Static Analysis for Java Security

Candidato
Massimo Bartoletti

Relatori
Prof. Pierpaolo Degano
Prof. Gian-Luigi Ferrari

Controrelatore
Prof. Laura Ricci

Anno Accademico 2000/2001
Contents

1 Introduction 5

2 Java security 9
   2.1 The class loader .......................................... 10
   2.2 The bytecode verifier .................................... 11
   2.3 The security manager .................................... 11
   2.4 An e-commerce example ................................. 13
   2.5 Related work .............................................. 16

3 Program model 23
   3.1 Syntax .................................................. 23
   3.2 Semantics ............................................... 28
   3.3 The access control policy ............................... 29
   3.4 Basic properties of control flow graphs ............... 32
   3.5 Abstract paths ........................................ 34
   3.6 Concrete paths ......................................... 44
   3.7 Valid paths ............................................. 57
   3.8 Traversable paths ..................................... 64

4 Data Flow Analysis 69
   4.1 Basic definitions ....................................... 70
   4.2 Data flow frameworks ................................. 72
   4.3 Solutions and their properties ......................... 74
   4.4 The Worklist-Iteration algorithm ....................... 79
5 Static analyses 81
5.1 The Denied Permissions Analysis 82
5.2 The Granted Permissions Analysis 84
5.3 The DP0 Analysis 86
5.4 The GP0 Analysis 95
5.5 The DP1 Analysis 99
5.6 The GP1 Analysis 109
5.7 The DP2 Analysis 115
5.8 The GP2 Analysis 125
5.9 Optimized stack inspection 131

6 Conclusions 133

A An e-commerce example 135

Index of symbols 141

Bibliography 143
Chapter 1

Introduction

A main innovation of the Java platform concerns its approach to security: the language comes equipped with constructs and mechanisms for expressing and enforcing security policies. Since the code actually executed is on the form of an intermediate object-oriented language – the bytecode – *bytecode verification* is the basic building block of Java security.

Over the past few years, there has been considerable effort in developing formal models of the Java bytecode verifier. Some authors showed that the problem of bytecode verification can be formally understood and described at static time using type systems [SA98, FM98, FM99a]. All the proposals are proved to enjoy the type soundness properties (on the bytecode fragments they consider). Also, the type inference algorithm can be turned into a correct bytecode verifier, see e.g. [CGQ98, Nip01].

Another crucial aspect of the Java security architecture is the dynamic check of the permissions granted to running code. Roughly, one has to make sure that whenever a principal invokes a certain method, it has the rights to. At run-time, the security policy is enforced by *stack inspection*: a permission is granted, provided that it belongs to *all* principals on the call stack. An exception are the so-called *privileged operations*, which are allowed to execute any code granted to their principal, regardless of the calling sequence.

Since the analysis of stack frames may be expensive, the run-time overhead due to stack inspection may grow very high: effective techniques which improve and optimize stack inspection are therefore in order.
In this thesis we develop a static analysis which improves run-time checking of permissions. We reduce the number of frames to be examined, while maintaining the same accuracy of the plain stack inspection algorithm. Also, our analysis may be used for optimizing bytecode, by moving checks where they are actually needed, and by removing redundant ones.

Our approach is based on Data Flow Analysis, a static technique for predicting safe and computable approximations of the set of values that the objects of a program may assume during its execution. These approximations are then used to analyze properties of programs in a safe manner: if a property holds at static time, then it will always hold at run-time. The vice-versa may not be true: the analysis may “err on the safe side”.

Our main technical contribution is the formulation of two families of data flow analyses with increasing accuracy. We call them Granted Permissions (GP) Analyses and Denied Permissions (DP) Analyses, respectively. The analyses are specified over an abstract representation of Java programs proposed in [JLT98b], which specializes the usual control flow graph by adding information about security checks, privileged operations and protection domains.

Control flow graphs are given an operational semantics: essentially, the states that a program can pass through are represented by stacks $\sigma$, made of nodes of the graph, each interpreted as an abstraction of the actual stack frames. The control point is the top $n$ of the stack $\sigma : n$, and a computation step is represented by a transition between stacks, written as $\sigma \triangleright \sigma'$. The operational semantics incorporates a specification of the Java stack inspection mechanism: this is done through a predicate $\text{JDK}$. The predicate is true for a state $\sigma$ and a permission $P$, when $P$ is granted to $\sigma$ according to the security policy: in this case, we write $\sigma \trianglerightJDK(P)$. Otherwise, if $P$ is denied to $\sigma$, the predicate is false, and we write $\sigma \ntriangleright JDK(P)$.

For each node $n$, a GP-analysis computes an approximation $\gamma(n)$ of those permissions that are granted to $n$ in any execution leading to it. Similarly, a DP-analysis computes an approximation $\delta(n)$ of the permissions denied to $n$ in every run leading to it. Both analyses are correct with respect to the operational semantics. Suppose that $n$ models a security check of permission
P, and that P ∈ γ(n) (resp. P ∈ δ(n)). Then, whenever there is a derivation
\[ [] \triangleright \cdots \triangleright \sigma : n, \] it always happens \( \sigma : n \vdash \text{JDK}(P) \) (resp. \( \sigma : n \not\vdash \text{JDK}(P) \)).

The approximations computed by our analyses are then used to reduce the
depth at which the stack inspection algorithm stops. When checking privileges
against a permission P, it suffices to reach a frame m such that P ∈ γ(m) or
P ∈ δ(m). In the first case the check succeeds, while in the second one an
AccessControlException is raised.

This thesis is organized as follows:

- in chapter 2 we review the Java security architecture, and we report some
  of the work done towards a better understanding of this model. Also,
  different security-aware frameworks are described, and their properties
  are compared to those featured by Java.

- in chapter 3 we introduce our program model, with its syntax and op-
  erational semantics. Through this chapter, we see how the translation
  from Java bytecode gives rise to particular contraints on the structure
  of control flow graphs. When a control flow graph respects these con-
  straints, its operational semantics enjoys some properties, needed for our
  subsequent formal development. A large portion of this chapter is then
devoted to the proof of these characterizing properties.

- in chapter 4 we give a brief overview of Data Flow Analysis. Our ap-
  proach slightly deviates from the standard one, because it is intended to
  fit with our program model and with the analyses we will build upon it.

- in chapter 5 we present our static analyses: for each of them, we prove a
  correctness result, and we show that an effectively computable solution
  always exists. The analyses are ordered according to their degree of ac-
  curacy. Furthermore, we show how the stack inspection algorithm can be
  optimized once the static information about granted/denied permissions
  has been computed.

- finally, chapter 6 contains some concluding remarks, and propose exten-
  sions to our analyses, in order to make them more effective in practice.
Chapter 2

Java security

Code mobility presents one of the major challenges in computer security. Java accepts the challenge by providing a customizable environment, called the sandbox, in which untrusted programs are placed: the sandbox prevents untrusted code from performing security-critical operations, according to a customizable fine-grained policy.

This chapter gives a brief overview of the Java security model. For more detailed information, we refer the reader to one of the several books that address this topic, e.g. [MF99, Gon99, Oak01].

A comprehensive example of a security-aware application is provided: it will be used throughout the thesis to test the effectiveness of our static analyses.

Finally, we review some of the work on the formalization of Java and its security model. We also present some extensions to the model.
2.1 The class loader

Since Java advocates code mobility, it is equipped with a class loading mechanism, which brings bytecode from the outside into the JVM. Code may come from different places, either trusted ones (e.g. files on the local disk) or untrusted ones (e.g. applets downloaded across the network). Therefore, the class loader architecture is the first line of defense against security attacks.

Class loaders are organized hierarchically. The bootstrap class loader is responsible for loading all the classes that are needed to start the JVM. The other classes (e.g. libraries in the classpath, downloaded applets, etc.) are loaded by user-defined class loaders. Class loaders form then a tree, having the bootstrap class loader as root.

Each class loader caches all classes it loads over time. When an user-defined class loader is asked for a class, it first checks if the class is in the cache: if so, the class is returned directly. Otherwise, the class loader pass the request to its parent. This process is repeated until the bootstrap class loader is reached. If no class loader up in the hierarchy has loaded the requested class before, the class loader it was first asked for finally attempts to load the class.

This delegation model prevents “class name spoofing” attacks, in which a malicious class loader replaces trusted classes with hostile ones.

Each class loader L, defines a name space, containing the names of all the classes that have been loaded by L. Once a class named C has been loaded by L, it is impossible for another class named C to be placed into the same name space. Moreover, classes belonging to different name spaces cannot interact: indeed, they cannot even detect each other’s presence, unless class loaders provide explicit mechanisms to let them doing so.

Name space separation not only relieves applet programmers from having to worry about name collisions, but also constitutes a main contribution to the security (and privacy, too) of the Java platform.

Another task is attained by class loaders: they place each loaded class into a protection domain. This information will be used by the security manager to enforce the access control policy.
2.2 The bytecode verifier

Once a class has been loaded, it has to be linked to the rest of the system before it can be executed. However, in order to preserve security, a verification step is performed on the loaded class: this is done by statically analyzing the bytecode, to guarantee that it satisfies some safety properties, e.g.:

- the loaded class file has the correct format;
- the loaded class has a non-final superclass;
- methods are invoked with the correct number and types of arguments;
- objects and variables are initialized before they are used;
- access to classes, methods and variables is done according to their respective access modifiers;
- the operand stack neither overflows nor underflows;
- the class preserves binary compatibility with the classes it refers to.

While any trusted Java compiler produces bytecode that satisfies these properties, there are several reasons that suggest a deferred verification phase:

- bytecode can be corrupted during network transmission;
- bytecode can be generated on-spot;
- bytecode can be written by hand;
- bytecode can be generated by an hostile compiler.

2.3 The security manager

While both the class loader and the bytecode verifier are mainly concerned with the safety facet of security, the security manager more directly address the problem of protecting critical resources from leakage and tampering threats.
A critical resource is protected by ensuring that no code can invoke its accessor methods without having the rights to. This is done by placing calls to the AccessController’s checkPermission() method before the accessor methods of the critical resource (for an example, see section 2.4 in this chapter).

A permission is an access right on a resource. The set of permissions granted to code is specified by a security policy. Code can be attributed different degrees of trust, according to its origin and digital signature. A policy is specified by means of a set of grant clauses of the form:

```java
grant signedBy signer_1, ..., signer_k codeBase origin {
    permission name_1 target_1, action_1
    ...
    permission name_n target_n, action_n
}
```

Each clause in the security policy defines a protection domain: by the foregoing definitions, it turns out that a protection domain is just a mapping from classes to sets of permissions. As mentioned above, it is responsibility of the class loader to place each class into the appropriate protection domain (according to the security policy).

The security policy is enforced by the security manager each time a checkPermission() is invoked. The security manager decides whether granting access to the protected resource or not by performing stack inspection.

The stack inspection algorithm, shown in Fig 2.1, is based on the execution context (i.e. the sequence of method invocations). Each method in the sequence belongs to a class, which in turn belongs to a protection domain. Basically, the stack inspection algorithm grants access to a resource if all protection domains in the current execution context have the required permission.

This strategy is slightly complicated by the presence of privileged code. When a method is executed in privileged mode, it exploits all of its permissions, regardless of the other methods in the execution context.

This is useful in client-server systems, where the server runs a security-critical resource that the client can only access through the interface provided by the server.
2.4. AN E-COMMERCE EXAMPLE

```java
void checkPermission(Permission P)
    throws AccessControlException {

    for each frame in the current call stack, 
        starting from the topmost {

        if permission P is not granted to the frame 
            throw an AccessControlException;
        if the frame is privileged 
            return;
    }
}
```

Table 2.1: The stack inspection algorithm.

2.4 An e-commerce example

Throughout this thesis we will make use of an example that models a small e-commerce application. Its UML class diagram if shown in Fig. 2.1.

The `ControlledVar` class implements a controlled integer variable. This means that not only the integer variable is properly encapsulated (i.e. it is only accessible through the public interface), but also its methods are protected by security checks, which ensure that only principals having appropriate permissions can access the variable.

We assume that `ControlledVar` is a standard extension, i.e. it is installed in the directory `lib/ext/` under the Java home. By default, all permissions are granted to standard extensions.

The `BankAccount` class uses the controlled variable `balance` to implement a simple account manager. One boolean query and three transactions are provided by the public interface: each of these methods is protected by an appropriate security check. Observe that, once a subject has gained access to one of the methods of the account manager, `balance` is accessed in privileged mode. Of course, we assume that the permissions to read and write the controlled variable are granted to `BankAccount`. 
public class ControlledVar {
    private int var;
    ...
    public void write(int newValue) {
        AccessController.checkPermission(write);
        var = newValue;
    }
    public int read() {
        AccessController.checkPermission(read);
        return var;
    }
}

Three clients aim at exploiting the services offered by the account manager: of these, two (Spender and Saver) are regarded as trusted clients, while the other one (Robber) is considered untrusted, and no permissions are granted to it. However, only Saver has enough permissions to accomplish its task (i.e. depositing money into the account). A Spender can perform its transactions (i.e. drawing on the account) as long as it has enough cash; as it tries to ask
public class BankAccount {
    private ControlledVar balance;
    ...
    public boolean canpay(final int amount) {
        AccessController.checkPermission(canpay);
        Object res = AccessController.doPrivileged(new PrivilegedAction() {
            public Object run() {
                return new Boolean(balance.read() > amount);
            }
        });
        return ((Boolean) res).booleanValue();
    }
    public void debit(final int amount) {
        AccessController.checkPermission(debit);
        if (this.canpay(amount)) {
            AccessController.doPrivileged(new PrivilegedAction() {
                public Object run() {
                    balance.write(balance.read() - amount);
                    return null;
                }
            });
        }
    }
    public void credit(final int amount) {
        AccessController.checkPermission(credit);
        AccessController.doPrivileged(new PrivilegedAction() {
            public Object run() {
                balance.write(balance.read() + amount);
                return null;
            }
        });
    }
    public void loan(final int amount) {
        AccessController.checkPermission(loan);
        credit(amount);
    }
}

for a loan, the access controller raises a security exception, because Spender is not granted the required loan permission.

Table 2.2 shows the relation between classes, protection domains, and permissions: this relation actually defines the security policy for the e-commerce application. The complete Java sources for the example, together with an appropriate policy file, are contained in appendix A.
public class Spender extends Client {
...  
  public void transact() {
    if (account.canPay(AMOUNT)) account.debit(AMOUNT);
    else account.loan(AMOUNT);
  }
}

public class Saver extends Client {
...  
  public void transact() {
    account.credit(AMOUNT);
  }
}

public class Robber extends Client {
...  
  public void transact() {
    account.loan(AMOUNT);
    account.debit(AMOUNT);
  }
}

<table>
<thead>
<tr>
<th>Class</th>
<th>Protection Domain</th>
<th>Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spender</td>
<td>Client</td>
<td>canPay, credit, debit</td>
</tr>
<tr>
<td>Saver</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robber</td>
<td>Unknown</td>
<td></td>
</tr>
</tbody>
</table>
| BankAccount    | Bank              | canPay, credit, debit, loan,  
|                |                   |  read, write         |
| ControlledVar  | System            | AllPermission       |
| Ecommerce      |                   |                     |

Table 2.2: Security policy for the e-commerce application.

2.5 Related work

Bytecode verification

Although bytecode verification is one the critical parts of the Java security architecture, only an informal specification [LY96] and a reference implementation of the verifier are supplied by Sun. Then, a lot of research has been done in order to provide a formal description of the Java bytecode verifier.

Stata and Abadi [SA98] take a minimal subset of the JVM language, including the “jump” and “return” instructions for JVM subroutines. Then,
they define a dynamic semantics for the language, and a type system which is proved to be sound with respect to the semantics. The same goal is reached by Hagiya and Tozawa [HT98] by means of a different type system, which enables a simpler proof of type soundness.

Freund and Mitchell extend the work of Stata and Abadi by adding constructors and object initialisation [FM98], exception handling [FM99b], and, finally, an almost-complete subset of the JVM language including classes, interfaces, methods and arrays [FM99a]. Each of these fragments of the JVM language is proved to enjoy the type soundness property. Moreover, with the insight gained by carrying out the soundness proofs, they show a bug in the Sun's implementation of the bytecode verifier: this strengthen the opinion that a formal specification of the bytecode verifier is needed in order to make the Java language secure.

Bigiardi and Laneve [BL00] give a further extension of the work of Freund and Mitchell, taking into account thread creation and the synchronization primitives. They prove that, for the pieces of bytecode accepted by the verifier, any critical section is executed in mutual exclusion.

Posegga and Vogt [PV98] show how bytecode verification can be reduced to a model checking problem. The semantics of bytecode programs is given by employing the formalism of Abstract State Machines (ASM). Then, it is shown how ASM rules can be translated into propositional formulae, which are directly amenable to the SMV model checker.

Coglio, Goldberg and Qian [CGQ98] specify bytecode verification as a data flow analysis problem. Then, the Specware system is used in order to convert their formalization into a provably-correct implementation of the verifier.

Nipkow [Nip01] develops an abstract framework to transform the type inference algorithm of bytecode verification into a correct bytecode verifier. This task is accomplished by using the Isabelle/HOL theorem prover.

Dean [Dea97, Dea99] models dynamic linking in PVS. Then, he proves that dynamic linking does not interfere with static type checking.

Jensen, Le Métayer and Thorn [JLT98a] and Qian, Goldberg and Coglio [QGC00] provide formal specifications of Java class loading. Jensen et al. show
how the type confusion bug found by Saraswat [Sar97] can be derived in their model. Qian et al. prove that their specification enjoys type safety.

Fong and Cameron [FC98, FC99, FC00] propose a framework called proof linking, that supports the distributed verification of Java bytecode. They state sufficient condition under which their implementation of proof linking is consistent with standard bytecode verification.

Stack inspection

Many authors advocated the use of static techniques in order to optimize the check of security properties.

Jensen, Le Métayer and Thorn [JLT98b] make one of the first attempts to apply static techniques to the verification of global security properties. They formalize classes of security properties through a linear time temporal logic. They show that a large class of policies (including Java stack inspection) can be expressed in this formalism, while more sophisticated ones (like the Chinese Wall policy) are not. Model checking is then used to prove that local security checks, inserted at certain points in the code, enforce a given global security policy. Under the additional hypothesis that programs do not contain mutual recursion, they prove that it is possible to perform the verification of a global security property in a finite number of steps. The problem with mutual recursion has been fixed in a subsequent paper by the same authors and Besson [BJLT01]. In this paper, they also propose a model checking technique for the verification of global security properties, based on the translation from linear-time temporal formulae to deterministic finite-state automata. The soundness and completeness of the method are proved, and an example shows how it can be applied to the analysis of Java programs.

