partially ordered set:

$$\frac{\ell_1 \sqcup \ell_2 = \ell_2}{\ell_1 \sqsubseteq^B \ell_2}$$

It can be shown that the defined relations are derivable in the original system.

Lemma 6.1 *The relations $=$ and $\sqsubseteq^B$ are sound with respect to the label equality and label order relation over an empty constraint set:*

1. $\ell_1 \sqsubseteq^B \ell_2 \Rightarrow \ell_1 \sqsubseteq_\emptyset \ell_2$ *and*
2. $\ell_1 = \ell_2 \Rightarrow \ell_1 \equiv_\emptyset \ell_2$.

It turns out that two labels are equal with respect to $\equiv_\emptyset$ if they are structurally equal. Also, the rule above conveniently links the label normalization algorithm to the inference of arbitrary tautologies, that is, label order judgements over an empty constraint set.

PROOF The second implication follows from the definitions. All equalities shown above can be derived as label equalities $\equiv_\emptyset$ in the original rules. This fact can be used in the first implication: we get $\ell_1 \sqcup \ell_2 \equiv_\emptyset \ell_2$, thus we can derive $\ell_1 \sqcup \ell_2 \sqsubseteq_\emptyset \ell_2$. Together with $\ell_1 \sqsubseteq_\emptyset \ell_1 \sqcup \ell_2$, we get by the transitivity rule $\ell_1 \sqsubseteq_\emptyset \ell_2$. □

I define another relation $\sqsubseteq_Q^H$ using the rules that do depend on Q. They only generate order information for labels that are in Q. In the algorithm, all labels generated by the rules are immediately normalized. This way, the number of derived $\sqsubseteq^H$ judgements is always finite, and can be used in the implementation to precompute a closure or "hull".

$$\frac{(\ell_1, \ell_2) \in Q}{\ell_1 \sqsubseteq_Q^H \ell_2} \qquad \frac{\ell_1 \sqsubseteq_Q^H \ell_3 \qquad \ell_2 \sqsubseteq_Q^H \ell_4}{\ell_1 \sqcup \ell_2 \sqsubseteq_Q^H \ell_3 \sqcup \ell_4} \qquad \frac{\ell_1 \sqsubseteq_Q^H \ell_2 \qquad \ell_2 \sqsubseteq^B \ell_2' \qquad \ell_2' \sqsubseteq_Q^H \ell_3}{\ell_1 \sqsubseteq_Q^H \ell_3}$$

Finally, the relation $\sqsubseteq_Q^A$ is defined in terms of $\sqsubseteq_Q^H$ and $\sqsubseteq^B$ to determine whether a given flow follows from a constraint set Q:

$$\frac{\ell_1 \sqsubseteq^B \ell_2}{\ell_1 \sqsubseteq_Q^A \ell_2} \qquad \frac{\ell_1 \sqsubseteq^B \ell_2 \qquad \ell_2 \sqsubseteq_Q^H \ell_3 \qquad \ell_3 \sqsubseteq^B \ell_4}{\ell_1 \sqsubseteq_Q^A \ell_4}$$

I use this relation in the algorithmic definition of constraint set implication:

$$Q \Rightarrow^A Q' \quad \text{if and only if} \quad \forall(\ell_1, \ell_2) \in Q.\ \ell_1 \sqsubseteq^A_Q \ell_2$$

The algorithmic relation $\sqsubseteq^A_Q$ and the algorithmic constraint set implication are correct with respect to the label order relation $\sqsubseteq_Q$, and the original constraint set implication. This is formalized in the following lemma.

Lemma 6.2 *The algorithmic label order and the algorithmic constraint set implication are correct with respect to their original definitions, that is,*

1. *If $\ell_1 \sqsubseteq^A_Q \ell_2$ then $\ell_1 \sqsubseteq_Q \ell_2$.*
2. *If $Q \Rightarrow^A Q'$ then $Q \Rightarrow Q'$.*

PROOF For the first implication, it is easy to see that all the rules above can be derived from the original rules, if one takes into account Lemma 6.1. The second implication follows from the first one and the definition of the original and the algorithmic constraint set implication. □

6.2 Algorithmic Version of the High-Level Type System

The goal of the implementation is to automatically determine whether a given DSD program is well-typed. For this, a type derivation for each method body that fits the method signature has to be found. A common practice for the type inference algorithm is to present the type system in an algorithmic form, from which a type inference algorithm can be directly "read off". The advantage is that the correctness of the rules of the algorithmic type system (and thus of the inference algorithm) is easy to justify — one only needs to show that the algorithmic rules are derivable in the original type system.

In fact, the separation of the typability specification from the inference is one of the strong points of type systems. This way, implementation aspects such as internal representations and efficiency considerations can be factored out, such that the presentation of the analysis can be modularized, and the proofs of correctness and other properties can be simplified.

Algorithmic type system The algorithmic version of the high-level type system is syntax-directed and derives statements of the form

$$\Gamma, pc \vdash_A \{Q\}\ S\ \{Q'\}.$$

The algorithmic typing rules are shown in Figure 6.2 on the facing page. For better clarity, I write $(\ell_1 \sqsubseteq \ell_2)$ instead of (ℓ_1, ℓ_2) for a label pair in a constraint set Q to underline

$$\text{TA-SKIP}\ \frac{}{\Gamma, pc \vdash_A \{Q\}\ \textbf{skip}\ \{Q\}}$$

$$\text{TA-SEQ}\ \frac{\Gamma, pc \vdash_A \{Q'\}\ S_2\ \{Q\} \qquad \Gamma, pc \vdash_A \{Q''\}\ S_1\ \{Q'\}}{\Gamma, pc \vdash_A \{Q''\}\ S_1\,;S_2\ \{Q\}}$$

$$\text{TA-WHILE1}\ \frac{\Gamma \vdash e : \ell \qquad \Gamma, pc \sqcup \ell \vdash_A \{Q'\}\ S\ \{Q\} \qquad Q \Rightarrow^A Q'}{\Gamma, pc \vdash_A \{Q\}\ \textbf{while}\ e\ \textbf{do}\ S\ \{Q\}}$$

$$\text{TA-WHILE2}\ \frac{\begin{array}{c}\Gamma \vdash e : \ell \qquad \Gamma, pc \sqcup \ell \vdash_A \{Q'\}\ S\ \{Q\} \qquad Q \not\Rightarrow^A Q' \\ \Gamma, pc \vdash_A \{Q''\}\ \textbf{while}\ e\ \textbf{do}\ S\ \{Q \cup Q'\}\end{array}}{\Gamma, pc \vdash_A \{Q''\}\ \textbf{while}\ e\ \textbf{do}\ S\ \{Q\}}$$

$$\text{TA-IF}\ \frac{\Gamma \vdash e : \ell \qquad \Gamma, pc \sqcup \ell \vdash_A \{Q_1\}\ S_1\ \{Q\} \qquad \Gamma, pc \sqcup \ell \vdash_A \{Q_2\}\ S_2\ \{Q\}}{\Gamma, pc \vdash_A \{Q_1 \cup Q_2\}\ \textbf{if}\ e\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}}$$

$$\text{TA-IFLABEL}\ \frac{\Gamma \vdash \ell_1 \sqsubseteq \ell_2 : \ell \qquad \Gamma, pc \sqcup \ell \vdash_A \{Q_1\}\ S_1\ \{Q\} \qquad \Gamma, pc \sqcup \ell \vdash_A \{Q_2\}\ S_2\ \{Q\}}{\Gamma, pc \vdash_A \{(Q_1 \backslash (\ell_1 \sqsubseteq \ell_2)) \cup Q_2\}\ \textbf{if}\ \ell_1 \sqsubseteq \ell_2\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}}$$

$$\text{TA-ASSIGN}\ \frac{\Gamma \vdash e : \ell \qquad x \not\in pc \qquad x \neq x_\delta}{\Gamma, pc \vdash_A \{Q[e/x], (\ell \sqcup pc \sqsubseteq \Gamma(x))\}\ x := e\ \{Q\}}$$

$$\text{TA-PUTF}\ \frac{\Gamma \vdash \pi : \ell_1 \qquad \Gamma \vdash e : \ell_2 \qquad f \not\in pc \qquad f \neq f_\delta \qquad f \not\in Q[e/\pi.f]}{\Gamma, pc \vdash_A \{Q[e/\pi.f], (\ell_1 \sqcup \ell_2 \sqcup pc \sqsubseteq \Phi^\pi(f))\}\ \pi.f := e\ \{Q\}}$$

$$\text{TA-NEW}\ \frac{\begin{array}{c}\Gamma \vdash \overline{e} : \overline{\ell} \\ \mathsf{fields}(C) = \overline{f} \qquad \Phi^* = \Phi^x[\overline{e}.1/x.f_\delta] \qquad x \not\in pc \qquad x \neq x_\delta \qquad x \not\in Q[\overline{e}/x.\overline{f}]\end{array}}{\Gamma, pc \vdash_A \{Q[\overline{e}/x.\overline{f}], (\overline{\ell} \sqsubseteq \Phi^*(\overline{f})), (pc \sqsubseteq \Gamma(x))\}\ x := \textbf{new}\ C(\overline{e})\ \{Q\}}$$

$$\text{TA-CALL}\ \frac{\begin{array}{c}\mathsf{msig}(m) = [\Gamma_m, pc_m, Q_m, Q'_m] \\ \mathsf{margs}(m) = \overline{x} \qquad \Gamma \vdash \pi : \ell_r \qquad \Gamma \vdash \overline{e} : \overline{\ell} \qquad \Gamma^* = \Gamma_m[\overline{e}.1/x_\delta] \\ Q' = \{\ell_r \sqsubseteq \Gamma^*(\mathit{this}),\ \overline{\ell} \sqsubseteq \Gamma^*(\overline{x}),\ pc \sqsubseteq pc_m[\overline{e}/\overline{x}][\pi/\mathit{this}]\} \\ Q'' = (Q \backslash Q'_m[x/\mathit{ret}])\ \cup\ (\Gamma^*(\mathit{ret}) \sqcup pc \sqsubseteq \Gamma(x)) \\ x \not\in pc, Q'' \qquad \forall f.\ f \neq f_\delta \Rightarrow f \not\in pc, Q'' \qquad x \neq x_\delta\end{array}}{\Gamma, pc \vdash_A \{Q_m[\overline{e}/\overline{x}][\pi/\mathit{this}] \cup Q' \cup Q''\}\ x := \pi.m(\overline{e})\ \{Q\}}$$

Figure 6.2: Algorithmic version of the high-level type system

that it is a label flow predicate. There are a number of differences from the original system:

- The weakening rule is built into the system where it is needed. In particular, at control flow branching points, the union of the different preconditions is taken as the overall precondition.
- There are two rules for **while**, TA-WHILE1 and TA-WHILE2 which together compute a least fixed point of the preconditions. Depending on whether $Q \Rightarrow^A Q'$, only one of the two rules fires.
- The side conditions needed for assignments are directly added to the precondition set.
- In the rule TA-IFLABEL for label tests "removes" the label test from the precondition of the first branch. This is defined syntactically as

  $$Q \backslash Q' = Q \setminus \{(\ell_1, \ell_2) \in Q \mid \ell'_1 \sqsubseteq^A_{Q'} \ell'_2\}$$

  and will be explained later.
- Likewise, the rule TA-CALL "removes" all information from the postcondition that does not follow from the postcondition of the called method.

Note that the algorithmic type system uses the normal typing rules for expressions to judgements of the form $\Gamma \vdash e : \ell$, because they are already in a syntax-directed form.

Inference algorithm The rules suggest an algorithm A that computes constraint sets in a backwards fashion, similar to the computation of weakest (liberal) preconditions in predicate transformer semantics [Dij75], which can be seen as an algorithmic reformulation of Hoare logic. Given a type environment, a program counter label, a DSD statement, and a postcondition, the algorithm returns the matching precondition:

$$A(\Gamma, pc, S, Q_{post}) = Q_{pre}$$

such that

$$\Gamma, pc \vdash_A \{Q_{pre}\}\ S\ \{Q_{post}\}$$

can be derived. The premises are processed from left to right. Each statement typing relation in a premise corresponds to the application of the algorithm to the respective subgoal. A syntactic side condition like $x \notin pc$ is considered as an assertion in the algorithm; the algorithm fails if the condition cannot be satisfied. An expression typing relation $\Gamma \vdash e : \ell$ corresponds to a subalgorithm that computes the label of the expression e, exploiting the fact that the expression typing rules are syntax-directed. Finally, the entire algorithm checks that all methods are well-typed — that is, for all methods m such that $\mathsf{msig}(m) = [\Gamma, pc, Q, Q']$, it checks that $Q \Rightarrow^A A(\Gamma, pc, \mathsf{mbody}(m), Q')$.

Soundness The algorithmic typing rules are sound with respect to the original typing rules presented in Section 3.4 on page 39, so that whenever $\Gamma, pc \vdash_A \{Q\}\ S\ \{Q'\}$ can be derived in the system presented here, then $\Gamma, pc \vdash \{Q\}\ S\ \{Q'\}$ can also be derived from the original system.

Lemma 6.3 *If* $\Gamma, pc \vdash_A \{Q\}\ S\ \{Q'\}$*, then* $\Gamma, pc \vdash \{Q\}\ S\ \{Q'\}$.

PROOF The lemma is shown inductively over the structure of $\Gamma, pc \vdash_A \{Q\}\ S\ \{Q'\}$. Instead of the full proof, I describe the general proof pattern for each form of S.

As the judgements are syntax-directed, one can determine the algorithmic rule used for S and read off the premises. We get $\Gamma, pc \vdash \{Q\}\ S\ \{Q'\}$ by applying the corresponding rule of the original type system, but first, the premises have to be put into the appropriate form. There are three kinds of premises in the original system:

1. Statement typing judgements: we observe that we already have the corresponding algorithmic typing judgements, thus we can apply the lemma inductively. In most cases, the subsumption rule T-WEAK is then applied to the result to weaken the precondition appropriately.

2. For label order requirements of the form $\ell_1 \sqsubseteq_{Q'} \ell_2$, we can directly assume Q' to be the set of label order preconditions given in the algorithmic typing rule, which are identical to the required flows.

3. All other premises are already present in the rule for the algorithmic type system. For the algorithmic constraint set implication in TA-WHILE1, the original form follows from Lemma 6.2 on page 86.

The only (minor) difficulty is caused by the syntactic "removal" of flow information from a constraint set used in TA-IFLABEL and TA-CALL. By definition, however, it can be shown that $Q \backslash Q' \cup Q' \Rightarrow^A Q$.

To get the premise for S_1 in T-IFLABEL in the correct form, we can thus transform its precondition Q_1 using T-WEAK with the fact $Q_1 \backslash (\ell_1 \sqsubseteq \ell_2) \cup Q_2 \cup (\ell_1 \sqsubseteq \ell_2) \Rightarrow^A Q_1 \backslash (\ell_1 \sqsubseteq \ell_2) \cup (\ell_1 \sqsubseteq \ell_2) \Rightarrow^A Q_1$. Likewise, T-CALL gives us $Q'' \cup Q_m[x/ret]$ as the postcondition of the entire call. With two applications of T-WEAK, this can be transformed to Q using the fact $Q'' \cup Q_m[x/ret] = (Q \backslash Q'_m[x/ret]) \cup (\Gamma^*(ret) \sqcup pc \sqsubseteq \Gamma(x)) \cup Q_m[x/ret] \Rightarrow Q \backslash Q_m[x/ret] \cup Q_m[x/ret] \Rightarrow^A Q$. □

Remarks on the relation to the weakest precondition calculus By means of the constraint set preconditions defined in the rules TA-IF and TA-IFLABEL, I will now explain how the algorithmic rules relate to predicate transformer semantics [Dij75], and describe the limits of the expressivity of constraint sets. The weakest precondition of a conditional statement is computed as follows:

$$wp(\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2, R) \quad = \quad (e \rightarrow wp(S_1, R)) \wedge (\neg e \rightarrow wp(S_2, R))$$

Constraint sets represent only conjunctive and positive label order predicates. Other types of expressions e, as well as negations and implications cannot be expressed in a constraint set. Therefore, the algorithmic type system infers a *stronger* precondition as follows. Let us treat Q as a postcondition in the calculus, and say that Q_1 corresponds to $wp(S_1, Q)$, and Q_2 corresponds to $wp(S_2, Q)$. The union of two constraint sets corresponds to a logical conjunction of predicates. The precondition of ordinary conditionals implies the one computed by the weakest precondition calculus:

$$\begin{array}{rcl} Q_1 \cup Q_2 & \simeq & wp(S_1, Q) \wedge wp(S_2, Q) \\ & \Rightarrow & (e \rightarrow wp(S_1, Q)) \wedge (\neg e \rightarrow wp(S_2, Q)) \\ & = & wp(\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2, Q) \end{array}$$

For the label test rule, the algorithmic type system can derive a better (i.e., weaker) precondition. Let $Q_1 = \{q_1, q_2, \ldots\}$, such that each q_i is a flow expression $\ell \sqsubseteq \ell'$, and let $I = \{1, \ldots, |Q_1|\}$. As a predicate, Q_1 thus corresponds to $\bigwedge_{i \in I} q_i$. We have previously defined $Q_1 \backslash (\ell_1 \sqsubseteq \ell_2) = Q_1 \setminus \{(\ell, \ell') \in Q_1 \mid \ell \sqsubseteq_{\{\ell_1 \sqsubseteq \ell_2\}} \ell'\}$. As a predicate, this corresponds to

$$\begin{array}{rcl} Q_1 \setminus \{(\ell, \ell') \in Q_1 \mid \ell \sqsubseteq_{\{\ell_1 \sqsubseteq \ell_2\}} \ell'\} & \simeq & \bigwedge_{i \in I} \{q_i \mid \neg((\ell_1 \sqsubseteq \ell_2) \rightarrow q_i)\} \\ & = & \bigwedge_{i \in I} \neg((\ell_1 \sqsubseteq \ell_2) \rightarrow q_i) \rightarrow q_i \\ & = & \bigwedge_{i \in I} ((\ell_1 \sqsubseteq \ell_2) \wedge \neg q_i) \rightarrow q_i \\ & = & \bigwedge_{i \in I} ((\ell_1 \sqsubseteq \ell_2) \rightarrow q_i) \\ & = & (\ell_1 \sqsubseteq \ell_2) \rightarrow \bigwedge_{i \in I} q_i \\ & = & (\ell_1 \sqsubseteq \ell_2) \rightarrow wp(S_1, Q) \end{array}$$

The precondition of label test statements is thus strictly stronger than the one derivable in the weakest precondition calculus:

$$\begin{array}{rcl} Q_1 \backslash (\ell_1 \sqsubseteq \ell_2) \cup Q_2 & \simeq & (\ell_1 \sqsubseteq \ell_2 \rightarrow wp(S_1, R)) \wedge wp(S_2, R) \\ & \Rightarrow & (\ell_1 \sqsubseteq \ell_2 \rightarrow wp(S_1, R)) \wedge (\neg(\ell_1 \sqsubseteq \ell_2) \rightarrow wp(S_2, R)) \\ & = & wp(\textbf{if } \ell_1 \sqsubseteq \ell_2 \textbf{ then } S_1 \textbf{ else } S_2, R) \end{array}$$

6.3 Type Inference for the Intermediate Representation

The type system for the intermediate representation is turned into *two* sets of algorithmic typing rules for all instructions. The first set defines an algorithm that computes all pc and Δ type annotations in a "forward" fashion; the second set uses this information to compute constraint sets Q in a "backward" way.

More precisely, the algorithm verifies the well-typedness of a method m by finding a suitable type mapping according to the requirements of the well-typedness Definition 5.9 on page 79. Let m be a method such that $\mathsf{msig}(m) = (\Gamma, pc, Q, Q')$.

1. A control flow graph of the method is computed, such that two instructions i and i' are linked whenever $(m, i) \overset{\mathrm{I}}{\longmapsto} (m, i')$.

2. Starting from the initial node $\mathsf{mentry}(m)$, the type information pc, and an empty Δ stack, the information pc and Δ is computed recursively for all nodes. This information can be directly derived according to the typing rules in Section 5.4 on page 75.

3. Starting from the final node $\mathsf{mexit}(m)$ and the postcondition Q', the precondition information is recursively computed for all nodes. For the nontrivial cases (assignment sequences and conditionals), the precondition is computed in the same way as in the algorithmic high-level type system in Figure 6.2 on page 87. This second phase relies on the information derived in the first phase.

4. It is checked that at the address $\mathsf{mexit}(m)$, the derived pc label is equal to the pc label declared in the signature, and that Δ is empty. Likewise, the derived precondition for the address $\mathsf{mentry}(m)$ must be implied by the set Q declared in the signature.

If the algorithm can derive a type mapping for each method, then the IR program is well-typed.

6.4 Implementation

I have developed a prototype implementation of the verification framework. The implementation is a single program called `dsdtool`, consisting of 4822 lines of OCaml code. The sources can be downloaded from my homepage [Gra11]. The tool combines all of the languages and type systems presented in this thesis, and processes a given DSD program in the following stages:

DSD Source Code Representation and Parser I have invented a concrete syntax for DSD programs, which resembles actual Java programs with class declarations and other syntactic structure (explained below). The parser reads the program from a file, and transforms it into a DSD program specification P_{DSD} as defined in Section 2.1 on page 15, checking all well-formedness conditions. I have generated the DSD parser with the `ocfg` parser generator, which is part of the `fjavac` project, a Java compiler by Stephen Tse and Steve Zdancewic [TZ06] written in OCaml.

DSD Type Inference After a program has been transformed into a DSD program specification, it is type checked as described earlier in this chapter (see Section 6.2 on page 86). The type checking phase essentially annotates the abstract syntax tree

with type information. If the type checking fails, the tool prints an informative error message.

DSD Program Interpreter For testing purposes, a simple interpreter is implemented to execute DSD programs. The interpreter expects an entry-point method with the name `main`, which is executed on an empty heap using default values for the program arguments. The return value of the `main` method is shown as the output value of the program.

Bytecode Compiler This step is a direct implementation of the $\mathsf{compile}(P_{\mathrm{DSD}})$ function presented in Section 4.2 on page 56, which compiles the DSD program into a bytecode program P_{BC}.

Bytecode Interpreter Another interpreter is used to run the bytecode program, with the `main` method as the entry point. Again, the return value of the `main` method is output.

Translation to Intermediate Representation The BC2IR algorithm to translate the bytecode program to IR code is implemented as presented in Section 5.3 on page 70.

IR Type Inference The IR program is type checked as presented above in Section 6.3 on page 90. The algorithm prints an error message if type checking fails. However, this does not happen for IR programs derived from DSD high-level programs due to the type preservation result (see Section 5.5 on page 80).

IR Interpreter For testing purposes, yet another interpreter is used to execute the IR program. As shown by the semantics preservation result in Section 5.3.3 on page 75, the execution always leads to the exact same return value as the bytecode execution.

The implementation is meant to be a proof of concept, and shows that it is indeed feasible to perform a verification as described in this thesis. With a functional language like OCaml, I was able to implement the languages, the interpreters, and the type inferences very closely according to the language definitions, semantics, and algorithmic typing rules, respectively.

One of the design goals of the software was code readability. I have not taken any particular care in terms of optimizing the code for efficiency and speed. However, performance is not an issue for the small example programs with which I have tested the tool, each of them consisting of less than 100 lines of DSD code.

```
class Buffer {
  fdelta : BOT;
  contents : FDELTA;

  [ this.fdelta ~> xdelta ]          // precondition
  (BOT)                              // pc label
  BOT read(xdelta: BOT) : XDELTA {  // types of this, arguments, ret
    ret := this.contents;            // sample implementation
  }
  [ ]                                // postcondition

  [ xdelta ~> this.fdelta ]
  (BOT)
  BOT write(xdelta: BOT, s: XDELTA) : BOT {
    this.contents := s;
    ret := 0;
  }
  [ ]
}

class Main {
  fdelta : BOT;

  [ xdelta ~> file.fdelta, file.fdelta ~> xdelta ]
  (BOT)
  BOT sendFile(xdelta: BOT, file: BOT, srv: BOT, tmp: XDELTA) : BOT {
    if (file.fdelta ~> srv.fdelta) then {
      tmp := file.read(file.fdelta);
      ret := srv.write(file.fdelta, tmp);
    }
    else {
      ret := 0;
    }
  }
  [ ]

  [ ]
  (BOT)      // sample initialization of buffer objects
  BOT initAndSend(xdelta: BOT, f: BOT, s: BOT) : BOT {
    f := new Buffer(TOP, 42);
    s := new Buffer(BOT, 1234);
    ret := this.sendFile(TOP, f, s, 0);
    ret := s.contents;
  }
  [ ]

  [ ]
  (BOT)     // entry point
  BOT main(xdelta: BOT) : BOT {
    ret := new Main(BOT);
    ret := ret.initAndSend(BOT, 0, 0);
  }
  [ ]
}
```

Figure 6.3: Example program in textual representation

Concrete syntax The concrete syntax of DSD loosely follows well-known conventions from C, C++, and Java, such as comments starting with '//', and execution blocks enclosed in curly braces { and }. A program contains a list of class declarations, and each class declaration contains a list of field and method declarations. The code contains DSD-specific type annotations, namely method signatures (including pre- and post-conditions) and field types. As mentioned above, the `dsdtool` translates the program into a program specification P_{DSD}. Therefore, despite the richer structural information that is possible in the concrete syntax, the tool makes sure that, for example, two fields in different classes with the same name are declared with the same field type.

Figure 6.3 shows how the example program from Section 2.2 on page 22 looks in the concrete syntax. Note how `mbody(sendFile)` can be written directly in the concrete syntax (the operator `~>` is the textual representation of the $\sqsubseteq$ operator). In the example program, I have actually specified a bit more to make the program executable. The buffer operations are implemented using a simple `contents` field. Also, there is a sample environment that is explicitly created: a file buffer initialized with contents 42 of dynamic domain $\top$, and a server buffer initialized with contents 1234 of dynamic domain $\bot$.

As $\top$ and $\bot$ evaluate to HIGH and LOW in Sue's domain lattice from the introduction, a flow from the file to the server is not allowed. Consequently, the label test `sendFile` fails in this configuration, and the server contents is not overwritten. In contrast, $\top$ and $\bot$ both evaluate to DEF in Dave's domain lattice, thus the flow is allowed and the server contents is overwritten with 42. The contents field of the server is simply handed back to the main method and returned as the *ret* value, which is output by the DSD interpreter of the `dsdtool`. Thus, depending on the security environment, the program has the result 42 or 1234.

Usage The tool is simply run by providing the name of the DSD (high-level) source file as an argument. Alternatively, there is a small graphical front-end in Java, shown in Figure 6.4 on the facing page. The front-end enables an easier testing of programs. It lets the programmer edit a DSD program (upper part of the window), which is sent to `dsdtool` by pressing the "Analyze" button. For a better overview, the different parts of the output, representing the compilation and verification stages, are presented in different tabs.

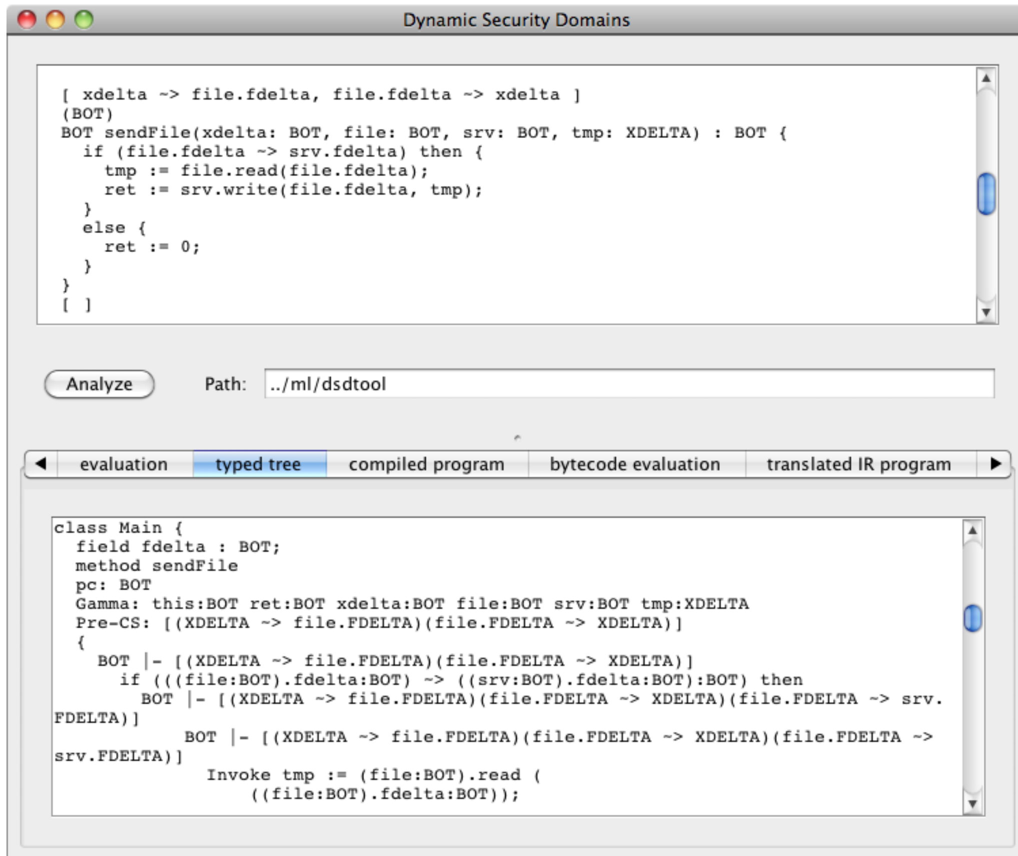

Figure 6.4: Screenshot of the `dsdtool` graphical frontend

7 Related Work

As mentioned in the introduction, the presented verification framework for universal noninterference is based on a number of previous works in the research field. In this chapter, I present and compare related language-based approaches, and explain to what extend they have influenced the DSD framework. I concentrate on those works that are most relevant to the thesis; a more complete and general overview of the field of language-based information flow security can be found in a survey paper [SM03].

7.1 Static Language-Based Information Flow Analysis

First static program analyses for secure information flow have been developed by Denning and Denning [DD77] and by Andrews and Reitman [AR80] for simple imperative WHILE languages. They classify program variables according to their security level, and automatically verify whether a given program does not leak information from higher to lower security levels.

However, it was not until the mid-1990s when the field saw a major increase in interest. A main reason was the information flow type system by Volpano, Smith, and Irvine [VSI96], who not only gave a natural type-based view on the Denning-style analysis, but also connected it to an extensional security property that was defined in terms of the semantics of a standard imperative WHILE language. More precisely, a program P is considered secure if

$$\forall s, s', t, t'.\ s, P \Downarrow s' \ \wedge\ t, P \Downarrow t' \ \wedge\ s =_L t \ \Rightarrow\ s' =_L t'$$

where s, s', t, t' are program states, the $=_L$ relation denotes the equality on all "low" variables, and $s, P \Downarrow t$ stands for the execution of the program. This definition is a natural extension of the Strong Dependency property developed by Cohen [Coh77]. The work by Volpano et al. led to a large number of other type-based approaches.

Type-based analyses Numerous works examined ways to extend the type system to other, more expressive languages. Banerjee and Naumann [BN05] presented an information flow type system for object-oriented languages with heaps, and parametrized the equivalence relation in the security property with bijections to account for different allocations of objects in the two executions of the program, an approach I have reused in this thesis.

A number of functional languages have been extended with mechanisms for secure information flows. The most prominent examples are the SLam calculus [HR98], which extends a typed λ-calculus, and CoreML^2 [PS02], which provides information flow inference for a core ML language. The latter work evolved into a complete implementation called Flow Caml [Sim03], an extension of the Objective Caml language with a type system tracing information flow.

Due to incompleteness, there are always secure programs that are not typable. To increase the precision of the analysis and thus reduce the number of such "false negatives", Hunt and Sands have presented a flow-sensitive version of the Volpano-Smith type system [HS06; HS11], where the analysis follows the program flow, and updates the security type (classification) of a variable when it is assigned with data of a different security type. For example, when a "high" variable is assigned a "low" value, then the variable is considered "low" after the update, which permits the verification of a larger range of programs.

Due to aliasing, the flow-sensitive approach does not easily transfer to types of fields used in object-oriented languages. In order to safely update a field type at a field assignment, one would need to make sure that the field type information is updated for all references to that object, and for no other references. Statically tracking references to an object requires some form of pointer analysis techniques, for example a region type system [BGH10]. Since such pointer analyses are only indirectly related to secure information flows, the type system presented here is flow-insensitive.

Program logics for secure information flows Program logics are in general more precise than type systems. A special challenge for the logic-based verification of non-interference is that it is formulated as a 2-safety property: the property talks about two executions of the program, which does not fit existing verification techniques that usually verify 1-safety properties.

Several relational logics have been developed that can relate the two initial and the two final states of both executions. Those include a relational logic for WHILE

languages [Ben04], for languages with heaps [Yan07], and for languages with objects [ABB06]. In comparison to type systems, the relations directly express equalities between related program states, thus the analyses do not necessarily depend on a given security lattice.

Alternatively, noninterference has been reformulated as a program property called self-composition [BDR04], which is based on a *single* execution of the program $P; P'$, where P' is a modification of P that operates on copies of the original variables. The idea has been refined by Terauchi and Aiken [TA05], such that program modifications only have to be applied to subprograms of P, which greatly simplifies the verification for automatic analyses. The self-composition approach has been embedded into a dynamic logic for Java programs [DHS05] as well as into a VDM-style program logic [BH07].

An existing program logic can also increase the precision of a type system. Indeed, the pre- and post-predicates for constraint sets used in the type systems of this thesis are based on classic (unary) program logics: the axiomatic definition of constraint sets a derived from Hoare logic [Hoa69], while their algorithmic inference is a simplified form of the weakest precondition calculus [Dij75].

7.2 Static Analysis of Privacy-Aware Software

All the above works assume a fixed assignment of security levels to variables, fields, or other data objects, as well as a given information flow policy. The static analyses are then presented with respect to the previously defined levels and policies.

Prior to this thesis, several approaches have been proposed to account for dynamic security environments, and privacy-aware software. Among the most notable works is Jif ("Java with Information Flow") by Andrew Myers [Mye99]. The Jif language is a superset of Java, where Java objects and variables can be annotated with "labels", which are an extended version of security levels. The Jif project includes an extensive information flow type system, and a fully implemented type inference. To date, Jif constitutes the most comprehensive implementation of an information-flow–secure language.

Similar to DSD, the Jif language includes a runtime representation of labels and the security policy which can be queried by a conditional, as shown by the following example from the Jif reference manual [Cho+06]:

```
if (L1 <= L2) { ... } else { ... }
```

The typing rule for such an *if-label* conditional resembles the corresponding label test rule in the DSD type system.

The development of the Jif language is steered by actual software examples, that is, the researchers aim to incorporate high-level information flow requirements into the language and the type system. On the other hand, less focus is put on the semantic

formalization of the security property that the type system is intended to verify. Consequently, there is no proof of correctness for the type system yet, although it should be added that the typing rules look reasonable and no obviously unintuitive behaviour has been found so far.

While there exists no soundness proof for the entire type system, the specific fragment of dynamic security labels and runtime inspections has been formalized by Lantian Zheng and Andrew Myers in a functional language called λ_{DSec} [ZM07]. The language features security labels as first-class values. Dependent pair and function types can be used to type arbitrary data and function arguments with a security label value. The authors give a precise security notion, and provide a soundness result for their type-based analysis.

The λ_{DSec} language strongly influenced the development of DSD. Indeed, DSD extends standard information flow concepts in object-oriented languages towards dynamic domains as featured in λ_{DSec}. A number of additional features such as higher-order functions and heaps make λ_{DSec} even more expressive than my language.

I have not fully followed the Jif and λ_{DSec} approaches in this thesis for two reasons: first, I wanted to formalize a soundness result for a Java-like language. As there does not exist a formal connection between the functional language λ_{DSec} to the Java-like language Jif, the soundness results for λ_{DSec} cannot be readily applied to Jif. Second, and more important, the translation to bytecode has neither been examined for Jif nor λ_{DSec}, and it would be very hard to do so. The Jif compiler is implemented by a transformation of Jif code to plain Java source code. This code may include calls to the Jif runtime library to handle dynamic flow checks, runtime representations of security labels, and others. The Java code is in turn compiled to JVM bytecode by a standard Java compiler. It is difficult to precisely pinpoint the end-to-end guarantee of compiled code, that is, the exact effect of the two compilers and the runtime system on the user-specified security policy.

Bandhakavi, Winsborough, and Winslett have developed the imperative language RTI [BWW08], which stands for Role-based Trust management for Information flow. In the RTI language, data are statically associated with roles (sets of principals), which may be queried at runtime to ensure that data flow securely. More recently, Broberg and Sands have proposed Paralocks [BS10], a mechanism where the security roles of data may depend on the state of parametrized flow locks, which are boolean values that are part of the program state. They have presented a functional language with a construct to inspect these locks at runtime. Both works include a type-based information flow analysis, with rules for the role or lock inspection that are similar to the T-IfLabel rule of DSD.

Nevertheless, there are a number of conceptional differences to the DSD approach. Roles in RTI and locks in Paralocks allow for more expressive security policies. However, they cannot be passed around in the language; instead, RTI and Paralocks extend a standard language with special language syntax and semantics for role and lock

operations, respectively. In contrast, DSD is a rather modest extension to a standard object-oriented language, with security domains as first-class values. This also enables a straight-forward translation to a standard bytecode language. Most important, however, is that the notion of "dynamic" security environments in Paralocks and RTI differs from this thesis: there, the focus lies on security environments that can be updated dynamically at runtime, possibly causing a declassification of data. (Indeed, the major purpose of roles is to provide a controlled way of declassification simply by adding a principal to a specific role.) The type system is defined with respect to a given initial policy and a set of principals, and then statically tracks and approximates the runtime changes to the security environment. In contrast, a DSD program is executed with respect to a security environment that is fixed for each particular execution; the type system analyses the program abstractly and gives a proof that the program is secure for any environment (domain values and policies) that may occur.

Finally, our own previous works [Gra08; GB09] have laid the foundations for a type-based analysis of privacy-aware programs. While most of the high-level DSD language has already been present in these papers, there has not been a proper parametrization of the security policy in the definition of noninterference, and the verification of DSD bytecode programs has only been outlined.

In summary, all the presented approaches tackle aspects of dynamic information flow security in expressive and sophisticated ways. However, they are in my opinion not directly applicable to the mobile code scenario, as they are all defined on the source code level only, and there is no translation given for an analysis on a lower level. Indeed, many of these works require a domain-specific language or larger extensions to the syntax, which makes it harder to build on existing techniques for the compilation to JVM-like bytecode and the subsequent information flow analysis for bytecode.

7.3 Bytecode Information Flow Analysis

A first information flow type system for an unstructured bytecode language has been presented by Kobayashi and Shirane [KS02]. They applied ideas from the field of Typed Assembly Languages [Mor+99] to information flow security. Bytecode languages are usually defined using a small-step semantics, hence the type system assigns typing information to each instruction address. Semantically, noninterference is defined in terms of a bisimulation: two bytecode executions continually need to reach low-equivalent program states at corresponding instruction addresses. (This is in contrast to the Volpano-Smith system, where the typing judgements and their interpretations refer to entire program statements.)

Following Kobayashi and Shirane, a number of similar approaches have been developed which handle more advanced language features. In particular, the MOBIUS project [Bar+06] has examined the type-based certification of bytecode for information

flows, as presented in the works by Gilles Barthe et al. [BPR07; BR05]. Additionally, the authors have presented a compilation from a fragment of Java that preserves security types [BRN06]. This research had a large influence on the bytecode language and the type-preserving compilation presented in this thesis.

To handle indirect information flows with the *pc* level correctly, these type systems rely on a computation of control dependence regions [Bal93] to determine the parts of the code that are governed by a conditional branching instruction. The soundness result for the type system relies on the correctness of the computed control dependence regions, expressed in form of safe over-approximation properties. In contrast, the typed assembly language SIF by Medel, Compagnoni, and Bonelli [MCB05] includes pseudo-instructions that are used as control dependence markers. Here, the correctness of these region markers can be directly checked by the type system. My low-level type system uses this latter approach with small modifications.

Beyond type systems, relational program logics for bytecode programs have been used to analyse information flow security in a more precise manner. An example are Lennart Beringer's Relational Shape Descriptions [Ber10], which provide an expressive way to describe correspondences between two related program states, such that fine-grained noninterference properties can be expressed. Additionally, the proof rules place no restrictions on the control flow structure of the program.

In the MOBIUS project, information flow type systems and logics have been expressed in a generic bytecode logic, such that one can build on a general framework for proof-carrying code (PCC) [Nec97]. In this thesis, I have followed the PCC paradigm, although the proof certificates are much simpler and require a larger trusted computing base: the certificates are just the type derivations, and the code consumer must trust the correctness of the soundness proof and the type checker.

In the presence of symbolic expressions in the analysis, such as used for the constraint sets here, it becomes rather cumbersome to define a type system on the bytecode level. For this reason, I have chosen a simplified version of the approach by Demange, Jensen, and Pichardie [DJP10] where a bytecode language is "disassembled" into an intermediate representation, which, among others, eases the symbolic reasoning required for DSD.

To my knowledge, this thesis presents the first definition of a privacy-aware bytecode language that contains constructs for inspecting the security environment. Furthermore, it is the first application of the stackless intermediate representation by Demange et al. to verify information flow security properties of bytecode programs.

7.4 Other Previous Work

In the remainder of this chapter, I give an overview of research that is in some way related to the DSD approach, but which differs in a fundamental way such that I have not pursued them in this thesis.

Runtime enforcement of information flow policies Instead of statically showing that a program respects any security environment in which it is eventually executed, one could also aim to ensure this behaviour directly at runtime, a technique also sometimes called "taint checking". As explained in the introduction, it is not possible to safely detect all violations against a security policy at runtime given an arbitrary program, because an indirect flow of information may occur when a control flow path is taken in which a certain action is *not* performed.

To address this problem, hybrid techniques have been proposed, where a program is first statically verified by a light-weight analysis which ensures that the program is in a form suitable for dynamic monitoring. In the functional language λ^{deps^+} [SST07], all values are explicitly paired with their security domains, such that an external monitor program may throw invalid flow exceptions as required. Another intriguing attempt is to modify the bytecode prior to execution in a way that even indirect flows are always detected by a purely dynamic monitoring mechanism. This approach is taken by the RIFLE framework [Vac+04].

A runtime monitor requires larger changes to the execution environment, and also needs to be included into the static security guarantee given by the code producer. Apart from the runtime overhead that arises from constantly checking information flows, a formal security guarantee also needs to include the correctness of the monitor.

Concurrent programs Concurrently-running programs that operate on the same shared data at runtime are an entire challenge on its own for information flow analysis. The problem is that compositionality cannot be preserved in general: the parallel execution of two secure programs may cause insecure information flows. The problem has been addressed in various ways; usually, the amount of parallelism is restricted in some form, for example by assuming certain scheduler properties, or by employing a specific synchronization mechanism (see e.g. [MS01; MSS11; SS00]). In any case, an information flow analysis for concurrent programs requires a more detailed execution model to talk about program configurations that may occur during the execution of the program. In contrast, it is sufficient for the verification of universal noninterference to define DSD in terms of a simple big-step semantics.

Termination-sensitivity and timing leaks The notion of universal noninterference defined in this thesis is termination-insensitive, as it only considers executions that

terminate eventually. This implies that programs that loop forever are trivially secure according to the definition. However, the termination behaviour itself is a covert channel that could depend on secret data and thus leak information, which could be exploited by running the program repeatedly [Ask+08]. More generally, termination leaks can be seen as a special case of timing leaks, where the observable execution time of a program or a subroutine depends on confidential data.

A definition of noninterference that is based on a bisimulation relation between the two executions can be used to verify that the termination or timing behaviour is the same in both states. As such bisimulations are standard for the analysis of concurrent programs, many of the works mentioned above [MS01; SS00] also provide an analysis of termination-sensitive or timing-based noninterference.

In general, a bisimulation-based security definition restricts the number of secure programs in contrast to an end-to-end property, since programs need to behave observationally equivalent during the entire execution, at least at specific control points. The advantage is not only that covert information leaks caused by timing or termination can be detected, but also that infinite execution traces can be analysed, so that programs that are meant to run forever can be certified for noninterference.

Declassification An important advancement of the definition of noninterference is declassification, which originates from the observation that the classic security notion is too restricted for practical purposes. In certain situations, it should be allowed to "violate" the information flow policy in a controlled fashion. One typical example is a password input prompt, which at minimum reveals whether an input string is equal to the (secret) password, thus leaking one bit of information about the secret. To express that this declassification of private information is acceptable in this specific example, a more relaxed security definition is required. This is a challenging quest, because declassification exceptions should be easy to define, but they should not be too permissive. As a result, a large number of works on declassification has been published. The approaches have been broadly classified according to the circumstances under which declassification is permitted [MS04; SS05]: among others, it may depend on the kind of information ("what"), the execution point ("where"), or the authority of a user ("who"). Examples for these three dimensions can be found in the type-based analyses [LZ05], [AB05], and [MSZ04], respectively, and there are many more approaches that account for multiple dimensions (e.g. [BNR07; MR07]).

Note that in most cases, a declassification policy cannot be defined as an end-to-end property, but instead requires some program execution model to precisely talk about situations at runtime when declassification is acceptable. Such a model may already be present when concurrent programs are to be verified, thus the analysis of declassification and of concurrency may benefit from each other.

Fine-grained label models Jif, RTI, and Paralocks all feature a policy model with complex labels that are more expressive than the simple domain lattices of DSD. Apart from the increased expressivity, each principal may define the information flow policy for the data he or she owns. This is in contrast to DSD policy lattices, which are meant to be defined by some administrator, for example, the owner of a smartphone. As long as the policy is fixed for each execution, however, I argue that this difference does not fundamentally affect the semantics of the ensured security property. In fact, DSD lattices can be easily expressed in these other label models, and the order on the fine-grained labels can be compiled into a (possibly exponentially large) lattice of simple security domains.

Data flow analysis and other language-based techniques Several works build on existing data and control flow analysis techniques. For example, abstract noninterference is a technique based on abstract interpretation [GM04] and relies on a PER model of secure information flow [SS01]. Similarly, data flow analysis has been applied to bytecode to detect dependencies between private data and public variables [GS05; Bia+07].

Another approach is an information flow analysis based on program dependence graphs (PDGs) and appropriate slicing techniques [HS09; HPR88] to precisely determine the dependencies between variables, values, and control flows. This increased precision has a number of advantages, for example, the flow of information can be followed path-sensitively. A common criticism of PDGs is the problematic scalability to larger programs.

Analyses not based on programming languages All the mentioned works so far are defined on a programming language level. Beyond that, there are numerous other works in which information flow properties are defined on a more abstract level. In fact, noninterference was first examined for systems given as automata [GM82; Rus92]. Other examples include the MAKS toolkit which enables the specification of a large number of noninterference properties for event systems [Man03], a security process algebra [FG01], or an extension of UML to define information flow properties [Jür02].

Unfortunately, there seems to be a certain gap between these abstract specifications of information flow security on the one side, and the corresponding semantic property on the programming language level on the other side. It requires a lot of (often manual, application-specific) work to formally define a refinement that preserves the security property. Nevertheless, it would be interesting to see how existing approaches can be used for the specification of mobile code with dynamic security domains.

8 Conclusion and Outlook

Preserving the confidentiality of personal information is a highly desirable function of modern smartphones. In particular, third-party applications may become a risk of information leakage, be it due to malicious developers or simple programming errors. With the rising amount of personal data stored on smartphones on the one hand and smartphone applications on the other hand, the importance of a formal information flow analysis is greater than ever.

As argued in the introduction, the typical application scenario of mobile software that is executed on many different devices poses a number of challenges: the security environment is user-defined, both the code producer and the consumer want a reliable security assertion, and the software is distributed as bytecode. None of the existing analyses cover all these aspects.

In this thesis, I have addressed these challenges by a novel verification framework. I have shown that a moderate extension of standard object-oriented high-level and bytecode languages enables privacy-aware software that can be executed safely in arbitrary security environments. I have explained how this universal noninterference property can be formally and statically verified by type systems for both high-level and low-level code. Finally, the prototypical implementation demonstrates the practical suitability of this approach.

This work thus presents an important step to the vision of downloadable mobile software that is certified for information flow security, and that is thus guaranteed to respect the security environment defined by the user. Throughout the thesis, I concentrated on the verification foundations and assumed several simplifications to convey the key ideas. Turn the vision into reality, two lines of future work can be

identified: first, how to embed the approach into real-life software architectures and development frameworks, and second, how to improve the analysis and extend the notion of universal noninterference to make it more usable for realistic requirements. In the remainder of this chapter, I therefore give an outlook on these two aspects.

8.1 The DSD Approach in the Real World

The presented verification framework touches various aspects of software development and distribution processes, thus there are a number of opportunities to integrate the work presented in this thesis into real-life frameworks.

Extending real programming languages To reduce complexity and avoid issues that are orthogonal to universal noninterference, the languages presented here are simplified theoretical toy languages. They cover only the core of full-featured languages like Java, C#, JVM or MSIL. However, as I opted for object-oriented, imperative languages instead of a core language with functional objects like Featherweight Java, the DSD language provides a strict subset of Java or C#. Thus, DSD directly proposes how to equip these languages with facilities to write universally noninterferent code.

As the required DSD-specific extensions are rather modest, they could alternatively be provided as a standardized library that implements the runtime representation of labels and the label test statement. This should also ease the transformation of legacy code to make it certifiable for flexible information flow requirements.

Software engineering with dynamic domains The encapsulation of data and security domains supports a modular software design. In the introductory example, the caller of the `sendFile` method does not need to deal with security-related aspects of sending the file's contents, at least not if an error recovery mechanism for the `else` case is implemented. Alternatively, it is possible to specify security requirements for subparts of a program by using constraint set annotations in method signatures. I believe that this makes it possible to specify privacy-aware software on a more abstract level, for example by extending the UMLsec approach [Jür02] with dynamic security domains and policies.

Integration into existing development and distribution architectures The DSD-specific extensions should be integrated into existing mobile software development and distribution frameworks. For example, the high-level information flow type system can be run alongside the usual data type system in the programmer's integrated development environment. The maintainers of mobile app distribution platforms, such as the Android or iOS app store, usually have a suite of verification tools that they apply to submitted applications anyway; this suite could be extended by the fully automatic

DSD bytecode verifier. Note that both type-based analyses are then to be performed with respect to the standardized DSD library.

Already today, mobile software is equipped with a list of access rights and other security-relevant demands which the software needs to run properly. The user has to grant these rights to download and install the software. This mechanism can be seen as a certification: it is guaranteed that the program will not perform other critical actions that do not occur in the list. This mechanism could be extended to certificates that state universal noninterference.

Alternatively, existing runtime security mechanisms could be leveraged to include information flow requests. Currently, for example, the mobile operating system may request an access permission from the user at runtime when the application tries to access some resource. Likewise, the label test conditionals could be implemented by generating a request whether a certain information flow should be granted by the user.

8.2 Towards a More Flexible Framework

In this section, I outline how the core verification framework could be improved to make it suitable for more application scenarios. This includes combinations with some related work described in the preceding chapter.

Improved analysis precision Several of the approaches that aim at a type-based analysis with an increased precision could be adapted to the DSD type system. For example, one could include a flow-sensitive analysis [HS06; HS11]. A different approach is to obtain a more exact heap representation by integrating concepts of alias or region type systems, or even from separation logic [TT97; BGH10; Rey02]. Another idea is to replace the Hoare-style pre- and postconditions by VDM-style assertions that can relate the states before and after an execution [BH07].

For an easier integration into an existing security model of an operating system that is based on user accounts, principals-based labels like the ones used in Jif, RTI or Paralocks [Cho+06; BWW08; BS10] could be used. Another way is to enhance information flow control in DSD with access control mechanisms [BN05].

More general type dependencies One could extend the DSD type environments to enable additional forms of security types that depend on domain values. As a straight-forward extension, one could allow multiple domain fields within a class, and multiple domain variables within a variable environment, which can all be used in the respective type environments. Another possibility is to allow the type $\Gamma(x) = f_\delta$ in the method signatures, which refers to the f_δ field of the object whose method is called, as illustrated by the introductory security API in Figure 1.3 on page 10.

The more flexible typing schemes are allowed, however, the more complicated becomes the definition of state equivalence. For example, if security types may refer to arbitrary domain fields and variables, such as a field whose type is the value of an f_δ field of a different object, the object equivalence relation cannot be defined on a local per-object basis anymore. Moreover, it becomes harder to ensure that the typing rules are monotone and well-founded. For these reasons, I have refrained from these extensions to the type dependencies.

Updating x_δ and f_δ The type system prevents updates to x_δ variables and f_δ fields. It should be noted, however, an assignment $x_\delta := \ell$ is fine as long as the contents of x_δ before the assignment is lower than the domain it is updated with, that is, the evaluation of ℓ. The same applies to the f_δ field of an object. For example, a public `Buffer` object that has the domain LOW in its f_δ field could be reclassified at runtime to become confidential, simply by setting its f_δ field to the domain HIGH. Since the confidentiality level of data is increased, the program stays universally noninterferent. The type system can check this "upwards" assignment by imposing condition of the form $x_\delta \sqsubseteq_Q \ell$ for such updates.

Erasure and declassification in DSD Updating domain fields and variables also enables other security-related operations. For example, information in a `Buffer` object can be erased by overwriting the `contents` field with a harmless value and then setting f_δ to LOW. This scenario is secure, but not typable with the presented type system.

Setting f_δ to a lower value without overwriting the `contents` field amounts to a declassification of data. Indeed, the explicit change of security types in form of overwriting f_δ fields resembles the manipulation of flow locks in the paralocks approach [BS10] (albeit with less syntactic overhead). It should thus be interesting to integrating their semantic security formalization based on increasing attacker knowledge to find an appropriate weakening of universal noninterference.

Security polymorphism The universal noninterference property states that a program is noninterferent for any values the domain variables x_δ and domain fields f_δ have, and for any security policy. It would be interesting to explore other techniques beyond DSD to express and verify that a program is secure for a larger number of security environments.

Interestingly, flow-sensitive systems [HS06; Ber10] can be used for this purpose: they can infer fine-grained dependencies between input and output values, for example, that the final value of variable y depends on the initial value of variable x. Thus, the program is secure (at least for variable y) for any concrete security environment that assigns x a lower security domain than y.

In general, there are different ways to use generalized types for information flow security for multiple security environments: DSD and Jif allow a program to adapt its behaviour according to the required type, while flow-sensitive systems take a standard program and determine the set of concrete types that can be assigned to it. This observation supports the idea that there is a connection to the different manifestations of classic data type polymorphism.

Types derived by flow-sensitive systems loosely correspond to parametric polymorphism: They express data dependencies and thus provide a schema that can be instantiated to concrete security level types, similar to parametric polymorphism, where universally quantified type schemata can be instantiated to concrete data types.

Privacy-aware programs, in contrast, can execute different code for different security environments, such that the effective policy can always be respected. There is a connection to inheritance polymorphism in object-oriented languages, where dynamic method dispatch can be used to always choose that method implementation which respects the invariants of the respective dynamic class. The explicit label check construct in DSD and Jif, however, seem to correspond most closely to intensional polymorphism with type introspection [HM95], where the type of some data can be inspected at runtime using a special `typecase` construct. The static analysis of the construct can use the refined type information for each case, just as my type system used the information about the inspected flow in the constraint set of the `then` branch.

Due to these parallels, it seems a promising approach to elaborate the works in the field of data type polymorphism, and examine to what extend they can be applied to type-based information flow analysis. This could provide a starting point to certify programs for a much wider range of user-defined security settings, including Dave's and Sue's.

A

Correctness Proof for the High-Level Type System

This appendix contains the complete soundness proof for the high-level type system. It is split into two parts: in the first part, the correctness of labels assigned by expression typing judgements is shown. The second part shows that typable statements are universally noninterferent; the main correctness theorem for well-typed programs follows as a corollary.

Unless otherwise noted, the identifiers and indices used in the proofs directly refer the respective identifiers and indices used in the operational semantics and the typing rules. Moreover, we simplify notation and ignore the class information of objects. That is, given a heap h and a location r, $h(r)$ shall refer directly to the field valuation.

A.1 Expression Typing Soundness

The following lemma states that the expression typing rules are sound, that is, if they assign to an expression a label whose evaluations in two equivalent states are visible, then the expression evaluates to indistinguishable values in both states.

Lemma A.1 *If* $\Gamma \vdash e : \ell$*, then for all states* $\sigma = (s, h)$ *and* $\sigma' = (s', h')$*, for all partial bijections* β*, for all domain lattices* $\diamond$ *and all domains* $k \in Dom^\diamond$ *such that* $\vdash^\diamond \sigma \sim_\beta^{\Gamma,k} \sigma'$*, if* $[\![\ell]\!]_\sigma^\diamond \leq^\diamond k$ *and* $[\![\ell]\!]_{\sigma'}^\diamond \leq^\diamond k$*, then* $[\![e]\!]_\sigma^\diamond \sim_\beta [\![e]\!]_{\sigma'}^\diamond$*.*

PROOF By induction over e.

- $e = n$ or $e = \top$ or $e = \bot$. Then e is a constant and $[\![e]\!]^{\diamond}_{\sigma} \sim_{\beta} [\![e]\!]^{\diamond}_{\sigma'}$ holds.
- $e = x$. Then $\ell = \Gamma(x)$. With Lemma 3.1 on page 33, we have $\langle\Gamma(x)\rangle^{\diamond}_{s} = [\![\Gamma(x)]\!]^{\diamond}_{\sigma} \leq^{\diamond} k$, thus by definition $s(x) \sim_{\beta} s'(x)$, hence $[\![x]\!]^{\diamond}_{\sigma} \sim_{\beta} [\![x]\!]^{\diamond}_{\sigma'}$.
- $e = \pi.f$. $\Gamma \vdash \pi : \ell_{\pi}$ and $\ell = \Phi^{\pi}(f) \sqcup \ell_{\pi}$. Since $[\![\ell]\!]^{\diamond}_{\sigma} \leq^{\diamond} k$ and $[\![\ell]\!]^{\diamond}_{\sigma'} \leq^{\diamond} k$, we get $[\![\ell_{\pi}]\!]^{\diamond}_{\sigma} \leq^{\diamond} k$ and $[\![\ell_{\pi}]\!]^{\diamond}_{\sigma'} \leq^{\diamond} k$ and by induction $[\![\pi]\!]^{\diamond}_{\sigma} \sim_{\beta} [\![\pi]\!]^{\diamond}_{\sigma'}$. We define $r = [\![\pi]\!]^{\diamond}_{\sigma}$ and $r' = [\![\pi]\!]^{\diamond}_{\sigma'}$. From $r \sim_{\beta} r'$ follows $\vdash^{\diamond} h(r) \sim^{k}_{\beta} h'(r')$. With Lemma 3.1 on page 33, we know $\left\langle\Phi(f)\right\rangle^{\diamond}_{r} = [\![\Phi^{\pi}(f)]\!]^{\diamond}_{\sigma}$, which is lower than k since $[\![\ell]\!]^{\diamond}_{\sigma} \leq^{\diamond} k$. With the definition of object equivalence, we have $h(r)(f) \sim_{\beta} h'(r')(f)$ and thus $[\![\pi.f]\!]^{\diamond}_{\sigma} \sim_{\beta} [\![\pi.f]\!]^{\diamond}_{\sigma'}$.
- $e = e_1 \circ e_2$. We have $\Gamma \vdash e_1 : \ell_1$ and $\Gamma \vdash e_2 : \ell_2$ and $\ell = \ell_1 \sqcup \ell_2$. Since $[\![\ell]\!]^{\diamond}_{\sigma} \leq^{\diamond} k$ and $[\![\ell]\!]^{\diamond}_{\sigma'} \leq^{\diamond} k$, the same holds for the sublabels ℓ_1 and ℓ_2. Therefore, we can apply the lemma inductively, and get $[\![e_1]\!]^{\diamond}_{\sigma} \sim_{\beta} [\![e_1]\!]^{\diamond}_{\sigma'}$ and $[\![e_2]\!]^{\diamond}_{\sigma} \sim_{\beta} [\![e_2]\!]^{\diamond}_{\sigma'}$. Since the language does not feature any pointer operations, we have indeed equalities, and thus get $[\![e_1 \circ e_2]\!]^{\diamond}_{\sigma} = [\![e_1 \circ e_2]\!]^{\diamond}_{\sigma'}$, hence $[\![e]\!]^{\diamond}_{\sigma} \sim_{\beta} [\![e]\!]^{\diamond}_{\sigma'}$. ■

We now show that the labelling function $\mathsf{labelof}_{\Gamma}$ is monotone and well-founded.