Nitta, Takata and Seki [NTS01] address the same problem of Jensen’s, except that they use regular languages instead of temporal logic to express security properties. This formalization is more powerful than Jensen’s, because the class of regular languages strictly contains the class of languages generated by linear-time temporal formulae. Moreover, they show that the verification problem is decidable also for problems with mutual recursion.
2.5. RELATED WORK

The problem with these approaches is that, given an arbitrary program, it seems hard to mechanically guess a “suitable” global security property for that program, i.e. a property that, if enforced, would guarantee safe execution.

Wallach, Appel and Felten [WF98, Wal99, WAF00] formalize stack inspection by exploiting the ABLP belief logic presented in [ABLP93]. They propose a technique called security-passing style, which allows standard optimizations, such as method inlining and elimination of tail recursion, to be performed without interfering with the stack inspection mechanism. They suggest some optimizations to reduce the overhead due to security-passing style.

Pottier, Skalka and Smith [SS00, PSS01] model the Java security architecture by a λ-calculus, whose operational semantics incorporates a specification of Java stack inspection. Then, they show how the security-passing style proposed by Wallach and Felten gives rise to a type system, which can statically predict the outcome of some of the access control decisions contained in a term. Observe that the last two approaches implicitly characterize the checks that are redundant, while our data flow analyses will effectively compute them.

Fourner and Gordon [FG01] investigate the limitations of stack inspection in the abstract setting introduced by [PSS01]. They provide examples of how the interaction between trusted and untrusted code may give rise to security breaches. Moreover, they show how stack inspection affects standard program transformations, like function inlining and tail-call optimizations.

Extensions to the Java security model

Metha and Sollins [MS98] define a constraint language that makes possible to express history-based policies, like the Chinese Wall policy (i.e. “an applet cannot connect to the network after it has read a protected file”). A policy is a set of conditional rules, whose meaning can depend on past actions: for example, an applet can be labelled as “suspicious” after it has read a protected file; on the other hand, another rule may establish that only applets labelled as “trusted” can gain access to the network. A logging facility keeps track of all potentially dangerous actions performed by applets; this information is then used by the extended security manager in order to enforce the given policy.
Hauswirth, Kerer and Kurmanowytsch [HKK00] present the Java Secure Execution Framework (JSEF), which allows the definition of subtractive permissions, an hierarchical organization of security policies and the interactive negotiation of permissions.

Walker [Wal00] propose an approach which is orthogonal to Jensen’s [JLT98b]. When a security-unaware program is compiled, a centralized security policy dictates where to insert run-time checks, in order to obtain a provably-secure compiled code. An optimization phase follows: whenever a security check is removed, it is replaced by a proof that the optimized code is still safe. This is done by means of typed compilation schemata: types encode assertions about program security, ensuring that no run-time violation of the security properties will occur. Finally, a correct verification software ensures that any code obeys the centralized security policy, before it can be executed. Observe that, in this way, compilers are not required to be trusted. Security properties are specified by security automata [Sch98, ES99]. Schneider claims that this mechanism can enforce any safety property, by inserting appropriate run-time checks. However, other properties not involving safety (e.g. information flow, resource availability, liveness, performance) are not expressible by security automata.

Karjoth, Lange and Oshima [KLO97] present a security model for aglets, a Java-based paradigm for distributed computation. Mobile agents introduce new challenges in computer security, because both the aglets and their execution environment are mutually exposed to security threats.

Several work has also been done with the purpose of thwarting denial of service attacks, like e.g. resource stealing and antagonism. However, all of these approaches rely on empirical tecniques: in fact, it seems hard to formally capture the idea of “denial of service attack”, because, for example, there is no way to distinguish between an hostile applet trying to allocate a lot of memory and an image-processing applet trying to make useful job.

Shin and Mitchell [SM98] introduce bytecode modification, a tecneque that instantiates to Java applets the more general approach of software fault isolation by Lucco et al. [WLAG93]. Before an untrusted applet is executed by the Java virtual machine, a proxy server modifies the received bytecode in order
to perform some monitoring of security-critical resources. In this way, several annoyance attacks (e.g. window consuming attacks, e-mail forging, URL spoofing, annoying sound attacks) can be prevented.

Acharya, Ranganathan and Saltz [ARS97] implement Sumatra, an extension of Java that supports resource-aware mobile programs. A distributed resource monitor (called Komodo) provides information about the availability of given critical resources (e.g. network latency, network bandwidth, CPU cycles), and mobile Sumatra programs then react accordingly.

The Java Authentication and Authorization Service (JAAS), presented in [LGK+99], extends the Java security model by enabling more sophisticated access control policies, based on the principal who actually runs the code.
Chapter 3

Program model

3.1 Syntax

Since, in the Java security model, access control decisions are made by looking solely into the call stack, we can base our analyses on an abstraction of the bytecode language where only security checks and control flow are taken into account. Thus, we model a bytecode program as an oriented graph, whose structure is specified by the following definition:

Definition 3.1.1. A control flow graph is a triple $G = \langle N, E, N_{entry} \rangle$, where:

- $N$ is the set of nodes, including a distinguished element $\perp_N$. Each node $n \in N \setminus \{\perp_N\}$ is associated with a label $\ell(n)$, describing the control flow primitive represented by the node. Labels give rise to three kinds of nodes: call nodes, representing method invocation, return nodes, which represent return from a method, and check nodes, which enforce the access control policy. We can think of a node labelled check(P) as having the same meaning of an AccessController.checkPermission(P) instruction in the Java language. The distinguished node $\perp_N$ plays the technical role of a single, isolated entry point.

- $E = E_{call} \uplus E_{trans} \uplus E_{entry} \subseteq N \times (N \setminus \{\perp_N\})$ is the set of edges. Edges are split into call edges $n \rightarrow n' \in E_{call}$, modelling interprocedural flow, and transfer edges $n \rightarrowrightarrow n' \in E_{trans}$, which instead correspond to intra-procedural flow. Moreover, we have the set of entry edges $\bullet \rightarrow n \in E_{entry}$,
containing all pairs \((\bot_N, n)\) for \(n \in N_{\text{entry}}\). The \(\bot_N\) element only appears in entry edges. Since the flow from a return node \(m\) to next instruction \(n'\) of the respective call node \(n\) is not explicitly represented by an element \((m, n')\) \(\in E\), we indicate \(E\) as the set of abstract edges (cf. def. 3.5.17).

- \(N_{\text{entry}} \subseteq N \setminus \{\bot_N\}\) is the non-empty set of entry nodes. We assume that a program may have many entry points, as it actually happens with programs designed to be launched both as applets and as stand-alone applications.

In what follows, we assume that all the information above is extracted from the bytecode, e.g. by the Control Flow Analysis techniques presented in [GDDC97, GHM00, Muc97, NNH99, SHR+00, TP00]. Unfortunately, these analyses are always approximated for object-oriented languages with dynamic resolution of method invocations. In Java, for example, when a program invokes an instance method on an object, the virtual machine may have to choose among various implementations of that method. The decision is not based on the declared type of the object, but on the actual class the object belongs to: since this is statically undecidable, Control Flow Analyses over-approximate the set of methods that can be invoked at each program point. This approximation is safe, in the sense that any actual execution flow is represented by a path in the control flow graph. However, the converse may not be true: some paths may exist which do not correspond to any actual execution. This is a main source of approximation for the static analyses built over control flow graphs. Another source of approximation is the fact that control flow graphs hide any data flow information.

**Example 3.1.2.** The control flow graph extracted from the Java program for the e-commerce application is shown in figure 3.1. Observe that the conditional constructs are modelled by non-determinism in the interprocedural flow. Circled nodes represent calls that enable their privileges (i.e. each call within the `run()` method of a `PrivilegedAction` object). Solid boxes enclose nodes belonging to the same methods, while dashed boxes enclose methods belonging to the same classes.
Figure 3.1: Control flow graph for the e-commerce application.
Throughout this chapter, we will put constraints on the structure of control flow graphs, so that they more appropriately reflect some peculiarities of the Java bytecode. When a control flow graph will satisfies each of the given constraints, we say that it is well-formed. Now, whenever one of these constraints is stated, it will be clear that, for any admissible Java program, the constraint will be automatically satisfied by the derived control flow graph. Therefore, in what follows we shall always assume that control flow graphs are well-formed.

**Well-formedness constraint 1.** Edges do not overlap. This means that, if a call edge connects two nodes \( n \) and \( n' \), then no transfer edge may exist between \( n \) and \( n' \) (with the same direction), and vice versa. Formally:

\[
E_{\text{call}} \cap E_{\text{trans}} = \emptyset
\]

This constraint is always satisfied by control flow graphs derived from actual bytecode: in fact, the instruction which follows a method invocation cannot be the first instruction of that method.

**Well-formedness constraint 2.** Each caller has a callee, i.e.:

\[
\ell(n) = \text{call} \implies \exists n' \in N. n \rightarrow n'
\]

Since we assume that the whole program is available prior the construction of its control flow graph, this constraint is clearly satisfied. If we were dealing with dynamically linkable control flow graphs, this constraint should be relaxed.

**Well-formedness constraint 3.** Checks do not admit outgoing calls, i.e.:

\[
\ell(n) = \text{check}(P) \land (n,n') \in E \implies n \rightarrow n'
\]

This implies that, if a call stack contains a check node, then it is the topmost node. This constraint is always satisfied, because control flow graphs model all the code inside a `checkPermission` method as a single check node.
Well-formedness constraint 4. Return nodes have not outgoing edges, i.e.:

\[
\ell(n) = \text{return} \implies \neg \exists n' \in N. (n, n') \in E
\]

This implies that, if a call stack contains a return node, then it is the topmost node. Since return nodes model exit points of methods (e.g. the ireturn and areturn instructions) they cannot have outgoing call edges, because this kind of flow only originates from method invocations (e.g. the invokevirtual and invokespecial instructions). Return nodes cannot have outgoing transfer edges, either: in fact, transfer edges only model interprocedural flow. Then, this constraint is satisfied by any bytecode-derived control flow graph.
\[
\begin{align*}
\text{Table 3.1: Operational semantics of control flow graphs.}
\end{align*}
\]

\begin{align*}
\frac{n \in N_{\text{entry}}}{[[ \exists \sigma \exists [n]]} & [\triangleright_{\varnothing}] \\
\frac{\ell(n) = \text{call } n \rightarrow n'}{\sigma : n \triangleright \sigma : n'} & [\triangleright_{\text{call}}] \\
\frac{\ell(n) = \text{check}(P) \quad \sigma : n \vdash \text{JDK}(P) \quad n \rightarrow n'}{\sigma : n \triangleright \sigma : n'} & [\triangleright_{\text{check}}] \\
\frac{\ell(m) = \text{return } n \rightarrow n'}{\sigma : n : m \triangleright \sigma : n'} & [\triangleright_{\text{return}}]
\end{align*}

\section{Semantics}

The operational semantics of the language is defined by a transition system whose configurations are sequences of nodes, modelling call stacks. More formally, we define the set of states as the Kleene closure of \( N \setminus \{ \bot_N \} \), i.e.:

\[
\Sigma = (N \setminus \{ \bot_N \})^* \]

A state is represented as a sequence of nodes enclosed in square brackets: for example, \( \sigma = [n_0, \ldots, n_k] \) is a state whose top node is \( n_k \). Concatenation between states is implemented by an infix colon operator: for example, if \( \sigma \in \Sigma \) and \( n \in N \setminus \{ \bot_N \} \), then \( \sigma : n \) is the concatenation between \( \sigma \) and \( [n] \).

The transition relation \( \triangleright \subseteq \Sigma \times \Sigma \) is defined in Table 3.1 (the JDK predicate is defined in Table 3.3). We write \( \sigma \triangleright \sigma' \) when the execution can lead from \( \sigma \) to \( \sigma' \) in one step. For the purposes of our analyses, we define a reachability relation \( \vdash \) stating when the execution of program \( G \) can lead to a given state:

\[
\frac{G \vdash \sigma \quad \sigma \triangleright \sigma'}{G \vdash \sigma}
\]

We say that a state \( \sigma \) is \( G \)-reachable iff \( G \vdash \sigma \). We say that a node \( n \) is \( G \)-reachable iff \( G \vdash \sigma : n \) for some state \( \sigma \).
3.3 The access control policy

In order to give a specification of the access control policy being consistent with the one introduced in chapter 2, we endow each node \( n \in \mathbb{N} \setminus \{ \bot \} \) with the following additional information:

- \( \text{Perm}(n) \), the set of permissions associated with \( n \).
- \( \text{Priv}(n) \), a boolean predicate telling whether \( n \) represents privileged code.

The mapping between nodes and permissions for the e-commerce application in Fig. 3.1 is illustrated in Table 3.2, where we denote with \( \text{Permission} \) the set of permissions which actually occur in the control flow graph of interest:

\[
\text{Permission} = \bigcup_{n \in \mathbb{N}} \text{Perm}(n) \cup \bigcup_{n \in \mathbb{N}} \{ P \mid \ell(n) = \text{check}(P) \}
\]

We assume that permissions are drawn out from a finite set. Although the Java security model allows the dynamic instantiation of permissions (imagine an application that iteratively asks the user for a file name and then tries to open the corresponding file), we only consider the permissions that can be determined statically. This makes sense, because we are only interested in control flow graphs which are amenable to static analysis.

**Well-formedness constraint 5.** As described in chapter 2, the Java security model bounds permissions to whole protection domains: then, all nodes belonging to the same protection domain carry the same permission set. Since our model does not handle protection domains explicitly, we only require that nodes belonging to the same method have the same permissions, i.e.:

\[
n \rightarrow n' \implies \text{Perm}(n) = \text{Perm}(n')
\]
Well-formedness constraint 6. Only call nodes can be privileged, i.e.:

\[ Priv(n) \implies \ell(n) = \text{call} \]

In general, also security checks can be enclosed with privileged actions; however, privileged check nodes make little sense, because it is always possible, during the construction of the control flow graph, to determine whether a privileged check will succeed. Similarly, there is not point in enabling return nodes to be privileged, because a return node will never be on the call stack when stack inspection is performed.

In our formalization, we use a slightly simplified version of the full access control algorithm presented in chapter 2, as we let privileged frames to exploit all of their own permissions. The simplified algorithm, described in Fig. 3.2, performs a top-down scan of the call stack. Each frame in the stack refers to the protection domain containing the class to which the called method belongs. As soon as a frame is found whose protection domain has not the required permission, an AccessControlException is raised. The algorithm succeeds when a privileged frame is found that carries the required permission, or when all frames have been visited. We formally specify this behaviour by means of the inference rules in Table 3.3.

Algorithm 1: Stack inspection

\begin{verbatim}
CHECK-PERMISSION(P, σ)
1   while σ ≠ NIL do
2       n ← POP(σ)
3       if P /∈ Perm(n) then throw "access control exception"
4       if Priv(n) then return
\end{verbatim}

Figure 3.2: The stack inspection algorithm.
Table 3.3: Specification of the access control policy.

There are several differences between our model and the Java security model introduced in chapter 2, in addition to those mentioned above:

- in the Java security model, a permission $P$ may be granted to a piece of code, lying inside a protection domain $D$, even if $P$ does not belong to the permissions explicitly associated with $D$ (this may happen through the `implies()` method). Our model prevents this behaviour, because the JDK rules ensure that, for each state $\sigma$ and node $n$:

$P \notin \text{Perm}(n) \implies \sigma : n \not\vdash \text{JDK}(P)$

- since our inference rules for JDK are fixed, as well as those for the transition relation $\triangleright$, we are prevented from modelling permissions like `AllPermission` and `FilePermission("*", "write")`, as they may breach security by altering the Java system binaries.

- in our model, the mapping from nodes to permissions is static: once the security policy is fixed, it cannot change any more. On the contrary, the Java security model allows the policy to be updated via the `refresh()` method of the `Policy` object.
3.4 Basic properties of control flow graphs

Lemma 3.4.1. Let:

\[ \mathbb{0} = \sigma_0 \triangleright \sigma_1 \triangleright \cdots \triangleright \sigma_k = \sigma : n \]

be a derivation. Then:

\[ \exists i \in 0..k-1. \sigma_i = \sigma \]

Proof. We proceed by induction on the length of derivations. For the base case, we have

\( k = 1, \ n \in N_{\text{entry}} \) and \( \sigma = \mathbb{0} \): then \( \sigma_0 = \sigma \) by premises. For the inductive case, assume the lemma is true for all derivations of length lower than \( k \). By case analysis on transition \( \sigma_{k-1} \triangleright \sigma_k \), we have:

- case [call]:

  \[ \text{check}: \quad \quad \quad \quad \quad \]

  \[ \ell(n') = \text{call} \quad n' \rightarrow n \quad \text{where} \quad \sigma_{k-1} = \sigma' : n' \]

  Here, the \( i \) we are looking for is just \( k - 1 \).

- case [check]:

  \[ \text{check}: \quad \quad \quad \quad \quad \]

  \[ \ell(n') = \text{check}(P) \quad \sigma : n' \vdash \text{JDK}(P) \quad n' \rightarrow n \]

  By the inductive hypothesis, \( \exists i \in 0..k-2. \sigma_i = \sigma \).

- case [return]:

  \[ \text{return}: \quad \quad \quad \quad \quad \]

  \[ \ell(m) = \text{return} \quad n' \rightarrow n \]

  By the inductive hypothesis, \( \exists j \in 0..k-2. \sigma_j = \sigma : n' \) (clearly, \( j \neq 0 \)). Applying the inductive hypothesis again, we find an \( i \in 0..j-1 \) such that \( \sigma_i = \sigma \).

\[ \square \]

Corollary 3.4.2.

\[ G \vdash \sigma : n \quad \implies \quad G \vdash \sigma \]

Proof. Let \( \sigma : n \) be a \( G \)-reachable state, i.e. \( G \vdash \sigma : n \). By definition, this means that a derivation \( \sigma_0 \triangleright \sigma_1 \triangleright \cdots \triangleright \sigma_k \) exists, with \( \sigma_0 = \mathbb{0} \) and \( \sigma_k = \sigma : n \). Then, by lemma 3.4.1, we find that \( \sigma_i = \sigma \) for some \( i \in 0..k-1 \). Now, \( \sigma_0 \triangleright \cdots \triangleright \sigma_i \) is a derivation for \( \sigma \): thus, \( G \vdash \sigma \). \( \square \)

Lemma 3.4.3. Let:

\[ \mathbb{0} = \sigma_0 \triangleright \sigma_1 \triangleright \cdots \triangleright \sigma_k = \sigma : n : m \]

be a derivation, and define:

\[ i^* = \max \{ i \in 0..k-1 \mid \sigma_i = \sigma : n \} \]

Then:

\[ \forall i \in i^*..k. \exists \tau_i \in N^*. \sigma_i = \sigma : n : \tau_i \]
Proof. By lemma 3.4.1, we know that there is at least one index i such that \( \sigma_i = \sigma : n \), hence \( t^* \) is well defined. We proceed by induction on the length of the derivation. Clearly, in any case we will have \( \tau_k = [m] \). The base case is \( n \in N_{\text{entry}}, \ell([n]) = \text{call} \) and \( n \rightarrow \rightarrow m \); here \( t^* = 1 \) and we define \( \tau_1 = [ ] \). For the inductive case, we proceed by case analysis on the transition \( \sigma_{k-1} \triangleright \sigma_k \), yielding:

- **case [call]:**
  \[
  \ell([n]) = \text{call} \quad n \rightarrow \rightarrow m \quad \sigma : n \triangleright \sigma : n : m \quad \text{where} \quad \sigma_{k-1} = \sigma : n
  \]
  Here, by definition of \( t^* \), we have \( t^* = k - 1 \), and the lemma holds by letting \( \tau_{t^*} = [ ] \).

- **case [check]:**
  \[
  \ell([m']) = \text{check}(P) \quad \sigma : n : m' \vdash \text{DK}(P) \quad m' \rightarrow \rightarrow m \quad \sigma : n : m' \triangleright \sigma : n : m
  \]
  By the inductive hypothesis, \( \forall i \in t^* . k - 1. \exists \tau_i \in N^* . \sigma_i = \sigma : n : \tau_i \).

- **case [return]:**
  \[
  \ell([n']) = \text{return} \quad m' \rightarrow \rightarrow m \\
  \sigma : n : m' \triangleright \sigma : n : m
  \]
  Define \( j^* = \max \{ j \in 0 . k - 2 \mid \sigma_j = \sigma : n : m' \} \). By the inductive hypothesis, \( \forall i \in j^* . k - 1. \exists \tau_i \in N^* . \sigma_i = \sigma : n : \tau_i \). Applying again the inductive hypothesis, we have that: \( \forall i \in j^* . j^* - 1. \exists \tau_i'' \in N^* . \sigma_i = \sigma : n : \tau_i'' \). Then we define:

\[
\tau_i = \begin{cases} 
\tau_i'' & \text{if } t^* \leq i < j^* \\
\tau_i' & \text{if } j^* \leq i < k 
\end{cases}
\]

\[
\square
\]

**Lemma 3.4.4.**

\[
G \vdash \sigma : n : m \quad \Rightarrow \quad \ell([n]) = \text{call}
\]

**Proof.** Consider an arbitrary derivation for \( \sigma : n : m \), say: \( \sigma_0 \triangleright \cdots \triangleright \sigma_k \), where \( \sigma_0 = [ ] \) and \( \sigma_k = \sigma : n : m \). By lemma 3.4.3, we know that, choosing \( t^* = \max \{ i \in 0 . k - 1 \mid \sigma_i = \sigma : n \} \), we have \( \forall i \in t^* . k. \exists \tau_i \in N^* . \sigma_i = \sigma : n : \tau_i \). By contradiction, assume \( \ell([n]) \not= \text{call} \). This leaves the following cases for the transition \( \sigma_i \triangleright \sigma_{i+1} \):

- **if** \( \ell([n]) = \text{check}(P) \), then \( \exists \tau \in N . \sigma : n \triangleright \sigma : n' \). This is a contradiction, as \( \neg \exists \tau \in N^* . \sigma : n' = \sigma : n : \tau \).

- **if** \( \ell([n]) = \text{return} \), then \( \exists \tau \in \Sigma . n', n'' \in N . \sigma = \sigma' : n' \) and \( \sigma' : n' : n \triangleright \sigma' : n'' \). Again, this is a contradiction, because \( \neg \exists \tau \in N^* . \sigma' : n'' = \sigma' : n' : \tau \).

\[
\square
\]
3.5 Abstract paths

A path is a sequence of edges in the graph, corresponding to a partial trace of a program’s potential control flow. More formally, let $G = (N,E,N_{entry})$ be a control flow graph. Then, the sequence:

$$\langle (n_0,n_1),(n_1,n_2),\ldots,(n_{k-1},n_k) \rangle$$

is an abstract path on $G$ if $n_0 \neq \bot_N$, and $(n_i,n_{i+1}) \in E$ for each $i \in 0..k-1$.

Since we have assumed that control flow graphs are well-formed, by constraint 1 it follows that, for each $i \in 0..k-1$, an unique edge may exist from node $n_i$ to $n_{i+1}$. Therefore, each abstract path is fully characterized by the sequence of nodes it traverses, as stated by the following definition.

**Definition 3.5.1.** An abstract path from node $n_0 \in N \setminus \{ \bot_N \}$ to node $n_k$ is a sequence $\langle n_0,\ldots,n_k \rangle$ where $k = 0$, or:

$$\forall i \in 0..k-1. (n_i,n_{i+1}) \in E$$

We denote with $\Pi_{n_0,n_k}$ the set of all abstract paths from $n_0$ to $n_k$, and with $\Pi$ the set of all abstract paths. An abstract entry path is an abstract path which starts from some entry node $n_0 \in N_{entry}$. We denote with $\Pi_n$ the set of all abstract entry paths leading to $n$, and with $\Pi_{entry}$ the set of all abstract entry paths. The empty sequence $\langle \rangle$ is a (non-entry) abstract path. We write $\Pi_{n_0,n_k}(G)$, $\Pi(G)$, $\Pi_n(G)$ and $\Pi_{entry}(G)$ when we want to make clear that the control flow graph under consideration is $G$.

**Lemma 3.5.2.**

$$G \vdash \sigma : n \implies \Pi_n \neq \emptyset$$

*Proof.* The proof is carried out by induction on the derivation used to establish $G \vdash \sigma : n$. For the base case, if $\sigma = \emptyset$ and $n \in N_{entry}$, then $\langle n \rangle \in \Pi_n$. For the inductive case, we proceed by case analysis on the last rule used to derive $G \vdash \sigma : n$, yielding:

- case [call]:

$$\ell(n') = \text{call} \quad n' \rightarrow n$$

where $\sigma' : n' \triangleright \sigma' : n' : n$.

By the inductive hypothesis, $G \vdash \sigma : n'$ implies $\Pi_{n'} \neq \emptyset$. So, take $\pi \in \Pi_{n'}$. Since $n' \rightarrow n$, the path $\pi : n$ is in $\Pi_n$. 


3.5. ABSTRACT PATHS

- **case [check]:**
  \[
  \ell([n]) = \text{check}(P) \quad \sigma : n' \vdash JDK[P] \quad n' \rightarrow n
  \]
  \[
  \sigma : n' \triangleright \sigma : n
  \]
  By the inductive hypothesis, \( G \vdash \sigma : n' \) implies \( \Pi_{n'} \neq \emptyset \). Again, take \( \pi \in \Pi_{n'} \). Here \( n' \rightarrow n \), hence \( \pi : n \in \Pi_n \).

- **case [return]:**
  \[
  \ell([m]) = \text{return} \quad n' \rightarrow n
  \]
  \[
  \sigma : n' : m \triangleright \sigma : n
  \]
  By lemma 3.4.1, any derivation of \( \sigma : n' : m \) is of the form:
  \[
  [] \triangleright \cdots \triangleright \sigma : n' \triangleright \cdots \triangleright \sigma : n' : m
  \]
  Therefore, we can apply the inductive hypothesis to \( \sigma : n' \), obtaining \( \Pi_{n'} \neq \emptyset \). Again, if we take \( \pi \in \Pi_{n'} \), then \( n' \rightarrow n \) implies \( \pi : n \in \Pi_n \).

\( \square \)

**Definition 3.5.3.** The weight of an edge \((n, n') \in E\) is defined as follows:

\[
\begin{align*}
\text{w}(n, n') &= \begin{cases} 
1 & \text{if } n \rightarrow n' \\
0 & \text{if } n \rightarrow n'
\end{cases}
\end{align*}
\]

Since entry edges \((\bot_N, n)\) are never present in abstract paths, their weight is left undefined.

**Definition 3.5.4.** The weight of an abstract path is defined as follows:

\[
\begin{align*}
\text{w}(\emptyset) &= 0 \\
\text{w}(\langle n_0, \ldots, n_k \rangle) &= 1 + \sum_{i=0}^{k-1} \text{w}(n_i, n_{i+1})
\end{align*}
\]

**Lemma 3.5.5.** Let \( \pi = \pi_0 : \pi_1 = \langle n_0, \ldots, n_h \rangle : \langle n_{h+1}, \ldots, n_k \rangle \) be an abstract path, with \( 0 < h < k \). Then:

\[
\text{w}(\pi) = \text{w}(\pi_0) + \text{w}(\pi_1) + \text{w}(n_h, n_{h+1}) - 1
\]

**Proof.** Since \( \pi_0 \) and \( \pi_1 \) are non-empty, by definition 3.5.4 we have:

\[
\begin{align*}
\text{w}(\pi_0) + \text{w}(\pi_1) &= \left( 1 + \sum_{i=0}^{n-1} \text{w}(n_i, n_{i+1}) \right) + \left( 1 + \sum_{i=h+1}^{k-1} \text{w}(n_i, n_{i+1}) \right) \\
&= \left( 1 + \sum_{i=0}^{k-1} \text{w}(n_i, n_{i+1}) \right) + 1 - \text{w}(n_h, n_{h+1}) \\
&= \text{w}(\pi) + 1 - \text{w}(n_h, n_{h+1})
\end{align*}
\]

\( \square \)
Definition 3.5.6. Let $\pi$ be an abstract path, $n, n' \in \mathbb{N}$, and $\sigma \in \Sigma$. We define the transition system $\chi = \langle \Pi \cup [\Sigma], \rightarrow_\chi \rangle$ by the following rules:

\[
\begin{align*}
\text{n} \in \mathbb{N}_{entry} & \quad \langle n \rangle \rightarrow_\chi [n] \quad \tag{[X_{entry}]} \\
\pi : n' \rightarrow_\chi \sigma & \quad n' \rightarrow n \\
\pi : n' : n \rightarrow_\chi \sigma : n \\
\pi : n' \sigma : n' \rightarrow_\chi n & \rightarrow_\chi \sigma : n \quad \tag{[X_{call}]} \\
\pi : n' \sigma : n' \rightarrow_\chi n & \rightarrow_\chi \sigma : n \quad \tag{[X_{trans}]} 
\end{align*}
\]

Lemma 3.5.7. Let $\pi = \langle n_0, \ldots, n_k \rangle$ be an abstract path, such that:

\[\pi \rightarrow_\chi [m_0, \ldots, m_n]\]

Then, a strictly increasing function $\varphi : 0..h \rightarrow 0..k$ exists, satisfying:

\[\forall i \in 0..h. m_i = n_{\varphi(i)} \land \varphi(h) = k \quad \tag{3.1}\]

Proof. The proof is carried out by mathematical induction on the length of the abstract path $\pi$. For the base case, if $\pi = \langle n_0 \rangle$ and $n_0 \in \mathbb{N}_{entry}$, then it must be $\pi \rightarrow_\chi [n_0]$, because only the first rule of $\chi$ is applicable. Then, the function $\varphi = \{ 0 \mapsto 0 \}$ trivially satisfies (3.1). For the inductive case, we proceed by case analysis on the last edge of the abstract path:

- case [call]:

\[
\begin{align*}
\langle n_0, \ldots, n_{k-1} \rangle & \rightarrow_\chi \sigma \quad n_{k-1} \rightarrow n_k \\
\langle n_0, \ldots, n_k \rangle & \rightarrow_\chi \sigma : n_k
\end{align*}
\]

By the inductive hypothesis, we have a strictly increasing function:

\[\varphi' : 0.|\sigma| - 1 \rightarrow 0..k - 1\]

which satisfies (3.1). Then, we define $\varphi : 0.|\sigma| \rightarrow 0..k$ as follows:

\[
\varphi[x] = \begin{cases} 
\begin{array}{ll}
k & \text{if } x = |\sigma| \\
\varphi'[x] & \text{otherwise} 
\end{array}
\end{cases}
\]

Now, $\varphi$ is strictly increasing on $0.|\sigma|-1$, because, inside that interval, it coincides with $\varphi'$. Since $\varphi'(|\sigma|-1) = k-1$ by (3.1), we have that:

\[\varphi(|\sigma|) = k > k-1 = \varphi'(|\sigma|-1) = \varphi(|\sigma|-1)\]

Hence, $\varphi$ is strictly increasing inside the whole domain $0.|\sigma|$. Since we also have $n_k = n_{\varphi(|\sigma|)}$, we can conclude that $\varphi$ satisfies (3.1).
3.5. ABSTRACT PATHS

- case [transfer]:

\[
\langle n_0, \ldots, n_{k-1} \rangle \rightarrow \chi \sigma : n_{k-1} \rightarrow n_k \\
\langle n_0, \ldots, n_k \rangle \rightarrow \chi \sigma : n_k
\]

By the inductive hypothesis, we have a strictly increasing function:

\[ \varphi' : 0..|\sigma| \rightarrow 0..k - 1 \]

which satisfies (3.1). Then, we define \( \varphi : 0..|\sigma| \rightarrow 0..k \) as follows:

\[
\varphi(x) = \begin{cases} 
  k & \text{if } x = |\sigma| \\
  \varphi'(x) & \text{otherwise}
\end{cases}
\]

Now, \( \varphi \) is strictly increasing on 0..|\sigma| - 1, because, inside that interval, it coincides with \( \varphi' \). Since \( \varphi'(|\sigma|) = k - 1 \) and \( \varphi'(|\sigma|) > \varphi'(|\sigma| - 1) \) by (3.1), we have that:

\[ \varphi(|\sigma|) = k > k - 1 = \varphi'(|\sigma|) > \varphi'(|\sigma| - 1) = \varphi(|\sigma| - 1) \]

Hence, \( \varphi \) is strictly increasing inside the whole domain 0..|\sigma|. Since we also have \( n_k = n_{\varphi(|\sigma|)} \), we can conclude that \( \varphi \) satisfies (3.1).

\[
\square
\]

Lemma 3.5.8. Let \( \pi = \langle n_0, \ldots, n_k \rangle \) be an abstract path, such that:

\[
\pi \rightarrow \chi [m_0, \ldots, m_h]
\]

Then, if \( \varphi \) is the function defined in lemma 3.5.7 and \( i \in 0..h \), we have:

\[
\langle n_0, \ldots, n_{\varphi(i)} \rangle \rightarrow \chi [m_0, \ldots, m_i]
\]

(3.2)

Proof. The proof is carried out by mathematical induction on the length of the abstract path \( \pi \). First of all, observe that (3.2) is immediately satisfied when \( i = h \): in fact, lemma 3.5.7 states that \( \varphi(h) = k \). For the base case, if \( \pi = \langle n_0 \rangle \) and \( n_0 \in \text{N}_{\text{entry}} \), then, by the first rule of \( \chi \), we have \( \pi \rightarrow \chi [n_0] \); hence \( h = 0 \), and (3.2) is trivially satisfied. For the inductive case, we proceed by case analysis on the last edge of the abstract path:

- case [call]:

\[
\langle n_0, \ldots, n_{k-1} \rangle \rightarrow \chi \sigma : n_{k-1} \rightarrow n_k \\
\langle n_0, \ldots, n_k \rangle \rightarrow \chi \sigma : n_k
\]

Let \( \sigma = [m'_0, \ldots, m'_{h'}] \). Then, we have \( h = h' + 1 \), and:

\[
m_k = \begin{cases} 
  n_k & \text{if } x = h \\
  m'_{h'} & \text{otherwise}
\end{cases}
\]

By the inductive hypothesis, we find a function \( \varphi' : 0..h' \rightarrow 0..k - 1 \), such that:

\[ \forall i \in 0..h'. \langle n_0, \ldots, n_{\varphi'(i)} \rangle \rightarrow \chi [m'_0, \ldots, m'_i] \]

Following the construction of lemma 3.5.7, we define \( \varphi : 0..h \rightarrow 0..k \) as:

\[
\varphi(x) = \begin{cases} 
  k & \text{if } x = h \\
  \varphi'(x) & \text{otherwise}
\end{cases}
\]

Then, if \( i < h \), we have that \( i \leq h' \), and:

\[
\langle n_0, \ldots, n_{\varphi(i)} \rangle = \langle n_0, \ldots, n_{\varphi'(i)} \rangle \rightarrow \chi [m'_0, \ldots, m'_i] = [m_0, \ldots, m_i]
\]
• case [transfer]:

\[
\langle n_0, \ldots, n_{k-1} \rangle \xrightarrow{\chi} \sigma : n_{k-1} \xrightarrow{\alpha} n_k \\
\langle n_0, \ldots, n_k \rangle \xrightarrow{\chi} \sigma : n_k
\]

Let \( \sigma : n_{k-1} = [m'_0, \ldots, m'_{k-1}] \), with \( m'_{k-1} = n_{k-1} \). Then, we have \( h = h' \), and:

\[
m_k = \begin{cases} 
  n_k & \text{if } x = h \\
  m'_k & \text{otherwise}
\end{cases}
\]

By the inductive hypothesis, we find a function \( \varphi' : 0..h' \rightarrow 0..k-1 \), such that:

\[
\forall i \in 0..h'. \quad \langle n_0, \ldots, n_{\varphi(i)} \rangle \xrightarrow{\chi} [m'_0, \ldots, m'_i]
\]

Following the construction of lemma 3.5.7, we define \( \varphi : 0..h \rightarrow 0..k \) as:

\[
\varphi[x] = \begin{cases} 
  k & \text{if } x = h \\
  \varphi'[x] & \text{otherwise}
\end{cases}
\]

Then, if \( i < h \), we have that \( i < h' \), and:

\[
\langle n_0, \ldots, n_{\varphi(i)} \rangle = \langle n_0, \ldots, n_{\varphi'(i)} \rangle \xrightarrow{\chi} [m'_0, \ldots, m'_i] = [m_0, \ldots, m_i]
\]

Lemma 3.5.9. The evaluation of \( \chi \) is deterministic, i.e.:

\[
\pi \xrightarrow{\chi} \sigma \land \pi \xrightarrow{\chi} \sigma' \implies \sigma = \sigma'
\]

for any abstract path \( \pi \in \Pi \) and \( \sigma, \sigma' \in \Sigma \).