Lemma A.2 *If $\Gamma \vdash e : \ell$ and $\Gamma \vdash \ell : \ell'$, then $\ell' \sqsubseteq_{\emptyset} \ell$.*

PROOF By induction over e.

- $e = n$ or $e \in \{\top, \bot\}$. Then $\ell = \bot$, hence $\ell' = \bot$. The label order $\bot \sqsubseteq_{\emptyset} \ell$ follows by definition.
- $e = x$. Then $\ell = \Gamma(x) \in \{x_{\delta}, \top, \bot\}$, thus $\ell' \in \{\Gamma(x_{\delta}), \bot, \bot\} = \{\bot\}$. The label order $\bot \sqsubseteq_{\emptyset} \ell$ follows by definition.
- $e = \pi.f$. Then $\ell = \mathsf{labelof}_{\Gamma}(\pi.f) = \Phi^{\pi}(f) \sqcup \mathsf{labelof}_{\Gamma}(\pi)$. It follows

  $$\ell' = \mathsf{labelof}_{\Gamma}(\Phi^{\pi}(f) \sqcup \mathsf{labelof}_{\Gamma}(\pi)) = \mathsf{labelof}_{\Gamma}(\Phi^{\pi}(f)) \sqcup \mathsf{labelof}_{\Gamma}(\mathsf{labelof}_{\Gamma}(\pi)).$$

  Since π is a subexpression of e, we can apply the lemma inductively and get $\mathsf{labelof}_{\Gamma}(\mathsf{labelof}_{\Gamma}(\pi)) \sqsubseteq_{\emptyset} \mathsf{labelof}_{\Gamma}(\pi)$. (*)

  We now make a case distinction on $\Phi^{\pi}(f)$:

  - If $\Phi^{\pi}(f) \in \{\bot, \top\}$, then $\mathsf{labelof}_{\Gamma}(\Phi^{\pi}(f)) = \bot$.
  - If $\Phi^{\pi}(f) = \pi.f_{\delta}$, then $\mathsf{labelof}_{\Gamma}(\Phi^{\pi}(f)) = \bot \sqcup \mathsf{labelof}_{\Gamma}(\pi)$.

  It follows $\mathsf{labelof}_{\Gamma}(\Phi^{\pi}(f)) \sqsubseteq_{\emptyset} \mathsf{labelof}_{\Gamma}(\pi)$. Together with (*), we get $\ell' \sqsubseteq_{\emptyset} \ell$.
- $e = e_1 \circ e_2$. Then $\ell = \ell_1 \sqcup \ell_2$ and $\ell' = \ell'_1 \sqcup \ell'_2$. By induction $\ell'_1 \sqsubseteq_{\emptyset} \ell_1$ and $\ell'_2 \sqsubseteq_{\emptyset} \ell_2$, we get $\ell' \sqsubseteq_{\emptyset} \ell$. ■

Lemma A.3 *For all expressions $e \in Exp$, there exists an $n \in \mathbb{N}$ such that for all $m \geq n$,* $\mathsf{labelof}^m_\Gamma(e) \equiv_\emptyset \bot$.

Proof By induction over e. It suffices to show that there exists a number $n \in \mathbb{N}$ such that $\mathsf{labelof}^n_\Gamma(e) \equiv_\emptyset \bot$, because all subsequent applications of $\mathsf{labelof}_\Gamma$ do not change the label, due to $\Gamma \vdash \bot : \bot$.

- $e = n$ or $e \in \{\top, \bot\}$. Then $n = 1$, because $\mathsf{labelof}_\Gamma(e) = \bot$.
- $e = x$: Then n=2, since $\mathsf{labelof}_\Gamma(x) \in \{x_\delta, \bot, \top\}$ and $\mathsf{labelof}_\Gamma(\mathsf{labelof}_\Gamma(x)) = \bot$.
- $e = \pi.f$: By induction, there exist $n' \in \mathbb{N}$ such that for all $m \geq n'$, $\mathsf{labelof}^m_\Gamma(\pi) \equiv_\emptyset \bot$. Let $n = n' + 1$. Then

$$\begin{aligned}
\mathsf{labelof}^n_\Gamma(\pi.f) &\equiv_\emptyset \mathsf{labelof}^{n'}_\Gamma(\mathsf{labelof}_\Gamma(\pi.f)) \\
&\equiv_\emptyset \mathsf{labelof}^{n'}_\Gamma(\Phi^\pi(f) \sqcup \mathsf{labelof}_\Gamma(\pi)) \\
&\equiv_\emptyset \mathsf{labelof}^{n'}_\Gamma(\Phi^\pi(f)) \sqcup \mathsf{labelof}_\Gamma(\mathsf{labelof}^{n'}_\Gamma(\pi)) \\
&\equiv_\emptyset \mathsf{labelof}^{n'}_\Gamma(\Phi(f)[\pi.f_\delta / f_\delta]) \sqcup \bot \quad \text{(by induction)} \\
&\equiv_\emptyset \mathsf{labelof}^{n'}_\Gamma(\Phi(f)[\pi.f_\delta / f_\delta])
\end{aligned}$$

  We now make a case distinction over $\Phi(f)$.

  - $\Phi(f) \in \{\bot, \top\}$. Then $\Phi^\pi(f) \in \{\top, \bot\}$, hence by definition $\mathsf{labelof}^{n'}_\Gamma(\Phi(f)) \equiv_\emptyset \bot$ (because $n' > 1$).
  - $\Phi(f) = f_\delta$. Then

$$\begin{aligned}
\mathsf{labelof}^{n'}_\Gamma(\Phi(f)[\pi.f_\delta / f_\delta]) &\equiv_\emptyset \mathsf{labelof}^{n'}_\Gamma(\pi.f_\delta) \\
&\equiv_\emptyset \mathsf{labelof}^{n'-1}_\Gamma(\mathsf{labelof}_\Gamma(\pi.f_\delta)) \\
&\equiv_\emptyset \mathsf{labelof}^{n'-1}_\Gamma(\Phi^\pi(f_\delta) \sqcup \mathsf{labelof}_\Gamma(\pi)) \\
&\equiv_\emptyset \mathsf{labelof}^{n'-1}_\Gamma(\bot \sqcup \mathsf{labelof}_\Gamma(\pi)) \\
&\equiv_\emptyset \mathsf{labelof}^{n'}_\Gamma(\pi) \\
&\equiv_\emptyset \bot \quad \text{(by induction).}
\end{aligned}$$

- $e_1 \;\mathbf{op}\; e_2$. Then there exist n_1, n_2 such that by induction $\mathsf{labelof}^{n_i}_\Gamma(e_i) \equiv_\emptyset \bot$ for $i \in \{1,2\}$. Let $n = \max(n_1, n_2)$. Then $\mathsf{labelof}^n_\Gamma(e_1 \;\mathbf{op}\; e_2) \equiv_\emptyset \bot$. ■

For the soundness proofs of statement typing rules, we need a corollary that follows directly from meta-label monotonicity (Lemma A.2), well-foundedness (Lemma A.3), and soundness of expressions (Lemma A.1).

Corollary A.4 *If $\Gamma \vdash e : \ell$, then for all states σ and σ' and all partial bijections β, for all domain lattices $\diamond$ and all domains $k \in Dom^\diamond$ such that $\vdash^\diamond \sigma \sim^{\Gamma,k}_\beta \sigma'$, $[\![\ell]\!]^\diamond_\sigma \leq^\diamond k$ implies* $[\![\ell]\!]^\diamond_\sigma = [\![\ell]\!]^\diamond_{\sigma'}$.

PROOF By contradiction. Suppose $[\![\ell]\!]^{\diamond}_{\sigma} \neq [\![\ell]\!]^{\diamond}_{\sigma'}$. Then by definition $[\![\ell]\!]^{\diamond}_{\sigma} \not\simeq_{\beta} [\![\ell]\!]^{\diamond}_{\sigma'}$. (*)

Let $\Gamma \vdash \ell : \ell'$, i.e. $\mathsf{labelof}_{\Gamma}(\ell) = \ell'$. We get with Lemma A.2 that $\ell' \sqsubseteq_{\emptyset} \ell$. Since $\sigma \models^{\diamond} \emptyset$, we get with the label order soundness theorem 3.3 that $[\![\ell']\!]^{\diamond}_{\sigma} \leq^{\diamond} [\![\ell]\!]^{\diamond}_{\sigma} \leq^{\diamond} k$. By Lemma A.1 and with (*), it must be $[\![\ell']\!]^{\diamond}_{\sigma'} \not\leq^{\diamond} k$. We thus get $[\![\ell']\!]^{\diamond}_{\sigma} \neq [\![\ell']\!]^{\diamond}_{\sigma'}$. We can repeat this argument arbitrarily often and get that for all n, $[\![\mathsf{labelof}^{n}_{\Gamma}(\ell)]\!]^{\diamond}_{\sigma'} \not\leq^{\diamond} k$. However, this contradicts Lemma A.3, which says there is some n such that $\mathsf{labelof}^{n}_{\Gamma}(\ell) = \bot$, i.e. $[\![\mathsf{labelof}^{n}_{\Gamma}(\ell)]\!]^{\diamond}_{\sigma'} = k^{\diamond}_{\top} \leq^{\diamond} k$. ■

A.2 Statement Typing Soundness

We first show properties for single statement executions.

Lemma A.5 *Let σ, σ' be program states, and let $\diamond$ be a domain lattice. Let $\sigma \xrightarrow{S}{}^{\diamond} \sigma'$ and $\Gamma, pc \vdash \{Q\}\ S\ \{Q'\}$, and let all methods be well-typed with respect to their signatures. Then:*

1. $[\![pc]\!]^{\diamond}_{\sigma} = [\![pc]\!]^{\diamond}_{\sigma'}$
2. *if $\sigma \models^{\diamond} Q$, then $\sigma' \models^{\diamond} Q'$.*

PROOF We show the two parts separately.

1. By induction over the operational semantics. For T-SKIP, T-IF, T-WHILE, and sequence statements, the induction hypothesis is used. For all assignment statements, the premises $x \not\sqsubseteq pc$ and $f \not\sqsubseteq pc$ ensure that the program counter label is not changed. For T-CALL, we apply the induction hypothesis on the method body.

2. We observe that with the given interpretation (satisfiability) of constraint sets, Q are unary state predicates where $\cup$ works as a conjunction and $\Rightarrow$ is indeed an implication (see Lemma 3.5 on page 34). It is easy to see that the pre- and post-sets can be derived in Hoare logic using the consequence rule appropriately. We only examine one case closer:

   - For the T-CALL rule, the induction hypothesis can be applied, because it can be shown that if σ satisfies $Q_m[\overline{e}/\mathsf{margs}(m)][\pi/this]$ (where Q_m is the required constraint set of the method), then the initial state

     $$([this \mapsto r] \cup [\mathsf{margs}(m) \mapsto [\![\overline{e}]\!]^{\diamond}_{s_1,h_1}] \cup [ret \mapsto \mathsf{defval}], h)$$

     of the method satisfies Q_m. From the induction hypothesis and from well-typedness we get that the final state of the method satisfies the ensured constraint set Q'_m, and it is easy to see that $Q'_m[x/ret]$ holds in σ' (the final

state of the method call), because the *ret* variable is assigned to x, and Q'_m does not contain any other variable by the well-formedness requirement of method signatures. Additionally, we know that σ satisifes a set Q that does not contain x or any field f other than f_δ. Since f_δ cannot be overwritten, the final state of the method σ' also satisifes Q. ■

In the following, the notation id in $\vdash^\diamond (s,h) \sim^{\Gamma,k}_{\mathsf{id}} (s',h')$ refers to a partial bijection $\mathsf{id} = \{(r,r) \mid r \in \mathsf{dom}(h) \cap \mathsf{dom}(h')\}$, i.e., the identity relation ranging over the locations on which both heaps are defined. (Usually, h' is a post-heap and thus a proper extension of h, the pre-heap, so the identity is defined on the locations that already existed in h.) The following lemma states that no data at a domain below or disjoint from the value of pc is written.

Lemma A.6 *Let $\sigma_1 = (s_1,h_1)$ and $\sigma_2 = (s_2,h_2)$ be program states, and let $\diamond$ be a domain lattice. Let $\Gamma, pc \vdash \{Q\}\ S\ \{Q'\}$ and $(s_1,h_1) \xrightarrow{S}{}^\diamond (s_2,h_2)$. Let $\sigma_1 \models^\diamond Q$, and let all methods be well-typed with respect to their signature. Then:*

- *If $[\![pc]\!]^\diamond_{s_1,h_1} \not\leq^\diamond k$, then $\vdash^\diamond s_1 \sim^{\Gamma,k}_{\mathsf{id}} s_2$ and $\vdash^\diamond h_1 \sim^{k}_{\mathsf{id}} h_2$.*

PROOF By induction over the derivation of the operational semantics.

Without loss of generality, we assume $\Gamma, pc \vdash \{Q\}\ S\ \{Q'\}$ has not been derived by the subsumption rule. If it has, then there is a judgement $\Gamma, pc \vdash \{Q_0\}\ S\ \{Q'_0\}$ that has not been derived by the subsumption rule. Since $Q \Rightarrow Q_0$, σ_1 also satisifies Q_0, and we can show the theorem on that judgement.

Since all other rules are syntax-directed, we can identify them by the statement S, and make a case distinction on S:

- **skip**. The desired equalities $\vdash^\diamond s_1 \sim^{\Gamma,k}_{\mathsf{id}} s_1$ and $\vdash^\diamond h_1 \sim^{k}_{\mathsf{id}} h_1$ follow by definition.

- $S_1\ ;\ S_2$. Then we have $\sigma_1 \xrightarrow{S_1}{}^\diamond \sigma_2$ and $\sigma_2 \xrightarrow{S_2}{}^\diamond \sigma_3$ from the operational semantics and $\Gamma, pc \vdash \{Q\}\ S_1\ \{Q'\}$ and $\Gamma, pc \vdash \{Q'\}\ S_2\ \{Q''\}$ from the type rules. Store equivalence: By induction and since by Lemma A.5 the pc labels have not changed and Q' holds in σ_2, we get $\vdash^\diamond s_1 \sim^{\Gamma,k}_{\mathsf{id}} s_2$ and $\vdash^\diamond s_2 \sim^{\Gamma,k}_{\mathsf{id}} s_3$. Transitivity of equivalence results in $\vdash^\diamond s_1 \sim^{\Gamma,k}_{\mathsf{id}} s_3$. Heap equivalence is shown in a very similar way.

- **if** e **then** S_1 **else** S_2.

  Since $[\![pc]\!]^\diamond_{\sigma_1} \not\leq^\diamond k$, we know $[\![pc \sqcup \ell]\!]^\diamond_{\sigma_1} \not\leq^\diamond k$ for any label ℓ. Let S_i be the substatement S_1 or S_2 depending on the value of $[\![e]\!]^\diamond_{\sigma_1}$. Then by induction on S_i, $\vdash^\diamond \sigma_1 \sim^{\Gamma,k}_{\mathsf{id}} \sigma_2$.

- **if** $\ell_1 \sqsubseteq \ell_2$ **then** S_1 **else** S_2.

  This case is identical to the case **if** e **then** S_1 **else** S_2, but we additionally have to show that $\sigma_1 \models^\diamond Q, \ell_1 \sqsubseteq \ell_2$ if the **then** branch is taken, but this follows from the operational semantics.

- **while** e **do** S.

  If $[\![pc]\!]^\diamond_{\sigma_1} \not\leq^\diamond k$, we know $[\![pc \sqcup \ell]\!]^\diamond_{\sigma_1} \not\leq^\diamond k$ for any label ℓ.

  - Case $[\![e]\!]^\diamond_{\sigma_1} = 0$: $\vdash^\diamond s_1 \sim^{\Gamma,k}_{\mathsf{id}} s_1$ (or $\vdash^\diamond h_1 \sim^{k}_{\mathsf{id}} h_1$) holds trivially.
  - Case $[\![e]\!]^\diamond_{\sigma_1} \neq 0$: We have $\sigma_1 \xrightarrow{S}{}^\diamond \sigma_2$ and $\Gamma, pc \sqcup \ell \vdash \{Q\}\ S\ \{Q\}$, hence by induction $\vdash^\diamond \sigma_1 \sim^{\Gamma,k}_{\mathsf{id}} \sigma_2$. Also, we have $\sigma_2 \xrightarrow{\textbf{while}\ e\ \textbf{do}\ S}{}^\diamond \sigma_3$ and the original $\Gamma, pc \vdash \{Q\}$ **while** e **do** S $\{Q\}$. Again by induction and with Lemma A.5, $\vdash^\diamond \sigma_2 \sim^{\Gamma,k}_{\mathsf{id}} \sigma_3$. Transitivity gives $\vdash^\diamond \sigma_1 \sim^{\Gamma,k}_{\mathsf{id}} \sigma_3$.

- $x := e$. Store equivalence: We get $s_2 = s_1[x \mapsto [\![e]\!]^\diamond_{s_1,h_1}]$. Since $\ell \sqcup pc \sqsubseteq_Q \Gamma(x)$ and σ_1 satisfies Q, we get $[\![\Gamma(x)]\!]^\diamond_{\sigma_1} \not\leq^\diamond k$. The variable x cannot be x_δ, hence we get by the definition of store equivalence $\vdash^\diamond s_1 \sim^{\Gamma,k}_{\mathsf{id}} s_2$. Heap equivalence is trivial, since $\vdash^\diamond h_1 \sim^{k}_{\mathsf{id}} h_1$.

- $\pi.f := e$. Let $\ell_f = \Phi^\pi(f)$. Since $pc \sqsubseteq_Q \ell_f$, we know $[\![\ell_f]\!]^\diamond_{\sigma_1} \not\leq^\diamond k$. By the definition of heap equivalence and since no f_δ field is written, it follows $\vdash^\diamond h_1 \sim^{k}_{\mathsf{id}} h_2$. Store equivalence is trivial, since $\vdash^\diamond s_1 \sim^{\Gamma,k}_{\mathsf{id}} s_1$.

- $x :=$ **new** $C(\overline{e})$.

  As for store equivalence, we have an assignment of the variable x, hence we can use the same argument as for the ordinary assign rule. Heap equivalence is trivial, as the heap has not been changed for all $a \in \mathsf{dom}(h_1) \cap \mathsf{dom}(h_2)$, so we get $\vdash^\diamond h_1 \sim^{k}_{\mathsf{id}} h_2$.

- $x := \pi.m(\overline{e})$.

  As for store equivalence, we have an assignment of the variable x, hence we can use the same argument as for the ordinary assign rule. Other than that, no variables are changed in the caller's store.

  All that is left to show is heap equivalence. Since $pc \sqsubseteq_{Q'} pc_m[\overline{e}/\mathsf{margs}(m)][\pi/\mathit{this}]$ and σ_1 satisfies Q', we know $[\![pc_m]\!]^\diamond_{\hat{\sigma}_1} \not\leq^\diamond k$, where $\hat{\sigma}_1$ is the method's initial state. Since the method m is well-typed, we have $\Gamma_m, pc_m \vdash \{Q_m\}\ \mathsf{mbody}(m)\ \{Q'_m\}$. Hence we can apply the lemma inductively and get $\vdash^\diamond h_1 \sim^{k}_{\mathsf{id}} h_2$. ■

The following is the main soundness theorem.

Theorem A.7 *If* $\Gamma, pc \vdash \{Q\}\ S\ \{Q'\}$ *and all methods are well-typed with respect to their signatures, then for all states* $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$ *such that* σ_1 *and* σ_1' *satisfy* Q*, for all partial bijections* β*, for all domain lattices* $\diamond$ *and for all domains* $k \in Dom^\diamond$*,*

$$\vdash^\diamond \sigma_1 \sim_\beta^{\Gamma,k} \sigma_1' \wedge \sigma_1 \xrightarrow{S}{}^\diamond \sigma_2 \wedge \sigma_1' \xrightarrow{S}{}^\diamond \sigma_2' \Rightarrow\ \vdash^\diamond \sigma_2 \sim_\gamma^{\Gamma,k} \sigma_2'$$

for some partial bijection $\gamma \supseteq \beta$*.*

PROOF By induction over the derivation of the operational semantics.

Without loss of generality, we assume $\Gamma, pc \vdash \{Q\}\ S\ \{Q'\}$ has not been derived by the subsumption rule. If it has, then there is a judgement $\Gamma, pc \vdash \{Q_0\}\ S\ \{Q_0'\}$ that has not been derived by the subsumption rule. Since $Q \Rightarrow Q_0$, σ_1 also satisifies Q_0, and we can show the theorem with that judgement.

Since all other rules are syntax-directed, we can identify them by the statement S, and make a case distinction on S:

- **skip**.

  From the assumption it follows $\vdash^\diamond \sigma_1 \sim_\beta^{\Gamma,k} \sigma_1'$.

- $S_1 ; S_2$.

  Then we have $\sigma_1 \xrightarrow{S_1}{}^\diamond \sigma_2$ and $\sigma_2 \xrightarrow{S_2}{}^\diamond \sigma_3$ as well as $\sigma_1' \xrightarrow{S_1}{}^\diamond \sigma_2'$ and $\sigma_2' \xrightarrow{S_2}{}^\diamond \sigma_3'$ from the operational semantics and $\Gamma, pc \vdash \{Q\}\ S_1\ \{Q'\}$ and $\Gamma, pc \vdash \{Q'\}\ S_2\ \{Q''\}$ from the type rules. By induction, we get $\vdash^\diamond \sigma_2 \sim_\beta^{\Gamma,k} \sigma_2'$. From Lemma A.5, we know σ_2 and σ_2' satisfy Q'. Again by induction, we get $\vdash^\diamond \sigma_3 \sim_\beta^{\Gamma,k} \sigma_3'$.

- **if** e **then** S_1 **else** S_2.

  We have $\Gamma \vdash e : \ell$. We make a case distinction on the evaluation of ℓ:

  - Let $[\![\ell]\!]_{\sigma_1}^\diamond \not\leq^\diamond k$. With Corollary A.4 on page 115, we get $[\![\ell]\!]_{\sigma_1'}^\diamond \not\leq^\diamond k$. Therefore, $[\![pc \sqcup \ell]\!]_{\sigma_1}^\diamond \not\leq^\diamond k$ and $[\![pc \sqcup \ell]\!]_{\sigma_1'}^\diamond \not\leq^\diamond k$. We can thus infer by Lemma A.6 on page 117 that $\vdash^\diamond \sigma_1 \sim_{\mathsf{id}}^{\Gamma,k} \sigma_2$ and $\vdash^\diamond \sigma_1' \sim_{\mathsf{id}}^{\Gamma,k} \sigma_2'$. By definition of the equivalence relations, we get $\vdash^\diamond \sigma_2 \sim_\beta^{\Gamma,k} \sigma_2'$.
  - Let $[\![\ell]\!]_{\sigma_1}^\diamond \leq^\diamond k$. Then by Lemma A.1 on page 113, we have $[\![e]\!]_{\sigma_1}^\diamond \sim_\beta [\![e]\!]_{\sigma_1'}^\diamond$, and even $[\![e]\!]_{\sigma_1}^\diamond = [\![e]\!]_{\sigma_1'}^\diamond$, because one cannot branch over a memory location.

    Therefore, the same branch S_i is taken in both executions. $\sigma_1 \xrightarrow{S_i}{}^\diamond \sigma_2$ and $\sigma_1' \xrightarrow{S_i}{}^\diamond \sigma_2'$ and $\Gamma, pc \sqcup \ell \vdash \{Q\}\ S_i\ \{Q'\}$ imply by induction $\vdash^\diamond s_2 \sim_\beta^{\Gamma,k} s_2'$.

- **if** $\ell_1 \sqsubseteq \ell_2$ **then** S_1 **else** S_2.

  The same arguments can be used as for the case **if** e **then** S_1 **else** S_2; however, the induction hypothesis is only applicable if we show that σ_1 or σ_2 satisfy the

extended constraint set $Q, \ell_1 \sqsubseteq \ell_2$ when the **then** branch is taken, but this follows from the operational semantics.

- **while** e **do** S.

  This case is similar to the case for T-IF. We get $\Gamma \vdash e : \ell$. With Corollary A.4 on page 115, we know that if ℓ evaluates to a domain that is not below or equal to k, it also does so in the other state. Hence $pc \sqcup \ell$ evaluates to a domain that is not below or equal to k in both states, and with Lemma A.6 on page 117, each final state is id-equivalent to its corresponding initial state, and thus the final states are β-equivalent.

  Let therefore $[\![\ell]\!]^{\diamond}_{\sigma_1} \leq^{\diamond} k$. Then with Lemma A.1 on page 113 $[\![e]\!]^{\diamond}_{\sigma_1} = [\![e]\!]^{\diamond}_{\sigma'_1}$.

  - Case $[\![e]\!]^{\diamond}_{\sigma_1} = [\![e]\!]^{\diamond}_{\sigma'_1} = 0$: We have to show $\vdash^{\diamond} \sigma_1 \sim^{\Gamma,k}_{\beta} \sigma'_1$, which follows from the assumption.
  - Case $[\![e]\!]^{\diamond}_{\sigma_1} = [\![e]\!]^{\diamond}_{\sigma'_1} \neq 0$: In this case, we have $\sigma_1 \xrightarrow{S}{}^{\diamond} \sigma_2$ and $\sigma'_1 \xrightarrow{S}{}^{\diamond} \sigma'_2$ and $\Gamma, pc \sqcup \ell \vdash \{Q\}\ S\ \{Q\}$, hence by induction $\vdash^{\diamond} \sigma_2 \sim^{\Gamma,k}_{\beta} \sigma'_2$. Also, we have $\sigma_2 \xrightarrow{\textbf{while}\ e\ \textbf{do}\ S}{}^{\diamond} \sigma_3$ and $\sigma'_2 \xrightarrow{\textbf{while}\ e\ \textbf{do}\ S}{}^{\diamond} \sigma'_3$ and the original judgement $\Gamma, pc \vdash \{Q\}$ **while** e **do** S $\{Q\}$. Also, we know from Lemma A.5 on page 116 that σ_2 and σ'_2 satisfy Q, and the pc did not change. By induction, $\vdash^{\diamond} \sigma_3 \sim^{\Gamma,k}_{\beta} \sigma'_3$.

- $x := e$.

  Since the heaps do not change, they remain β-equivalent. We show store equivalence as follows: Let x^* be a variable such that $\langle\Gamma(x^*)\rangle^{\diamond}_{s_2} \leq^{\diamond} k$. We need to show $s_2(x^*) \sim_{\beta} s'_2(x^*)$.

  It is easy to see that $\langle\Gamma(x^*)\rangle^{\diamond}_{s_1} = \langle\Gamma(x^*)\rangle^{\diamond}_{s_2}$, because x_{δ} cannot be updated. Therefore $\langle\Gamma(x^*)\rangle^{\diamond}_{s_1} \leq^{\diamond} k$, and thus $s_1(x^*) \sim_{\beta} s'_1(x^*)$ by store equivalence.

  Let $x^* \neq x$. Then the variable is unchanged, therefore we have $s_2(x^*) = s_1(x^*)$ and $s_1(x^*) \sim_{\beta} s'_1(x^*)$ and $s'_1(x^*) = s'_2(x^*)$. If $x^* = x$, then with $pc \sqcup \ell \sqsubseteq_Q \Gamma(x)$, we get $[\![\ell]\!]^{\diamond}_{\sigma_1} \leq^{\diamond} k$, and with Lemma A.1 on page 113, $[\![e]\!]^{\diamond}_{\sigma_1} \sim_{\beta} [\![e]\!]^{\diamond}_{\sigma'_1}$, which implies $s_2(x) \sim_{\beta} s'_2(x)$.

- $\pi.f := e$.

  As the stores are not changed, they remain β-equivalent. We show heap equivalence as follows: Let $(r, r') \in \beta$ be a pair of β-related locations. Then by definition $\vdash^{\diamond} h_2(r) \sim^{k}_{\beta} h'_2(r')$.

  - If $[\![\pi]\!]^{\diamond}_{\sigma_1} \neq r$ and $[\![\pi]\!]^{\diamond}_{\sigma'_1} \neq r'$, then the objects are not changed, hence we get trivially $\vdash^{\diamond} h_2(r) \sim^{k}_{\beta} h'_2(r')$.

- Let $[\![\pi]\!]^{\diamond}_{\sigma} = r$. Let $f^* \in \mathsf{dom}(h_2(r))$ be a field. In this case, we show that if $\langle \Phi(f^*) \rangle^{\diamond}_{h_2(r)} \leq^{\diamond} k$, then $\vdash^{\diamond} h_2(r) \sim^k_{\beta} h'_2(r')$.

  Very similar to the argument for variable assignment above, we observe that $\langle \Phi(f^*) \rangle^{\diamond}_{h_1(r)} = \langle \Phi(f^*) \rangle^{\diamond}_{h_2(r)}$, because the f_δ field is not updated. It follows $\langle \Phi(f^*) \rangle^{\diamond}_{h_1(r)} \leq^{\diamond} k$ and thus $h_1(r)(f^*) \sim_{\beta} h'_1(r')(f^*)$.

  Let $f^* \neq f$. Since f^* is not changed, we get $h_2(r)(f^*) \sim_{\beta} h'_2(r')(f^*)$.

  Let $f^* = f$. With Lemma 3.1 on page 33, $[\![\Phi^{\pi}(f)]\!]^{\diamond}_{\sigma_1} = \langle \Phi(f) \rangle^{\diamond}_{h_1(r)} \leq^{\diamond} k$. With $\ell_1 \sqcup \ell_2 \sqsubseteq_Q \Phi^{\pi}(f)$, we get that $[\![\ell_1]\!]^{\diamond}_{\sigma_1} \leq^{\diamond} k$ and $[\![\ell_2]\!]^{\diamond}_{\sigma_1} \leq^{\diamond} k$, hence with Lemma A.1, $[\![e]\!]^{\diamond}_{\sigma_1} \sim_{\beta} [\![e]\!]^{\diamond}_{\sigma'_1}$ and $[\![\pi]\!]^{\diamond}_{\sigma_1} \sim_{\beta} [\![\pi]\!]^{\diamond}_{\sigma'_1}$. Since β is a bijection and $r \sim_{\beta} r'$, $[\![\pi]\!]^{\diamond}_{\sigma'_1} = r'$. We get $h_2(r)(f) = h_2([\![\pi]\!]^{\diamond}_{\sigma_1})(f) = [\![e]\!]^{\diamond}_{\sigma_1} \sim_{\beta} [\![e]\!]^{\diamond}_{\sigma'_1} = h'_2([\![\pi]\!]^{\diamond}_{\sigma'_1}) = h'_2(r')(f)$.