Proof. The proof is carried out by mathematical induction on the length of the abstract path \( \pi \). For the base case, let \( \pi = \langle n \rangle \), with \( n \in N_{entry} \). If \( \pi \xrightarrow{\chi} \sigma \) and \( \pi \xrightarrow{\chi} \sigma' \), then it must be \( \sigma = \sigma' = [n] \), because only the first rule of \( \chi \) is applicable. For the inductive case, let \( \pi = \pi' : n' : n \). We proceed by case analysis on the last edge of the abstract path:

• case [call]: consider the two transitions:

\[
\frac{\pi' : n' \xrightarrow{\chi} \sigma : n'}{\pi' : n' : n \xrightarrow{\chi} \sigma : n} \\
\frac{\pi' : n' \xrightarrow{\chi} \sigma' : n'}{\pi' : n' : n \xrightarrow{\chi} \sigma' : n}
\]

By the inductive hypothesis, we have \( \sigma = \sigma' \), then \( \sigma : n = \sigma' : n \) holds, too.

• case [transfer]: consider the two transitions:

\[
\frac{\pi' : n' \xrightarrow{\chi} \sigma : n' \xrightarrow{\alpha} n}{\pi' : n' : n \xrightarrow{\chi} \sigma : n} \\
\frac{\pi' : n' \xrightarrow{\chi} \sigma' : n' \xrightarrow{\alpha} n}{\pi' : n' : n \xrightarrow{\chi} \sigma' : n}
\]

By the inductive hypothesis, we have \( \sigma : n' = \sigma' : n' \), which clearly implies \( \sigma = \sigma' \). Then, also \( \sigma : n = \sigma' : n \) does hold.
3.5. ABSTRACT PATHS

**Lemma 3.5.10.** The evaluation of $\chi$ is total on abstract entry paths, i.e.:

$$\pi \in \Pi_{entry} \implies \exists \sigma \in \Sigma. \pi \rightarrow_{\chi} \sigma$$

**Proof.** We prove the stronger result:

$$\pi \in \Pi_n \implies \exists \sigma \in \Sigma. \pi \rightarrow_{\chi} \sigma : n$$

The proof is carried out by mathematical induction on the length of the abstract path $\pi$. For the base case, if $n \in N_{entry}$ and $\pi = \langle n \rangle \in \Pi_n$, then $\pi \rightarrow_{\chi} \langle n \rangle$ by the $\chi_{entry}$ rule. For the inductive case, let $\pi = \pi' : n' : n$, with $\pi'$ possibly empty. By the inductive hypothesis applied to $\pi' : n'$, it follows that:

$$\pi' : n' \in \Pi_n' \implies \exists \sigma' \in \Sigma. \pi' : n' \rightarrow_{\chi} \sigma' : n'$$

Now we proceed by case analysis on the edge $\langle n', n \rangle$:

- **case [call]:** if $n' \rightarrow n$, then the $\chi_{call}$ rule is applicable, and we obtain:

  $$\begin{align*}
  \pi' : n' & \rightarrow_{\chi} \sigma' : n' \quad n' \rightarrow n \\
  \pi' : n' : n & \rightarrow_{\chi} \sigma' : n' : n
  \end{align*}$$

  Then, the result is proved by letting $\sigma = \sigma' : n'$.

- **case [transfer]:** if $n' \rightarrow n$, we can apply the $\chi_{trans}$ rule, yielding:

  $$\begin{align*}
  \pi' : n' & \rightarrow_{\chi} \sigma' : n' \quad n' \rightarrow n \\
  \pi' : n' : n & \rightarrow_{\chi} \sigma' : n
  \end{align*}$$

  Here the result is proved by letting $\sigma = \sigma'$.

\[ \Box \]

Since we have proved that the evaluation of $\chi$ is both deterministic and total on abstract entry paths, we define the total function $\chi : \Pi_{entry} \rightarrow \Sigma$ as:

$$\chi(\pi) = \sigma \quad \text{if} \quad \pi \rightarrow_{\chi} \sigma$$

**Theorem 3.5.11.**

$$G \vdash \sigma : n \implies \exists \pi \in \Pi_n. \chi(\pi) = \sigma : n$$

**Proof.** The proof is carried out by induction on the derivation used to establish $G \vdash \sigma : n$. For the base case, if $\sigma = []$ and $n \in N_{entry}$, then $\langle n \rangle \in \Pi_n$ and $\chi(\langle n \rangle) = \langle n \rangle$. For the inductive case, we proceed by case analysis on the last rule used to derive $G \vdash \sigma : n$, yielding:
• case [call]:
\[
\begin{align*}
\ell(n') &= \text{call } n' \rightarrow n \\
\sigma' : n' &\triangleright \sigma' : n \\
\text{where } \sigma &= \sigma' : n'
\end{align*}
\]

By the inductive hypothesis applied to \( \sigma' : n' \), we have:
\[
\exists \pi \in \Pi_{n'} : \chi(\pi) = \sigma' : n'
\]

Since \( n' \rightarrow n \), the path \( \pi : n \) is in \( \Pi_n \), and \( \chi(\pi : n) = \sigma' : n' : n \).

• case [check]:
\[
\begin{align*}
\ell(n') &= \text{check(P) } \sigma' : n' \vdash \text{DK(P)} \\
\sigma : n' &\triangleright \sigma : n \\
\end{align*}
\]

By the inductive hypothesis applied to \( \sigma : n' \), we have:
\[
\exists \pi \in \Pi_{n'} : \chi(\pi) = \sigma : n'
\]

Here we have \( n' \rightarrow n \), hence \( \pi : n \in \Pi_n \), and \( \chi(\pi : n) = \sigma : n \).

• case [return]:
\[
\begin{align*}
\ell(m) &= \text{return } n' \rightarrow n \\
\sigma : n' &\triangleright \sigma : n \\
\end{align*}
\]

By lemma 3.4.1, any derivation of \( \sigma : n' : m \) is of the form:
\[
[\] \triangleright \cdots \triangleright \sigma : n' \triangleright \cdots \triangleright \sigma : n' : m
\]

Therefore, we can apply the inductive hypothesis to \( \sigma : n' \), obtaining:
\[
\exists \pi \in \Pi_{n'} : \chi(\pi) = \sigma : n'
\]

As in the previous case, \( n' \rightarrow n \) implies \( \pi : n \in \Pi_n \), and \( \chi(\pi : n) = \sigma : n \).

\[\Box\]

**Theorem 3.5.12.** Let \( \pi \) be an abstract entry path. Then:

\[
|\chi(\pi)| = w(\pi)
\]

*Proof.* The proof is carried out by mathematical induction on the length of the abstract path \( \pi \). For the base case, if \( \pi = \langle n \rangle \) and \( n \in N_{entry} \), then \( w(\langle n \rangle) = 1 \) and \( |\chi(\langle n \rangle)| = |\langle n \rangle| = 1 \). For the inductive case, we proceed by case analysis on the last edge of the abstract path:

• case [call]:
\[
\begin{align*}
\chi(\pi' : n') &= \sigma \quad n' \rightarrow n \\
\chi(\pi' : n' : n) &= \sigma : n \\
\text{where } \pi &= \pi' : n' : n
\end{align*}
\]

By the inductive hypothesis, we have \( |\chi(\pi' : n')| = |\sigma| = w(\pi' : n') \). Then:
\[
|\chi(\pi' : n' : n)| = |\sigma : n| = |\sigma| + 1 = w(\pi' : n') + 1 = w(\pi' : n' : n)
\]
3.5. ABSTRACT PATHS

- case [transfer]:

\[
\chi(\pi' : n') = \sigma : n' \quad n' \rightarrow n
\]

where \( \pi = \pi' : n' : n \)

By the inductive hypothesis, we have \( |\chi(\pi' : n')| = |\sigma : n'| = w(\pi' : n') \). Then:

\[
|\chi(\pi' : n')| = |\sigma : n| = |\sigma : n'| = w(\pi' : n') = w(\pi' : n' : n)
\]

\[\square\]

Definition 3.5.13. We say that \( n \in N \) is an exit node if:

\[\exists \pi \in \Pi_n, \ w(\pi) = 1\]

We denote with \( N_{exit} \) the set of exit nodes.

Well-formedness constraint 7. An exit node cannot be the return of any call. This constraint is formalized by saying that each abstract path leading to an exit node does not contain any call node, i.e. the path has unit weight:

\[n \in N_{exit} \implies \forall \pi \in \Pi_n, \ w(\pi) = 1\]

This constraint follows by the fact that methods are only accessible through their entry points.

Lemma 3.5.14. Let \( n \) be an exit node. Then:

\[G \vdash \sigma : n \implies \neg \exists \sigma' \in \Sigma. \, \sigma : n \triangleright \sigma'\]

Proof. By contradiction, assume \( \sigma : n \triangleright \sigma' \) for some \( \sigma' \in \Sigma \). Then, since \( t(n) = \text{return} \), only the \( \triangleright \text{return} \) rule is applicable, and we obtain:

\[
\frac{t(n) = \text{return} \quad m \rightarrow n'}{\sigma'' : m : n \triangleright \sigma'' : n'} \quad \text{where} \ \sigma = \sigma'' : m \ \text{and} \ \sigma' = \sigma'' : n'
\]

By theorem 3.5.11, we find an abstract path \( \pi \in \Pi_n \) such that \( \chi(\pi) = \sigma'' : m : n \). Now, by well-formedness constraint 7, we have that any abstract path leading to \( n \) has unit weight; therefore, \( w(\pi) = 1 \). On the other hand, theorem 3.5.12 tells that \( w(\pi) = |\chi(\pi)| \), which raises the contradiction:

\[1 = w(\pi) = |\sigma'' : m : n| = |\sigma''| + 2
\]

\[\square\]
**Definition 3.5.15.** The set $\rho(n, n')$ of return nodes associated to a call edge $n \to n'$ is defined as:

$$\rho(n, n') = \{ m \in N \mid \ell(m) = \text{return} \land \exists \pi \in \Pi_{n', m}, w(\pi) = 1 \}$$

The set $\rho(n)$ of return nodes associated to a call node $n$ is defined as:

$$\rho(n) = \bigcup \{ \rho(n, n') \mid n \to n' \}$$

**Lemma 3.5.16.**

$$G \vdash \sigma : n : m \land \ell(m) = \text{return} \implies m \in \rho(n)$$

**Proof.** We prove the stronger result:

$$G \vdash \sigma : n : m \implies \exists n' \in N, \pi \in \Pi_{n', m}, n \to n' \land w(\pi) = 1$$

The proof is carried out by induction on the derivation used to establish $G \vdash \sigma : n : m$. For the base case, we have $\sigma = [], n \in N_{\text{entry}}, \ell(n) = \text{call}$ and $n \to m$: then $\langle m \rangle$ is an abstract path from $m$ to itself having unit weight. For the inductive case, we proceed by case analysis on the last rule used to derive $\sigma : n : m$, yielding:

- **case [call]:**

  $$\ell([n]) = \text{call} \quad n \to m \quad \sigma : n \vdash \sigma : n : m$$

  Here $n \to m$ and $\langle m \rangle$ is clearly an abstract path from $m$ to itself having unit weight.

- **case [check]:**

  $$\ell([m']) = \text{check(P)} \quad \sigma : n : m' \vdash \text{JDK}(P) \quad m' \to m$$

  $$\sigma : n : m' \vdash \sigma : n : m$$

  By the inductive hypothesis:

  $$\exists n' \in N, \pi' \in \Pi_{n', m'}, n \to n' \land w(\pi') = 1$$

  Then the path $\pi = \pi' : m$ is in $\Pi_{n', m}$, and $w(\pi) = w(\pi') = 1$ as $m' \to m$.

- **case [return]:**

  $$\ell([m'']) = \text{return} \quad m'' \to m$$

  $$\sigma : n : m' : m'' \vdash \sigma : n : m$$

  By lemma 3.4.1, any derivation of $\sigma : n : m' : m''$ is of the form:

  $$[] \vdash \cdots \vdash \sigma : n : m' \vdash \cdots \vdash \sigma : n : m' : m''$$

  Therefore, we can apply the inductive hypothesis to $\sigma : n : m'$, obtaining:

  $$\exists n' \in N, \pi' \in \Pi_{n', m'}, n \to n' \land w(\pi') = 1$$

  Again, the path $\pi = \pi' : m$ is in $\Pi_{n', m}$, and $w(\pi) = w(\pi') = 1$. 


It is immediate that our lemma follows from this result, by hypothesis $\ell(m) = \text{return}$ and by definition of $\rho(n)$.

\textbf{Well-formedness constraint 8.} Each call has a return. This constraint is formalized by saying that each method entry point (i.e. a node $n'$ such that $n \rightarrow n'$ for some $n$) is connected to a return node by a path with unit weight:

$$n \rightarrow n' \implies \rho(n, n') \neq \emptyset$$

\textbf{Definition 3.5.17.} We say that $m \leftrightarrow n'$ is a \textit{return edge} iff:

$$\exists n \in \mathbb{N}. \ m \in \rho(n) \land n \rightarrow n'$$

We denote with $E_{\text{return}}$ the set of return edges. The set $\tilde{E}$ of all \textit{concrete edges} is defined as: $\tilde{E} = E_{\text{call}} \cup \{ (m, n) \in E_{\text{trans}} \mid \ell(m) \neq \text{call} \} \cup E_{\text{entry}} \cup E_{\text{return}}$. The set of all edges (either abstract or concrete) is denoted with $E_p = E \cup \tilde{E}$. The weight $w(m, n')$ of a return edge $m \leftrightarrow n'$ is defined as $-1$.

\textbf{Lemma 3.5.18.}

$$m \in N_{\text{exit}} \land \exists n \in \mathbb{N}. \ m \in \rho(n) \implies \Pi_n = \emptyset$$

\textit{Proof.} By contradiction, assume $m$ is an exit node, and $m \in \rho(n)$ for some node $n$ such that $\Pi_n \neq \emptyset$. By definition 3.5.15, this means that $n \rightarrow n'$ for some node $n'$, and:

$$\exists \pi' \in \Pi_{n', m}. \ w(\pi') = 1$$

(3.3)

Moreover, by hypothesis $\Pi_n \neq \emptyset$, we can choose an abstract path $\pi''$ leading to $n$. Then, the path $\pi = \pi'' : \pi'$ is in $\Pi_m$, and we have:

$$1 = w(\pi) \quad \text{by w.f. constraint 7}$$

$$= w(\pi'') + w(\pi') + w(n, n') - 1 \quad \text{by lemma 3.5.5}$$

$$= w(\pi'') + w(\pi') + 1 - 1 \quad \text{as } n \rightarrow n'$$

$$= w(\pi'') + 1 \quad \text{by (3.3)}$$

Then, we find $w(\pi'') = 0$: this is a contradiction, since, by definition, non-empty abstract paths have strictly positive weight. \qed
3.6 Concrete paths

**Definition 3.6.1.** A concrete path from node $n_0 \in N \setminus \{\bot_N\}$ to node $n_k$ is a sequence $\langle n_0, \ldots, n_k \rangle$ where $k = 0$, or:

$$\forall i \in 0..k-1. \quad \ell(n_i) = \text{call} \quad \land \quad n_i \to n_{i+1}, \quad \text{or} \quad \ell(n_i) = \text{check}(P) \quad \land \quad n_i \to n_{i+1}, \quad \text{or} \quad \ell(n_i) = \text{return} \quad \land \quad n_i \leftarrow n_{i+1}$$

We denote with $\tilde{\Pi}_{n_0,n_k}$ the set of all concrete paths from $n_0$ to $n_k$, and with $\tilde{\Pi}$ the set of all concrete paths. A concrete entry path is a concrete path which starts from some entry node $n_0 \in N_{\text{entry}}$. We denote with $\tilde{\Pi}_n$ the set of all concrete entry paths leading to $n$, and with $\tilde{\Pi}_{\text{entry}}$ the set of all concrete entry paths. The empty sequence $\langle \rangle$ is a concrete (non-entry) path. We write $\tilde{\Pi}_{n_0,n_k}(G)$, $\tilde{\Pi}(G)$, $\tilde{\Pi}_n(G)$ and $\tilde{\Pi}_{\text{entry}}(G)$ when we want to make clear that the control flow graph under consideration is $G$.

**Lemma 3.6.2.**

$$G \vdash \sigma : n \implies \tilde{\Pi}_n \neq \emptyset$$

**Proof.** The proof is carried out by induction on the derivation used to establish $G \vdash \sigma : n$. For the base case, if $\sigma = [\]$ and $n \in N_{\text{entry}}$, then $\langle n \rangle \in \tilde{\Pi}_n$. For the inductive case, we proceed by case analysis on the last rule used to derive $G \vdash \sigma : n$, yielding:

- **case [call]:**

$$\ell(n') = \text{call} \quad n' \to n \quad \text{where} \quad \sigma : n' \vdash \sigma' : n$$

By the inductive hypothesis, $G \vdash \sigma : n'$ implies $\tilde{\Pi}_{n'} \neq \emptyset$. So, take $\tilde{n} \in \tilde{\Pi}_{n'}$. Since $\ell(n') = \text{call}$ and $n' \to n$, the path $\tilde{n} \cdot n$ is in $\tilde{\Pi}_n$.

- **case [check]:**

$$\ell(n') = \text{check}(P) \quad \sigma : n' \vdash \text{DK}(P) \quad n' \to n$$

$$\sigma : n' \vdash \sigma : n$$

By the inductive hypothesis, $G \vdash \sigma : n'$ implies $\tilde{\Pi}_{n'} \neq \emptyset$. Again, take $\tilde{n} \in \tilde{\Pi}_{n'}$. Here $\ell(n') = \text{check}(P)$ and $n' \to n$, hence $\tilde{n} \cdot n$ is in $\tilde{\Pi}_n$.

- **case [return]:**

$$\ell(m) = \text{return} \quad n' \to n$$

$$\sigma : n' : m \vdash \sigma : n$$

By the inductive hypothesis, $G \vdash \sigma : n' : m$ implies $\tilde{\Pi}_m \neq \emptyset$. Moreover, by lemma 3.5.16, $G \vdash \sigma : n' : m$ and $\ell(m) = \text{return}$ imply $m \in \rho(n')$. Now, take $\tilde{n} \in \tilde{\Pi}_m$. From $m \in \rho(n')$ and $n' \to n$ we deduce $m \leftarrow n$, hence $\tilde{n} \cdot n \in \tilde{\Pi}_n$. 


3.6. CONCRETE PATHS

\[ \tilde{w}(\langle \rangle) = 0 \]
\[ \tilde{w}(\langle n_0, \ldots, n_k \rangle) = 1 + \sum_{i=0}^{k-1} w(n_i, n_{i+1}) \]

**Definition 3.6.3.** The weight of a concrete path is defined as follows:

**Lemma 3.6.4.** Let \( \tilde{\pi} = \tilde{\pi}_0 : \tilde{\pi}_1 : \langle n_0, \ldots, n_h \rangle : \langle n_{h+1}, \ldots, n_k \rangle \) be a concrete path, with \( 0 < h < k \). Then:

\[ \tilde{w}(\tilde{\pi}) = \tilde{w}(\tilde{\pi}_0) + \tilde{w}(\tilde{\pi}_1) + w(n_h, n_{h+1}) - 1 \]

**Proof.** Since \( \tilde{\pi}_0 \) and \( \tilde{\pi}_1 \) are non-empty, by definition 3.6.3 we have:

\[ \tilde{w}(\tilde{\pi}_0) + \tilde{w}(\tilde{\pi}_1) = \left( 1 + \sum_{i=0}^{h-1} \tilde{w}(n_i, n_{i+1}) \right) + \left( 1 + \sum_{i=h+1}^{k-1} \tilde{w}(n_i, n_{i+1}) \right) \]
\[ = \left( 1 + \sum_{i=0}^{k-1} \tilde{w}(n_i, n_{i+1}) \right) + 1 - w(n_h, n_{h+1}) \]
\[ = \tilde{w}(\tilde{\pi}) + 1 - w(n_h, n_{h+1}) \]

**Definition 3.6.5.** Let \( \tilde{\pi} \) be a concrete path, \( n, n', m \in \mathbb{N} \), and \( \sigma \in \Sigma \). We define the transition system \( \tilde{\chi} = \langle \tilde{T} \cup \Sigma, \Sigma, \rightarrow_{\tilde{\chi}} \rangle \) by the following rules:

\[ \langle n \rangle \rightarrow_{\tilde{\chi}} [m] \]
\[ \tilde{\pi} : n' \rightarrow_{\chi} \sigma : n \]
\[ \tilde{\pi} : n' : n \rightarrow_{\chi} \sigma : n \]
\[ \tilde{\pi} : n' \rightarrow_{\chi} \sigma : n' \rightarrow_{\chi} n \]
\[ \tilde{\pi} : m \rightarrow_{\chi} \sigma : m \rightarrow_{\chi} n \]

**Lemma 3.6.6.** Let \( \tilde{\pi} = \langle n_0, \ldots, n_k \rangle \) be a concrete path, such that:

\[ \tilde{\pi} \rightarrow_{\tilde{\chi}} [m_0, \ldots, m_h] \]

Then, a strictly increasing function \( \varphi : 0..h \rightarrow 0..k \) exists, satisfying:

\[ \forall i \in 0..h. \ m_i = n_{\varphi(i)} \land \ \varphi(h) = k \]

(3.4)
Proof. Since, except for the \( \bar{\chi}_{\text{return}} \) rule, the rules of \( \bar{\chi} \) coincide with those of \( \chi \), the proof is identical to the proof of lemma 3.5.7 for the base, \([\text{call}]\) and \([\text{transfer}]\) cases. Hence, we have to treat only the following case:

- case \([\text{return}]\):

\[
\begin{align*}
\langle n_0, \ldots, n_{k-1} \rangle \rightarrow_{\bar{\chi}} \sigma : m : n_{k-1} & \quad n_{k-1} \leftrightarrow n_k \\
\langle n_0, \ldots, n_k \rangle \rightarrow_{\bar{\chi}} \sigma : n_k
\end{align*}
\]

By the inductive hypothesis, we have a strictly increasing function:

\[
\varphi' : 0..|\sigma| + 1 \rightarrow 0..k - 1
\]

which satisfies (3.4). Then, we define \( \varphi : 0..|\sigma| \rightarrow 0..k \) as follows:

\[
\varphi(x) = \begin{cases} 
  k & \text{if } x = |\sigma| \\
  \varphi'(x) & \text{otherwise}
\end{cases}
\]

Now, \( \varphi \) is strictly increasing on \( 0..|\sigma| - 1 \), because, inside that interval, it coincides with \( \varphi' \). Since \( \varphi'(|\sigma| + 1) = k - 1 \) and \( \varphi'(|\sigma| + 1) > \varphi'(|\sigma| - 1) \) by (3.4), we have:

\[
\varphi(|\sigma|) = k > k - 1 = \varphi'(|\sigma| + 1) > \varphi'(|\sigma| - 1) = \varphi(|\sigma| - 1)
\]

Hence, \( \varphi \) is strictly increasing inside the whole domain \( 0..|\sigma| \). Since we also have \( n_k = n_{\varphi(|\sigma|)} \), we can conclude that \( \varphi \) satisfies (3.4).

\[\square\]

Lemma 3.6.7. Let \( \bar{\pi} = \langle n_0, \ldots, n_k \rangle \) be a concrete path, such that:

\[
\bar{\pi} \rightarrow_{\bar{\chi}} [m_0, \ldots, m_h]
\]

Then, if \( \varphi \) is the function defined in lemma 3.6.6, and \( i \in 0..h \), we have:

\[
\langle n_0, \ldots, n_{\varphi(i)} \rangle \rightarrow_{\bar{\chi}} [m_0, \ldots, m_i]
\]

(3.5)

Proof. Since, except for the \( \bar{\chi}_{\text{return}} \) rule, the rules of \( \bar{\chi} \) coincide with those of \( \chi \), the proof is identical to the proof of lemma 3.5.8 for the base, \([\text{call}]\) and \([\text{transfer}]\) cases. Hence, we have to treat only the following case:

- case \([\text{return}]\):

\[
\begin{align*}
\langle n_0, \ldots, n_{k-1} \rangle \rightarrow_{\bar{\chi}} \sigma : m : n_{k-1} & \quad n_{k-1} \leftrightarrow n_k \\
\langle n_0, \ldots, n_k \rangle \rightarrow_{\bar{\chi}} \sigma : n_k
\end{align*}
\]

Let \( \sigma : m : n_{k-1} = [m'_0, \ldots, m'_h] \), with \( m'_{i+1} = m \) and \( m'_{i'} = n_{k-1} \). Then, we have \( h = h' - 1 \), and:

\[
m_x = \begin{cases} 
  n_k & \text{if } x = h \\
  m'_x & \text{otherwise}
\end{cases}
\]

By the inductive hypothesis, we find a function \( \varphi' : 0..h' \rightarrow 0..k - 1 \), such that:

\[
\forall i \in 0..h'. \langle n_0, \ldots, n_{\varphi'(i)} \rangle \rightarrow_{\bar{\chi}} [m'_0, \ldots, m'_i]
\]
Following the construction of lemma 3.6.6, we define $\varphi : 0..h \rightarrow 0..k$ as:

$$\varphi(x) = \begin{cases} k & \text{if } x = h \\ \varphi'(x) & \text{otherwise} \end{cases}$$

Then, if $i < h$, we have that $i < h' - 1$, and:

$$\langle n_0, \ldots, n_{\varphi(i)} \rangle = \langle n_0, \ldots, n_{\varphi'(i)} \rangle \rightarrow^\chi m_0' \ldots m_k' = [m_0, \ldots, m_k]$$

□

Lemma 3.6.8. The evaluation of $\tilde{\chi}$ is deterministic, i.e.:

$$\tilde{\pi} \rightarrow^{\tilde{\chi}} \sigma \land \tilde{\pi} \rightarrow^{\tilde{\chi}} \sigma' \implies \sigma = \sigma'$$

for any concrete path $\tilde{\pi} \in \tilde{\Pi}$ and $\sigma, \sigma' \in \Sigma$.

Proof. Since, except for the $\tilde{\chi}_{\text{return}}$ rule, the rules of $\tilde{\chi}$ coincide with those of $\chi$, the proof is identical to the proof of lemma 3.5.9 for the base, $[\text{call}]$ and $[\text{transfer}]$ cases. Hence, we have to treat only the following case:

- case $[\text{return}]$: consider the two transitions:

  $\tilde{\pi}': n' \rightarrow^{\tilde{\chi}} \sigma' : m' : n' \leftrightarrow n \\
  \tilde{\pi} : n \rightarrow^{\tilde{\chi}} \sigma : n$

By the inductive hypothesis, we have $\sigma : m : n' = \sigma' : m' : n'$, which clearly implies $\sigma = \sigma'$ and $m = m'$. Then, also $\sigma : n = \sigma' : n$ does hold.

□

Counterexample 3.6.9. The evaluation of $\tilde{\chi}$ is not total on concrete paths.
Proof. To see why, consider the control flow graph in Fig. 3.3. Since \( n_0 \in \text{N}_{\text{entry}} \) and \( n_i \rightarrow n_{i+1} \) for \( i \in \{0, 2\} \), by applying the \( \tilde{\chi}_{\text{entry}} \) rule once and \( \tilde{\chi}_{\text{call}} \) three times, we obtain:

\[
\langle n_0, n_1, n_2, n_3 \rangle \rightarrow^{\tilde{\chi}} \langle n_0, n_1, n_2, n_3 \rangle
\]

Now, the return edge \( n_3 \rightarrow n_3 \) follows by the fact that \( w((n_1, n_2, n_3)) = 1 \) implies \( n_3 \in \rho(n_1, n_2) \), and \( n_1 \rightarrow n_3 \). Then, by rule \( \tilde{\chi}_{\text{return}} \), we have the following transition:

\[
\langle n_0, n_1, n_2, n_3, n_3 \rangle \rightarrow^{\tilde{\chi}} \langle n_0, n_1, n_2, n_3 \rangle, n_3 \leftarrow n_3
\]

It is evident that, since \( n_3 \rightarrow n_3 \) is a self-loop, we can append to the concrete path \( \langle n_0, n_1, n_2, n_3 \rangle \) a sequence \( \langle n_3 \rangle^* \) of arbitrary length, still obtaining a concrete path. Each time the \( \tilde{\chi}_{\text{return}} \) rule is applied, we derive a state whose length is an unit shorter: eventually, the state will become of unit length, and the \( \tilde{\chi}_{\text{return}} \) rule will no more be applicable. In our case, to attain this aim it suffices to repeat the foregoing process two times more:

\[
\langle n_0, n_1, n_2, n_3, n_3 \rangle \rightarrow^{\tilde{\chi}} \langle n_0, n_1, n_3 \rangle, n_3 \leftarrow n_3
\]

Again, if we append \( n_3 \) to \( \langle n_0, n_1, n_2, n_3, n_3, n_3 \rangle \), we obtain:

\[
\langle n_0, n_1, n_2, n_3, n_3, n_3 \rangle \rightarrow^{\tilde{\chi}} \langle n_0, n_3 \rangle, n_3 \leftarrow n_3
\]

At this point, none of the rules of \( \tilde{\chi} \) is applicable, so we cannot go any further. In conclusion, we have shown that the transition system \( \tilde{\chi} \) does not reach a terminal configuration on the concrete entry path \( \langle n_0, n_1, n_2, n_3, n_3, n_3 \rangle \).

Since we have proved that the evaluation of \( \tilde{\chi} \) is deterministic (but not total) on concrete paths, we can define the partial function \( \tilde{\chi} : \Pi \rightarrow \Sigma \) as:

\[
\tilde{\chi}(\bar{\pi}) = \sigma \quad \text{if} \quad \bar{\pi} \rightarrow^{\tilde{\chi}} \sigma
\]

Theorem 3.6.10. Let \( \bar{\pi} \) be a concrete path, such that \( \tilde{\chi}(\bar{\pi}) \) is defined. Then:

\[
|\tilde{\chi}(\bar{\pi})| = \bar{w}(\bar{\pi})
\]

Proof. The proof is carried out by mathematical induction on the length of the concrete entry path \( \bar{\pi} \). For the base case, if \( \bar{\pi} = \langle n \rangle \) and \( n \in \text{N}_{\text{entry}} \), then \( \bar{w}(\langle n \rangle) = 1 \) and \( |\tilde{\chi}(\langle n \rangle)| = |\langle n \rangle| = 1 \). For the inductive case, we proceed by case analysis on the last edge of the path:

- case [call]:

\[
\tilde{\chi}(\bar{\pi}' : n') = \sigma \quad n' \rightarrow n
\]

where \( \bar{\pi} = \bar{\pi}' : n' : n \). By the inductive hypothesis, we have \( |\tilde{\chi}(\bar{\pi}' : n')| = |\sigma| = \bar{w}(\bar{\pi}' : n') \). Then:

\[
|\tilde{\chi}(\bar{\pi}' : n' : n)| = |\sigma : n| = |\sigma| + 1 = \bar{w}(\bar{\pi}' : n') + 1 = \bar{w}(\bar{\pi}' : n' : n)
\]
3.6. CONCRETE PATHS

• case [transfer]:

\[
\chi(\pi': n') = \sigma : n' \quad \text{n' ---} \quad n
\]

\[
\chi(\pi': n': n) = \sigma : n
\]

where \( \tilde{\pi} = \pi' : n' : n \)

By the inductive hypothesis, we have \( |\chi(\pi': n')| = |\pi : n'| = \tilde{w}(\pi': n') \). Then:

\[
|\chi(\pi': n': n)| = |\sigma : n| = |\sigma : n'| = \tilde{w}(\pi': n') = \tilde{w}(\pi': n': n)
\]

• case [return]:

\[
\chi(\pi': m) = \sigma : n' : m \quad m \rightarrow n
\]

\[
\chi(\pi': m : n) = \sigma : n
\]

By the inductive hypothesis, we have \( |\chi(\pi': m)| = |\sigma : n' : m| = \tilde{w}(\pi': m) \). Then:

\[
|\chi(\pi': m : n)| = |\sigma : n| = |\sigma : n'| - 1 = \tilde{w}(\pi': m) - 1 = \tilde{w}(\pi': m : n)
\]

\(\Box\)

**Definition 3.6.11.** Let \( \tilde{\pi} = \langle n_0, \ldots, n_k \rangle \) be a concrete path, \( 0 \leq i < j \leq k - 1 \), and \( \ell(n_i) = \text{call}, \ell(n_j) = \text{return} \). Then we say that \( n_j \) is **bound** by \( n_i \) in \( \tilde{\pi} \), and write \( n_i \triangleright \tilde{\pi} n_j \), when \( \tilde{w}(\langle n_{i+1}, \ldots, n_j \rangle) = 1 \) and:

\[
\forall i' < j. \quad \ell(n_{i'}) = \text{call} \land \tilde{w}(\langle n_{i'+1}, \ldots, n_j \rangle) = 1 \implies i' \leq i
\]

**Lemma 3.6.12.** Let \( \tilde{\pi} = \langle n_0, \ldots, n_k \rangle \) be a concrete path, and \( i, i', j, j' \in 0..k \) such that \( \max\{i, i'\} < \min\{j, j'\} \). Then:

\[
n_i \triangleright \tilde{\pi} n_j \land n_{i'} \triangleright \tilde{\pi} n_{j'} \implies i = i'
\]

\[
n_i \triangleright \tilde{\pi} n_j \land n_i \triangleright \tilde{\pi} n_j' \implies j = j'
\]

**Proof.** For the first part, hypotheses \( n_i \triangleright \tilde{\pi} n_j \) and \( n_{i'} \triangleright \tilde{\pi} n_{j'} \) imply:

\[
\tilde{w}(\langle n_{i+1}, \ldots, n_j \rangle) = 1 = \tilde{w}(\langle n_{i'+1}, \ldots, n_j \rangle) \implies i \leq i' \land i' \leq i \implies i = i'
\]

For the second part, assume, by contradiction, that \( j < j' \). By hypotheses \( n_i \triangleright \tilde{\pi} n_j \) and \( n_i \triangleright \tilde{\pi} n_j' \), after simple calculations we obtain:

\[
\tilde{w}(\langle n_{i+1}, \ldots, n_j \rangle) = 1 = \tilde{w}(\langle n_{i+1}, \ldots, n_{j'} \rangle) \implies \sum_{y=i}^{j'-1} w(n_y, n_{y+1}) = 0
\]

Since \( n_j \) is a return node, we have \( n_j \leftarrow n_{j+1} \). So \( w(n_j, n_{j+1}) = -1 \), and:

\[
\sum_{y=j}^{j'-1} w(n_y, n_{y+1}) = 1
\]  \hspace{1cm} (3.6)
Now, for $x \in j + 1..j'$, define the function:

$$f[x] = \sum_{y=x}^{j'-1} w(n_y, n_{y+1})$$

By definition, we have $f(j') = 0$, and $f(j + 1) = 1$ follows by equation (3.6). Actually $f$ is a step function, where the width of a jump between two consecutive steps is at most 1, i.e.:

$$|f(x + 1) - f(x)| \leq 1 \quad (3.7)$$

This condition ensures the existence of intermediate values, that is, for any $x_a < x_b$ in $j + 1..j'$, and any $y$ between $f(x_a)$ and $f(x_b)$, some $x \in x_a..x_b$ exists such that $f(x) = y$. Now take:

$$x^* = \max \{ x \in j + 1..j' | f(x) = 1 \} \quad (3.8)$$

Observe that $x^*$ is well-defined, as $f(j + 1) = 1$ by (3.6). Besides, it must be $f(x^* + 1) = 0$. By condition (3.7), we know that $f(x^* + 1)$ can only assume three values, i.e. $0, 1$ or 2. The second option is prevented, as it contradicts $x^*$ being defined as the greatest index for which $f$ evaluates to 1. Indeed, also the last option is prevented, because, by existence of intermediate values, $f(j') = 0$ implies $2x \in x^* + 1..j' - 1$. $f(x) = 1$. Again, this would violate definition (3.8). Now, by definition of $f$, we have:

$$f(x^*) = f(x^* + 1) + w(n_{x^*}, n_{x^* + 1})$$

Therefore $w(n_{x^*}, n_{x^* + 1}) = 1$, and $n_{x^*} \rightarrow n_{x^* + 1}$ as only call edges have unit weight. Actually, we have proved that $\ell(n_{x^*}) = \text{call}$ and $\bar{w}(\langle n_{x^* + 1}, \ldots, n_i \rangle) = 1$: by definition 3.6.11, this is in contradiction with hypothesis $n_i \xrightarrow{\bar{\pi}} n_i'$, because $x^* > j > i$. 