- $[\![\pi]\!]^{\diamond}_{\sigma'_1} = r'$ can be shown in a similar fashion.

• $x := \textbf{new}\, C(\overline{e})$.

  Let r and r' be the fresh locations for the new object in h_1 and h'_1. We define a partial bijection $\gamma = \beta \cup \{(r, r')\} \supseteq \beta$. For store equivalence, it holds a similar argument as for the ordinary variable assignment rule; for the assigned variable, we have $s_2(x) = r \sim_{\gamma} r' = s'_2(x)$, thus $\vdash^{\diamond} s_2 \sim^{\Gamma,k}_{\gamma} s'_2$.

  Let $\overline{f} = \mathsf{fields}(C)$. We have $h_2(r) = (C, F)$ and $h'_2(r') = (C, F')$, where the field valuations are $F = [\overline{f} \mapsto [\![\overline{e}]\!]^{\diamond}_{\sigma_1}]$ and $F' = [\overline{f} \mapsto [\![\overline{e}]\!]^{\diamond}_{\sigma'_1}]$. We know that $\vdash^{\diamond} h_2 \sim^k_{\beta} h'_2$, since the heaps are unchanged on all locations except for r and r'. All we need to show is that $\vdash^{\diamond} h_2(r) \sim^k_{\beta} h'_2(r')$. This is true if for all $f \in \mathsf{fields}(C)$ such that $\langle \Phi(f) \rangle^{\diamond}_{F} \leq^{\diamond} k$, we get $F(f) \sim_{\beta} F'(f)$.

  Thus, let f be a field with $\langle \Phi(f) \rangle^{\diamond}_{F} \leq^{\diamond} k$. Let e be the element from the list $\overline{e}$ that initializes the field f, i.e., $F(f) = [\![e]\!]^{\diamond}_{\sigma_1}$ and $F'(f) = [\![e]\!]^{\diamond}_{\sigma'_1}$. Let $\Gamma \vdash e : \ell$. With Corollary A.4 on page 115, we have $[\![\ell]\!]^{\diamond}_{\sigma_1} = [\![\ell]\!]^{\diamond}_{\sigma'_1}$.

  From the premises of the typing rule, we know $\ell \sqsubseteq_{Q'} \Phi(f)[\overline{e}.1/f_\delta]$. Since σ_1 satisfies Q', we get with Lemma 3.3 on page 34 that $[\![\ell]\!]^{\diamond}_{\sigma_1} \leq^{\diamond} [\![\Phi(f)]\!]^{\diamond}_{\sigma_1}[\overline{e}.1/f_\delta] = \langle \Phi(f) \rangle^{\diamond}_{F} \leq^{\diamond} k$. Hence, $[\![\ell]\!]^{\diamond}_{\sigma_1} = [\![\ell]\!]^{\diamond}_{\sigma'_1} \leq^{\diamond} k$, so we get with Lemma A.1 on page 113 that $[\![e]\!]^{\diamond}_{\sigma_1} \sim_{\beta} [\![e]\!]^{\diamond}_{\sigma'_1}$, thus we have shown $F(f) \sim_{\beta} F'(f)$. It follows $\vdash^{\diamond} h_2(r) \sim^k_{\beta} h'_2(r)$, hence $\vdash^{\diamond} h_2(r) \sim^k_{\gamma} h'_2(r)$ and thus $\vdash^{\diamond} h_2 \sim^k_{\gamma} h'_2$.

• $x := \pi.m(\overline{e})$.

  Let Γ_m be the type environment for the method, and $\hat{s_1}$ and $\hat{s'_1}$ be the initial stores of the called methods build from the arguments. To apply the lemma inductively, we need to show that the initial states of the method are β-equivalent. Since

the heaps are passed without change, we only need to show equivalence for the stores, i.e., $\vdash^\diamond \hat{s_1} \sim_\beta^{\Gamma_m,k} \hat{s_1'}$.

Let x_i be a formal method parameter from the sequence $\mathsf{margs}(m)$, and let e be the expression with which x_i is initialized, and let $\Gamma \vdash e : \ell$. We observe $[\![\Gamma_m(x_i)]\!]^\diamond_{(\hat{s_1},h_1)} = \Gamma_m(x_i)[[\![\overline{e}.1]\!]^\diamond_{\sigma_1}/x_\delta] = [\![\Gamma^*(x_i)]\!]^\diamond_{\sigma_1}$. With the premises, we get $\ell \sqsubseteq_{Q'} \Gamma^*(x_i)$. Since $\sigma_1 \models^\diamond Q'$, we get with Theorem 3.3 on page 34 that $[\![\ell]\!]^\diamond_{\sigma_1} \leq^\diamond [\![\Gamma^*(x_i)]\!]^\diamond_{\sigma_1}$. Furthermore, we get with Corollary A.4 that $[\![\ell]\!]^\diamond_{\sigma_1} = [\![\ell]\!]^\diamond_{\sigma_1'}$. Thus, if $[\![\Gamma_m(x_i)]\!]^\diamond_{(\hat{s_1},h_1)} = [\![\Gamma^*(x_i)]\!]^\diamond_{\sigma_1} \leq^\diamond k$, then also $[\![\ell]\!]^\diamond_{\sigma_1} = [\![\ell]\!]^\diamond_{\sigma_1'} \leq^\diamond k$. We get with Lemma A.1 that $[\![e]\!]^\diamond_{\sigma_1} \sim_\beta [\![e]\!]^\diamond_{\sigma_1'}$, hence $\hat{s_1}(x_i) \sim_\beta \hat{s_1'}(x_i)$.

A similar argument holds for the *this* variable and the ℓ_r label. Since the return value is initialized with the same value, $\hat{s_1}(\mathit{ret}) \sim_\beta \hat{s_1'}(\mathit{ret})$ holds trivially. Therefore, $\vdash^\diamond \hat{s_1} \sim_\beta^{\Gamma_m,k} \hat{s_1'}$.

With the argument of Lemma A.5, the new states also satisfy the new constraint sets. Therefore, we can apply the lemma inductively, and get for the final states of the method call that they are γ-equivalent for some $\gamma \supseteq \beta$ with respect to Γ_m: $\vdash^\diamond \hat{s_2} \sim_\gamma^{\Gamma_m,k} \hat{s_2}'$ and $\vdash^\diamond h_2 \sim_\gamma^k h_2'$.

As the heaps are directly passed back to the caller, we get γ-equivalence of the final heaps immediately. We still need to show $\vdash^\diamond s_2 \sim_\gamma^{\Gamma,k} s_2'$.

For the assignment of the variable x, we need to show that if $\langle\Gamma(x)\rangle^\diamond_{s_2} \leq^\diamond k$, then $s_2(x) \sim_\gamma s_2'(x)$.

We observe $\langle\Gamma_m(\mathit{ret})\rangle^\diamond_{(\hat{s_2})} = [\![\Gamma_m(\mathit{ret})]\!]^\diamond_{(\hat{s_2},h_2)} = \Gamma_m(\mathit{ret})[[\![\overline{e}.1]\!]^\diamond_{\sigma_1}/x_\delta] = [\![\Gamma^*(\mathit{ret})]\!]^\diamond_{\sigma_1} = [\![\Gamma^*(\mathit{ret})]\!]^\diamond_{\sigma_1}$. From the premises, we have $\Gamma^*(\mathit{ret}) \sqsubseteq_Q \Gamma(x)$. As shown in Lemma A.5, the final state σ_1 satisfies Q. Thus we get immediately $[\![\Gamma^*(\mathit{ret})]\!]^\diamond_{\sigma_1} \leq^\diamond [\![\Gamma(x)]\!]^\diamond_{\sigma_1} = \langle\Gamma(x)\rangle^\diamond_{s_1}$. We also know $\langle\Gamma(x)\rangle^\diamond_{s_1} = \langle\Gamma(x)\rangle^\diamond_{s_2}$, because $s_2(x_\delta) = s_1(x_\delta)$ (the variable x_δ cannot be changed).

Putting it all together, we have that from $\langle\Gamma(x)\rangle^\diamond_{s_2} \leq^\diamond k$ follows $\langle\Gamma(x)\rangle^\diamond_{s_1} \leq^\diamond k$ and thus $[\![\Gamma^*(\mathit{ret})]\!]^\diamond_{\sigma_1} \leq^\diamond k$, so $\langle\Gamma_m(\mathit{ret})\rangle^\diamond_{(\hat{s_2})} \leq^\diamond k$. Since $\vdash^\diamond \hat{s_2} \sim_\gamma^{\Gamma_m,k} \hat{s_2}'$, it follows by definition $\hat{s_2}(\mathit{ret}) \sim_\gamma \hat{s_2'}(\mathit{ret})$, therefore $s_2(x) \sim_\gamma s_2'(x)$, thus $\vdash^\diamond s_2 \sim_\gamma^{\Gamma,k} s_2'$.

The actual main soundness theorem is a corollary of the theorem above:

Corollary A.8 *If P_{DSD} is a well-typed DSD program, then it is universally noninterferent.*

PROOF Since every method is well-typed with respect to its signature, we can apply Theorem A.7 on page 119. Together with the fact that the final states satisfy Q (Lemma A.5 on page 116), we get that every method is (Q,Q')-universally noninterferent. By definition, the entire program P_{DSD} is universally noninterferent. ■

B
Proof of the Semantics Preservation Result

This appendix contains the complete proof that the translation of bytecode programs to the intermediate representation with the BC2IR algorithm preserves the semantics of the program.

B.1 Call Depth Annotations

For the proof, the operational semantics of bytecode and IR programs is redefined to include call depth information, which indicates the maximum size of the call stack during an execution, that is, the maximum number of times the method call rule has been used at the leaves of the semantics derivation tree. This is a standard technique often used in proofs for program executions.

B.1.1 Bytecode Call Depth

We define the relation $(m, i, s, h, \rho) \xrightarrow{\text{B}}{}^{\diamond}_{n} (m, i', s', h', \rho')$, which means the maximal call depth of this single step is n. This is done as follows: all simple instructions have call depth 0. Method calls increase the call depth:

$$\frac{\begin{array}{c}\text{B} = \texttt{call}\ m' \\ s' = [\mathit{this} \mapsto r] \cup [\mathsf{margs}(m') \mapsto \overline{v}] \cup [\mathit{ret} \mapsto \mathsf{defval}] \\ (s', h) \overset{m'}{\underset{BC}{\Longrightarrow}}{}^{\diamond}_{n} (s'', h')\end{array}}{(m, i, s, h, \overline{v} :: r :: \rho) \xrightarrow{\text{B}}{}^{\diamond}_{n+1} (m, i+1, s, h', s''(\mathit{ret}) :: \rho)}$$

The rule for method calls relies on a big-step execution of the method. More formally, a big-step execution relation is defined as follows:

$$\frac{}{(m,i,s,h,\rho) \underset{BC}{\Longrightarrow}{}^{\diamond}_{n} (m,i,s,h,\rho)} \qquad \frac{(m,i,s,h,\rho) \xrightarrow{BC(m,i)}{}^{\diamond}_{n_1} (m,i',s',h',\rho') \quad (m,i',s',h',\rho') \underset{BC}{\Longrightarrow}{}^{\diamond}_{n_2} (m,i'',s'',h'',\rho'') \quad n = \mathsf{max}(n_1,n_2)}{(m,i,s,h,\rho) \underset{BC}{\Longrightarrow}{}^{\diamond}_{n} (m,i'',s'',h'',\rho'')}$$

The following notation is a shorthand for the execution of entire method bodies:

$$(s,h) \overset{m}{\underset{BC}{\Longrightarrow}}{}^{\diamond}_{n} (s',h') \Longleftrightarrow \exists \rho.\ (m,\mathsf{mentry}(m),s,h,\epsilon) \underset{BC}{\Longrightarrow}{}^{\diamond}_{n} (m,\mathsf{mexit}(m),s',h',\rho)$$

There is always a way to annotate an existing big-step bytecode execution with approriate call depths:

Theorem B.1 *For all states* (s,h) *and* (s',h'),

$$(s,h) \overset{m}{\underset{BC}{\Longrightarrow}}{}^{\diamond} (s',h') \Longleftrightarrow \exists n \in \mathbb{N}.\ (s,h) \overset{m}{\underset{BC}{\Longrightarrow}}{}^{\diamond}_{n} (s',h')$$

PROOF By the definition of unannotated and annotated semantics. ■

It follows that everything that is shown in this appendix for annotated bytecode executions also hold for unannotated executions as defined in Chapter 4.

B.1.2 IR Call Depth

The transitions of the IR instruction semantics are likewise extended with a call depth n. For small-step IR instructions, there is a transition $(m,i,s,s_t,h) \xrightarrow{\mathrm{I}}{}^{\diamond}_{n} (m,i',s',s_t',h')$, such that all instructions have a call depth of 0, with the exception of assignment blocks:

$$\frac{\mathrm{I} = \mathtt{block}\ \overline{a} \qquad (s,s_t,h) \overset{\overline{a}}{\Rightarrow}{}^{\diamond}_{n}(s',s_t',h)}{(m,i,s,s_t,h) \xrightarrow{\mathrm{I}}{}^{\diamond}_{n} (m,i+1,s',s_t',h')}$$

The semantics of assignment blocks is annotated as follows:

$$\frac{S = a \in \{x := e, e.f := e, x := \textbf{new}\ C(\overline{e})\} \quad (s \cup s_t, h) \xrightarrow{S}{}^{\diamond} (s' \cup s'_t, h')}{(s, s_t, h) \xrightarrow{a}{}^{\diamond}_0 (s', s'_t, h')}$$

$$\frac{\begin{array}{c} a = x := e.m(\overline{e}) \\ s_0 = [\mathit{this} \mapsto [\![e]\!]^{\diamond}_{s,s_t,h}] \cup [\mathsf{margs}(m) \mapsto [\![\overline{e}]\!]^{\diamond}_{s,s_t,h}] \cup [\mathit{ret} \mapsto \mathsf{defval}] \\ (s_0, h) \underset{IR}{\overset{m}{\Longrightarrow}}{}^{\diamond}_n (s_1, h') \\ (s' \cup s'_t) = (s \cup s_t)[x \mapsto s_1(\mathit{ret})] \end{array}}{(s, s_t, h) \xrightarrow{a}{}^{\diamond}_{n+1} (s', s'_t, h')}$$

$$\frac{}{(s, s_t, h) \overset{\epsilon}{\Rightarrow}{}^{\diamond}_0 (s, s_t, h)} \qquad \frac{\begin{array}{c} (s, s_t, h) \xrightarrow{a}{}^{\diamond}_{n_1} (s', s'_t, h') \\ (s', s'_t, h') \overset{as}{\Longrightarrow}{}^{\diamond}_{n_2} (s'', s''_t, h'') \\ n = \mathsf{max}(n_1, n_2) \end{array}}{(s, s_t, h) \overset{a::as}{\Longrightarrow}{}^{\diamond}_n (s'', s''_t, h'')}$$

The big-step relation is annotated as follows:

$$\frac{}{(m, i, s, s_t, h) \underset{IR}{\Longrightarrow}{}^{\diamond}_n (m, i, s, s_t, h)} \qquad \frac{\begin{array}{c} (m, i, s, s_t, h) \xrightarrow{IR[m,i]}{}^{\diamond}_{n_1} (m, i', s', s'_t, h') \\ (m, i', s', s'_t, h') \underset{IR}{\Longrightarrow}{}^{\diamond}_{n_2} (m, i'', s'', s''_t, h'') \\ n = \mathsf{max}(n_1, n_2) \end{array}}{(m, i, s, h, \rho) \underset{IR}{\Longrightarrow}{}^{\diamond}_n (m, i'', s'', s''_t, h'')}$$

The big-step relation for entire method bodies has the following definition:

$$(s, h) \underset{IR}{\overset{m}{\Longrightarrow}}{}^{\diamond}_n (s', h') \quad \text{if and only if}$$
$$\exists s_t.\ (m, \mathsf{mentry}(m), s, h, [\mathsf{tvars}(m) \mapsto \mathsf{defval}]) \underset{IR}{\Longrightarrow}{}^{\diamond}_n (m, \mathsf{mexit}(m), s', s_t, h')$$

Again, there is always a way to annotate an existing big-step IR execution with approriate call depths:

Theorem B.2 *For all states (s, h) and (s', h'),*

$$(s, h) \underset{IR}{\overset{m}{\Longrightarrow}}{}^{\diamond} (s', h') \Longleftrightarrow \exists n \in \mathbb{N}.\ (s, h) \underset{IR}{\overset{m}{\Longrightarrow}}{}^{\diamond}_n (s', h')$$

PROOF By the definition of unannotated and annotated semantics. ∎

It follows that everything that is shown in this appendix for annotated bytecode executions also hold for unannotated executions as defined in Section 5.1.

B.2 Proof of Semantics Preservation

First, we show that a single call-free IR step simulates a single call-free bytecode step, where the abstract stacks from the $\mathsf{BC2IR}_{\text{instr}}$ algorithm describe the operand stacks in the bytecode semantics.

Proposition B.3 *Suppose we have the transition* $(m,i,s,h,\rho) \xrightarrow{\text{B}}{}_0^{\diamond} (m,i',s',h',\rho')$. *Let* s_t *be a temporary variable store, and* as, as' *be abstract stacks and* I *be an IR instruction such that*

$$[\![as]\!]^{\diamond}_{s,s_t,h} = \rho \quad \textit{and} \quad \mathsf{BC2IR}_{instr}(i,\text{B},as) = (\text{I},as').$$

Then there exists a temporary variable store s'_t *such that* $(m,i,s,s_t,h) \xrightarrow{\text{I}}{}_0^{\diamond} (m,i',s',s'_t,h')$ *and*

$$[\![as']\!]^{\diamond}_{s',s'_t,h'} = \rho'.$$

PROOF By case distinction of the bytecode instruction B.

- B = `nop`. Then I = `block` ϵ, and $as' = as$. By bytecode and IR semantics, we get $s' = s$ and $h' = h$ and $\rho' = \rho$ and $s'_t = s_t$, hence the proposition follows from the assumptions.

- B = `push` c. Then I = `block` ϵ and $as' = c :: as$ by definition of $\mathsf{BC2IR}_{\text{instr}}$. We have the bytecode transition $(m,i,s,h,\rho) \xrightarrow{\text{B}}{}_0^{\diamond} (m,i+1,s,h,[\![c]\!]^{\diamond} :: \rho)$ and the IR transition $(m,i,s,s_t,h) \xrightarrow{\text{I}}{}_0^{\diamond} (m,i+1,s,s_t,h)$. Since $[\![as]\!]^{\diamond}_{s,s_t,h} = \rho$, it follows $[\![as']\!]^{\diamond}_{s,s_t,h} = [\![c :: as]\!]^{\diamond}_{s,s_t,h} = [\![c]\!]^{\diamond}_{s,s_t,h} :: [\![as]\!]^{\diamond}_{s,s_t,h} = [\![c]\!]^{\diamond} :: \rho = \rho'$.

- B = `pop`. Then I = `block` ϵ and $as = e :: as'$. Stores and heaps remain unchanged in both the bytecode and IR transitions. We only need to show $[\![as']\!]^{\diamond}_{s,s_t,h} = \rho'$, but this follows from $[\![e :: as']\!]^{\diamond}_{s,s_t,h} = v :: \rho'$, where v is the top-most value on the initial stack.

- B = `prim` *op*. Then I = `block` ϵ and $as = e_2 :: e_1 :: \overline{e}$ and $as' = (e_1 \,\mathbf{op}\, e_2) :: \overline{e}$ for some $\overline{e}$. We have $(m,i,s,h,v_2 :: v_1 :: \overline{v}) \xrightarrow{\text{B}}{}_0^{\diamond} (m,i+1,s,h,(v_1 \operatorname{op} v_2 :: \overline{v}))$ for some $\overline{v}$, and $(m,i,s,s_t,h) \xrightarrow{\text{I}}{}_0^{\diamond} (m,i+1,s,s_t,h)$. With the assumption, we have $\overline{v} = [\![\overline{e}]\!]^{\diamond}_{s,s_t,h}$ and $v_1 = [\![e_1]\!]^{\diamond}_{s,s_t,h}$ and $v_2 = [\![e_2]\!]^{\diamond}_{s,s_t,h}$, hence $v_1 \operatorname{op} v2 = [\![e_1 \,\mathbf{op}\, e_2]\!]^{\diamond}_{s,s_t,h}$. Therefore, $[\![as']\!]^{\diamond}_{s,s_t,h} = \rho$.

- B = `load` x. Since x appears in the bytecode, it cannot be a temporary variable, but rather $x \in \mathsf{dom}(s)$. We have I = `block` ϵ and $as' = x :: as$. The transitions are $(m,i,s,h,\rho) \xrightarrow{\text{B}}{}_0^{\diamond} (m,i+1,s,h,s(x) :: \rho)$ and $(m,i,s,s_t,h) \xrightarrow{\text{I}}{}_0^{\diamond} (m,i+1,s,s_t,h)$. Thus we have $[\![as']\!]^{\diamond}_{s,s_t,h} = [\![x]\!]^{\diamond}_{s,s_t,h} :: [\![as]\!]^{\diamond}_{s,s_t,h} = s(x) :: \rho$.

- $\mathtt{B} = \mathtt{store}\ x$. Since x appears in the bytecode, it cannot be a temporary variable, but rather $x \in \mathsf{dom}(s)$. We have $v :: \rho = [\![e :: as]\!]^{\diamond}_{s,s_t,h}$, hence $v = [\![e]\!]^{\diamond}_{s,s_t,h}$. We define $s[x \mapsto v] = s[x \mapsto [\![e]\!]^{\diamond}_{s,s_t,h}] = s'$. Then $(m, i, s, h, v :: \rho) \xrightarrow{\mathrm{B}}{}^{\diamond}_{0} (m, i+1, s', h, \rho)$.

  By definition, $\mathsf{BC2IR}_{\mathrm{instr}}(i, \mathtt{B}, e :: as) = (\mathtt{block}\ [x := e], as[t^0_i / x])$. We get the IR transition $(m, i, s, s_t, h) \xrightarrow{\mathrm{I}}{}^{\diamond}_{0} (m, i+1, s', s'_t, h)$, where $s'_t = s_t[t^0_i \mapsto s(x)]$. In other words, the IR step results in the same store s', and the heap h is unchanged. We still have to show that $[\![as[t^0_i/x]]\!]^{\diamond}_{s',s'_t,h} = \rho$. Because of the side condition $t^0_i \notin as$, we get $[\![as[t^0_i/x]]\!]^{\diamond}_{s',s'_t,h} = [\![as[t^0_i/x]]\!]^{\diamond}_{s[x\mapsto v],s_t[t^0_i \mapsto s(x)],h} = [\![as[s(x)/x]]\!]^{\diamond}_{s[x\mapsto v],s_t,h} = [\![as]\!]^{\diamond}_{s,s_t,h} = \rho$.

- $\mathtt{B} = \mathtt{bnz}\ j$. Let $\rho = v :: \overline{v}$. There are two cases, depending on the value v on top of the stack. We only treat the jump case here, i.e., where $v \neq 0$. We have the bytecode transition $(m, i, s, h, v :: \overline{v}) \xrightarrow{\mathrm{B}}{}^{\diamond}_{0} (m, j, s, h, \overline{v})$. Also, by definition $\mathsf{BC2IR}_{\mathrm{instr}}(i, \mathtt{B}, e :: \overline{e}) = (\mathtt{if}\ e\ j, \overline{e})$ and $as' = \overline{e}$. As $[\![e :: \overline{e}]\!]^{\diamond}_{s,s_t,h} = [\![as]\!]^{\diamond}_{s,s_t,h} = v :: \overline{v}$, we get $[\![e]\!]^{\diamond}_{s,s_t,h} = v \neq 0$. Therefore, the IR step is $(m, i, s, s_t, h) \xrightarrow{\mathrm{I}}{}^{\diamond}_{0} (m, j, s, s_t, h)$. Stores and heaps remain unchanged, and $\rho' = \overline{v} = [\![\overline{e}]\!]^{\diamond}_{s,s_t,h} = [\![as']\!]^{\diamond}_{s,s_t,h}$.

- $\mathtt{B} = \mathtt{jmp}\ j$. Then $\mathsf{BC2IR}_{\mathrm{instr}}(i, \mathtt{B}, as) = (\mathtt{jmp}\ j, as)$. We have the two transitions $(m, i, s, h, \rho) \xrightarrow{\mathrm{B}}{}^{\diamond}_{0} (m, j, s, h, \rho)$ and $(m, i, s, s_t, h) \xrightarrow{\mathrm{I}}{}^{\diamond}_{0} (m, j, s, s_t, h)$. Stores, heaps, and stacks remain unchanged.

- $\mathtt{B} = \mathtt{cpush}\ j$. This case is the same as for $\mathtt{B} = \mathtt{nop}$, since the IR semantics of $\mathtt{cpush}\ j$ is the same as for $\mathtt{block}\ \epsilon$.

- $\mathtt{B} = \mathtt{cjmp}\ j$. This case is the same as for $\mathtt{B} = \mathtt{jmp}\ j$, since the IR semantics of $\mathtt{cjmp}\ j$ is the sam as for $\mathtt{jmp}\ j$.

- $\mathtt{B} = \mathtt{new}\ C$. Let r be fresh in h, and $h' = h \cup [r \mapsto (C, [\mathsf{fields}(C) \mapsto \overline{v}])]$. We have the bytecode transition $(m, i, s, h, \overline{v} :: \rho) \xrightarrow{\mathrm{B}}{}^{\diamond}_{0} (m, i+1, s, h', r :: \rho)$. Also, by definition $\mathsf{BC2IR}_{\mathrm{instr}} = (i, \mathtt{B}, \overline{e} :: as) = (\mathtt{block}\ [t^0_i := \mathbf{new}\ C(\overline{e})], t^0_i :: as)$. Since we can use the same memory location r in the IR semantics, we get the IR transition $(m, i, s, s_t, h) \xrightarrow{\mathrm{I}}{}^{\diamond}_{0} (m, i+1, s, s'_t, h')$, where $s'_t = s_t[t^0_i \mapsto r]$. That is, both resulting heaps are the same, and the store s remains unchanged. Since $t^0_i \notin as$ by the side condition, we get $[\![as']\!]^{\diamond}_{s,s'_t,h'} = [\![t^0_i]\!]^{\diamond}_{s,s'_t,h'} :: [\![as]\!]^{\diamond}_{s,s_t,h} = r :: \rho = \rho'$.

- $\mathtt{B} = \mathtt{getf}\ f$. We have $\mathsf{BC2IR}_{\mathrm{instr}}(i, \mathtt{B}, e :: \overline{e}) = (\mathtt{block}\ \epsilon, e.f :: \overline{e})$. Thus, we have the bytecode transition $(m, i, s, h, r :: \rho) \xrightarrow{\mathrm{B}}{}^{\diamond}_{0} (m, i+1, s, h, h(r)(f) :: \rho)$ and the IR transition $(m, i, s, s_t, h) \xrightarrow{\mathrm{I}}{}^{\diamond}_{0} (m, i+1, s, s_t, h)$. Stores and heaps remain unchanged. By assumption $[\![as]\!]^{\diamond}_{s,s_t,h} = [\![e :: \overline{e}]\!]^{\diamond}_{s,s_t,h} = r :: \rho$, thus $[\![e]\!]^{\diamond}_{s,s_t,h} = r$. It follows $[\![as']\!]^{\diamond}_{s,s_t,h} = [\![e.f :: \overline{e}]\!]^{\diamond}_{s,s_t,h} = h([\![e]\!]^{\diamond}_{s,s_t,h})(f) :: \rho = h(r)(f) :: \rho = \rho'$.

- $\texttt{B} = \texttt{putf}\ f$. We have the transition $(m,i,s,h,v::r::\overline{v}) \xrightarrow{\texttt{B}}{}^{\diamond}_{0} (m,i+1,s,h',\overline{v})$, where $h' = h[r \mapsto h(r)[f \mapsto v]]$. The input stack is $as = e' :: e :: \overline{e}$, such that $[\![as]\!]^{\diamond}_{s,s_t,h} = [\![e' :: e :: \overline{e}]\!]^{\diamond}_{s,s_t,h} = v :: r :: \overline{v}$.

  By definition $\mathsf{BC2IR}_{\mathrm{instr}}(i,\texttt{B},as) = (\texttt{block}\ [\overline{t_i} := \overline{e}, e.f := e'], \overline{t_i})$. When we execute the IR instruction, we get exactly the heap h'. Since $[\![\overline{e}]\!]^{\diamond}_{s,s_t,h} = \overline{v}$ and $\overline{t_i} \not\in \overline{e}$ by the side condition, we get the IR transition $(m,i,s,s_t,h) \xrightarrow{\texttt{I}}{}^{\diamond}_{0} (m,i+1,s,s'_t,h')$, where $s'_t = s_t[\overline{t_i} \mapsto \overline{v}]$. Finally, we have $[\![as']\!]^{\diamond}_{s,s'_t,h'} = [\![\overline{t_i}]\!]^{\diamond}_{s,s'_t,h'} = \overline{v} = \rho'$. ■

To show the semantic preservation for big-step transitions, we now consider a fixed IR program that has been created by the BC2IR algorithm, i.e., $P_{\mathrm{IR}} = \mathsf{BC2IR}(P_{\mathrm{BC}})$, along with the arrays $AS_{in}[m,i]$ and $AS_{out}[m,i]$. We first show a result for call-free big-step transitions.

Proposition B.4 *Suppose we have* $(m,i,s,h,\rho) \underset{BC}{\Longrightarrow}{}^{\diamond}_{0} (m,i',s',h',\rho')$. *Let* s_t *be a temporary variable store such that* $\rho = [\![AS_{in}[m,i]]\!]^{\diamond}_{s,s_t,h}$. *Then there exists a store* s'_t *such that* $(m,i,s,s_t,h) \underset{IR}{\Longrightarrow}{}^{\diamond}_{0} (m,i',s',s'_t,h')$ *and* $\rho' = [\![AS_{in}[m,i']]\!]^{\diamond}_{s',s'_t,h'}$.

PROOF By induction on the number of execution steps, i.e., the definition of big-step semantics.

- If zero steps are taken, i.e., $(m,i,s,h,\rho) \underset{BC}{\Longrightarrow}{}^{\diamond}_{0} (m,i,s,h,\rho)$, then we can derive $(m,i,s,s_t,h) \underset{IR}{\Longrightarrow}{}^{\diamond}_{0} (m,i,s,s_t,h)$. Since $i' = i$, we get the desired properties from the assumptions.

- Suppose more than zero steps have been taken, that is, we have the small-step transition $(m,i,s,h,\rho) \xrightarrow{IR[m,i]}{}^{\diamond}_{0} (m,i'',s'',h'',\rho'')$ and the big-step transition $(m,i'',s'',h'',\rho'') \underset{BC}{\Longrightarrow}{}^{\diamond}_{0} (m,i',s',h',\rho')$. Since $P_{\mathrm{IR}} = \mathsf{BC2IR}(P_{\mathrm{BC}})$, we get by definition of the algorithm $\mathsf{BC2IR}_{\mathrm{instr}}(i, BC(m,i), AS_{in}[m,i]) = (IR[m,i], AS_{out}[m,i])$. With Proposition B.3, we immediately get $(m,i,s,s_t,h) \xrightarrow{IR[m,i]}{}^{\diamond}_{0} (m,i'',s'',s''_t,h'')$ and $\rho'' = [\![AS_{out}[m,i]]\!]^{\diamond}_{s'',s''_t,h''}$. Since $(m,i) \xmapsto{IR[m,i]} (m,i'')$, we get with Proposition 5.5 on page 73 $AS_{out}[m,i] = AS_{in}[m,i'']$, hence $\rho' = [\![AS_{in}[m,i'']]\!]^{\diamond}_{s'',s''_t,h''}$. Therefore, we can apply this theorem inductively, and get the transition relation $(m,i'',s'',s''_t,h'') \underset{IR}{\Longrightarrow}{}^{\diamond}_{0} (m,i',s',s'_t,h')$ and $\rho = [\![AS_{in}[m,i']]\!]^{\diamond}_{s',s'_t,h'}$. Also, by definition of IR big-step semantics, we get $(m,i,s,s_t,h) \underset{IR}{\Longrightarrow}{}^{\diamond}_{0} (m,i',s',s'_t,h')$. ■

The following proposition applies to arbitrary big-step transitions.

Proposition B.5 *Suppose we have* $(m,i,s,h,\rho) \underset{BC}{\Longrightarrow}{}^{\diamond}_{n} (m,i',s',h',\rho')$. *Let* s_t *be a temporary variable store such that* $\rho = [\![AS_{in}[m,i]]\!]^{\diamond}_{s,s_t,h}$. *Then there exists a store* s'_t *such that* $(m,i,s,s_t,h) \underset{IR}{\Longrightarrow}{}^{\diamond}_{n} (m,i',s',s'_t,h')$ *and* $\rho' = [\![AS_{in}[m,i']]\!]^{\diamond}_{s',s'_t,h'}$.

We call the previous proposition $\mathscr{P}(m,n)$, and prove it by strong induction on n. For this, we need the following auxiliary proposition.

Proposition B.6 *Let $n \geq 0$, and suppose $\mathscr{P}(m,k)$ holds for all methods m and all call depths $k < n$. Suppose $(m,i,s,h,\rho) \xrightarrow{BC(m,i)}{}^{\diamond}_{n} (m,i',s',h',\rho')$. Let s_t be a store such that $\rho = [\![AS_{in}[m,i]]\!]^{\diamond}_{s,s_t,h}$. Then there is an s'_t such that $(m,i,s,s_t,h) \xrightarrow{IR[m,i]}{}^{\diamond}_{n} (m,i',s',s'_t,h')$ and $\rho' = [\![AS_{in}[m,i']]\!]^{\diamond}_{s',s'_t,h'}$.*

PROOF If $n = 0$, we can directly apply Proposition B.3 and get the desired properties. Assume $n > 0$. By construction of the bytecode semantics, it follows $BC(m,i) = \texttt{call}\ m'$.

We have $(m,i,s,h,\overline{v} :: r :: \overline{v'}) \xrightarrow{\text{B}}{}^{\diamond}_{n} (m,i+1,s,h',s''(ret) :: \overline{v'})$ and the method execution $(s',h) \overset{m'}{\underset{BC}{\Longrightarrow}}{}^{\diamond}_{n-1} (s'',h')$, where $s' = [this \mapsto r] \cup [\mathsf{margs}(m') \mapsto \overline{v}] \cup [ret \mapsto \mathsf{defval}]$. By definition of big-step method executions, this means there exists ρ'' such that $(m',\mathsf{mentry}(m'),s',h,\epsilon) \underset{BC}{\Longrightarrow}{}^{\diamond}_{n-1} (m',\mathsf{mexit}(m'),s'',h',\rho'')$. From Proposition 5.4 on page 73, we know $AS_{in}[m',\mathsf{mentry}(m)] = \epsilon$. Therefore, we apply $\mathscr{P}(m',n-1)$, and get that there is a store s''_t such that

$$(\mathsf{mentry}(m'),i,s,[\mathsf{tvars}(m') \mapsto \mathsf{defval}],h) \underset{IR}{\Longrightarrow}{}^{\diamond}_{n-1} (m',\mathsf{mexit}(m'),s'',s''_t,h')$$

which means by definition $(s',h) \overset{m'}{\underset{IR}{\Longrightarrow}}{}^{\diamond}_{n-1} (s'',h')$. (*)

At the same time, we have $IR[m,i] = \texttt{block}\ [\overline{t_i} := \overline{e'}; t^0_i := e.m'(\overline{e})]$ and the stacks $AS_{in}[m,i] = (\overline{e} :: e :: \overline{e'})$ and $AS_{in}[m,i+1] = \overline{t_i}$. With the assumption, we know that $[\![\overline{e} :: e :: \overline{e'}]\!]^{\diamond}_{s,s_t,h} = \overline{v} :: r :: \overline{v}'$.

For the execution of the block, we get with IR semantics and result (*) the transition $(m,i,s,s_t,h) \xrightarrow{IR[m,i]}{}^{\diamond}_{n} (m,i+1,s,s'_t,h')$, where $s'_t = [\overline{t_i} \mapsto [\![\overline{e'}]\!]^{\diamond}_{s,s_t,h}] \cup [t^0_i \mapsto s''(ret)]$. All that is left to show is that $[\![AS_{in}[m,i+1]]\!]^{\diamond}_{s,s'_t,h'} = [\![t^0_i :: \overline{t_i}]\!]^{\diamond}_{s,s'_t,h'} = s''(ret) :: \overline{v'} = \rho'$. But this follows from $[\![t^0_i]\!]^{\diamond}_{s,s'_t,h'} = s''(ret)$ and $[\![\overline{t^0_i}]\!]^{\diamond}_{s,s'_t,h'} = [\![\overline{e'}]\!]^{\diamond}_{s,s_t,h} = \overline{v'}$. ∎

PROOF (OF PROPOSITION B.5) By induction on the length of the execution c.

- If zero steps are taken, we have $(m,i,s,h,\rho) \underset{BC}{\Longrightarrow}{}^{\diamond}_{n} (m,i,s,h,\rho)$. For any s_t, we can derive $(m,i,s,s_t,h) \underset{IR}{\Longrightarrow}{}^{\diamond}_{n} (m,i,s,s_t,h)$ and get the desired properties directly from the assumptions.
- If $c > 0$ steps are taken, then suppose this proposition $\mathscr{P}(m,n)$ holds for all $c' < c$. By definition of big-step semantics, we have

$$(m,i,s,h,\rho) \xrightarrow{BC(m,i)}{}^{\diamond}_{n_1} (m,i'',s'',h'',\rho'') \underset{BC}{\Longrightarrow}{}^{\diamond}_{n_2} (m,i',s',h',\rho')$$

where $n_1 \leq n$ and $n_2 \leq n$. We can apply Proposition B.6 to the first step and the induction hypothesis $\mathscr{P}(m,n)$ to the remaining execution, because it contains less execution steps. It follows

$$(m,i,s,s_t,h) \xrightarrow{IR[m,i]}{}^{\diamond}_{n_1} (m,i'',s'',s''_t,h'') \underset{IR}{\Longrightarrow}{}^{\diamond}_{n_2} (m,i',s',s'_t,h')$$

for some s'_t, s''_t, such that $[\![AS_{in}[m,i'']]\!]^{\diamond}_{s'',s''_t,h''} = \rho''$ and $[\![AS_{in}[m,i']]\!]^{\diamond}_{s',s'_t,h'} = \rho'$. Putting the executions together, we get $(m,i,s,s_t,h) \underset{IR}{\Longrightarrow}{}^{\diamond}_{n} (m,i',s',s'_t,h')$. ■

Theorem B.7 *Let P_{BC} be a bytecode program, and P_{IR} be an IR program such that $P_{\text{IR}} = \mathsf{BC2IR}(P_{\text{BC}})$. Let m be a method, s, s' be stores, and h, h' be heaps, and n be a call depth. If*

$$(s,h) \overset{m}{\underset{BC}{\Longrightarrow}}{}^{\diamond}_{n} (s',h')$$

then

$$(s,h) \overset{m}{\underset{IR}{\Longrightarrow}}{}^{\diamond}_{n} (s',h')$$

PROOF By definition of $(s,h) \overset{m}{\underset{BC}{\Longrightarrow}}{}^{\diamond}_{n} (s',h')$ semantics, we know there is a stack ρ such that $(m,\mathsf{mentry}(m),s,h,\epsilon) \underset{BC}{\Longrightarrow}{}^{\diamond}_{n} (m,\mathsf{mexit}(m),s',h',\rho)$. Let $s_t = [\mathsf{tvars}(m) \mapsto \mathsf{defval}]$. With Proposition 5.4 on page 73, we have $AS_{in}[m,\mathsf{mentry}(m)] = \epsilon$, therefore we know that $[\![AS_{in}[m,\mathsf{mentry}(m)]]\!]^{\diamond}_{s,s_t,h} = \epsilon$. We can apply Proposition B.5 and get that there is some temporary state s'_t such that $(m,\mathsf{mentry}(m),s,s_t,h) \underset{IR}{\Longrightarrow}{}^{\diamond}_{n} (m,\mathsf{mexit}(m),s',s'_t,h')$, which by definition means $(s,h) \overset{m}{\underset{IR}{\Longrightarrow}}{}^{\diamond}_{n} (s',h')$. ■

C

Correctness Proof for the IR Type System

This appendix contains the complete soundness proof for the type system of the intermediate representation. In the following, the meta-variable ω is used for IR state triples (s, s_t, h).

C.1 Properties of Single Executions

In this section, we first show how assignments in the IR language are related to assignments in the high-level DSD language. Then we define equivalence relations for confluence point stacks and for IR configurations. Finally, we show that a program does not change "low" variables and fields under a "high" pc label, and other properties that apply to single executions of well-typed IR instructions.

C.1.1 Connection to the High-Level Language

We extend some high-level definitions to IR program states:

$$\begin{array}{rccc} \text{constraint set satisfiability:} & (s, s_t, h) \models^{\diamond} Q & \iff & (s \cup s_t, h) \models^{\diamond} Q \\ \text{expression evaluation:} & [\![e]\!]^{\diamond}_{s,s_t,h} & = & [\![e]\!]^{\diamond}_{s \cup s_t,h} \end{array}$$

We formalize the similarity between assignment blocks and the respective high-level sequential compositions. We define an auxiliary function makestmt that combines a

sequence of assignment statements into a valid high-level statement (either a nested sequential composition or **skip**):

$$\mathsf{makestmt}(\overline{a}) = \begin{cases} \textbf{skip} & \text{if } \overline{a} = \epsilon \\ a & \text{if } \overline{a} = a \\ a\,;\mathsf{makestmt}(\overline{a_r}) & \text{if } \overline{a} = a :: \overline{a_r} \end{cases}$$

Then for call-free assignment blocks, the IR semantics corresponds directly to the high-level semantics.

Lemma C.1 *Let* $\texttt{block}\,\overline{a}$ *be an assignment block IR instruction. If the small-step IR transition* $(m, i, s, s_t, h) \xrightarrow{\texttt{block}\,\overline{a}}{}_0^\diamond (m, i+1, s', s'_t, h')$ *can be derived, then high-level big-step transition* $(s \cup s_t, h) \xrightarrow{\mathsf{makestmt}(\overline{a})}{}^\diamond (s' \cup s'_t, h')$ *can be derived.*

PROOF Follows directly from the definition of $\overset{\overline{a}}{\Rightarrow}{}_0^\diamond$. ■

The correspondence of the IR typing for assignment blocks and the high-level typing judgement is formalized by the following lemma.

Lemma C.2 *Let* $\mathrm{I} = \texttt{block}\,\overline{a}$ *be an assignment block instruction. If the IR typing judgement* $\Gamma, \Gamma_t \vdash i, pc, \Delta, Q \xrightarrow{\mathrm{I}} i+1, pc, \Delta, Q'$ *can be derived, then high-level typing judgement* $\Gamma \cup \Gamma_t, pc \vdash \{Q\}\ \mathsf{makestmt}(\overline{a})\ \{Q'\}$ *can be derived.*

PROOF Follows directly from the definition of $\Gamma, \Gamma_t, \Delta, pc \vdash \{Q\}\ \overline{a}\ \{Q'\}$. ■

Finally, we show that the evaluation of pc and Δ remains unchanged during the execution of assignment blocks.

Lemma C.3 *Let* $\mathrm{I} = \texttt{block}\,\overline{a}$. *If we can derive the transition* $(m, i, \omega) \xrightarrow{\mathrm{I}}{}_n^\diamond (m, i+1, \omega')$ *and the typing judgmenet* $\Gamma, \Gamma_t \vdash i, pc, \Delta, Q \xrightarrow{\mathrm{I}} i+1, pc, \Delta, Q'$, *then* $[\![pc]\!]_\omega^\diamond = [\![pc]\!]_{\omega'}^\diamond$ *and* $[\![\Delta]\!]_\omega^\diamond = [\![\Delta]\!]_{\omega'}^\diamond$.

PROOF This follows from the definition of the typing rules. They require that the variables or fields that can possibly change do not occur syntactically in Δ or pc. ■

C.1.2 Equivalences

We define the following equivalence relation for IR program states, which naturally extends the definition of DSD program states:

$$\vdash^\diamond (s, s_t, h) \sim_\beta^{\Gamma,\Gamma_t,k} (s', s'_t, h') \iff \vdash^\diamond s \sim_\beta^{\Gamma,k} s' \wedge \vdash^\diamond s_t \sim_\beta^{\Gamma_t,k} s'_t \wedge \vdash^\diamond h \sim_\beta^k h'$$

We also give a definition for confluence point stack equivalences: we require that there exists a common "low" prefix such that the program points and evaluated labels in this "low" prefix are equal, the labels evaluate to some domain that is less or equal to k, and in the remaining "high" parts of the stacks, the labels evaluate to a domain that is neither less nor equal to k.

Definition C.4 *Let ω_1, ω_2 be IR states, and $\diamond$ be a domain lattice, and $k \in Dom^{\diamond}$. Two evaluated confluence point stacks $[\![\Delta_1]\!]^{\diamond}_{\omega_1} = \overline{i_1, k_1}$ and $[\![\Delta_2]\!]^{\diamond}_{\omega_2} = \overline{i_2, k_2}$ are $\diamond, k$-*equivalent*, written*

$$\vdash^{\diamond} [\![\Delta_1]\!]^{\diamond}_{\omega_1} \overset{k}{\sim} [\![\Delta_2]\!]^{\diamond}_{\omega_2}$$

if there exists a prefix index p such that

- $0 \le p \le \min(|\overline{k_1}|, |\overline{k_2}|)$,
- $\forall j \in 1, \ldots, p.\ \overline{k_1}.j = \overline{k_2}.j \le^{\diamond} k \ \wedge\ \overline{i_1}.j = \overline{i_2}.j$,
- $\forall j \in \{p+1, \ldots, |\overline{k_1}|\}.\ \overline{k_1}.j \not\le^{\diamond} k$, *and*
- $\forall j \in \{p+1, \ldots, |\overline{k_2}|\}.\ \overline{k_2}.j \not\le^{\diamond} k$.

Definition C.5 (Equivalent IR configurations) *Let $\diamond$ be a domain lattice, and let k be a domain with $k \in Dom^{\diamond}$. Two IR configurations are $\diamond, k$-*equivalent*, written*

$$\Gamma, \Gamma_t \vdash (m, i_1, \omega_1) : pc_1, \Delta_1, Q_1 \approx^{\diamond}_{k,\beta} (m, i_2, \omega_2) : pc_2, \Delta_2, Q_2$$

if and only if:

- $\vdash^{\diamond} \omega_1 \sim^{\Gamma,\Gamma_t,k}_{\beta} \omega_2$,
- $\omega_1 \models^{\diamond} Q_1$ *and* $\omega_2 \models^{\diamond} Q_2$,
- $\vdash^{\diamond} [\![\Delta_1]\!]^{\diamond}_{\omega_1} \overset{k}{\sim} [\![\Delta_2]\!]^{\diamond}_{\omega_2}$, *and*
- *either* $[\![pc_1]\!]^{\diamond}_{\omega_1} = [\![pc_2]\!]^{\diamond}_{\omega_2} \le^{\diamond} k$ *and* $i_1 = i_2$, *or* $[\![pc_1]\!]^{\diamond}_{\omega_1} \not\le^{\diamond} k$ *and* $[\![pc_2]\!]^{\diamond}_{\omega_2} \not\le^{\diamond} k$

Lemma C.6 *All equivalence relations defined in this section are reflexive, transitive, and symmetric.*

PROOF Follows by definition of the equivalences. ∎

C.1.3 Properties of single executions

Then we show in two lemmas that for a single execution of a small step,

- if the pre-state satisfies Q, then the post-state satisfies Q',
- if the step is executed under a "high" pc, then the states are indistinguishable,
- there is an invariant on Δ: it contains ascending pc labels, and if there is a bottom element, it does not change during the execution, and if the final pc is "high", the stack does not change its "low" prefix.

Using these lemmas, we then show that the satisfiability of constraint sets Q and the indistinguishability of states under a high pc remain invariant for single executions of an arbitrary chain of steps.

Lemma C.7 *Let* $\Gamma, \Gamma_t \vdash i, pc, \Delta, Q \xrightarrow{\text{I}} i', pc', \Delta', Q'$ *and* $(m, i, \omega) \xrightarrow{\text{I}}{}^{\diamond}_{0} (m, i', \omega')$.
If $\omega \models^{\diamond} Q$*, then* $\omega' \models^{\diamond} Q'$.

PROOF By induction over the type derivation.

- If the typing judgement has been derived by the weakening rule, we have the judgement $\Gamma, \Gamma_t \vdash i, pc, \Delta, Q_0 \xrightarrow{\text{I}} i', pc', \Delta', Q'_0$ with $Q \Rightarrow Q_0$ and $Q'_0 \Rightarrow Q'$. Since $\omega \models^{\diamond} Q$, we get with Lemma 3.5 on page 34 that $\omega \models^{\diamond} Q_0$. Hence we can apply the theorem inductively and get $\omega' \models^{\diamond} Q'_0$, and hence with lemma $\omega \models^{\diamond} Q'$.
- $\text{I} = \texttt{if}\ e\ j$.

  We have $(m, i, \omega) \xrightarrow{\text{I}}{}^{\diamond}_{0} (m, i', \omega)$. We invert the rule for the instruction and make a case distinction over the shape of the postcondition:

  - If $Q' = Q$, then $\omega \models^{\diamond} Q'$ follows trivially.
  - If $Q' = Q \cup \{\ell_1 \sqsubseteq \ell_2\}$, then $e = \ell_1 \sqsubseteq \ell_2$. Also, $i' = j$, thus $(m, i, \omega) \xrightarrow{\text{I}}{}^{\diamond}_{0} (m, j, \omega)$, thus $[\![\ell_1 \sqsubseteq \ell_2]\!]^{\diamond}_{\omega} \neq 0$. Therefore, $[\![\ell_1]\!]^{\diamond}_{\omega} \leq^{\diamond} [\![\ell_2]\!]^{\diamond}_{\omega}$, and with $\omega \models^{\diamond} Q$, we get $\omega \models^{\diamond} Q \cup \{\ell_1 \sqsubseteq \ell_2\}$.
- $\text{I} = \texttt{block}\ \overline{a}$.

  Let $S = \mathsf{makestmt}(\overline{a})$. We get $(s \cup s_t, h) \xrightarrow{S}{}^{\diamond} (s' \cup s'_t, h')$ with Lemma C.1, and $\Gamma \cup \Gamma_t, pc \vdash \{Q\}\ P\ \{Q'\}$ with Lemma C.2. With Lemma A.5 on page 116, we get $(s' \cup s'_t, h') \models^{\diamond} Q'$, hence $\omega' \models^{\diamond} Q'$.
- $\text{I} \in \{\texttt{jmp}\ j, \texttt{cjmp}\ j, \texttt{cpush}\ j\}$.

  Then $\omega' = \omega$ and $Q = Q'$, hence $\omega' \models^{\diamond} Q'$ follows trivially from the assumptions. ∎

Lemma C.8 *Let* $\Gamma, \Gamma_t \vdash i, pc, \Delta, Q \xrightarrow{\mathrm{I}} i', pc', \Delta', Q'$ *and* $(m, i, \omega) \xrightarrow{\mathrm{I}}{}^{\diamond}_{0} (m, i', \omega')$.
If $\omega \models^{\diamond} Q$ *and* $[\![pc]\!]^{\diamond}_{\omega} \not\leq^{\diamond} k$*, then* $\vdash^{\diamond} \omega \sim^{\Gamma,\Gamma_t,k}_{\mathsf{id}} \omega'$.

PROOF By induction over the type derivation.

- If the typing judgement has been derived by the weakening rule, we have the judgement $\Gamma, \Gamma_t \vdash i, pc, \Delta, Q_0 \xrightarrow{\mathrm{I}} i', pc', \Delta', Q'_0$ with $Q \Rightarrow Q_0$. Since $\omega \models^{\diamond} Q$, we get with Lemma 3.5 on page 34 that $\omega \models^{\diamond} Q_0$. Hence we can apply the theorem inductively and get $\vdash^{\diamond} \omega \sim^{\Gamma,\Gamma_t,k}_{\mathsf{id}} \omega'$.
- $\mathrm{I} = \mathtt{block}\ \overline{a}$.

  Let $P = \mathsf{makestmt}(\overline{a})$. We get $(s \cup s_t, h) \xrightarrow{P}{}^{\diamond} (s' \cup s'_t, h')$ with Lemma C.1 and $\Gamma \cup \Gamma_t, pc \vdash \{Q\}\ P\ \{Q'\}$ with Lemma C.2.

  With Lemma A.6 on page 117 (high-level soundness under high pc), we can conclude $\vdash^{\diamond} (s \cup s_t, h) \sim^{\Gamma\cup\Gamma_t,k}_{\mathsf{id}} (s' \cup s'_t, h')$, hence $\vdash^{\diamond} \omega \sim^{\Gamma,\Gamma_t,k}_{\mathsf{id}} \omega'$.
- $\mathrm{I} \in \{\mathtt{jmp}\ j, \mathtt{cjmp}\ j, \mathtt{if}\ e\ j, \mathtt{cpush}\ j\}$.

  Then $(m, i, \omega) \xrightarrow{\mathrm{I}}{}^{\diamond}_{0} (m, i', \omega')$ with $\omega' = \omega$ for some i', so $\vdash^{\diamond} \omega \sim^{\Gamma,\Gamma_t,k}_{\mathsf{id}} \omega'$ follows trivially. ∎

Confluence point stacks reflect the high-level control flow structure (conditionals and loops) at a specific point in an IR program. More precisely, they indicate the addresses where the structures end, and the corresponding pc labels that are to be restored at these addresses. The innermost structure is on top of the stack, the outermost structure is at the bottom. Therefore, the row of pc labels on the stack is always in ascending order, if we start to count from the bottom (right-most) element. (The rationale is that Δ is a stack that "grows" to the left. In contrast, if Δ was regarded as a sequence of pc labels, a more natural description of the order would be *descending*.)

Definition C.9 *A stack of domains is* ascending *if its domains are sorted in ascending order with respect to* $\leq^{\diamond}$ *from bottom (right-most) to top. More precisely,* $\overline{k}$ *is ascending if and only if*

- $\overline{k}$ *is empty or contains only one element, or*
- $\overline{k} = k_1 :: k_2 :: ks$ *and* $k_2 \leq^{\diamond} k_1$ *and* $k_2 :: ks$ *is ascending.*

In the following, we write Δ_{pc} to refer only to the stack of pc labels in a confluence point stack, thereby projecting away the program points. Also, $\mathsf{bottom}(\overline{x})$ refers to the right-most (last, bottom) element of the stack or sequence $\overline{x}$.

Lemma C.10 *Let* $\Gamma, \Gamma_t \vdash i, pc, \Delta, Q \xrightarrow{\mathrm{I}} i', pc', \Delta', Q'$ *and* $(m, i, \omega) \xrightarrow{\mathrm{I}}{}^{\diamond}_{n} (m, i', \omega')$.
If $[\![pc :: \Delta_{pc}]\!]^{\diamond}_{\omega}$ *is ascending, then*

1. $\llbracket pc' :: \Delta'_{pc} \rrbracket^{\diamond}_{\omega'}$ *is ascending, and*

2. $\mathsf{bottom}(\llbracket pc :: \Delta_{pc} \rrbracket^{\diamond}_{\omega}) \leq^{\diamond} \mathsf{bottom}(\llbracket pc' :: \Delta'_{pc} \rrbracket^{\diamond}_{\omega'})$*, and*

3. *if* $\llbracket pc' \rrbracket^{\diamond}_{\omega'} \not\leq^{\diamond} k$*, then* $\vdash^{\diamond} \llbracket \Delta \rrbracket^{\diamond}_{\omega} \overset{k}{\sim} \llbracket \Delta' \rrbracket^{\diamond}_{\omega'}$.

PROOF By induction over the type derivation.

- If the typing judgement has been derived by the weakening rule, we get by rule inversion $\Gamma, \Gamma_t \vdash i, pc, \Delta, Q_0 \xrightarrow{\text{I}} i', pc', \Delta', Q'_0$, hence we can inductively apply this lemma and get the desired properties.

- $\text{I} = \texttt{if}\ e\ j$.

  We have $\Delta' = \Delta$ and $\omega' = \omega$ and $\llbracket pc \rrbracket^{\diamond}_{\omega} \leq^{\diamond} \llbracket pc \rrbracket^{\diamond}_{\omega} \vee^{\diamond} \llbracket \ell \rrbracket^{\diamond}_{\omega} = \llbracket pc \sqcup \ell \rrbracket^{\diamond}_{\omega} = \llbracket pc' \rrbracket^{\diamond}_{\omega'}$.

  1. With the assumptions, we get $\llbracket (pc \sqcup \ell) :: \Delta'_{pc} \rrbracket^{\diamond}_{\omega'}$ is ascending.
  2. If Δ is not empty, we get the assumption trivially, since $\Delta' = \Delta$ and $\omega' = \omega$. Otherwise, we can conclude that $\mathsf{bottom}(\llbracket pc :: \Delta_{pc} \rrbracket^{\diamond}_{\omega}) = \llbracket pc \rrbracket^{\diamond}_{\omega} \leq^{\diamond} \llbracket pc' \rrbracket^{\diamond}_{\omega'} = \mathsf{bottom}(\llbracket pc' :: \Delta'_{pc} \rrbracket^{\diamond}_{\omega'}$.
  3. This holds trivially, since $\Delta' = \Delta$ and $\omega = \omega'$.

- $\text{I} = \texttt{jmp}\ j$.

  We have $\Delta' = \Delta$ and $\omega' = \omega$ and $pc' = pc$, therefore the stack stays in ascending order, the bottom element does not change, and the pre-stack is $\diamond, k$-equivalent to the post-stack.

- $\text{I} = \texttt{cpush}\ j$.

  We have $\omega' = \omega$ and $pc' = pc$.

  1. Since $\llbracket pc :: \Delta_{pc} \rrbracket^{\diamond}_{\omega}$ is ascending, $\llbracket pc :: pc :: \Delta_{pc} \rrbracket^{\diamond}_{\omega'}$ is ascending.
  2. We have $\mathsf{bottom}(\llbracket pc :: \Delta_{pc} \rrbracket^{\diamond}_{\omega}) = \mathsf{bottom}(\llbracket pc :: pc :: \Delta_{pc} \rrbracket^{\diamond}_{\omega})$.
  3. If $\llbracket pc \rrbracket^{\diamond}_{\omega} \not\leq^{\diamond} k$, we get by definition of $\diamond, k$-equivalence $\vdash^{\diamond} \llbracket \Delta \rrbracket^{\diamond}_{\omega} \overset{k}{\sim} \llbracket pc :: \Delta \rrbracket^{\diamond}_{\omega}$.

- $\text{I} = \texttt{cjmp}\ j$.

  We have $\omega' = \omega$.

  1. Since $\llbracket pc :: pc' :: \Delta \rrbracket^{\diamond}_{\omega}$ is ascending, $\llbracket pc' :: \Delta_{pc} \rrbracket^{\diamond}_{\omega}$ is ascending.
  2. We have $\mathsf{bottom}(\llbracket pc :: pc' :: \Delta_{pc} \rrbracket^{\diamond}_{\omega}) = \mathsf{bottom}(\llbracket pc' :: \Delta_{pc} \rrbracket^{\diamond}_{\omega})$.
  3. If $\llbracket pc' \rrbracket^{\diamond}_{\omega} \not\leq^{\diamond} k$, then by definition of $\diamond, k$-equivalence $\vdash^{\diamond} \llbracket pc' :: \Delta \rrbracket^{\diamond}_{\omega} \overset{k}{\sim} \llbracket \Delta \rrbracket^{\diamond}_{\omega}$.

- $\text{I} = \texttt{block}\ \overline{a}$.