**Lemma 3.6.13.** Let $\bar{\pi} = \langle n_0, \ldots, n_i, \ldots, n_j \rangle$ be a concrete path, such that $\ell(n_i) = \text{return}$. Then:

$$\bar{\chi}(\bar{\pi}) = \sigma : n_i : n_j \implies n_i \xrightarrow{\bar{\pi}} n_j$$

**Proof.** We prove the stronger result:

$$\bar{\chi}(\bar{\pi}) = \sigma : n_i : n_j \implies n_i \rightarrow n_{i+1} \quad (3.9a)$$

$$\wedge \quad \bar{w}(\langle n_{i+1}, \ldots, n_i \rangle) = 1 \quad (3.9b)$$

$$\wedge \quad \forall i' > i. \ell(n_{i'}) = \text{call} \implies \bar{w}(\langle n_{i'+1}, \ldots, n_i \rangle) < 1 \quad (3.9c)$$

By definition 3.6.11 and hypothesis $\ell(n_i) = \text{return}$, this clearly implies $n_i \xrightarrow{\bar{\pi}} n_i$. The proof of the stronger statement is carried out by mathematical induction on the length of the concrete path. For the base case, if $\sigma = [], n_i \in N_{entry}$, $\ell(n_i) = \text{call}$ and $n_i \rightarrow n_i$, then $\bar{w}(\langle n_{i+1} \rangle) = 1$, while both conjuncts (3.9a) and (3.9c) are trivially satisfied. For the inductive case, we proceed by case analysis on the last rule used to derive $\bar{\chi}(\bar{\pi}) = \sigma : n_i : n_j$, yielding:

- case [call]:

$$\bar{\chi}(\langle n_0, \ldots, n_{i-1} \rangle) = \sigma : n_{i-1} \quad n_{i-1} \rightarrow n_i$$

$$\bar{\chi}(\langle n_0, \ldots, n_i \rangle) = \sigma : n_{i-1} : n_i$$

Here we have $i = j - 1$: hence, $\bar{w}(\langle n_{i+1}, \ldots, n_i \rangle) = \bar{w}(\langle n_i \rangle) = 1$ satisfies (3.9b), while (3.9a) and (3.9c) are trivially true.
3.6. CONCRETE PATHS

- case [check]:

\[ \bar{X}(\langle n_0, \ldots, n_{i-1} \rangle) = \sigma : n_i : n_{j-1} \quad n_{j-1} \rightarrow n_j \]

\[ \bar{X}(\langle n_0, \ldots, n_i \rangle) = \sigma : n_i : n_i \]

By the inductive hypothesis applied to the concrete path \( \langle n_0, \ldots, n_{i-1} \rangle \), and premise \( \bar{X}(\langle n_0, \ldots, n_{i-1} \rangle) = \sigma : n_i : n_{j-1} \), we have:

\[ n_i \rightarrow n_{i+1} \quad (3.10a) \]

\[ \bar{w}(\langle n_{i+1}, \ldots, n_{j-1} \rangle) = 1 \quad (3.10b) \]

\[ \forall i' > i. \ell(\langle n_{i'} \rangle) = \text{call} \implies \bar{w}(\langle n_{i'+1}, \ldots, n_{j-1} \rangle) < 1 \quad (3.10c) \]

Therefore (3.9a) is trivially implied by (3.10a), while (3.9b) is fixed by:

\[ \bar{w}(\langle n_{i+1}, \ldots, n_i \rangle) = \bar{w}(\langle n_{i+1}, \ldots, n_{j-1} \rangle) + w(n_{j-1}, n_i) \quad \text{by def. } \bar{w} \]

\[ = \bar{w}(\langle n_{i+1}, \ldots, n_{j-1} \rangle) \quad \text{as } n_{j-1} \rightarrow n_i \]

\[ = 1 \quad \text{by (3.10b)} \]

To prove (3.9c), let \( i' > i \) such that \( \ell(\langle n_{i'} \rangle) = \text{call} \). Then:

\[ \bar{w}(\langle n_{i'+1}, \ldots, n_i \rangle) = \bar{w}(\langle n_{i'+1}, \ldots, n_{j-1} \rangle) + w(n_{j-1}, n_i) \quad \text{by def. } \bar{w} \]

\[ = \bar{w}(\langle n_{i'+1}, \ldots, n_{j-1} \rangle) \quad \text{as } n_{j-1} \rightarrow n_i \]

\[ < 1 \quad \text{by (3.10c)} \]

- case [return]:

\[ \bar{X}(\langle n_0, \ldots, n_{i-1} \rangle) = \sigma : n_i : m : n_{j-1} \quad n_{j-1} \leftarrow n_j \]

\[ \bar{X}(\langle n_0, \ldots, n_i \rangle) = \sigma : n_i : n_i \]

By lemma 3.6.6, it must be \( m = n_h \), where \( h = \varphi(|\sigma|+1) \) and \( i < h < j - 1 \). Again, by the inductive hypothesis applied to the concrete path \( \langle n_0, \ldots, n_h, \ldots, n_{j-1} \rangle \), and premise \( \bar{X}(\langle n_0, \ldots, n_{j-1} \rangle) = \sigma' : n_h : n_{j-1} \) (where \( \sigma' = \sigma : n_i \)), we have:

\[ n_h \rightarrow n_{h+1} \quad (3.11a) \]

\[ \bar{w}(\langle n_{h+1}, \ldots, n_{j-1} \rangle) = 1 \quad (3.11b) \]

\[ \forall i' > h. \ell(\langle n_{i'} \rangle) = \text{call} \implies \bar{w}(\langle n_{i'+1}, \ldots, n_{j-1} \rangle) < 1 \quad (3.11c) \]

On the other hand, by lemma 3.6.7, we have \( \bar{X}(\langle n_0, \ldots, n_h \rangle) = \sigma : n_i : n_h \). Hence, the inductive hypothesis can also be applied to \( \langle n_0, \ldots, n_h \rangle \), and we obtain:

\[ n_i \rightarrow n_{i+1} \quad (3.12a) \]

\[ \bar{w}(\langle n_{i+1}, \ldots, n_h \rangle) = 1 \quad (3.12b) \]

\[ \forall i' > i. \ell(\langle n_{i'} \rangle) = \text{call} \implies \bar{w}(\langle n_{i'+1}, \ldots, n_h \rangle) < 1 \quad (3.12c) \]

Then, conjunct (3.9a) is clearly implied by (3.12a), while (3.9b) is fixed by:

\[ \bar{w}(\langle n_{i+1}, \ldots, n_i \rangle) = \bar{w}(\langle n_{i+1}, \ldots, n_{j-1} \rangle) + w(n_{j-1}, n_i) \quad \text{by def. } \bar{w} \]

\[ = \bar{w}(\langle n_{i+1}, \ldots, n_{j-1} \rangle) - 1 \quad \text{as } n_{j-1} \leftarrow n_j \]

\[ = \bar{w}(\langle n_{i+1}, \ldots, n_h \rangle) + \bar{w}(\langle n_{h+1}, \ldots, n_{j-1} \rangle) \]

\[ + w(n_h, n_{h+1}) - 2 \quad \text{by lemma 3.6.4} \]

\[ = \bar{w}(\langle n_{i+1}, \ldots, n_h \rangle) + \bar{w}(\langle n_{h+1}, \ldots, n_{j-1} \rangle) - 1 \quad \text{by (3.11a)} \]

\[ = \bar{w}(\langle n_{i+1}, \ldots, n_h \rangle) + 1 - 1 \quad \text{by (3.11b)} \]

\[ = 1 + 1 - 1 \quad \text{by (3.12b)} \]

\[ = 1 \]
To prove (3.9c), let \( i' \geq i \) such that \( I[n_i'] = \text{call} \). If \( i' < h \), then:

\[
\begin{align*}
\tilde{w}(\langle n_{i'+1}, \ldots, n_i \rangle) &= \tilde{w}(\langle n_{i'+1}, \ldots, n_{i-1} \rangle) + w(n_{i-1}, n_i) & \text{by def. } \tilde{w} \\
&= \tilde{w}(\langle n_{i'+1}, \ldots, n_{i-1} \rangle) - 1 & \text{as } n_{i-1} \rightarrow n_i \\
&= \tilde{w}(\langle n_{i'+1}, \ldots, n_h \rangle) + \tilde{w}(\langle n_{h+1}, \ldots, n_{i-1} \rangle) \\
&\quad + w(n_h, n_{h+1}) - 2 & \text{by lemma 3.6.4} \\
&= \tilde{w}(\langle n_{i'+1}, \ldots, n_h \rangle) + \tilde{w}(\langle n_{h+1}, \ldots, n_{i-1} \rangle) - 1 & \text{by (3.11a)} \\
&= \tilde{w}(\langle n_{i'+1}, \ldots, n_h \rangle) + 1 - 1 & \text{by (3.11b)} \\
&< 1 + 1 - 1 & \text{by (3.12c)} \\
&= 1
\end{align*}
\]

Otherwise, if \( i' > h \), we obtain:

\[
\begin{align*}
\tilde{w}(\langle n_{i'+1}, \ldots, n_i \rangle) &= \tilde{w}(\langle n_{i'+1}, \ldots, n_{i-1} \rangle) + w(n_{i-1}, n_i) & \text{by def. } \tilde{w} \\
&= \tilde{w}(\langle n_{i'+1}, \ldots, n_{i-1} \rangle) - 1 & \text{as } n_{i-1} \rightarrow n_i \\
&< 1 - 1 & \text{by (3.11c)} \\
&< 1
\end{align*}
\]

\[\square\]

**Definition 3.6.14.** Let \( \tilde{\pi} = \langle n_0, \ldots, n_k \rangle \) be a concrete path. We define the transition system \( \alpha = \langle \tilde{\Pi} \cup \Pi, \Pi, \rightarrow_\alpha \rangle \) by the following rules:

\[
\begin{align*}
k &= 0 \\
\tilde{\pi} &\rightarrow_\alpha \langle n_0 \rangle \\
\langle n_0, \ldots, n_{k-1} \rangle &\rightarrow_\alpha \pi \quad (n_{k-1}, n_k) \in E \\
\tilde{\pi} &\rightarrow_\alpha \pi : n_k \\
\langle n_0, \ldots, n_{i'} \rangle &\rightarrow_\alpha \pi \quad n_{i'} \rightarrow_{\tilde{\alpha}} n_{k-1} \quad n_{i'} \rightarrow n_k \\
\tilde{\pi} &\rightarrow_\alpha \pi : n_k
\end{align*}
\]

**Lemma 3.6.15.** Let \( \tilde{\pi} = \langle n_0, \ldots, n_k \rangle \) be a concrete path, such that:

\[\tilde{\pi} \rightarrow_\alpha \langle m_0, \ldots, m_h \rangle\]

Then, a strictly increasing function \( \varphi : 0..h \rightarrow 0..k \) exists, satisfying:

\[
\forall i \in 0..h. \ m_i = n_{\varphi(i)} \land \varphi(h) = k
\]

**Proof.** The proof is carried out by mathematical induction on the length of the concrete path \( \tilde{\pi} \). For the base case, if \( \tilde{\pi} = \langle n_0 \rangle \), then it must be \( \tilde{\pi} \rightarrow_\alpha \langle n_0 \rangle \), because only the first rule of \( \alpha \) is applicable. Then, the function \( \varphi = \{ 0 \mapsto 0 \} \) trivially satisfies (3.13). For the inductive case, we proceed by case analysis on the last edge of the concrete path:
3.6. CONCRETE PATHS

- cases [call, transfer]:

\[
\begin{align*}
\langle n_0, \ldots, n_{k-1} \rangle & \rightarrow_\alpha \pi \\
\langle n_0, \ldots, n_k \rangle & \rightarrow_\alpha \pi : n_k
\end{align*}
\]

By the inductive hypothesis, we have a strictly increasing function:

\[
\varphi' : 0..|\pi| - 1 \rightarrow 0..k - 1
\]

which satisfies (3.13). Then, we define \( \varphi : 0..|\pi| \rightarrow 0..k \) as follows:

\[
\varphi(i) = \begin{cases} 
  k & \text{if } i = |\pi| \\
  \varphi'(i) & \text{otherwise}
\end{cases}
\]

Now, \( \varphi \) is strictly increasing on \( 0..|\pi| - 1 \), because, inside that interval, it coincides with \( \varphi' \). Since \( \varphi'(|\pi| - 1) = k - 1 \) by (3.13), we have that:

\[
\varphi(|\pi|) = k > k - 1 = \varphi'(|\pi| - 1) = \varphi(|\pi| - 1)
\]

Hence, \( \varphi \) is strictly increasing inside the whole domain \( 0..|\pi| \). Since we also have \( n_k = n_{\varphi(|\pi|)} \), we can conclude that \( \varphi \) satisfies (3.13).

- case [return]:

\[
\begin{align*}
\langle n_0, \ldots, n_i \rangle & \rightarrow_\alpha \pi \\
\langle n_0, \ldots, n_k \rangle & \rightarrow_\alpha \pi : n_k
\end{align*}
\]

By the inductive hypothesis, we have a strictly increasing function:

\[
\varphi' : 0..|\pi| - 1 \rightarrow 0..i^*
\]

which satisfies (3.13). Then, we define \( \varphi : 0..|\pi| \rightarrow 0..k \) as follows:

\[
\varphi(i) = \begin{cases} 
  k & \text{if } i = |\pi| \\
  \varphi'(i) & \text{otherwise}
\end{cases}
\]

Now, \( \varphi \) is strictly increasing on \( 0..|\pi| - 1 \), because, inside that interval, it coincides with \( \varphi' \). Since \( \varphi'(|\pi| - 1) = i^* \) by (3.13) and \( k > i^* \) by definition 3.6.11, we have:

\[
\varphi(|\pi|) = k > i^* = \varphi'(|\pi| - 1) = \varphi(|\pi| - 1)
\]

Hence, \( \varphi \) is strictly increasing inside the whole domain \( 0..|\pi| \). Since we also have \( n_k = n_{\varphi(|\pi|)} \), we can conclude that \( \varphi \) satisfies (3.13).

\[ \square \]

**Lemma 3.6.16.** The evaluation of \( \alpha \) is deterministic, i.e.:

\[
\hat{\pi} \rightarrow_\alpha \pi \land \hat{\pi} \rightarrow_\alpha \pi' \implies \pi = \pi'
\]

for any concrete path \( \hat{\pi} \in \hat{\Pi} \) and abstract path \( \pi, \pi' \in \Pi \).

**Proof.** The proof is carried out by mathematical induction on the length of the concrete path \( \hat{\pi} \). For the base case, let \( \hat{\pi} = \langle n_0 \rangle \). If \( \hat{\pi} \rightarrow_\alpha \pi \) and \( \hat{\pi} \rightarrow_\alpha \pi' \), then it must be \( \pi = \pi' = \langle n_0 \rangle \), because only the first rule of \( \alpha \) is applicable. For the inductive case, let \( \hat{\pi} = \langle n_0, \ldots, n_k \rangle \). We proceed by case analysis on the last edge of the concrete path:
Figure 3.4: Control flow graph for counterexample 3.6.17

- cases [call, transfer]: consider the two transitions:
  \[
  \langle n_0, \ldots, n_{k-1} \rangle \rightarrow_{\alpha} \pi \quad (n', n) \in E
  \]
  \[
  \langle n_0, \ldots, n_k \rangle \rightarrow_{\alpha} \pi : n_k
  \]
  \[
  \langle n_0, \ldots, n_{k-1} \rangle \rightarrow_{\alpha} \pi' \quad (n', n) \in E
  \]
  \[
  \langle n_0, \ldots, n_k \rangle \rightarrow_{\alpha} \pi' : n_k
  \]
  By the inductive hypothesis, we have \( \pi = \pi' \), then \( \pi : n_k = \pi' : n_k \) holds, too.

- case [return]: consider the two transitions:
  \[
  \langle n_0, \ldots, n_{i-1} \rangle \rightarrow_{\alpha} \pi \quad n_i \vdash_{\pi} n_{k-1} \quad n_i \rightarrow n_k
  \]
  \[
  \langle n_0, \ldots, n_k \rangle \rightarrow_{\alpha} \pi : n_k
  \]
  \[
  \langle n_0, \ldots, n_{i-1} \rangle \rightarrow_{\alpha} \pi' \quad n_i \vdash_{\pi'} n_{k-1} \quad n_i \rightarrow n_k
  \]
  \[
  \langle n_0, \ldots, n_k \rangle \rightarrow_{\alpha} \pi' : n_k
  \]
  By the inductive hypothesis, we have \( \pi = \pi' \), which clearly implies \( \pi : n_k = \pi' : n_k \).

\[\square\]

**Counterexample 3.6.17.** The evaluation of \( \alpha \) is not total on concrete paths.

*Proof.* To see why, consider the control flow graph in Fig. 3.4. If we take the concrete path \( \bar{\pi} = \langle n_0, n_2, n_4 \rangle \), then \( \alpha(\langle n_0, n_2 \rangle) \) is defined, but \( \alpha(\bar{\pi}) \) is not, because \( n_0 \vdash_{\alpha} n_2 \) but \( n_0 \not\rightarrow_{\alpha} n_4 \).

\[\square\]

Since we have proved that the evaluation of \( \alpha \) is deterministic (but not total) on concrete paths, we can define the partial function \( \alpha : \tilde{\Pi} \rightarrow \Pi \) as:

\[
\alpha(\bar{\pi}) = \pi \quad \text{if} \quad \bar{\pi} \rightarrow_{\alpha} \pi
\]
Lemma 3.6.18. Let \( \bar{\pi} \) be a concrete path, such that \( \alpha(\bar{\pi}) \) is defined. Then:

\[
\bar{\pi} \in \Pi_{n_0,n_k} \implies \alpha(\bar{\pi}) \in \Pi_{n_0,n_k}
\]

for any \( n_0, n_k \in \mathbb{N} \).