  We have $\Delta' = \Delta$, and $pc' = pc$. From Lemma C.3, it follows that $[\![\Delta]\!]^{\diamond}_{\omega} = [\![\Delta]\!]^{\diamond}_{\omega'}$ and $[\![pc]\!]^{\diamond}_{\omega} = [\![pc]\!]^{\diamond}_{\omega'}$, therefore we get the required propositions trivially. ∎

Theorem C.11 *Let $\diamond$ be a domain lattice, and let $k \in Dom^{\diamond}$ be a domain. Given an IR program P_{IR} that is well-typed. Let m be a method with a signature $\mathsf{msig}(m) = [\Gamma, pc_m, Q_m, Q'_m]$. Suppose that $i, i' \in \mathsf{dom}(IR(m))$ are addresses in m, such that $\Lambda(i) = (pc, \Delta, Q)$ and $\Lambda(i') = (pc', \Delta', Q')$. If*

- $(m, i, \omega) \underset{IR}{\Longrightarrow}{}^{\diamond}_{n} (m, i', \omega')$,
- $\omega \models^{\diamond} Q$,
- $[\![pc :: \Delta_{pc}]\!]^{\diamond}_{\omega}$ *is ascending and* $\mathsf{bottom}([\![pc :: \Delta_{pc}]\!]^{\diamond}_{\omega}) \not\leq^{\diamond} k$,

then there exists a type environment Γ_t such that

- $\omega' \models^{\diamond} Q'$, *and*
- $\vdash^{\diamond} \omega \sim^{\Gamma, \Gamma_t, k}_{\mathsf{id}} \omega'$.

We call this theorem $\mathscr{P}(m, n)$, and prove it by strong induction on n. For this, we need the following auxiliary lemma.

Lemma C.12 *Let $\Gamma, \Gamma_t \vdash i, pc, \Delta, Q \xrightarrow{\text{I}} i', pc', \Delta', Q'$ and $(m, i, \omega) \xrightarrow{\text{I}}{}^{\diamond}_{n} (m, i', \omega')$. Furthermore, let $\mathscr{P}(m, p)$ for all $p < n$. Let $\diamond$ be a domain lattice, and let $k \in Dom^{\diamond}$.*
If $\omega \models^{\diamond} Q$ and $[\![pc]\!]^{\diamond}_{\omega} \not\leq^{\diamond} k$, then $\omega' \models^{\diamond} Q'$, and $\vdash^{\diamond} \omega \sim^{\Gamma, \Gamma_t, k}_{\mathsf{id}} \omega'$.

Proof By case distinction over n.

- Let $n = 0$. Then we can apply Lemma C.7, and get $\omega' \models^{\diamond} Q'$, and Lemma C.8 and get $\vdash^{\diamond} \omega \sim^{\Gamma, \Gamma_t, k}_{\mathsf{id}} \omega'$.
- Let $n \neq 0$. Then $\text{I} = \texttt{block}\ \overline{a}$. For all $a \in \overline{a}$, if $\omega \xrightarrow{a}{}^{\diamond}_{0} \omega'$, we get $\omega' \models^{\diamond} Q'$ and $\vdash^{\diamond} \omega \sim^{\Gamma, \Gamma_t, k}_{\mathsf{id}} \omega'$. If $\omega \xrightarrow{a}{}^{\diamond}_{n} \omega'$ and $n \neq 0$, then it must be $a = x := e.m(\overline{e})$. With a similar argument as in Lemma A.6 on page 117 for high-level method calls and using $\mathscr{P}(m, n-1)$ and the fact that pc_m for the called method must be invisible at k in $\diamond$, we get $\omega' \models^{\diamond} Q'$, and $\vdash^{\diamond} \omega \sim^{\Gamma, \Gamma_t, k}_{\mathsf{id}} \omega'$. ∎

PROOF (OF THEOREM C.11) Since $pc :: \Delta_{pc}$ is ascending and the bottom element is not $\diamond$, k-visible, we get $[\![pc]\!]^{\diamond}_{\omega} \not\leq^{\diamond} k$. As the method is well-typed, we know by definition that there exists a type environment Γ_t such that a type mapping Λ is derivable for Γ from $\mathsf{msig}(m)$ and that Γ_t, that is, there are suitable small-step typing judgements for each successor relation $(m,i) \overset{\mathrm{I}}{\mapsto} (m,i')$.

We proceed by induction on n and the number of execution steps c.

- If $c = 0$, then zero steps have been taken, i.e. $i = i'$. We have $\omega' = \omega$, and get the desired propositions directly from the assumptions and by reflexivity of the state equivalence relation.
- If $c > 0$, then $(m,i,\omega) \overset{\mathrm{I}}{\longrightarrow}^{\diamond}_{n_1} (m,i'',\omega'') \underset{IR}{\Longrightarrow}^{\diamond}_{n_2} (m,i',\omega')$ and $n_1 \leq n$ and $n_2 \leq n$. Let $\Lambda(i'') = (pc'', \Delta'', Q'')$.
  - If $n = 0$, then $n_1 = 0$ and $n_2 = 0$. Hence we get with Lemmas C.7 and C.8 that $\omega'' \models^{\diamond} Q''$ and $\vdash^{\diamond} \omega \sim^{\Gamma,\Gamma_t,k}_{\mathsf{id}} \omega''$.
  - Let $n > 0$, and $\mathscr{P}(m,p)$ for all $p < n$. Hence we can apply Lemma C.12 and get $\omega'' \models^{\diamond} Q''$ and $\vdash^{\diamond} \omega \sim^{\Gamma,\Gamma_t,k}_{\mathsf{id}} \omega''$. ∎

  By Lemma C.10, $[\![pc' :: \Delta'_{pc}]\!]^{\diamond}_{\omega'}$ is ascending, and $\mathsf{bottom}([\![pc' :: \Delta'_{pc}]\!]^{\diamond}_{\omega'}) \not\leq^{\diamond} k$. The remaining execution $(m,i'',\omega'') \underset{IR}{\Longrightarrow}^{\diamond}_{n_2} (m,i',\omega')$ is shorter, hence we can apply the theorem inductively on it and get $\omega' \models^{\diamond} Q'$ and $\vdash^{\diamond} \omega'' \sim^{\Gamma,\Gamma_t,k}_{\mathsf{id}} \omega'$. By transitivity of the state equivalence relation we finally get $\vdash^{\diamond} \omega \sim^{\Gamma,\Gamma_t,k}_{\mathsf{id}} \omega'$.

C.2 Small-Step and Big-Step Universal Noninterference

In this section, we first show universal noninterference for a single IR execution step on the left side. Then we use this result to show the soundness for entire method bodies.

C.2.1 Small-step noninterference

This subsection presents two auxiliary lemmas for two executions.

The first lemma states that there is a “strong low-bisimulation”: Whenever one can make one step in the left execution, one can make one step in the right execution, such that when the initial states were equivalent and had a low *pc*, then the final configurations are equivalent.

The second lemma states that there is a “weak high-bisimulation”: Whenever one can make one step in the left execution, one can make zero or more steps in the right execution, such that when the initial states were equivalent and had a high *pc*, then the final configurations are equivalent. (The main part of the lemma has already been shown in the previous section.)

Lemma C.13 (Small-step noninterference under low pc) *Given a program P_{IR} that is well-typed, and a method m with a signature* $\mathsf{msig}(m) = [\Gamma, pc_m, Q_m, Q'_m]$*. Let Γ_t be the temporary variable type environment for which m is well-typed. Let $\diamond$ be a domain lattice, and $k \in Dom^\diamond$ be a domain. For $v \in \{1,2\}$, suppose $(pc_v, \Delta_v, Q_v) = \Lambda(i_v)$ and $(pc'_v, \Delta'_v, Q'_v) = \Lambda(i'_v)$. If*

- $(m, i_1, \omega_1) \xrightarrow{\mathrm{I}}{}^{\diamond}_{0} (m, i'_1, \omega'_1)$ *and* $(m, i_2, \omega_2) \xrightarrow{\mathrm{I}}{}^{\diamond}_{0} (m, i'_2, \omega'_2)$*, and*
- $[\![pc_1]\!]^\diamond_{\omega_1} = [\![pc_2]\!]^\diamond_{\omega_2} \leq^\diamond k$*, and*
- $\Gamma, \Gamma_t \vdash (m, i_1, \omega_1) : pc_1, \Delta_1, Q_1 \approx^\diamond_{k,\beta} (m, i_2, \omega_2) : pc_2, \Delta_2, Q_2$*,*

then there exists a partial bijection $\beta' \supseteq \beta$ such that

- $\Gamma, \Gamma_t \vdash (m, i'_1, \omega'_1) : pc'_1, \Delta'_1, Q'_1 \approx^\diamond_{k,\beta'} (m, i'_2, \omega'_2) : pc'_2, \Delta'_2, Q'_2$.

PROOF We get by definition of configuration equivalence that $i_1 = i_2$, i.e., we consider the same instruction for both executions. This program point will be called i, and we denote the initial types as $\Lambda(i) = (pc, \Delta, Q)$.

1. With Lemma C.7, we get $\omega'_1 \models^\diamond Q'_1$ and $\omega'_2 \models^\diamond Q'_2$.
2. We show that there is a bijection $\beta' \supseteq \beta$ such that $\vdash^\diamond \omega'_1 \sim^{\Gamma,\Gamma_t,k}_{\beta'} \omega'_2$.
   - Let $\mathrm{I} = \mathtt{block}\,\overline{a}$. Then $i'_1 = i'_2$, and thus $Q'_1 = Q'_2 = Q'$. Let $S = \mathsf{makestmt}(\overline{a})$. We get $(s_1 \cup s_{t1}, h_1) \xrightarrow{S}{}^\diamond (s'_1 \cup s'_{t1}, h'_1)$ and $(s_1 \cup s_{t1}, h_1) \xrightarrow{S}{}^\diamond (s'_1 \cup s'_{t1}, h'_1)$ with Lemma C.1 and $\Gamma \cup \Gamma_t, pc \vdash \{Q\}\ S\ \{Q'\}$ with Lemma C.2. With Theorem A.7 on page 119, we get that there is a bijection $\beta' \supseteq \beta$ such that $\vdash^\diamond \omega'_1 \sim^{\Gamma,\Gamma_t,k}_{\beta'} \omega'_2$.
   - If $\mathrm{I} \neq \mathtt{block}\,\overline{a}$, then $\omega'_1 = \omega_1$ and $\omega'_2 = \omega_2$. Let $\beta' = \beta$. From the assumptions, it follows $\vdash^\diamond \omega'_1 \sim^{\Gamma,\Gamma_t,k}_{\beta'} \omega'_2$.
3. Now we show that $\vdash^\diamond [\![\Delta]\!]^\diamond_{\omega'_1} \overset{k}{\sim} [\![\Delta]\!]^\diamond_{\omega'_2}$.
   - Let $\mathrm{I} = \mathtt{block}\,\overline{a}$. We have $\Delta' = \Delta$. By Lemma C.3, we have for $v \in \{1,2\}$ that $[\![\Delta]\!]^\diamond_{\omega_v} = [\![\Delta]\!]^\diamond_{\omega'_v}$, hence $\vdash^\diamond [\![\Delta]\!]^\diamond_{\omega'_v} \overset{k}{\sim} [\![\Delta]\!]^\diamond_{\omega'_v}$.
   - Let $\mathrm{I} = \mathtt{cpush}\ j$. We have $\omega'_1 = \omega_1$ and $\omega'_2 = \omega_2$, hence with the assumptions we get $\vdash^\diamond [\![(j, pc) :: \Delta]\!]^\diamond_{\omega'_1} \overset{k}{\sim} [\![(j, pc) :: \Delta]\!]^\diamond_{\omega'_2}$.
   - Let $\mathrm{I} = \mathtt{cjmp}\ j$. We have $\omega'_1 = \omega_1$ and $\omega'_2 = \omega_2$. From the stack equivalence $\vdash^\diamond [\![(j, pc) :: \Delta]\!]^\diamond_{\omega_1} \overset{k}{\sim} [\![(j, pc) :: \Delta]\!]^\diamond_{\omega_2}$, we can conclude $\vdash^\diamond [\![\Delta]\!]^\diamond_{\omega'_1} \overset{k}{\sim} [\![\Delta]\!]^\diamond_{\omega'_2}$.
   - Let $\mathrm{I} \in \{\mathtt{if}\ e\ j, \mathtt{jmp}\ j\}$. We have $\omega'_1 = \omega_1$ and $\omega'_2 = \omega_2$ and $\Delta' = \Delta$, hence with the assumptions we get the desired confluence stack equivalence.

4. Finally, we show that either $[\![pc'_1]\!]^{\diamond}_{\omega'_1} = [\![pc'_2]\!]^{\diamond}_{\omega'_2} \leq^{\diamond} k$ and $i'_1 = i'_2$, or $[\![pc'_1]\!]^{\diamond}_{\omega'_1} \not\leq^{\diamond} k$ and $[\![pc'_2]\!]^{\diamond}_{\omega'_2} \not\leq^{\diamond} k$.

   - Let $\text{I} = \texttt{if}\ e\ j$. We have $\omega'_1 = \omega_1$ and $\omega'_2 = \omega_2$. We have $\Gamma \cup \Gamma_t \vdash e : \ell$. By Corollary A.4 on page 115, we know that either $[\![\ell]\!]^{\diamond}_{\omega_1} = [\![\ell]\!]^{\diamond}_{\omega_2} \leq^{\diamond} k$, or $[\![\ell]\!]^{\diamond}_{\omega_1} \not\leq^{\diamond} k$ and $[\![\ell]\!]^{\diamond}_{\omega_2} \not\leq^{\diamond} k$.
     - If $[\![\ell]\!]^{\diamond}_{\omega_1} = [\![\ell]\!]^{\diamond}_{\omega_2} \leq^{\diamond} k$, we get with Lemma A.1 on page 113 that the expression evaluates to the same value: $[\![e]\!]^{\diamond}_{\omega_1} = [\![e]\!]^{\diamond}_{\omega_2}$. Therefore, the same branch is taken in both steps, and we get $i'_1 = i'_2$. Also, we get $[\![pc \sqcup \ell]\!]^{\diamond}_{\omega'_1} = [\![pc \sqcup \ell]\!]^{\diamond}_{\omega'_2} \leq^{\diamond} k$.
     - If $[\![\ell]\!]^{\diamond}_{\omega_1} \not\leq^{\diamond} k$ and $[\![\ell]\!]^{\diamond}_{\omega_2} \not\leq^{\diamond} k$, then $[\![pc \sqcup \ell]\!]^{\diamond}_{\omega'_1} \not\leq^{\diamond} k$ and $[\![pc \sqcup \ell]\!]^{\diamond}_{\omega'_2} \not\leq^{\diamond} k$.
   - Let $\text{I} = \texttt{cjmp}\ j$. Then $i'_1 = i'_2 = j$. By assumption, we have the stack equivalence $\vdash^{\diamond} [\![(j, pc') :: \Delta]\!]^{\diamond}_{\omega_1} \overset{k}{\sim} [\![(j, pc') :: \Delta]\!]^{\diamond}_{\omega_2}$, so we get by definition of confluence stack equivalence that either $[\![pc'_1]\!]^{\diamond}_{\omega'_1} = [\![pc'_2]\!]^{\diamond}_{\omega'_2} \leq^{\diamond} k$, or $[\![pc'_1]\!]^{\diamond}_{\omega'_1} \not\leq^{\diamond} k$ and $[\![pc'_2]\!]^{\diamond}_{\omega'_2} \not\leq^{\diamond} k$.
   - Let $\text{I} = \texttt{block}\ \overline{a}$. Then $i'_1 = i'_2$. With Lemma C.3, we get for $v \in \{1,2\}$ that $[\![pc]\!]^{\diamond}_{\omega_v} = [\![pc]\!]^{\diamond}_{\omega'_v}$, hence $[\![pc]\!]^{\diamond}_{\omega'_1} = [\![pc]\!]^{\diamond}_{\omega'_2} \leq^{\diamond} k$.
   - Let $\text{I} \in \{\texttt{jmp}\ j, \texttt{cpush}\ j\}$. Then $i'_1 = i'_2$ and $pc'_1 = pc'_2 = pc$ and $\omega'_1 = \omega_1$ and $\omega'_2 = \omega_2$. Therefore, $[\![pc'_1]\!]^{\diamond}_{\omega'_1} = [\![pc'_2]\!]^{\diamond}_{\omega'_2} \leq^{\diamond} k$ and $i'_1 = i'_2$.

From points (1)-(4) above, it follows there exists a partial bijection $\beta' \supseteq \beta$ such that $\Gamma, \Gamma_t \vdash (m, i'_1, \omega'_1) : pc'_1, \Delta'_1, Q'_1 \approx^{\diamond}_{k,\beta'} (m, i'_2, \omega'_2) : pc'_2, \Delta'_2, Q'_2$. ■

Lemma C.14 (Small-step noninterference under high pc) *Given a program P_{IR} that is well-typed, and a method m with a signature* $\mathsf{msig}(m) = [\Gamma, pc_m, Q_m, Q'_m]$. *Let Γ_t be the temporary variable typing for which m is well-typed. Let $\diamond$ be a domain lattice, and $k \in Dom^{\diamond}$. For $v \in \{1,2\}$, suppose $(pc_v, \Delta_v, Q_v) = \Lambda(i_v)$ and $(pc'_v, \Delta'_v, Q'_v) = \Lambda(i'_v)$. If*

- $(m, i_1, \omega_1) \xrightarrow{\text{I}}{}^{\diamond}_{n_1} (m, i'_1, \omega'_1)$,
- $[\![pc_1]\!]^{\diamond}_{\omega_1} \not\leq^{\diamond} k$ *and* $[\![pc_2]\!]^{\diamond}_{\omega_2} \not\leq^{\diamond} k$,
- $\Gamma, \Gamma_t \vdash (m, i_1, \omega_1) : pc_1, \Delta_1, Q_1 \approx^{\diamond}_{k,\beta} (m, i_2, \omega_2) : pc_2, \Delta_2, Q_2$,

then either (m, i_2, ω_2) diverges, or there exists a state (m, i'_2, ω'_2), a call depth n_2 and $\beta' \supseteq \beta$ such that

- $(m, i_2, \omega_2) \underset{IR}{\Longrightarrow}{}^{\diamond}_{n_2} (m, i'_2, \omega'_2)$, *and*
- $\Gamma \vdash (m, i_1, \omega'_1) : pc'_1, \Delta'_1, Q'_1 \approx^{\diamond}_{k,\beta'} (m, i_2, \omega'_2) : pc'_2, \Delta'_2, Q'_2$.

PROOF By induction over the execution of the second program.

- If $[\![pc'_1]\!]^\diamond_{\omega'_1} \not\leq^\diamond k$, we choose $(i'_2, \omega'_2) = (i_2, \omega_2)$ and $\beta' = \beta$, i.e., zero execution steps. From the assumptions, we have $\omega'_2 \models^\diamond Q'_2$. By Lemma C.12, we know $\omega'_1 \models^\diamond Q'_1$ and $\vdash^\diamond \omega_1 \sim^{\Gamma,\Gamma_t,k}_{\mathsf{id}} \omega'_1$. Since $[\![pc'_1]\!]^\diamond_{\omega'_1} \not\leq^\diamond k$, we can conclude with Lemma C.10 that $\vdash^\diamond [\![\Delta'_1]\!]^\diamond_{\omega'_1} \overset{k}{\sim} [\![\Delta_1]\!]^\diamond_{\omega_1}$. By transitivity, $\vdash^\diamond [\![\Delta'_1]\!]^\diamond_{\omega'_1} \overset{k}{\sim} [\![\Delta'_2]\!]^\diamond_{\omega'_2}$ and $\vdash^\diamond \omega'_1 \sim^{\Gamma,\Gamma_t,k}_{\beta'} \omega'_2$. Therefore, the configuration equivalence relation holds.

- If $[\![pc'_1]\!]^\diamond_{\omega'_1} \leq^\diamond k$, then it must be $\mathtt{I} = \mathtt{cjmp}\ i'_1$:
  - If $\mathtt{I} = \mathtt{if}\ e\ j$, then with $[\![pc_1]\!]^\diamond_{\omega_1} \not\leq^\diamond k$, we get $[\![pc'_1]\!]^\diamond_{\omega'_1} = [\![pc_1 \sqcup \ell]\!]^\diamond_{\omega'_1} \not\leq^\diamond k$.
  - If $\mathtt{I} = \mathtt{block}\ \overline{a}$, then with Lemma C.3, we have $[\![pc_1]\!]^\diamond_{\omega_1} = [\![pc'_1]\!]^\diamond_{\omega'_1} \not\leq^\diamond k$.
  - If $\mathtt{I} \in \{\mathtt{cpush}\ j, \mathtt{jmp}\ j\}$, we have $pc_1 = pc'_1$ and $\omega'_1 = \omega_1$, hence $[\![pc'_1]\!]^\diamond_{\omega'_1} \not\leq^\diamond k$.

  We thus have $\Delta_1 = (i'_1, pc'_1) :: \Delta'_1$, and $\omega'_1 = \omega_1$. We now execute the second program, starting at (i_2, ω_2). Assuming that it does not diverge, we now show that execution eventually reaches the confluence point i'_1.

  First, we assume $i_2 = \mathsf{mexit}(m)$, i.e., we cannot make a single step. By design of Λ, $\Delta_2 = \Lambda_\Delta(\mathsf{mexit}(m)) = \epsilon$, and $\Delta_1 = (i'_1, pc'_1) :: \Delta'_1$. As $[\![pc'_1]\!]^\diamond_{\omega_1} \leq^\diamond k$, this contradicts the assumption $\vdash^\diamond [\![\Delta_1]\!]^\diamond_{\omega_1} \overset{k}{\sim} [\![\Delta_2]\!]^\diamond_{\omega_2}$. So it must be $i_2 \neq \mathsf{mexit}(m)$.

  Let $(m, i_2, \omega_2) \xrightarrow{\mathtt{I}_2}{}^\diamond_{n_2} (m, i'_2, \omega'_2)$ for some n_2. It follows $(m, i_2, \omega_2) \underset{IR}{\Longrightarrow}{}^\diamond_{n_2} (m, i'_2, \omega'_2)$.
  - If $[\![pc'_2]\!]^\diamond_{\omega'_2} \not\leq^\diamond k$, then with the same arguments as above, we get

    $$\Gamma, \Gamma_t \vdash (m, i_2, \omega_2) : pc_2, \Delta_2, Q_2 \approx^\diamond_{k,\mathsf{id}} (m, i'_2, \omega'_2) : pc'_2, \Delta'_2, Q'_2.$$

    Therefore, with the original configuration equivalence and by transitivity,

    $$\Gamma, \Gamma_t \vdash (m, i_1, \omega_1) : pc_1, \Delta_1, Q_1 \approx^\diamond_{k,\beta} (m, i'_2, \omega'_2) : pc'_2, \Delta'_2, Q'_2.$$

    We can then apply the theorem inductively to the initial state (i_1, ω_1) and (i'_2, ω'_2), since the execution path of the second program is now shorter.
  - If $[\![pc'_2]\!]^\diamond_{\omega'_2} \leq^\diamond k$, we get with the same argument as above that $\mathtt{I}_2 = \mathtt{cjmp}\ i'_2$. We thus have $\omega'_2 = \omega_2$ and $\Delta_2 = (i'_2, pc'_2) :: \Delta'_2$. As

    $$\vdash^\diamond [\![(i'_1, pc'_1) :: \Delta_1]\!]^\diamond_{\omega'_1} \overset{k}{\sim} [\![(i'_2, pc'_2) :: \Delta_2]\!]^\diamond_{\omega'_2},$$

    we get by definition of confluence point stack equivalence that $i'_1 = i'_2$. Since we have a unique mapping of types to program points, $\Delta'_2 = \Delta'_1$ and $pc'_2 = pc'_1$ and $Q'_2 = Q'_1$. For configuration equivalence, it remains

to be shown that the final states are equivalent. We choose $\beta' = \beta$. With $[\![pc_1]\!]^{\diamond}_{\omega_1} \not\leq^{\diamond} k$ and $[\![pc_2]\!]^{\diamond}_{\omega_2} \not\leq^{\diamond} k$, we get by Lemma C.12 that $\vdash^{\diamond} \omega_1 \sim^{\Gamma,\Gamma_t,k}_{\mathsf{id}} \omega'_1$ and $\vdash^{\diamond} \omega_2 \sim^{\Gamma,\Gamma_t,k}_{\mathsf{id}} \omega'_2$, and with the assumption and transitivity of state equivalence we get $\vdash^{\diamond} \omega'_1 \sim^{\Gamma,\Gamma_t,k}_{\beta} \omega'_2$. ∎

C.2.2 Universal Noninterference

The universal noninterference theorem applies the two lemmas from the previous section repeatedly to get a noninterference result for the execution of entire methods (or other pairs of execution chains that start at the same program points).

Theorem C.15 *Given a program P_{IR} that is well-typed, and a method m with a signature* $\mathsf{msig}(m) = [\Gamma, pc_m, Q_m, Q'_m]$. *Let Γ_t be the temporary variable type environment for which m is well-typed. Let $i_{ret} = \mathsf{mexit}(m)$. Let $\diamond$ be a domain lattice, and $k \in Dom^{\diamond}$. For $v \in \{1,2\}$, let $(pc_v, \Delta_v, Q_v) = \Lambda(i_v)$, and let $(pc_{ret}, \Delta_{ret}, Q_{ret}) = \Lambda(i_{ret})$. Then if*

- $(m, i_1, \omega_1) \underset{IR}{\Longrightarrow}^{\diamond}_{n_1} (m, i_{ret}, \omega_{ret1})$ *and*
- $(m, i_2, \omega_2) \underset{IR}{\Longrightarrow}^{\diamond}_{n_2} (m, i_{ret}, \omega_{ret2})$ *and*
- $\Gamma, \Gamma_t \vdash (m, i_1, \omega_1) : pc_1, \Delta_1, Q_1 \approx^{\diamond}_{k,\beta} (m, i_2, \omega_2) : pc_2, \Delta_2, Q_2$

then there exists a partial bijection $\beta_{ret} \supseteq \beta$ such that

- $\Gamma, \Gamma_t \vdash (m, i_{ret}, \omega_{ret1}) : pc_{ret}, \Delta_{ret}, Q_{ret} \approx^{\diamond}_{k,\beta_{ret}} (m, i_{ret}, \omega_{ret2}) : pc_{ret}, \Delta_{ret}, Q_{ret}$.

We call this previous theorem $\mathscr{P}(m, n_1, n_2)$, and prove it by strong induction on n_1 and n_2. For this, we need an auxiliary lemma.

Lemma C.16 *Given a program P_{IR} that is well-typed, and a method m with a signature* $\mathsf{msig}(m) = [\Gamma, pc_m, Q_m, Q'_m]$. *Let Γ_t be the temporary variable type environment for which m is well-typed. Let $\diamond$ be a domain lattice, and $k \in Dom^{\diamond}$ be a domain. For $v \in \{1,2\}$, let $(pc_v, \Delta_v, Q_v) = \Lambda(i_v)$ and $(pc'_v, \Delta'_v, Q'_v) = \Lambda(i'_v)$. Suppose $\mathscr{P}(m, p_1, p_2)$ holds for all $p_1 < n_1$ and $p_2 < n_2$. If*

- $(m, i_1, \omega_1) \xrightarrow{\mathrm{I}_1}^{\diamond}_{n_1} (m, i'_1, \omega'_1)$ *and* $(m, i_2, \omega_2) \xrightarrow{\mathrm{I}_2}^{\diamond}_{n_2} (m, i'_2, \omega'_2)$,
- $[\![pc_1]\!]^{\diamond}_{\omega_1} = [\![pc_2]\!]^{\diamond}_{\omega_2} \leq^{\diamond} k$,
- $\Gamma, \Gamma_t \vdash (m, i_1, \omega_1) : pc_1, \Delta_1, Q_1 \approx^{\diamond}_{k,\beta} (m, i_2, \omega_2) : pc_2, \Delta_2, Q_2$,

then there exists a partial bijection $\beta' \supseteq \beta$ such that

- $\Gamma, \Gamma_t \vdash (m, i'_1, \omega'_1) : pc'_1, \Delta'_1, Q'_1 \approx^{\diamond}_{k,\beta'} (m, i'_2, \omega'_2) : pc'_2, \Delta'_2, Q'_2$.

PROOF As $[\![pc_1]\!]^{\diamond}_{\omega_1} = [\![pc_2]\!]^{\diamond}_{\omega_2} \leq^{\diamond} k$, we get by definition of configuration equivalence that $i_1 = i_2$. This also means that $\mathrm{I}_1 = \mathrm{I}_2 = \mathrm{I}$.

If $n_1 = n_2 = 0$, then we can directly apply Lemma C.13. Let $n_1 > 0$, without loss of generality. (The case $n_2 > 0$ is similar.) Then $\mathrm{I} = \texttt{block}\,\overline{a}$, and $\overline{a}$ contains a method call. Hence it must be $n_2 \neq 0$. Using the same argument as in the proof of Lemma C.12, we only need to show the lemma for method calls. We use the same argument as for the proof of Theorem A.7 on page 119 (high-level program execution soundness), relying on well-typedness of P_{IR} and the fact that we can use $\mathscr{P}(m, n_1 - 1, n_2 - 1)$. ∎

PROOF (OF THEOREM C.15) If $i_1 = i_2 = i_{ret}$, then the theorem follows trivially. Let therefore be $i_1 \neq i_{ret}$ or $i_2 \neq i_{ret}$. By definition of configuration equivalence, we know either $[\![pc_1]\!]^{\diamond}_{\omega_1} = [\![pc_2]\!]^{\diamond}_{\omega_2} \leq^{\diamond} k$, or $[\![pc_1]\!]^{\diamond}_{\omega_1} \not\leq^{\diamond} k$ and $[\![pc_2]\!]^{\diamond}_{\omega_2} \not\leq^{\diamond} k$.

- Let $[\![pc_1]\!]^{\diamond}_{\omega_1} = [\![pc_2]\!]^{\diamond}_{\omega_2} \leq^{\diamond} k$. Then $i_1 = i_2$, and thus $i_1 \neq i_{ret}$ and $i_2 \neq i_{ret}$. Hence we can make a step in both executions, that is,

$$(m, i_1, \omega_1) \xrightarrow{\mathrm{I}_1}{}^{\diamond}_{n'_1} (m, i'_1, \omega'_1) \underset{IR}{\Longrightarrow}{}^{\diamond}_{n''_1} (m, i_{ret}, \omega_{ret1})$$

  and

$$(m, i_2, \omega_2) \xrightarrow{\mathrm{I}_2}{}^{\diamond}_{n'_2} (m, i'_2, \omega'_2) \underset{IR}{\Longrightarrow}{}^{\diamond}_{n''_2} (m, i_{ret}, \omega_{ret2})$$

  We proceed by induction over both n_1 and n_2.

  – If $n_1 = n_2 = 0$, then by definition of big-step semantics, $n'_1 < n_1 = 0$ and $n'_2 \leq n_2 = 0$. Hence we can apply Lemma C.13 and get that there exists a bijection $\beta' \supseteq \beta$ such that

$$\Gamma, \Gamma_t \vdash (m, i'_1, \omega'_1) : pc'_1, \Delta'_1, Q'_1 \approx^{\diamond}_{k,\beta'} (m, i'_2, \omega'_2) : pc'_2, \Delta'_2, Q'_2.$$

  – Assume $\mathscr{P}(m, p_1, p_2)$ for all $p_1 < n_1$ and $p_2 < n_2$. Since $n'_1 \leq n_1$ and $n'_2 \leq n_2$, we have $\mathscr{P}(m, p_1, p_2)$ for all $p_1 < n'_1$ and $p_2 < n'_2$. We can apply Lemma C.16 to the first steps, and get that there exists a bijection $\beta' \supseteq \beta$ such that

$$\Gamma, \Gamma_t \vdash (m, i'_1, \omega'_1) : pc'_1, \Delta'_1, Q'_1 \approx^{\diamond}_{k,\beta'} (m, i'_2, \omega'_2) : pc'_2, \Delta'_2, Q'_2.$$

  Since the remaining execution is now shorter, we can apply the theorem inductively, and get by transitivity of configuration equivalence that there is some $\beta_{ret} \supseteq \beta$ the final configurations are equivalent.

- Let $[\![pc_1]\!]^{\diamond}_{\omega_1} \not\leq^{\diamond} k$ and $[\![pc_2]\!]^{\diamond}_{\omega_2} \not\leq^{\diamond} k$. Let $i_1 \neq i_{ret}$ without loss of generality. (If $i_2 \neq i_{ret}$, then we can swap the two configurations and apply the theorem, as configuration equivalence is symmetric modulo inverse bijections.)

We then have

$$(m, i_1, \omega_1) \xrightarrow{I_1}{}^{\diamond}_{n'_1} (m, i'_1, \omega'_1) \underset{IR}{\Longrightarrow}{}^{\diamond}_{n''_1} (m, i_{ret}, \omega_{ret1}).$$

Since (m, i_2, ω_2) does not diverge, we get with Lemma C.14 that there is some (m, i'_2, ω'_2) such that $(m, i_2, \omega_2) \xrightarrow{I_2}{}^{\diamond}_{n_2} (m, i'_2, \omega'_2)$ and

$$\Gamma, \Gamma_t \vdash (m, i'_1, \omega'_1) : \Lambda(i'_1) \approx^{\diamond}_{k,\beta'} (m, i'_2, \omega'_2) : \Lambda(i'_2).$$

Also, since program execution is deterministic, $(m, i'_2, \omega'_2) \underset{IR}{\Longrightarrow}{}^{\diamond}_{n''_2} (m, i_{ret}, \omega_{ret2})$. Since the executions are now shorter, we can apply the theorem inductively, and get by transitivity that there is some $\beta_{ret} \supseteq \beta$ such that the final configurations are equivalent. ■

Lemma C.17 *Let P_{IR} be a well-typed IR program, and m be a method. Then m is universally noninterferent.*

PROOF Let $\mathsf{msig}(m) = [\Gamma, pc, Q, Q']$.

Let k be a domain, and $(s_1, h_1), (s_2, h_2)$ be states such that

- $\vdash^{\diamond} (s_1, h_1) \sim^{\Gamma,k}_{\beta} (s_2, h_2)$, and
- $(s_1, h_1) \overset{m}{\underset{IR}{\Longrightarrow}}{}^{\diamond}_{n_1} (s'_1, h'_1)$, and
- $(s_2, h_2) \overset{m}{\underset{IR}{\Longrightarrow}}{}^{\diamond}_{n_2} (s'_2, h'_2)$, and
- $(s_1, h_1) \models^{\diamond} Q$ and $(s_2, h_2) \models^{\diamond} Q$.

We need to show that there exists a partial bijection $\beta' \supseteq \beta$ such that

- $\vdash^{\diamond} (s'_1, h'_1) \sim^{\Gamma,k}_{\beta'} (s'_2, h'_2)$, and
- $(s'_1, h'_1) \models^{\diamond} Q'$ and $(s'_2, h'_2) \models^{\diamond} Q'$.

We define $i_0 = \mathsf{mentry}(m)$ and $i_{ret} = \mathsf{mexit}(m)$ and $s_t = [\mathsf{tvars}(m) \mapsto \mathsf{defval}]$. By definition of method execution, we have $(m, i_0, s_1, s_t, h_1) \underset{IR}{\Longrightarrow}{}^{\diamond}_{n_1} (m, i_{ret}, s'_1, s_{t1}, h'_1)$ and $(m, i_0, s_2, s_t, h_2) \underset{IR}{\Longrightarrow}{}^{\diamond}_{n_2} (m, i_{ret}, s'_2, s_{t2}, h'_2)$.

Since the program is well-typed, there exists a type environment Γ_t, and a type mapping Λ which is derivable for $IR(m)$ for Γ and Γ_t, such that $\Lambda(i_0) = (pc, \epsilon, Q)$ and $\Lambda(i_{ret}) = (pc, \epsilon, Q')$. We have trivially $\vdash^{\diamond} s_t \sim^{\Gamma_t,k}_{\beta} s_t$. Let $\omega_1 = (s_1, s_t, h_1)$ and $\omega_2 = (s_2, s_t, h_2)$ and $\omega'_1 = (s'_1, s_{t1}, h'_1)$ and $\omega'_2 = (s'_2, s_{t2}, h'_2)$. We get $\vdash^{\diamond} \omega_1 \sim^{\Gamma,\Gamma_t,k}_{\beta} \omega_2$.

Since the method signature is well-formed, we know that there exists an expression e such that $\Gamma \vdash e : pc$. From Corollary A.4, it follows that either $[\![pc]\!]^{\diamond}_{\omega_1} = [\![pc]\!]^{\diamond}_{\omega_2} \leq^{\diamond} k$, or $[\![pc]\!]^{\diamond}_{\omega_1} \not\leq^{\diamond} k$ and $[\![pc]\!]^{\diamond}_{\omega_2} \not\leq^{\diamond} k$. With all the preceding facts, we can conclude by definition

$$\Gamma, \Gamma_t \vdash (i_1, \omega_1) : pc_1, \epsilon, Q \approx^{\diamond}_{k,\beta} (i_2, \omega_2) : pc_2, \epsilon, Q$$

and can thus apply Theorem C.15, which gives us that there exists a bijection $\beta' \supseteq \beta$ such that $\Gamma, \Gamma_t \vdash (i_{ret}, \omega'_1) : pc, \epsilon, Q' \approx^{\diamond}_{k,\beta'} (i_{ret}, \omega'_2) : pc, \epsilon, Q'$. By definition, it follows that $\vdash^{\diamond} \omega'_1 \sim^{\Gamma,\Gamma_t,k}_{\beta'} \omega'_2$, thus $\vdash^{\diamond} (s'_1, h'_1) \sim^{\Gamma,k}_{\beta'} (s_2, h'_2)$. Also, from the configuration equivalence it follows $\omega'_1 \models^{\diamond} Q'$ and $\omega'_2 \models^{\diamond} Q'$. Since $\mathsf{msig}(m)$ is well-formed, Q and Q' do not contain any variables from *TVar*, so $(s'_1, h'_1) \models^{\diamond} Q'$ and $(s'_2, h'_2) \models^{\diamond} Q'$.

This means the method m is universally noninterferent. ■

Corollary C.18 *If P_{IR} is a well-typed IR program, then it is universally noninterferent.*

PROOF The corollary follows immediately from Lemma C.17 and definition of universally noninterferent IR programs.

D
Type-Preserving Compilation

In this appendix, we show that if P_{DSD} is a well-typed DSD program with respect to the high-level type system, then $\mathsf{BC2IR}(\mathsf{compile}(P_{\mathrm{DSD}}))$ exists and is well-typed with respect to the IR type system.

The proof consists of two parts:

1. We show that $\mathsf{BC2IR}(\mathsf{compile}(P_{\mathrm{DSD}}))$ indeed exists, and has the following properties: stacks are empty between compiled statements, and high-level expressions are completely recovered by the BC2IR algorithm.
2. Then we show that there exists a valid type mapping Λ for $\mathsf{BC2IR}(\mathsf{compile}(P_{\mathrm{DSD}}))$, i.e., the program is well-typed.

D.1 Properties of the IR Program

Proposition D.1 *Let* $(BC, i_1) = \mathsf{compile}_{\mathrm{exp}}(m, e, i_0)$ *and* $AS_{in}[m, i_0] = as$. *Then*

1. $IR = \mathsf{BC2IR}_{rng}(BC, m, [i_0, i_1[)$ *exists, and*
2. $\forall i \in [i_0, i_1[.\ IR[m, i] = \mathtt{block}\,\epsilon$*, and*
3. $AS_{in}[m, i_1] = e :: as$.

PROOF First, we observe that by Proposition 5.3 on page 73, $\mathsf{dom}(BC(m)) = [i_0, i_1[$, and $\mathsf{jmpTgt}^{BC}_{m} = \emptyset$. Therefore, $\mathsf{BC2IR}_{\mathrm{rng}}(BC, m, [i_0, i_1[)$ cannot fail, hence IR exists.

We continue by induction over the structure of the expression e.

- $e = c$. Then $(BC, i_1) = ([(m, i_0) \mapsto \texttt{push}\ c], i_0 + 1)$. By definition of the algorithm, $\mathsf{BC2IR}_{\text{instr}}(i_0, \texttt{push}\ c, AS_{in}[m, i_0]) = (\texttt{block}\ \epsilon, c :: as) = (IR[m, i_0], AS_{in}[m, i_1])$.

- $e = x$. Then $(BC, i_1) = ([(m, i_0) \mapsto \texttt{load}\ x], i_0 + 1)$. By definition of the algorithm, $\mathsf{BC2IR}_{\text{instr}}(i_0, \texttt{load}\ x, AS_{in}[m, i_0]) = (\texttt{block}\ \epsilon, x :: as) = (IR[m, i_0], AS_{in}[m, i_1])$.

- $e = e.f$. We have $(BC', i') = \mathsf{compile}_{\text{exp}}(m, e, i_0)$ and $BC = BC' \cup [(m, i') \mapsto \texttt{getf}\ f]$ and $i_1 = i' + 1$. By induction, we get $AS_{in}[m, i'] = e :: as$, and $IR[m, i] = \texttt{block}\ \epsilon$ for all $i \in [i_0, i'[$. We have $\mathsf{BC2IR}_{\text{instr}}(i', \texttt{getf}\ f, e :: as) = (\texttt{block}\ \epsilon, e.f :: as) = (IR[m, i'], AS_{in}[m, i_1])$. Thus, $IR[m, i] = \texttt{block}\ \epsilon$ for all $i \in [i_0, i_1[$.

- $e = e_1\ \mathbf{op}\ e_2$. By definition of the compilation, $(BC', i') = \mathsf{compile}_{\text{exp}}(m, e_1, i_0)$ and $(BC'', i'') = \mathsf{compile}_{\text{exp}}(m, e_2, i')$ and $BC = BC' \cup BC'' \cup [(m, i'') \mapsto \texttt{prim}\ op]$ and $i_1 = i'' + 1$. Applying the proposition inductively twice, we get $AS_{in}[m, i'] = e_1 :: as$ and $AS_{in}[m, i''] = e_2 :: e_1 :: as$ and $IR[m, i] = \texttt{block}\ \epsilon$ for all $i \in [i_0, i''[$. We have

$$\begin{aligned} \mathsf{BC2IR}_{\text{instr}}(i'', \texttt{prim}\ op, e_2 :: e_1 :: as) &= (\texttt{block}\ \epsilon, (e_1\ \mathbf{op}\ e_2) :: as) \\ &= (IR[m, i''], AS_{in}[m, i_1]). \end{aligned}$$

  Thus, $IR[m, i] = \texttt{block}\ \epsilon$ for all $i \in [i_0, i_1[$. ∎

Proposition D.2 *Let* $(BC, i_1) = \mathsf{compile}_{\text{stmt}}(m, S, i_0)$, *and* $AS_{in}[m, i_0] = \epsilon$. *Then*

1. $IR = \mathsf{BC2IR}_{rng}(BC, m, [i_0, i_1[)$ *exists, and*

2. $AS_{in}[m, i_1] = \epsilon$.

PROOF By induction over the structure of S.

Suppose $S = \mathbf{skip}$. Then BC is empty, and $i_1 = i_0$. Hence IR trivially exists, and $AS_{in}[m, i_1] = AS_{in}[m, i_0] = \epsilon$.

All other cases for S are explained by Tables D.1 to D.7, respectively. For reference, the tables present the layout of the compiled bytecode program BC as defined by the compilation function. Moreover, they show how the corresponding IR instructions and abstract stacks look like, thereby proving that IR indeed exists, and that the final abstract state is ϵ.

For each instruction address i, the tables list the compiled bytecode instruction(s) starting at i, then initial abstract stack $AS_{in}[m, i]$, the corresponding IR instruction(s) starting at i, and the resulting abstract stack $AS_{in}[m, i^+]$, where i^+ is the first address of the following block (indicated by the next line).

Each row corresponds to an instruction sequence starting at a specific address i. The symbols in the "remarks" column show how to obtain the IR instruction(s) and abstract stacks:

i	$BC(m, i \text{ to } i^+ - 1)$	$AS_{in}[m,i]$	$IR[m, i \text{ to } i^+ - 1]$	$AS_{in}[m,i^+]$	remarks
i_0	$[\text{compile}_{\text{exp}}(m,e,i_0)]$ ⋮	ϵ	[sequence of `block` ϵ]	$[e]$	(Init), (Expr)
i'	`store` x	$[e]$	`block` $[x := e]$	ϵ	(Instr)
i_1	…	ϵ			(Result)

Table D.1: Proof for $x := e$

i	$BC(m, i \text{ to } i^+ - 1)$	$AS_{in}[m,i]$	$IR[m, i \text{ to } i^+ - 1]$	$AS_{in}[m,i^+]$	remarks
i_0	$[\text{compile}_{\text{exp}}(m,e_r,i_0)]$ ⋮	ϵ	[sequence of `block` ϵ]	$[e_r]$	(Init), (Expr)
i'	$[\text{compile}_{\text{exp}}(m,e,i')]$ ⋮	$[e_r]$	[sequence of] `block` ϵ]	$[e :: e_r]$	(Expr)
i''	`putf` f	$[e :: e_r]$	`block` $[e_r.f := e]$	ϵ	(Instr)
i_1	…	ϵ			(Result)

Table D.2: Proof for $S = e_r.f := e$

i	$BC(m, i \text{ to } i^+ - 1)$	$AS_{in}[m,i]$	$IR[m, i \text{ to } i^+ - 1]$	$AS_{in}[m,i^+]$	remarks
i_0	$[\text{compile}_{\text{exp}}(m,\overline{e},i_0)]$	ϵ	[sequence of `block` ϵ]	$[\overline{e}]$	(Init), (Expr)
i'	`new` C	$[\overline{e}]$	`block` $[t^0_{i'} := \mathbf{new}\ C(\overline{e})]$	$[t^0_{i'}]$	(Instr)
$i'+1$	`store` x	$[t^0_{i'}]$	`block` $[x := t^0_{i'}]$	ϵ	(Instr)
i_1	…	ϵ			(Result)

Table D.3: Proof for $S = x := \mathbf{new}\ C(\overline{e})$

i	$BC(m, i \text{ to } i^+ - 1)$	$AS_{in}[m, i]$	$IR[m, i \text{ to } i^+ - 1]$	$AS_{in}[m, i^+]$	remarks
i_0	[compile$_{\text{exp}}(m, e_r, i_0)$] ⋮	ϵ	[sequence of `block` ϵ]	$[e_r]$	(Init), (Expr)
i'	[compile$_{\text{exp}}(m, \overline{e}, i')$] ⋮	$[e_r]$	[sequence of `block` ϵ]	$[\overline{e} :: e_r]$	(Expr)
i''	`call` m'	$[\overline{e} :: e_r]$	`block` $[t^0_{i''} := e_r.m'(\overline{e})]$	$[t^0_{i''}]$	(Instr)
$i'' + 1$	`store` x	$[t^0_{i''}]$	`block` $[x := t^0_{i''}]$	ϵ	(Instr)
i_1	…	ϵ			(Result)

Table D.4: Proof for $S = x := e_r.m(\overline{e})$

i	$BC(m, i \text{ to } i^+ - 1)$	AS$_{in}$ $[m, i]$	$IR[m, i \text{ to } i^+ - 1]$	AS$_{in}$ $[m, i^+]$	remarks
i_0	`cpush` i_1	ϵ	`cpush` i_1	ϵ	(Init), (Instr)
$i_0 + 1$	[compile$_{\text{exp}}(m, e, i_0 + 1)$] ⋮	ϵ	[sequence of `block` ϵ]	$[e]$	(Expr)
i'	`bnz` $i'' + 1$	$[e]$	`if` e $i'' + 1$	ϵ	(Instr), (Jump)
$i' + 1$	[compile$_{\text{stmt}}(m, S_2, i' + 1)$] ⋮	ϵ	[BC2IR$_{\text{rng}}$ $(BC, m, [i' + 1, i''[))$]	ϵ	(Ind)
i''	`cjmp` i_1	ϵ	`cjmp` i_1	ϵ	(Instr), (Jump)
$i'' + 1$	[compile$_{\text{stmt}}(m, S_2, i'' + 1)$] ⋮	ϵ	[BC2IR$_{\text{rng}}$ $(BC, m, [i'' + 1, i'''[))$]	ϵ	(Ind)
i'''	`cjmp` i_1	ϵ	`cjmp` i_1	ϵ	(Instr), (Jump)
i_1	…	ϵ			(Result)

Table D.5: Proof for $S =$ **if** e **then** S_1 **else** S_2

i	$BC(m, i \text{ to } i^+ - 1)$	$AS_{in}[m,i]$	$IR[m, i \text{ to } i^+ - 1]$	$AS_{in}[m,i^+]$	remarks
i_0	[$\mathsf{compile}_{\mathsf{stmt}}(m, S_1, i_0)$] ⋮	ϵ	[$\mathsf{BC2IR}_{\mathsf{rng}}$ $(BC, m, [i_0, i'[)$]	ϵ	(Init), (Ind)
i'	[$\mathsf{compile}_{\mathsf{stmt}}(m, S_2, i')$] ⋮	ϵ	[$\mathsf{BC2IR}_{\mathsf{rng}}$ $(BC, m, [i', i_1[)$]	ϵ	(Ind)
i_1	…	ϵ			(Result)

Table D.6: Proof for $S = S_1 \, ; S_2$

i	$BC(m, i \text{ to } i^+ - 1)$	AS_{in} $[m,i]$	$IR[m, i \text{ to } i^+ - 1]$	AS_{in} $[m,i^+]$	remarks
i_0	`cpush` i_1	ϵ	`cpush` i_1	ϵ	(Init), (Instr)
i_0+1	[$\mathsf{compile}_{\mathsf{exp}}(m, e, i_0+1)$] ⋮	ϵ	[sequence of `block` ϵ]	$[e]$	(Expr)
i'	`bnz` $i'+2$	$[e]$	`if` e $i'+2$	ϵ	(Instr), (Jump)
$i'+1$	`cjmp` i_1	ϵ	`cjmp` i_1	ϵ	(Instr), (Jump)
$i'+2$	[$\mathsf{compile}_{\mathsf{stmt}}(m, S, i'+2)$] ⋮	ϵ	[$\mathsf{BC2IR}_{\mathsf{rng}}$ $(BC, m, [i'+2, i''[)$]	ϵ	(Ind)
i''	[$\mathsf{compile}_{\mathsf{exp}}(m, e, i'')$] ⋮	ϵ	[sequence of `block` ϵ]	$[e]$	(Expr)
i'''	`bnz` $i'+2$	$[e]$	`if` e $i'+2$	ϵ	(Instr), (Jump)
$i'''+1$	`cjmp` i_1	ϵ	`cjmp` i_1	ϵ	(Instr), (Jump)
i_1	…	ϵ			(Result)

Table D.7: Proof for $S =$ **while** e **do** S

(Init) We have $AS_{in}[m, i_0] = \epsilon$ by assumption.

(Instr) This line contains a single bytecode instruction, hence we can simply apply the definition of $\mathsf{BC2IR}_{\mathrm{rng}}(BC, m, [i])$. In all cases, we do not need to rescue abstract stack values in temporary variables, since the bottom of the abstract stack, i.e., the part that does not contain instruction-relevant values, is empty. Also, since $i^+ = i + 1$ in these cases, we get $AS_{in}[m, i^+] = AS_{in}[m, i+1] = AS_{out}[m, i]$.

(Jump) In this case, there is a $j \in \mathsf{succ}(m, i) \cap \mathsf{jmpTgt}_m^{BC}$, but $AS_{in}[m, j] = \epsilon$, so the compilation to IR does not fail.

(Expr) This line contains compiled expression block, thus we apply Proposition D.1.

(Ind) This line contains a block of a compiled substatement, and $AS_{in}[m, i] = \epsilon$. Therefore, we can apply this proposition inductively.

(Result) This line shows $AS_{in}[m, i_1] = \epsilon$.

From Proposition 4.3 on page 59, we get that no jumps occur within compiled subexpressions. Hence the lines marked with (Jump) are the only instructions where jumps do occur, and we have shown that the abstract stacks at the jumps are empty. Therefore, $\mathsf{BC2IR}_{\mathrm{rng}}(BC, [i_0, i_1[)$ does not fail, so the compiled IR program exists. ∎

D.2 Type Derivation for the IR Program

This final section shows that for every type derivation for the high-level DSD program, there exists a corresponding type derivation for the corresponding IR program. "Coresponding" means that there exists an IR type mapping for each method that matches its method signature for a (fixed) variable type environment Γ_t.

Proposition D.3 *Let* $(BC, i_1) = \mathsf{compile}_{\mathrm{exp}}(m, e, i_0)$*, and* $IR = \mathsf{BC2IR}_{rng}(BC, m, [i_0, i_1[)$. *Let* $\mathsf{msig}(m) = [\Gamma, pc^m, Q_0^m, Q_1^m]$. *Let* Γ_t *be an arbitrary type environment for temporary variables, and let* pc, Δ, Q *be any* pc *label, confluence point stack, and constraint set. Then there exists a type mapping* Λ *derivable for IR and* Γ, Γ_t *such that* $\Lambda(i_0) = (pc, \Delta, Q)$ *and* $\Lambda(i_1) = (pc, \Delta, Q)$.

Proof By Proposition D.1, we know $IR[m, i] = \texttt{block}\ \epsilon$; therefore, we get the type derivations $\Gamma, \Gamma_t \vdash i, pc, \Delta, Q \xrightarrow{\mathrm{I}} i+1, pc, \Delta, Q$ for all $i \in [i_0, i_1[$. This means we can derive a type mapping Λ such that $(\Lambda(i_0) = \Lambda(i_0 + 1) = \Lambda(i_0 + 2) = \ldots = \Lambda(i_1)$. ∎

In the proof for compiled statements, we extend the definitions from Section 3.4.1 on page 39 with the set of all identifiers that occur in an expression: $\mathsf{ids}(e) = \mathsf{vars}(e) \cup \mathsf{flds}(e)$. This definition is lifted to confluence point stacks $\mathsf{ids}(\Delta)$, which refers to all fields and all variables that occur syntactically in the pc labels on the Δ stack.

Proposition D.4 *Let P_{DSD} be a high-level DSD program, with given signatures for the methods. Let m be a method, and S be a high-level statement that occurs in the method body. Let* $(BC, i_1) = \mathsf{compile}_{\mathsf{stmt}}(m, S, i_0)$. *Suppose this compiled program is translated to IR code, such that* $AS_{in}[m, i_0] = \epsilon$ *and* $IR = \mathsf{BC2IR}_{rng}(BC, m, [i_0, i_1[)$. *Suppose* $\mathsf{msig}(m) = [\Gamma, pc^m, Q_0^m, Q_1^m]$ *and* $\Gamma, pc \vdash \{Q_0\}\ S\ \{Q_1\}$. *Let Δ be an arbitrary confluence point stack, such that* $\mathsf{ids}(\Delta) \subseteq \mathsf{ids}(pc)$. *Then there exists a type environment Γ_t and a type mapping Λ derivable for IR and Γ, Γ_t such that* $\Lambda(i_0) = (pc, \Delta, Q_0)$ *and* $\Lambda(i_1) = (pc, \Delta, Q_1)$.

PROOF By induction over the type deriviation. If $\Gamma, pc \vdash \{Q_0\}\ S\ \{Q_1\}$ has been derived by the subsumption rule T-WEAK, then we get by rule inversion that there is some other typing judgement $\Gamma, pc \vdash \{Q_0'\}\ S\ \{Q_1'\}$ with $Q_0 \Rightarrow Q_0'$ and $Q_1' \Rightarrow Q_1$. We can apply the proposition inductively and get a type mapping derivable for IR and Γ, Γ_t. By definition, there must exist a number of small-step IR typing rules for each instruction, such that we can apply the weakening rule of the IR type system to get the desired pre- and postconditions, which again means that there is a derivable type mapping Λ for Q_0 and Q_1.

In the following, suppose $\Gamma, pc \vdash \{Q_0\}\ S\ \{Q_1\}$ has not been derived by the subsumption rule. We proceed by case distinction over S.

- $S_1\ ;\ S_2$: The IR type derivation is shown in Table D.8 on the next page. We can directly apply this proposition inductively to both substatements, and get suitable type mappings Λ_1, Λ_2 and environments Γ_{t1}, Γ_{t2} for each of the substatements. Since the IR parts that correspond to the statements do not overlap, and since the temporary variables used in both parts are disjoint thanks to the notation, we can combine them and get the desired type mapping Λ and environment Γ_t.

- $x := e$: Let $Q_0 = Q \cup Q'[e/x]$ and $Q_1 = Q'$. The IR type derivation is shown in Table D.9 on the following page. We set $\Gamma_t(t_{i'}^0) = \Gamma(x)$. This is well-defined, because $t_{i'}^0$ only appears at this compiled IR fragment in the entire IR program. Since $t_{i'}^0$ is a temporary variable, it cannot occur in pc, and thus, with $\mathsf{ids}(\Delta) \subseteq \mathsf{ids}(pc)$, it also cannot occur in pc. We can use these facts and the premises of the T-ASSIGN rule to type the IR block of assignments. Note that the substitution of $t_{i'}^0$ with x can be ignored, since $t_{i'}^0$ could have occurred neither in the high-level data structures $Q' = Q_1$ nor in e, since it is a temporary variable.

- $e_r.f := e$: Let $Q_0 = Q \cup Q'[e/e_r.f]$ and $Q_1 = Q'$. The IR type derivation is shown in Table D.10 on the next page. Since the abstract input stack at the field assignment is empty, there is no need to save it in temporary variables $\overline{t_{i''}}$. The field update block can be typed directly by using the premises of T-PUTFIELD. Also, since $f \not\in pc$ and $\mathsf{ids}(\Delta) \subseteq \mathsf{ids}(pc)$, we get $f \not\in \Delta$.

i	$\Lambda(i)$	$IR[m, i \text{ to } i^+ - 1]$	$\Lambda(\text{succ}(m,i))$	i^+	remarks
i_0	(pc, Δ, Q_0)	$[\text{BC2IR}_{\text{rng}}(BC, m, [i_0, i'[)]$ $\vdots$	(pc, Δ, Q')	i'	by induction
i'	(pc, Δ, Q')	$[\text{BC2IR}_{\text{rng}}(BC, m, [i', i_1[)]$ $\vdots$	(pc, Δ, Q_1)	i_1	by induction
i_1	(pc, Δ, Q_1)				

Table D.8: Proof for $S = S_1 \,;\, S_2$

i	$\Lambda(i)$	$IR[m, i \text{ to } i^+ - 1]$	$\Lambda(\text{succ}(m,i))$	i^+	remarks
i_0	$(pc, \Delta, Q \cup Q'[e/x])$ $\vdots$	[sequence of `block` ϵ]	$(pc, \Delta, Q \cup Q'[e/x][x/t^0_{i'}])$	i'	Proposition D.3 and with $t^0_{i'} \not\in Q', e$
i'	$(pc, \Delta, Q \cup Q'[e/x][x/t^0_{i'}])$	`block` $[t^0_{i'} := x \,;\, x := e]$	(pc, Δ, Q')	i_1	by premises of T-ASSIGN and $\Gamma_t(t^0_{i'}) = \Gamma(x)$ and $x, t^0_{i'} \not\in \Delta, pc$ and $t^0_{i'} \neq x_\delta$
i_1	(pc, Δ, Q')				

Table D.9: Proof for $x := e$

i	$\Lambda(i)$	$IR[m, i \text{ to } i^+ - 1]$	$\Lambda(\text{succ}(m,i))$	i^+	remarks
i_0	$(pc, \Delta, Q \cup Q'[e/e_r.f])$	[sequence of `block` ϵ] $\vdots$	$(pc, \Delta, Q \cup Q'[e/e_r.f])$	i'	Proposition D.3
i'	$(pc, \Delta, Q \cup Q'[e/e_r.f])$	[sequence of `block` ϵ] $\vdots$	$(pc, \Delta, Q \cup Q'[e/e_r.f])$	i''	Proposition D.3
i''	$(pc, \Delta, Q \cup Q'[e/e_r.f])$	`block` $[e_r.f := e]$	(pc, Δ, Q')	i_1	by premises of T-PUTFIELD and with $f \not\in \Delta$
i_1	(pc, Δ, Q')				

Table D.10: Proof for $S = e_r.f := e$

- $x := e_r.m(\overline{e})$: Let Q_0 and Q_1 be the pre- and postcondition sets as defined in the high-level T-CALL rule. We define $\Gamma_t(t^0_{i''}) = \Gamma(x)$. This is well-defined, because $t^0_{i''}$ only appears at this compiled IR fragment in the entire IR program. Table D.11 on the following page shows the IR type derivation. As $t^0_{i''}$ is a temporary variable, $t_{i''} \not\in pc, \Delta$. From the premises of the T-CALL rule, we get the label order information $\Gamma^*(ret) \sqcup pc \sqcup \ell_r \sqsubseteq_{Q_1} \Gamma(x)$, hence $\Gamma^*(ret) \sqcup pc \sqcup \ell_r \sqsubseteq_{Q_1} (\Gamma \cup \Gamma_t)(t^0_{i''})$. Note that the postconditions of the two assignments are chained, such that we get the desired overall postcondition Q_1.