Proof. The proof is carried out by mathematical induction on the length of the concrete path \( \bar{\pi} \). For the base case, if \( \bar{\pi} = \langle n_0 \rangle \in \Pi_{n_0,n_0} \), then \( \alpha(\bar{\pi}) = \langle n_0 \rangle \), and the statement holds because \( \langle n_0 \rangle \in \Pi_{n_0,n_0} \). For the inductive case, let \( \bar{\pi} = \langle n_0, \ldots, n_k \rangle \). We then proceed by case analysis on the last edge of the concrete path:

- cases [call, transfer]:

\[
\begin{align*}
\alpha(\langle n_0, \ldots, n_{k-1} \rangle) = \pi & \quad (n_{k-1}, n_k) \in E \\
\alpha(\langle n_0, \ldots, n_k \rangle) = \pi : n_k
\end{align*}
\]

By the inductive hypothesis, we have \( \pi = \alpha(\langle n_0, \ldots, n_{k-1} \rangle) \in \Pi_{n_0,n_{k-1}} \). Therefore, \( (n_{k-1}, n_k) \in E \) implies \( \pi : n_k \in \Pi_{n_0,n_k} \).

- case [return]:

\[
\begin{align*}
\alpha(\langle n_0, \ldots, n_{i-1} \rangle) = \pi & \quad n_i \rightarrow_{\bar{\pi}} n_{k-1} \quad n_i \rightarrow n_k \\
\alpha(\langle n_0, \ldots, n_k \rangle) = \pi : n_k
\end{align*}
\]

By the inductive hypothesis, we have \( \pi = \alpha(\langle n_0, \ldots, n_{i-1} \rangle) \in \Pi_{n_0,n_i} \). Therefore, by premise \( n_i \rightarrow n_k \), it follows \( \pi : n_k \in \Pi_{n_0,n_k} \).

\[\square\]

Theorem 3.6.19. Let \( \bar{\pi} \) be a concrete path, such that \( \alpha(\bar{\pi}) \) is defined. Then:

\[
\bar{w}(\bar{\pi}) = w(\alpha(\bar{\pi}))
\]

Proof. The proof is carried out by mathematical induction on the length of the concrete path. For the base case, if \( \bar{\pi} = \langle n \rangle \) then \( \alpha(\langle n \rangle) = \langle n \rangle \), and \( \bar{w}(\langle n \rangle) = 1 = w(\langle n \rangle) \). For the inductive case, let \( \bar{\pi} = \langle n_0, \ldots, n_k \rangle \). We then proceed by case analysis on the last edge of the concrete path:

- cases [call, transfer]:

\[
\begin{align*}
\alpha(\langle n_0, \ldots, n_{k-1} \rangle) = \pi & \quad (n_{k-1}, n_k) \in E \\
\alpha(\langle n_0, \ldots, n_k \rangle) = \pi : n_k
\end{align*}
\]

By the inductive hypothesis, we have \( \bar{w}(\langle n_0, \ldots, n_{k-1} \rangle) = w(\pi) \). Moreover, by lemma 3.6.18, \( \langle n_0, \ldots, n_{k-1} \rangle \in \Pi_{n_0,n_{k-1}} \) implies \( \pi \in \Pi_{n_0,n_{k-1}} \). Then:

\[
\bar{w}(\langle n_0, \ldots, n_k \rangle) = \bar{w}(\langle n_0, \ldots, n_{k-1} \rangle) + w(n_{k-1}, n_k)
\]

by def. \( \bar{w} \)

\[
= w(\pi) + w(n_{k-1}, n_k)
\]

by ind. hyp.

\[
= w(\pi : n_k)
\]

by def. \( w \)
\[ \begin{array}{c}
\{P\} \\
\downarrow \hspace{1cm} \Updownarrow \\
n_0: \text{call} \\
\downarrow \hspace{1cm} \Updownarrow \\
n_1: \text{call} \overset{\cdots}{\longrightarrow} n_2: \text{check}(P) \\
\downarrow \hspace{1cm} \Updownarrow \\
n_3: \text{return} \\
\end{array} \]

Figure 3.5: Control flow graph for counterexample 3.6.20

- case [return]:

\[
\alpha(n_0, \ldots, n_t) = \pi \quad n_1 \xrightarrow{\pi} n_{k-1} \quad n_t \rightarrow n_k
\]

\[\alpha(n_0, \ldots, n_k) = \pi : n_k\]

By the inductive hypothesis, we have \( \hat{w}(n_0, \ldots, n_{t'}) = w(\pi) \). Moreover, by lemma 3.6.18, \( \langle n_0, \ldots, n_t \rangle \in \tilde{\Pi}_{n_0, n_t} \) implies \( \pi \in \pi_{n_0, n_t} \). Then:

\[
\begin{align*}
\hat{w}(n_0, \ldots, n_k) &= \hat{w}(n_0, \ldots, n_{k-1}) + w(n_{k-1}, n_k) \\
&= \hat{w}(n_0, \ldots, n_{k-1}) - 1 \\
&= \hat{w}(n_0, \ldots, n_{t'}) + \hat{w}(n_{t'+1}, \ldots, n_{k-1}) + w(n_{t'+1}, n_{k-1}) - 1 \\
&= \hat{w}(n_0, \ldots, n_{t'}) + w(n_{t'+1}, n_{k}) - 1 \\
&= w(\pi) + w(n_{t'}, n_k) \\
&= w(\pi : n_k)
\end{align*}
\]

\[\hat{w}(n_0, \ldots, n_k) = w(\pi) \quad \text{by def.} \quad \hat{w}
\]

Counterexample 3.6.20. The evaluation of \( \alpha \) is not surjective, i.e. some control flow graph \( G \) and abstract path \( \pi \in \Pi(G) \) exist, such that:

\[ \exists \exists \pi \in \tilde{\Pi}. \pi = \alpha(\tilde{\pi}) \]

Proof. To see why, consider the control flow graph in Fig. 3.5: the return edge \( n_3 \rightarrow n_4 \) follows by the fact that \( w(n_2, n_1, n_3) = 1 \) implies \( n_3 \in p(n_1, n_2) \), and \( n_1 \rightarrow n_4 \).

Now, any concrete (non-empty) path \( \tilde{\pi} \) leaving from \( n_1 \) is of the form:

\[ n_1 : (n_2, n_1)^* \mid (n_1, n_2)^+ \]

Therefore, the abstract path \( \pi = \langle n_1, n_4 \rangle \) does not belong to the image of \( \alpha \), because, by lemma 3.6.18, for this to be possible it should be \( \tilde{\pi} \in \tilde{\Pi}_{n_1, n_4} \). \( \square \)
3.7 Valid paths

Definition 3.7.1. A concrete path \( \hat{\pi} = \langle n_0, \ldots, n_k \rangle \) is valid if, for any \( j < k \):

\[
n_j \leftrightarrow n_{j+1} \quad \implies \quad \exists i \in \langle 0..k \rangle. \ n_i \xrightarrow{\hat{\pi}} n_j \ \land \ n_i \xrightarrow{} n_{j+1}
\]

We denote with \( \tilde{\Pi}^n_{n_0,n_k} \) the set of all valid paths from \( n_0 \) to \( n_k \), and with \( \tilde{\Pi}^v \) the set of all valid paths. We denote with \( \bar{\Pi}^n \) the set of all valid entry paths leading to \( n \), and with \( \bar{\Pi}^v_{\text{entry}} \) the set of all valid entry paths. We write \( \tilde{\Pi}^n_{n_0,n_k}(G), \bar{\Pi}^v(G), \tilde{\Pi}^n(G) \) and \( \bar{\Pi}^v_{\text{entry}}(G) \) when we want to make clear that the control flow graph under consideration is \( G \).

Lemma 3.7.2. Let \( \hat{\pi} = \langle n_0, \ldots, n_k \rangle \) be a valid path, and \( h \in \langle 0..k \rangle \). Then, also \( \langle n_0, \ldots, n_h \rangle \) is a valid path.

Proof. This result clearly follows by the fact that \( n_i \xrightarrow{\hat{\pi}} n_j \) implies \( i < j \).

Lemma 3.7.3. Let \( \hat{\pi} = \langle n_0, \ldots, n_k \rangle \) be a valid path, and \( n_i \xrightarrow{\hat{\pi}} n_j \) for some \( 0 \leq i < j \leq k \). Then, also \( \langle n_i, \ldots, n_j \rangle \) is a valid path.

Proof. Let \( \hat{\pi}' = \langle n_i, \ldots, n_j \rangle \), and choose an index \( j' \in \langle i..j-1 \rangle \) such that \( n_{j'} \mapsto n_{j'+1} \). If \( j' = j \) or no return node exists between \( i \) and \( j \), we are done. Otherwise, for \( x \in \langle i+1..j \rangle \), let:

\[
g(x) = \sum_{y=i+1}^{x-1} w(n_y, n_{y+1})
\]

By definition, we have \( g(i+1) = 0 \), and \( g(j) = 0 \) follows by the hypothesis \( n_i \xrightarrow{\hat{\pi}} n_j \). Similarly to the proof of lemma 3.6.12, \( g(x) \) is a step function which ensures the existence of intermediate values. Now define:

\[
G = \{ x \in i+1..j | g(x) < 0 \}
\]

By contradiction, assume \( G \) is non-empty, and let \( x^* = \min G \). Then, it must be \( g(x^*) = -1 \): to see why, assume \( g(x^*) < -h \), for some \( h > 1 \). Then, by the fact \( g(i+1) = 0 \), existence of intermediate values implies that \( \exists x \in \langle i+1..x^*-1 \rangle. \ g(x) = -h + 1 < 0 \), contradicting the definition of \( x^* \). Besides, \( g(x^* - 1) = 0 \): this happens because \( g(x^* - 1) \) can only take three values (i.e., 0, -1 and -2), and both the second and the last one are prevented by definition of \( x^* \). Actually, we have proved that \( \ell(n_{x^* - 1}) = \text{return} \) and \( w((n_{i+1}, \ldots, n_{x^* - 1})) = 1 \). By definition 3.6.11, this would imply \( n_i \xrightarrow{\hat{\pi}} n_{x^* - 1} \): then, by lemma 3.6.12, we should also have \( x^* - 1 = j \), as \( n_i \xrightarrow{n_j} \) by hypothesis. Indeed, it is \( x^* - 1 < j \), hence we have a contradiction with assumption \( G \neq \emptyset \): then, it must be \( g(x) \geq 0 \) for any \( x \in i+1..j \). Since \( n_i \xrightarrow{n_j} \), we have \( n_i \xrightarrow{} n_{i+1} \). Then, if we let \( x = j' \), we actually deduce:

\[
\sum_{y=i}^{j'-1} w(n_y, n_{y+1}) = 1 + g(j') > 0 \quad (3.14)
\]
Now, for \( x \in i \cdot j' \), define the function:

\[
f(x) = \sum_{y=x}^{j'-1} w(n_y, n_{y+1})
\]

By definition, we have \( f(j') = 0 \), and \( f(i) = h > 0 \) follows by equation (3.14). Again, \( f(x) \) is a step function which ensures the existence of intermediate values. Now define:

\[
F = \{ x \in i \cdot j' | f(x) = 1 \}
\]

By existence of intermediate values, we have that \( \exists x \in i \cdot j' - 1 \). \( f(x) = 1 \), therefore \( F \) is non-empty. Now, define \( i' = \max F \). Since \( f(i') = 1 \), we have that \( f(i' + 1) \) can only be 0, 1 or 2. The second option is prevented, as it contradicts \( i' \) being defined as the greatest index for which \( f \) evaluates to 1. Indeed, also the last option is prevented, because, by existence of intermediate values, \( f(j') = 0 \) implies \( \exists x \in i' + 1 \cdot j' - 1 \). \( f(x) = 1 \). Again, this would contradict definition of \( i' \). By definition of \( f(x) \), we have:

\[
f(i') = f(i' + 1) + w(n_{i'}, n_{i'+1})
\]

Therefore \( w(n_{i'}, n_{i'+1}) = 1 \), and, since only call edges have unit weight, \( n_{i'} \rightarrow n_{i'+1} \). Actually, we have proved that \( f(n_{i'}) = \text{call} \) and \( \hat{w}(n_{i'+1}, \ldots, n_{i'}) = 1 \). Moreover, we have that \( \hat{w}(n_{i'+1}, \ldots, n_{i'}) \neq 1 \) for any \( i'' > i' \). By definition 3.6.11, this indeed suffices to state \( n_{i'} \leftarrow \pi, n_{i'} \). On the other hand, by definition 3.7.1, \( n_{i' \rightarrow} \rightarrow n_{i'+1} \) immediately follows by the fact that \( \pi \) is valid and hypothesis \( n_{i'} \leftarrow n_{i'+1} \). Then, also \( \pi' \) is valid. \( \square \)

**Lemma 3.7.4.** Let \( \tilde{\pi} = \langle n_0, \ldots, n_i, \ldots, n_j \rangle \) be a valid entry path, such that \( \tilde{\chi}(\langle n_0, \ldots, n_i \rangle) = \sigma : n_i \) for some state \( \sigma \in \Sigma \). Then:

\[
n_i \leftarrow \tilde{\pi}, n_j \implies \tilde{\chi}(\tilde{\pi}) = \sigma : n_i : n_j
\]

**Proof.** Assume \( n_i \leftarrow \pi, n_j \). We prove the stronger statement:

\[
\forall i' > i. \exists \pi \in \Sigma. \tilde{\chi}(\langle n_0, \ldots, n_i \rangle) = \sigma : n_i : n_{i'} \land |\pi| = \hat{w}(n_{i'+1}, \ldots, n_i) - 1
\tag{3.15}
\]

Observe that the main result follows from (3.15): in fact, if \( i' = j \), then we have:

\[
\tilde{\chi}(\langle n_0, \ldots, n_i \rangle) = \sigma : n_i : n_j \land |\pi| = \hat{w}(n_{i'+1}, \ldots, n_i) - 1
\]

By definition 3.6.11, \( n_i \leftarrow \tilde{\pi}, n_i \) implies \( \hat{w}(n_{i'+1}, \ldots, n_i) = 1 \), hence \( |\pi| = 0 \). Actually, this can only happen if \( \tau = [] \); therefore, \( \tilde{\chi}(\langle n_0, \ldots, n_i \rangle) = \sigma : n_i : n_i \). The proof of (3.15) is carried out by mathematical induction on the value of \( i' \). For the base case, if \( i' = i + 1 \), then \( \tilde{\chi}(\langle n_0, \ldots, n_i \rangle) = \sigma : n_i \) is ensured by the premises of the lemma. Hence, the following transition occurs:

\[
\tilde{\chi}(\langle n_0, \ldots, n_i \rangle) = \sigma : n_i \quad n_i \rightarrow n_{i+1}
\]

\[
\tilde{\chi}(\langle n_0, \ldots, n_{i+1} \rangle) = \sigma : n_i : n_{i+1}
\]

If we define \( \tau = [] \) we are done, because \( |\pi| = 0 = \hat{w}(n_{i+1}) - 1 \). For the inductive case, we assume the statement is true for an arbitrary \( i' \in i+1 \cdot j-1 \); then, we show it also holds for \( i'+1 \). We proceed by case analysis on the last edge of the path \( \langle n_0, \ldots, n_{i'+1} \rangle \), yielding:
3.7. VALID PATHS

- case [call]: by the inductive hypothesis, we have \( \hat{\mathcal{X}}((n_0, \ldots, n_{i'}) \tau \colon n_{i'} \mapsto n_{i'+1}) \). Hence, the following transition occurs:

\[
\hat{\mathcal{X}}((n_0, \ldots, n_{i'})) \mapsto n_{i'} \mapsto n_{i'+1}
\]

If we define \( \tau = \tau' : n_{i'}, \) then \( \hat{\mathcal{X}}((n_0, \ldots, n_{i'+1})) = \sigma : n_{i'} : n_{i'+1}, \) and:

\[
|\tau| = |\tau'| = 1 = \hat{\omega}(n_{i+1}, \ldots, n_{i+1}) - 1
\]

- case [check]: by the inductive hypothesis, we have \( \hat{\mathcal{X}}((n_0, \ldots, n_{i'})) = \sigma : n_{i'} : n_{i'+1}, \) with \( |\tau'| = \hat{\omega}(n_{i+1}, \ldots, n_{i+1}) \). Hence, the following transition occurs:

\[
\hat{\mathcal{X}}((n_0, \ldots, n_{i'})) \mapsto n_{i'} \mapsto n_{i'+1}
\]

If we define \( \tau = \tau' : n_{i'} \), then \( \hat{\mathcal{X}}((n_0, \ldots, n_{i'+1})) = \sigma : n_{i'} : n_{i'+1}, \) and:

\[
|\tau| = |\tau'| = 1 = \hat{\omega}(n_{i+1}, \ldots, n_{i+1}) - 1
\]

- case [return]: by the inductive hypothesis, \( \hat{\mathcal{X}}((n_0, \ldots, n_{i'})) = \sigma : n_{i'} : n_{i'+1}, \) with \( |\tau'| = \hat{\omega}(n_{i+1}, \ldots, n_{i+1}) \). Now, since \( \pi \) is a valid path and \( n_1 \rightarrow \pi n_1 \), by the proof of lemma 3.7.3 it turns out that \( \hat{\omega}(n_{i+1}, \ldots, n_{i'+1}) \geq 1 \). Since \( n_{i'} \mapsto n_{i'+1} \), we then have \( \hat{\omega}(n_{i+1}, \ldots, n_{i'+1}) \geq 1 \). Therefore \( |\tau'| > 0 \), so we can write \( \tau' = \tau'' : n_{i''} \) for some \( i'' \in \Sigma \) and \( i'' = i + 1 \). The following transition occurs:

\[
\hat{\mathcal{X}}((n_0, \ldots, n_{i'})) \mapsto n_{i'} \mapsto n_{i'+1}
\]

If we define \( \tau = \tau'' : n_{i''} \), then \( \hat{\mathcal{X}}((n_0, \ldots, n_{i'+1})) = \sigma : n_{i''} : n_{i'+1}, \) and:

\[
|\tau| = |\tau'| = 1 = \hat{\omega}(n_{i+1}, \ldots, n_{i+1}) - 1
\]

\[\square\]

**Lemma 3.7.5.** The evaluation of \( \hat{\mathcal{X}} \) is total on valid entry paths, i.e.:

\[\tilde{\pi} \in \mathcal{P}^*_{\text{entry}} \implies \exists \sigma \in \Sigma. \tilde{\pi} \rightarrow_{\hat{\mathcal{X}}} \sigma\]

**Proof.** We prove the stronger result:

\[\pi \in \tilde{\Pi}^*_{\text{entry}} \implies \exists \sigma \in \Sigma. \pi \rightarrow_{\hat{\mathcal{X}}} \sigma : n\]

The proof is carried out by mathematical induction on the length of the valid entry path. For the base case, if \( n \in \Pi^*_{\text{entry}} \), then \( \pi = [n] \in \tilde{\Pi}^*_{\text{entry}} \) by the \( \mathcal{H}^*_{\text{entry}} \) rule. For the inductive case, let \( \pi = \tilde{\pi}' : n' : n \), with \( \tilde{\pi}' \) possibly empty. By the inductive hypothesis applied to \( \tilde{\pi}' : n' \), it follows that:

\[\tilde{\pi}' : n' \in \tilde{\Pi}^*_{\text{entry}} \implies \exists \sigma' \in \Sigma. \tilde{\pi}' : n' \rightarrow_{\hat{\mathcal{X}}} \sigma' : n'\]

Now we proceed by case analysis on the edge \( \langle n', n \rangle \):
• case \texttt{[call]}: if \( n' \rightarrow n \), then the \( \hat{\chi}_\text{call} \) rule is applicable, and we obtain:

\[
\frac{\pi' : n' \rightarrow \chi \quad \sigma' : n' \quad n' \rightarrow n}{\pi' : n : n' \rightarrow \chi \quad \sigma' : n' : n}
\]

Then, the result is proved by letting \( \sigma = \sigma' : n' \).