- $x := \textbf{new}\ C(\overline{e})$: Let $Q_0 = Q \cup Q'[\overline{e}/x.\overline{f}]$ and $Q_1 = Q$. The IR type derivation is shown in Table D.11 on the next page. We define $\Gamma_t(t^0_{i_0}) = \Gamma(x)$. This is well-defined, because $t^0_{i_0}$ only appears at this compiled IR fragment in the entire IR program. Since $t^0_{i_0}$ is a temporary variable, we have $t^0_{i_0} \not\in Q, pc, \Delta$. By rule inversion, we have $pc \sqsubseteq_Q \Gamma(x)$, hence $pc \sqsubseteq_Q \Gamma_t(t^0_{i_0})$. With these facts and the premises of T-NEW, we can type the IR object creation instruction. For the second assignment, we use $pc \sqsubseteq_Q \Gamma(x)$. Also, note that the substitutions that occur in the preconditions of the assignments are chained, such that we get the desired overall precondition Q_0.

- **if** e **then** S_1 **else** S_2: We have $Q_0 = Q$ and $Q_1 = Q'$. Table D.13 on page 157 shows the type derivation. Essentially, we use Proposition D.3 for expressions, and this proposition inductively for substatements. To apply the proposition inductively, we also need to show that the set of identifiers in the new Δ stack is smaller than the one of the pc label, but this follows from the premises. For the same reasons as described for sequential compositions, the resulting type mappings and temporary variable type environments can be combined together with the information shown in the table to obtain the desired mapping Λ and environment Γ_t.

  The case for label test conditionals is not shown in a table, but it is very similar, because the constraint sets are updated in the same way for the **then** branch in the high-level and IR typing rules.

- **while** e **do** S: We have $Q_0 = Q_1 = Q$. See Table D.14 on page 158 for the IR type derivation. Essentially, we use Proposition D.3 for expressions, and this proposition inductively for substatements. To apply the proposition inductively, we also need to show that the set of identifiers in the new Δ stack is smaller than the one of the pc label, but this follows from the premises. We get a suitable type environment Γ_t. The type mapping that we get for the compiled part that corresponds to S can be extended with the information shown in the table to get the desired type mapping Λ. ∎

i	$\Lambda(i)$	$IR[m, i \text{ to } i^+ - 1]$	$\Lambda(\text{succ}(m,i))$	i^+	remarks
i_0	(pc, Δ, Q_0)	[sequence of `block` ϵ] ⋮	(pc, Δ, Q_0)	i'	Proposition D.3
i'	(pc, Δ, Q_0)	[sequence of `block` ϵ] ⋮	(pc, Δ, Q_0)	i''	Proposition D.3
i''	(pc, Δ, Q_0)	`block` $[t^0_{i''} := e_r.m'(\overline{e})]$	$(pc, \Delta, Q \cup Q'_m[t^0_{i''}/ret])$	$i'' + 1$	by premises of T-CALL and with $t^0_{i''} \notin pc, \Delta, Q$ and $t^0_{i''} \neq x_\delta$
$i'' + 1$	$(pc, \Delta, Q \cup Q'_m[t^0_{i''}/ret])$	`block` $[x := t^0_{i''}]$	$(pc, \Delta, Q \cup Q'_m[x/ret])$	i_1	by premises of T-ASSIGN and with $x \notin \Delta$
i_1	$(pc, \Delta, Q \cup Q'_m[x/ret])$				

Table D.11: Proof for $S = x := e_r.m(\overline{e})$

i	$\Lambda(i)$	$IR[m, i \text{ to } i^+ - 1]$	$\Lambda(\text{succ}(m,i))$	i^+	remarks
i_0	$(pc, \Delta, Q \cup Q'[\overline{e}/x.\overline{f}]$	`block` $[t^0_{i_0} := \textbf{new}\ C(\overline{e})]$	$(pc, \Delta, Q \cup Q'[t^0_{i_0}/x]$	$i_0 + 1$	by premises of T-NEW and $t^0_{i_0} \notin Q, pc, \Delta$ and $t^0_{i_0} \neq x_\delta$ and $pc \sqsubseteq_Q \Gamma_t(t^0_{i_0})$
$i_0 + 1$	$(pc, \Delta, Q \cup Q'[t^0_{i_0}/x]$	`block` $[x := t^0_{i_0}]$	(pc, Δ, Q)	i_1	by premises of T-ASSIGN, and $\Gamma_t(t^0_{i_0}) = \Gamma(x)$ and $t^0_{i_0} \notin \Delta, pc$ and $pc \sqsubseteq_Q \Gamma(x)$
i_1	(pc, Δ, Q)				

Table D.12: Proof for $S = x := \textbf{new}\ C(\overline{e})$

i	$\Lambda(i)$	$IR[m, i \text{ to } i^+ - 1]$	$\Lambda(\mathsf{succ}(m,i))$	i^+	remarks
i_0	(pc, Δ, Q_0)	`cpush` i_1	$(pc, (i_1, pc) :: \Delta, Q_0)$	$i_0 + 1$	by definition
$i_0 + 1$	$(pc, (i_1, pc) :: \Delta, Q_0)$	[sequence of `block` ϵ] ⋮	$(pc, (i_1, pc) :: \Delta, Q_0)$	i'	Proposition D.3
i'	$(pc, (i_1, pc) :: \Delta, Q_0)$	`if` $e\ i'' + 1$	$(pc \sqcup \ell, (i_1, pc) :: \Delta, Q_0)$	$i' + 1$	by definition; $\Gamma \cup \Gamma_t \vdash e : \ell$
$i' + 1$	$(pc \sqcup \ell, (i_1, pc) :: \Delta, Q_0)$	$[\mathsf{BC2IR}_{\text{rng}}(BC, m, [i'+1, i''[)]$ ⋮	$(pc \sqcup \ell, (i_1, pc) :: \Delta, Q_1)$	i''	by induction and $\mathsf{ids}((i_1, pc) :: \Delta) \subseteq \mathsf{ids}(pc \sqcup \ell)$
i''	$(pc \sqcup \ell, (i_1, pc) :: \Delta, Q_1)$	`cjmp` i_1	(pc, Δ, Q_1)	$i'' + 1$	by definition
$i'' + 1$	$(pc \sqcup \ell, (i_1, pc) :: \Delta, Q_0)$	$[\mathsf{BC2IR}_{\text{rng}}(BC, m, [i''+1, i'''[)]$ ⋮	$(pc \sqcup \ell, (i_1, pc) :: \Delta, Q_1)$	i'''	by induction and $\mathsf{ids}((i_1, pc) :: \Delta) \subseteq \mathsf{ids}(pc \sqcup \ell)$
i'''	$(pc \sqcup \ell, (i_1, pc) :: \Delta), Q_1)$	`cjmp` i_1	(pc, Δ, Q_1)	i_1	by definition
i_1	(pc, Δ, Q_1)				

Table D.13: Proof for $S = \textbf{if}\ e\ \textbf{then}\ S_1\ \textbf{else}\ S_2$

i	$\Lambda(i)$	$IR[m, i \text{ to } i^+ - 1]$	$\Lambda(\mathsf{succ}(m,i))$	i^+	remarks
i_0	(pc, Δ, Q)	`cpush` i_1	$(pc, (i_1, pc) :: \Delta, Q)$	$i_0 + 1$	by definition
$i_0 + 1$	$(pc, (i_1, pc) :: \Delta, Q)$	[sequence of `block` ϵ]	$(pc, (i_1, pc) :: \Delta, Q)$	i'	Proposition D.3
		⋮			
i'	$(pc, (i_1, pc) :: \Delta, Q)$	`if` e $i' + 2$	$(pc \sqcup \ell, (i_1, pc) :: \Delta, Q)$	$i' + 1$	by definition; $\Gamma \cup \Gamma_t \vdash e : \ell$
$i' + 1$	$(pc \sqcup \ell, (i_1, pc) :: \Delta, Q)$	`cjmp` i_1	(pc, Δ, Q)	$i' + 2$	by definition
$i' + 2$	$(pc \sqcup \ell, (i_1, pc) :: \Delta, Q)$	[$\mathsf{BC2IR}_{\mathrm{rng}}(BC, m, [i' + 2, i''[)$]	$(pc \sqcup \ell, (i_1, pc) :: \Delta, Q)$	i''	by induction and $\mathsf{ids}((i_1, pc) :: \Delta) \subseteq \mathsf{ids}(pc \sqcup \ell)$
		⋮			
i''	$(pc \sqcup \ell, (i_1, pc) :: \Delta, Q)$	[sequence of `block` ϵ]	$(pc \sqcup \ell, (i_1, pc) :: \Delta, Q)$	i'''	Proposition D.3
		⋮			
i'''	$(pc \sqcup \ell, (i_1, pc) :: \Delta, Q)$	`if` e $i' + 2$	$(pc \sqcup \ell, (i_1, pc) :: \Delta, Q)$	$i''' + 1$	by definition, and since $pc \sqcup \ell \sqcup \ell = pc \sqcup \ell$
$i''' + 1$	$(pc \sqcup \ell, (i_1, pc) :: \Delta, Q)$	`cjmp` i_1	(pc, Δ, Q)	i_1	by definition
i_1	(pc, Δ, Q)				

Table D.14: Proof for $S = $ **while** e **do** S

Proposition D.5 *Let P_{DSD} be a well-typed DSD program, and m be a method with* $\mathsf{msig}(m) = [\Gamma, pc, Q, Q']$. *Let* $P_{\mathrm{BC}} = \mathsf{compile}(P_{\mathrm{DSD}})$. *Then* $P_{\mathrm{IR}} = \mathsf{BC2IR}_{mtd}(P_{\mathrm{BC}}, m)$ *exists, and there exists a type environment Γ_t such that the method m of the compiled program P_{IR} is well-typed with respect to this method signature.*

PROOF Since P_{DSD} is well-typed, we get by definition that $\Gamma, pc \vdash \{Q\}\ \mathsf{mbody}(m)\ \{Q'\}$.
By definition $P_{\mathrm{BC}} = (\prec, \mathsf{fields}, \mathsf{methods}, \mathsf{margs}, BC, \mathsf{mentry}, \mathsf{mexit})$ where

$$(BC(m), \mathsf{mexit}(m)) = \mathsf{compile}_{\mathrm{stmt}}(m, \mathsf{mbody}(m), \mathsf{mentry}(m))$$

By Proposition 4.3 on page 59, we know $\mathsf{dom}(BC(m)) = [\mathsf{mentry}(m), \mathsf{mexit}(m)[$. By definition of the algorithm $\mathsf{BC2IR}_{\mathrm{mtd}}(P_{\mathrm{BC}}, m)$, we have $AS_{in}[m, \mathsf{mentry}(m)] = \epsilon$, and also $IR(m) = \mathsf{BC2IR}_{\mathrm{rng}}(BC, m, [\mathsf{mentry}(m), \mathsf{mexit}(m)[)$. With Proposition D.2, we know $IR(m)$ exists. Let $\Delta = \epsilon$. With Proposition D.4, we know there is a type environment Γ_t and a type mapping Λ derivable for $IR(m)$ and Γ, Γ_t such that $\Lambda(\mathsf{mentry}(m)) = (pc, \epsilon, Q)$ and $\Lambda(\mathsf{mexit}(m)) = (pc, \epsilon, Q')$. But this means that m is well-typed with respect to signature $\mathsf{msig}(m) = [\Gamma, pc, Q, Q']$. ■

Theorem D.6 *Let P_{DSD} be well-typed with respect to given method signatures. Then* $\mathsf{BC2IR}(\mathsf{compile}(P_{\mathrm{DSD}}))$ *exists and is well-typed with respect to these signatures.*

PROOF Let $P_{\mathrm{BC}} = \mathsf{compile}(P_{\mathrm{DSD}})$. We get with Proposition D.5 that for all methods m, $IR(m) = \mathsf{BC2IR}_{\mathrm{mtd}}(P_{\mathrm{BC}}, m)$ exists, hence $P_{\mathrm{IR}} = \mathsf{BC2IR}(P_{\mathrm{DSD}})$ exists by definition of the algorithm BC2IR.

With the same proposition, we get that for all methods m with signature $\mathsf{msig}(m) = [\Gamma, pc, Q, Q']$, the fragment $\mathsf{BC2IR}(\mathsf{compile}(P_{\mathrm{DSD}}))$ is well-typed with respect to this signature. By definition, the entire program P_{IR} is thus well-typed. ■

Bibliography

[AB05] A. Almeida Matos and Gerard Boudol. “On Declassification and the Non-Disclosure Policy”. In: *18th IEEE Computer Security Foundations Workshop (CSFW 2005)*. IEEE Computer Society, 2005, pp. 226–240.

[ABB06] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. “A Logic for Information Flow in Object-Oriented Programs”. In: *33rd ACM Symposium on Principles of Programming Languages (POPL 2006)*. Charleston, South Carolina, USA: ACM Press, 2006, pp. 91–102.

[AR80] Gregory R. Andrews and Richard P. Reitman. “An Axiomatic Approach to Information Flow in Programs”. In: *ACM Transactions on Programming Languages and Systems* 2 (1 Jan. 1980), pp. 56–76.

[Ask+08] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. “Termination-Insensitive Noninterference Leaks More Than Just a Bit”. In: *13th European Symposium on Research in Computer Security (ESORICS 2008)*. Lecture Notes in Computer Science 5283. Springer-Verlag, 2008, pp. 333–348.

[Bal93] Thomas Ball. “What’s in a region?: or computing control dependence regions in near-linear time for reducible control flow”. In: *ACM Letters on Programming Languages and Systems* 2 (1-4 Mar. 1993), pp. 1–16.

[Bar+06] Gilles Barthe, Lennart Beringer, Pierre Crégut, Benjamin Grégoire, Martin Hofmann, Peter Müller, Erik Poll, Germán Puebla, Ian Stark, and Eric Vétillard. “MOBIUS: Mobility, Ubiquity, Security”. In: *Trustworthy Global Computing*. 2006, pp. 10–29.

[BBR04] Gilles Barthe, Amitabh Basu, and Tamara Rezk. “Security Types Preserving Compilation (Extended Abstract)”. In: *Verification, Model Checking and Abstract Interpretation (VMCAI 2004)*. 2004, pp. 2–15.

[BDR04] Gilles Barthe, Pedro R. D’Argenio, and Tamara Rezk. “Secure Information Flow by Self-Composition”. In: *17th IEEE Computer Security Foundations Workshop (CSFW 2004)*. Pacific Grove, California, USA: IEEE Computer Society Press, 2004, pp. 100–114.

[Ben04] Nick Benton. “Simple relational correctness proofs for static analyses and program transformations”. In: *31th ACM Symposium on Principles of Programming Languages (POPL 2004)*. Ed. by N. D. Jones and X. Leroy. ACM Press, 2004, pp. 14–25.

[Ber10] Lennart Beringer. “Relational bytecode correlations”. In: *Journal of Logic and Algebraic Programming* 79.7 (2010). The 20th Nordic Workshop on Programming Theory (NWPT 2008), pp. 483 –514.

[BGH10] Lennart Beringer, Robert Grabowski, and Martin Hofmann. “Verifying Pointer and String Analyses with Region Type Systems”. In: *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*. Ed. by Edmund Clarke and Andrei Voronkov. Vol. 6355. Lecture Notes in Computer Science. Springer-Verlag, 2010, pp. 82–102.

[BH07] Lennart Beringer and Martin Hofmann. “Secure information flow and program logics”. In: *20th IEEE Computer Security Foundations Symposium (CSF 2007)*. IEEE Computer Society, 2007, pp. 233–248.

[Bia+07] Gaowei Bian, Ken Nakayama, Yoshitake Kobayashi, and Mamoru Maekawa. “Java Bytecode Dependence Analysis for Secure Information Flow”. In: *International Journal on Network Security* (2007), pp. 59–68.

[BL73] David Elliott Bell and Leonard J. LaPadula. *Secure computer systems: Mathematical foundations*. Technical Report MTR-2547. Bedford, MA: MITRE Corporation, 1973.

[BMS03] Lennart Beringer, Kenneth MacKenzie, and Ian Stark. “Grail: a Functional Form for Imperative Mobile Code”. In: vol. 85. Electronic Notes in Theoretical Computer Science 1. 2nd EATCS Workshop on Foundations of Global Computing. Elsevier, June 2003, pp. 3–23.

[BN05] Anindya Banerjee and David A. Naumann. “Stack-based access control and secure information flow”. In: *Journal of Functional Programming* 15.2 (2005), pp. 131–177.

[BNR07] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. “Towards a logical account of declassification”. In: *2007 Workshop on Programming Languages and Analysis for Security (PLAS 2007)*. San Diego, California, USA: ACM, 2007, pp. 61–66.

[BPR07] Gilles Barthe, David Pichardie, and Tamara Rezk. “A Certified Lightweight Non-interference Java Bytecode Verifier”. In: *16th European Symposium on Programming (ESOP 2007)*. Ed. by Rocco De Nicola. Vol. 4421. Lecture Notes in Computer Science. Springer-Verlag, 2007, pp. 125–140.

[BR05] Gilles Barthe and Tamara Rezk. "Non-interference for a JVM-like language". In: *2005 ACM International Workshop on Types in Languages Design and Implementation (TLDI 2005)*. 2005, pp. 103–112.

[BRN06] Gilles Barthe, Tamara Rezk, and David A. Naumann. "Deriving an Information Flow Checker and Certifying Compiler for Java". In: *2006 IEEE Symposium on Security and Privacy (S&P 2006)*. IEEE Computer Society, 2006, pp. 230–242.

[BS10] Niklas Broberg and David Sands. "Paralocks – Role-Based Information Flow Control and Beyond". In: *37th Annual ACM Symposium on Principles of Programming Languages (POPL 2010)*. Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, 2010.

[BWW08] Sruthi Bandhakavi, William Winsborough, and Marianne Winslett. "A Trust Management Approach for Flexible Policy Management in Security-Typed Languages". In: *21st IEEE Computer Security Foundations Symposium (CSF 2008)*. Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 33–47.

[Cho+06] Stephen Chong, Andrew C. Myers, Krishnaprasad Vikram, and Lantian Zheng. *Jif Reference Manual*. June 2006. URL: `http://www.truststc.org/pubs/548.html`.

[Coh77] Ellis Cohen. "Information transmission in computational systems". In: *Sixth ACM symposium on Operating systems principles (SOSP 1977)*. New York, NY, USA: ACM, 1977, pp. 133–139.

[DD77] Dorothy E. Denning and Peter J. Denning. "Certification of Programs for Secure Information Flow". In: *Communications of the ACM* 20.7 (1977), pp. 504–513.

[DHS05] Adam Darvas, Reiner Hähnle, and David Sands. "A Theorem Proving Approach to Analysis of Secure Information Flow". In: *Security in Pervasive Computing*. Ed. by D. Hutter and M. Ullmann. Vol. 3450. Lecture Notes in Computer Science. Springer-Verlag, 2005, pp. 193–209.

[Dij75] Edsger W. Dijkstra. "Guarded commands, nondeterminacy and formal derivation of programs". In: *Communications of the ACM* 18 (8 1975), pp. 453–457.

[DJP10] Delphine Demange, Thomas Jensen, and David Pichardie. "A Provably Correct Stackless Intermediate Representation for Java Bytecode". In: *8th Asian Symposium on Programming Languages and Systems (APLAS 2010)*. Vol. 6461. Lecture Notes in Computer Science. Springer-Verlag, Nov. 2010, pp. 97–113.

[Fen73] J.S. Fenton. "Information Protection Systems". PhD thesis. Cambridge, England: University of Cambridge, 1973.

[FG01] Riccardo Focardi and Roberto Gorrieri. “Classification of Security Properties (Part I: Information Flow)”. In: *International School on Foundations of Security Analysis and Design (FOSAD 2000)*. London, UK: Springer-Verlag, 2001, pp. 331–396.

[FLR77] Richard J. Feiertag, Karl N. Levitt, and Lawrence Robinson. “Proving multilevel security of a system design”. In: *SIGOPS Operating Systems Review* 11 (5 Nov. 1977), pp. 57–65.

[GB09] Robert Grabowski and Lennart Beringer. “Noninterference with Dynamic Security Domains and Policies”. In: *Advances in Computer Science - ASIAN 2009. Information Security and Privacy*. Vol. 5913. Lecture Notes in Computer Science. Springer-Verlag, 2009, pp. 54–68.

[GHL11] Robert Grabowski, Martin Hofmann, and Keqin Li. “Type-Based Enforcement of Secure Programming Guidelines – Code Injection Prevention at SAP”. In: *8th International Workshop on Formal Aspects of Security & Trust (FAST 2011)*. Lecture Notes in Computer Science 7140. Leuven, Belgium: Springer-Verlag, 2011, pp. 182–197.

[GM04] Roberto Giacobazzi and Isabella Mastroeni. “Abstract non-interference: parameterizing non-interference by abstract interpretation.” In: *31st ACM Symposium on Principles of Programming Languages (POPL 2004)*. Ed. by Neil D. Jones and Xavier Leroy. Venice, Italy: ACM Press, 2004, pp. 186–197.

[GM82] Joseph A. Goguen and José Meseguer. “Security Policies and Security Models”. In: *1982 Symposium on Security and Privacy (S&P ’82)*. IEEE Computer Society Press, 1982, pp. 11–20.

[Gra08] Robert Grabowski. “Noninterference for Mobile Code with Dynamic Security Domains”. In: *Second International Workshop on Proof-Carrying Code (PCC 2008)*. Pittsburgh, USA: Informatics Research Report 1256, University of Edinburgh, 2008.

[Gra11] Robert Grabowski. *Homepage of the author, with sources of the dsdtool implementation*. `http://www.tcs.ifi.lmu.de/~grabow/`. 2011.

[GS05] Samir Genaim and Fauto Spoto. “Information Flow Analysis for Java Bytecode”. In: *6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2005)*. Ed. by Radhia Cousot. Vol. 3385. Lecture Notes in Computer Science. Springer-Verlag, 2005, pp. 346–362.

[HJ06] Martin Hofmann and Steffen Jost. “Type-Based Amortised Heap-Space Analysis”. In: *16th European Symposium on Programming (ESOP 2006)*. Vol. 3924. Lecture Notes in Computer Science. Springer-Verlag, Mar. 2006, pp. 22–37.

[HM95] Robert Harper and Greg Morrisett. "Compiling polymorphism using intensional type analysis". In: *22th ACM Symposium on Principles of Programming Languages (POPL 1995)*. ACM Press, 1995, pp. 130–141.

[Hoa69] C. A. R. Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12 (10 Oct. 1969), pp. 576–580.

[HPR88] Susan Horwitz, Jan Prins, and Thomas Reps. "On the adequacy of program dependence graphs for representing programs". In: *15th ACM Symposium on Principles of programming languages (POPL 1988)*. New York, NY, USA: ACM, 1988, pp. 146–157.

[HR98] Nevin Heintze and Jon G. Riecke. "The SLam calculus: programming with secrecy and integrity". In: *25th ACM Symposium on Principles of Programming Languages (POPL 1998)*. Ed. by ACM. New York, NY, USA: ACM Press, 1998, pp. 365–377.

[HS06] Sebastian Hunt and David Sands. "On flow-sensitive security types." In: *33rd ACM Symposium on Principles of Programming Languages (POPL 2006)*. Charleston, South Carolina, USA: ACM Press, 2006, pp. 79–90.

[HS09] Christian Hammer and Gregor Snelting. "Flow-Sensitive, Context-Sensitive, and Object-Sensitive Information Flow Control Based on Program Dependence Graphs". In: *International Journal of Information Security* 8.6 (Dec. 2009), pp. 399–422.

[HS11] Sebastian Hunt and David Sands. "From Exponential to Polynomial-time Security Typing via Principal Types". In: *20th European Symposium on Programming (ESOP 2011)*. Lecture Notes in Computer Science 6602. Springer-Verlag, 2011.

[IPW99] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. "Featherweight Java: A Minimal Core Calculus for Java and GJ". In: *1999 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1999)*. ACM, 1999.

[Jür02] Jan Jürjens. "UMLsec: Extending UML for Secure Systems Development". In: *International Conference on The Unified Modeling Language*. Ed. by J.-M. Jézéquel, H. Hußmann, and S. Cook. Vol. 2460. Lecture Notes in Computer Science. Dresden, Germany: Springer-Verlag, Sept. 2002, pp. 412 –425.

[KS02] Naoki Kobayashi and Keita Shirane. "Type-Based Information Flow Analysis for Low-Level Languages". In: *3rd Asian Workshop on Programming Languages and Systems (APLAS 2002)*. 2002, pp. 2–21.

[Ler02] Xavier Leroy. "Bytecode verification on Java smart cards". In: *Software Practice and Experience* 32 (4 Apr. 2002), pp. 319–340.

[LZ05] Peng Li and Steve Zdancewic. “Downgrading policies and relaxed noninterference”. In: *32nd ACM Symposium on Principles of Programming Languages (POPL 2005)*. Long Beach, California, USA: ACM Press, 2005, pp. 158–170.

[Man03] Heiko Mantel. “A Uniform Framework for the Formal Specification and Verification of Information Flow Security”. PhD thesis. Saarbrücken, Germany: Universität des Saarlandes, July 2003.

[MCB05] Ricardo Medel, Adriana Compagnoni, and Eduardo Bonelli. “A Typed Assembly Language for Non-interference”. In: *Theoretical Computer Science*. Ed. by Mario Coppo, Elena Lodi, and G. Pinna. Vol. 3701. Lecture Notes in Computer Science. Springer-Verlag, 2005, pp. 360–374.

[Mor+99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. “From System F to Typed Assembly Language”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21 (May 1999), pp. 527–568.

[MR07] Heiko Mantel and Alexander Reinhard. “Controlling the what and where of declassification in language-based security”. In: *16th European conference on Programming (ESOP 2007)*. Braga, Portugal: Springer-Verlag, 2007, pp. 141–156.

[MS01] Heiko Mantel and Andrei Sabelfeld. “A Generic Approach to the Security of Multi-Threaded Programs”. In: *14th IEEE Computer Security Foundations Workshop (CSFW 2001)*. IEEE Computer Society Press, 2001, pp. 200–214.

[MS04] Heiko Mantel and David Sands. “Controlled Declassification based on Intransitive Noninterference”. In: *2nd Asian Symposium on Programming Languages and Systems (APLAS 2004)*. Vol. 3302. Lecture Notes in Computer Science. Springer-Verlag, Nov. 2004, pp. 129–145.

[MSS11] Heiko Mantel, David Sands, and Henning Sudbrock. “Assumptions and Guarantees for Compositional Noninterference”. In: *24th IEEE Computer Security Foundations Symposium (CSF 2011)*. Cernay-la-Ville, France: IEEE Computer Society, 2011, pp. 218–232.

[MSZ04] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. “Enforcing Robust Declassification.” In: *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004)*. Pacific Grove, CA, USA: IEEE Computer Society, 2004, pp. 172–186.

[Mye99] Andrew C. Myers. “JFlow: Practical Mostly-Static Information Flow Control.” In: *26th ACM Symposium on Principles of Programming Languages (POPL 1999)*. ACM Press, 1999, pp. 228–241.

[Nec97] George C. Necula. “Proof-carrying code”. In: *24th ACM Symposium on Principles of Programming Languages (POPL 1997)*. Paris, France: ACM Press, 1997, pp. 106–119.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PS02] François Pottier and Vincent Simonet. “Information flow inference for ML”. In: *29th ACM Symposium on Principles of Programming Languages (POPL 2002)*. Portland, January 16-18, 2002: ACM Press, 2002, pp. 319–330.

[Rey02] John C. Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures”. In: *17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*. IEEE Computer Society, 2002, pp. 55–74.

[Rus92] John Rushby. *Noninterference, Transitivity, and Channel-Control Security Policies*. Technical Report CSL-92-02. SRI International, Dec. 1992.

[SEL09] National Security Agency. *Security-Enhanced Linux*. Available on the website `http://www.nsa.gov/research/selinux/index.shtml`. 2009.

[Sim03] Vincent Simonet. *The Flow Caml System: documentation and user's manual*. Technical Report 0282. Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003.

[SM02] P. K. Shukla and A. A. Mamun. *Introduction to Dusty Plasma Physics*. Vol. 44. 3. Taylor & Francis, 2002.

[SM03] Andrei Sabelfeld and Andrew C. Myers. “Language-Based Information-Flow Security”. In: *IEEE Journal on Selected Areas in Communications – special issue on Formal Methods for Security* 21.1 (Jan. 2003), pp. 5 –19.

[SS00] Andrei Sabelfeld and David Sands. “Probabilistic Noninterference for Multi-Threaded Programs”. In: *13th IEEE Workshop on Computer Security Foundations (CSFW 2000)*. IEEE Computer Society, 2000.

[SS01] Andrei Sabelfeld and David Sands. “A Per Model of Secure Information Flow in Sequential Programs.” In: *Higher-Order and Symbolic Computation* 14.1 (2001), pp. 59–91.

[SS05] Andrei Sabelfeld and David Sands. “Dimensions and Principles of Declassification”. In: *18th IEEE Computer Security Foundations Workshop (CSFW 2005)*. Aix-en-Provence, June 20 - 22, 2005: IEEE Computer Society, 2005, pp. 255–269.

[SST07] Paritosh Shroff, Scott Smith, and Mark Thober. “Dynamic Dependency Monitoring to Secure Information Flow”. In: *20th IEEE Computer Security Foundations Symposium (CSF 2007)*. IEEE Computer Society, 2007, pp. 203–217.

[TA05] Tachio Terauchi and Alexander Aiken. “Secure Information Flow as a Safety Problem”. In: *12th International Symposium on Static Analysis (SAS 2005)*. Ed. by Chris Hankin and Igor Siveroni. Vol. 3672. Lecture Notes in Computer Science. Springer-Verlag, 2005, pp. 352–367.

[TT97] Mads Tofte and Jean-Pierre Talpin. “Region-based memory management”. In: *Information and Computation* 132 (2 Feb. 1997), pp. 109–176.

[TZ06] Stephen Tse and Steve Zdancewic. *Fjavac: a functional Java compiler*. `http://www.cis.upenn.edu/~stevez/stse-work/javac/`. 2006.

[Vac+04] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. “RIFLE: An architectural framework for user-centric information-flow security”. In: *37th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 37)*. IEEE Computer Society, 2004, pp. 243–254.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. “A Sound Type System for Secure Flow Analysis”. In: *Journal of Computer Security* 4.3 (1996), pp. 167–187.

[Yan07] Hongseok Yang. “Relational separation logic”. In: *Theoretical Computer Science* 375.1-3 (2007), pp. 308–334.

[ZM07] Lantian Zheng and Andrew C. Myers. “Dynamic security labels and static information flow control”. In: *International Journal of Information Security* 6.2 (2007), pp. 67–84.