• case \texttt{[transfer]}: if \( n' \rightarrow \rightarrow n \), we can apply the \( \hat{\chi}_\text{trans} \) rule, yielding:

\[
\frac{\pi' : n' \rightarrow \chi \quad \sigma' : n' \quad n' \rightarrow n}{\pi' : n' : n \rightarrow \chi \quad \sigma' : n}
\]

Here the result is proved by letting \( \sigma = \sigma' \).

• case \texttt{[return]}: if \( n' \leftarrow n \), write \( \pi = \langle n_0, \ldots, n_i \rangle \), with \( n' = n_i \) and \( n = n_{i+1} \) for some \( j \geq 0 \). By the inductive hypothesis, we have that:

\( \langle n_0, \ldots, n_i \rangle \rightarrow \chi \sigma' : n_i \)

Since \( \pi \) is valid, it must be \( n_i \approx \pi n_i \) for some \( i < j \). Then, \( \hat{\chi} \) must be defined also on \( \langle n_0, \ldots, n_i \rangle \), i.e. a state \( \sigma'' \in \Sigma \) exists, such that:

\( \langle n_0, \ldots, n_i \rangle \rightarrow \hat{\chi} \sigma'' : n_i \)

Now, lemma 3.7.4 ensures \( \sigma' = \sigma'' : n_i \). Then, we can apply the \( \hat{\chi}_\text{return} \) rule, yielding:

\[
\frac{\pi' : n' \rightarrow \hat{\chi} \quad \sigma'' : n_i : n' \quad n' \leftarrow n}{\pi' : n' : n \rightarrow \hat{\chi} \quad \sigma'' : n}
\]

Here the result is proved by letting \( \sigma = \sigma'' \).

\[\square\]

**Counterexample 3.7.6.** Validity is not necessary for \( \hat{\chi} \) to be defined.
3.7. VALID PATHS

Proof. To see why, consider the control flow graph in Fig 3.6, and take the concrete path \( \vec{\pi} = \langle n_0, n_2, n_4 \rangle \). Now, this path is not valid, because, with regard to the return edge \( n_2 \leftarrow n_4 \), we have \( n_0 \nleq_{\vec{\pi}} n_2 \) but \( n_0 \nrightarrow n_4 \). However, \( \check{X}(\vec{\pi}) \) is defined, because \( \langle n_0, n_2 \rangle \rightarrow \check{X} \ [n_0, n_2] \) by \( \check{X}_{\text{entry}} \) and \( \check{X}_{\text{call}} \), and \( \langle n_0, n_2, n_4 \rangle \rightarrow \check{X} \ [n_4] \) by \( \check{X}_{\text{return}} \). □

Theorem 3.7.7.

\[
G \vdash \sigma : n \implies \exists \vec{\pi} \in \tilde{\Pi}_n^\nu, \ \check{X}(\vec{\pi}) = \sigma : n
\]

Proof. The proof is carried out by induction on the derivation used to establish \( G \vdash \sigma : n \). For the base case, if \( \sigma = [] \) and \( n \in N_{\text{entry}} \), then \( \langle n \rangle \in \tilde{\Pi}_n^\nu \) and \( \check{X}(\langle n \rangle) = \nu \). For the inductive case, we proceed by case analysis on the last rule used to derive \( G \vdash \sigma : n \), yielding:

- case [call] :
  \[
  \ell(n') = \text{call} \quad n' \rightarrow n \\
  \sigma' : n' \triangleright \sigma' : n
  \]
  where \( \sigma = \sigma' : n' \)

  By the inductive hypothesis applied to \( \sigma' : n' \), we have:
  \[
  \exists \vec{\pi} \in \tilde{\Pi}_n^\nu, \ \check{X}(\vec{\pi}) = \sigma' : n'
  \]
  Since \( \ell(n') = \text{call} \) and \( n' \rightarrow n \), the path \( \vec{\pi} : n \) is in \( \tilde{\Pi}_n^\nu \), and:
  \[
  \check{X}(\vec{\pi} : n) = \sigma' : n' 
  \]

- case [check] :
  \[
  \ell(n') = \text{check}(P) \quad \sigma : n' \vdash \text{JDK}(P) \\
  n' \rightarrow n \\
  \sigma : n' \triangleright \sigma : n
  \]

  By the inductive hypothesis applied to \( \sigma : n' \), we have:
  \[
  \exists \vec{\pi} \in \tilde{\Pi}_n^\nu, \ \check{X}(\vec{\pi}) = \sigma' : n'
  \]
  Here we have \( \ell(n') = \text{check}(P) \) and \( n' \rightarrow n \), hence \( \vec{\pi} : n \in \tilde{\Pi}_n^\nu \), and:
  \[
  \check{X}(\vec{\pi} : n) = \sigma : n
  \]

- case [return] :
  \[
  \ell(m) = \text{return} \\
  n' \rightarrow n \\
  \sigma : n' : m \triangleright \sigma : n
  \]

  By the inductive hypothesis applied to \( \sigma : n' : m \), we have:
  \[
  \exists \vec{\pi} \in \tilde{\Pi}_n^\nu, \ \check{X}(\vec{\pi}) = \sigma : n' : m
  \]
  By lemma 3.5.16, we have \( m \in p(n') \), hence \( n' \rightarrow n \) implies that \( m \leftarrow n \) is a return edge. Now, lemma 3.6.13 ensures \( n' \nleq_{\vec{\pi}} m \), then the path \( \vec{\pi} : n \) is still valid. Therefore, \( \vec{\pi} : n \in \tilde{\Pi}_n^\nu \), and:
  \[
  \check{X}(\vec{\pi} : n) = \sigma : n
  \]
  \[
  \check{X}(\vec{\pi} : n) = \sigma : n
  \]
Lemma 3.7.8. The evaluation of $\alpha$ is total on valid paths, i.e.:

$$\bar{\pi} \in \tilde{\Pi}^v \implies \exists \pi \in \Pi. \ \bar{\pi} \rightarrow_\alpha \pi$$

Proof. The proof is carried out by mathematical induction on the length of the valid path. For the base case, if $\bar{\pi} = \langle n \rangle \in \tilde{\Pi}^v$, then $\bar{\pi} \rightarrow_\alpha \langle n \rangle$ by the first rule of $\alpha$. For the inductive case, let $\bar{\pi} = \langle n_0, \ldots, n_k \rangle$. We then proceed by case analysis on the last edge of the valid path:

- cases [call, transfer]: by the inductive hypothesis, we have $\langle n_0, \ldots, n_{k-1} \rangle \rightarrow_\alpha \pi$ for some abstract path $\pi \in \Pi_{n_2, n_{k-1}}$. Then, the following transition occurs:

  $$\langle n_0, \ldots, n_{k-1} \rangle \rightarrow_\alpha \pi \quad \langle n_{k-1}, n_k \rangle \in \mathbb{E}$$

  $$\langle n_0, \ldots, n_k \rangle \rightarrow_\alpha \pi : n_k$$

- case [return]: assume $n_{k-1} \rightarrow n_k$. Since $\bar{\pi}$ is a valid path, it must be $n_{i^*} \rightarrow_\pi n_{k-1}$ and $n_{i^*} \rightarrow n_k$ for some $i^* < k$. By Lemma 3.7.2, $\langle n_0, \ldots, n_{i^*} \rangle$ is a valid path, too; hence, by the inductive hypothesis, we find an abstract path $\pi$ such that $\langle n_0, \ldots, n_{i^*} \rangle \rightarrow_\alpha \pi$. Therefore, the following transition occurs:

  $$\alpha(\langle n_0, \ldots, n_{i^*} \rangle) = \pi \quad n_{i^*} \rightarrow_\pi n_{k-1} \quad n_{i^*} \rightarrow n_k$$

  $$\alpha(\langle n_0, \ldots, n_k \rangle) = \pi : n_k$$

Theorem 3.7.9. Let $\bar{\pi}$ be a valid entry path. Then:

$$\bar{\chi}(\bar{\pi}) = \chi(\alpha(\bar{\pi}))$$

Proof. The proof is carried out by mathematical induction on the length of $\bar{\pi}$. For the base case, if $\bar{\pi} = \langle n \rangle$ and $n \in N_{\text{entry}}$, then $\alpha(\langle n \rangle) = \langle n \rangle$ and $\bar{\chi}(\langle n \rangle) = [n] = \chi(\langle n \rangle)$. For the inductive case, let $\bar{\pi} = \langle n_0, \ldots, n_k \rangle$. We proceed by case analysis on the last edge $\langle n_{k-1}, n_k \rangle$ of $\bar{\pi}$:

- case [call]: by Lemma 3.7.2, since $\langle n_0, \ldots, n_k \rangle$ is valid, then $\langle n_0, \ldots, n_{k-1} \rangle$ is a valid path, too. Therefore, both $\bar{\chi}(\langle n_0, \ldots, n_{k-1} \rangle)$ and $\alpha(\langle n_0, \ldots, n_{k-1} \rangle)$ are defined. By definition of $\bar{\chi}$ and $\alpha$, we have:

  $$\bar{\chi}(\langle n_0, \ldots, n_{k-1} \rangle) = \sigma\ n_{k-1} \rightarrow n_k$$

  $$\alpha(\langle n_0, \ldots, n_{k-1} \rangle) = \pi\ n_{k-1} \rightarrow n_k$$

  $$\alpha(\langle n_0, \ldots, n_k \rangle) = \pi : n_k$$

  By the inductive hypothesis, $\bar{\chi}(\langle n_0, \ldots, n_{k-1} \rangle) = \sigma = \chi(\pi)$. Then, by definition of $\chi$:

  $$\chi(\bar{\pi}) = \sigma\ n_{k-1} \rightarrow n_k$$

  $$\chi(\pi) = \sigma : n_k$$
3.7. VALID PATHS

- case [transfer]: again, by lemma 3.7.2, \((n_0, \ldots, n_{k-1})\) is a valid entry path. Then, both \(\hat{\chi}(n_0, \ldots, n_{k-1})\) and \(\alpha(n_0, \ldots, n_{k-1})\) are defined, and we have:

\[
\begin{align*}
\hat{\chi}(n_0, \ldots, n_{k-1}) &= \sigma : n_{k-1} \rightarrow n_k \\
\hat{\chi}(n_0, \ldots, n_k) &= \sigma : n_k \\
\alpha(n_0, \ldots, n_{k-1}) &= \pi : n_{k-1} \rightarrow n_k \\
\alpha(n_0, \ldots, n_k) &= \pi : n_k
\end{align*}
\]

By the inductive hypothesis, \(\hat{\chi}(n_0, \ldots, n_{k-1}) = \sigma : n_{k-1} = \chi(\pi)\). Then:

\[
\chi[\pi] = \sigma : n_{k-1} \rightarrow n_k
\]

\[
\chi(\pi : n_k) = \sigma : n_k
\]

- case [return]: again, by lemma 3.7.2, \((n_0, \ldots, n_{k-1})\) is a valid path, then:

\[
\begin{align*}
\hat{\chi}(n_0, \ldots, n_{k-1}) &= \sigma : n_i : n_{k-1} \leftrightarrow n_k \\
\hat{\chi}(n_0, \ldots, n_k) &= \sigma : n_k \\
\alpha(n_0, \ldots, n_i) &= \pi : n_i \rightarrow n_{k-1} \rightarrow n_k \\
\alpha(n_0, \ldots, n_k) &= \pi : n_k
\end{align*}
\]

Actually, \((n_0, \ldots, n_i)\) is a valid entry path, too: hence, by lemma 3.7.5, we have

\[
\hat{\chi}(n_0, \ldots, n_i) = \sigma' : n_i, \text{ for some state } \sigma' \in \Sigma.
\]

Then, by hypothesis \(n_i \rightarrow n_{k-1}\), lemma 3.7.4 ensures that:

\[
\hat{\chi}(n_0, \ldots, n_i, \ldots, n_{k-1}) = \sigma' : n_i : n_{k-1}
\]

Now, since the evaluation of \(\hat{\chi}\) is deterministic (lemma 3.6.8), and we have already assumed \(\hat{\chi}(n_0, \ldots, n_{k-1}) = \sigma : n_{k-1}\), it must be \(\sigma = \sigma'\) and \(i = i^*\). Then, by the inductive hypothesis, we have \(\hat{\chi}(n_0, \ldots, n_i) = \sigma : n_i = \chi[\pi]\). Therefore:

\[
\begin{align*}
\chi[\pi] &= \sigma : n_i \\
\chi(\pi : n_k) &= \sigma : n_k
\end{align*}
\]

\[
\square
\]

**Lemma 3.7.10.** Let \(\hat{\pi}\) be a valid path. Then:

\[
n \rightarrow_{\hat{\pi}} m \implies m \in \rho(n)
\]

**Proof.** Let \(\hat{\pi} = (n_0, \ldots, n_k)\), \(n = n_i, m = n_j\) for some \(0 \leq i < j \leq k\). Consider the subpath \(\hat{\pi}^* = (n_i, \ldots, n_j) \in \Pi_{n_i,n_j}\). By lemma 3.7.3, \(\hat{\pi}^*\) is a valid path: hence, by 3.7.8 and 3.6.18, we can compute its abstraction \(\pi^* = \alpha(\hat{\pi}^*) \in \Pi_{n_i,n_j}\). Here we can write \(\pi^* = n_i : \pi\) for some \(\pi \in \Pi_{n_{i+1},n_j}\), because \(i \neq j\) implies \(\pi^*\) is non-empty. Now, \(\hat{w}(\pi^*) = w(\pi^*)\) is ensured by theorem 3.6.19, and, by hypothesis \(n_i \rightarrow_{\hat{\pi}} n_j\), we know \(\hat{w}(\pi^*) = 2\). Then:

\[
w(\pi) = w(\pi^*) - w(n_i,n_{i+1}) = \hat{w}(\pi^*) - 1 = 1
\]

In conclusion, we have that \(n_i \rightarrow n_{i+1}\), and \(\exists \pi \in \Pi_{n_{i+1},n_j}, w(\pi) = 1\). Then, by definition 3.5.15, it turns out that \(n_i \in \rho(n_i)\), i.e. \(m \in \rho(n)\). \(\square\)
3.8 Traversable paths

**Definition 3.8.1.** Let \( \sigma \in \Sigma \), and \( n, n', m \in \mathbb{N} \). We define \( \eta : \Sigma \times \Sigma \to \mathbb{E} \) as:

\[
\eta([\ell], [n]) = (\bot, n) \quad \text{if} \quad n \in N_{\text{entry}}
\]

\[
\eta(\sigma : n, \sigma : n : n') = (n, n') \quad \text{if} \quad \ell(n) = \text{call} \quad \text{and} \quad n \rightarrow n'
\]

\[
\eta(\sigma : n, \sigma : n') = (n, n') \quad \text{if} \quad \ell(n) = \text{check}(P) \quad \text{and} \quad n \rightarrow n'
\]

\[
\eta(\sigma : n : m, \sigma : n') = (m, n') \quad \text{if} \quad \ell(m) = \text{return} \quad \text{and} \quad m \leftarrow n'
\]

In any other case, \( \eta \) is left undefined.

**Lemma 3.8.2.**

\[
\sigma' \gg \sigma : n \quad \implies \quad \exists (m, n) \in \mathbb{E}. \eta(\sigma', \sigma : n) = (m, n)
\]

**Proof.** If \( \sigma' = [], n \in N_{\text{entry}} \) and \( \sigma = [n] \), then \( (\bot, n) \in E_{\text{entry}} \) and \( \eta([\ell], [n]) = (\bot, n) \). Otherwise, we proceed by case analysis on the rule used in transition \( \sigma' \gg \sigma : n \), yielding:

- **case [call]:**
  \[
  \ell(n') = \text{call} \quad n' \rightarrow n
  \]
  where \( \sigma = \sigma' : n' \)

  Here \( \eta(\sigma' : n', \sigma' : n' : n) = (n', n) \in \mathbb{E} \).

- **case [check]:**
  \[
  \ell(n') = \text{check}(P) \quad \sigma : n' \vdash DK[P] \quad n' \rightarrow n
  \]
  \( \sigma : n' \gg \sigma : n \)

  Here \( \eta(\sigma : n', \sigma : n) = (n', n) \in \mathbb{E} \).

- **case [return]:**
  \[
  \ell(m) = \text{return} \quad n' \rightarrow n
  \]
  \( \sigma : n' : m \gg \sigma : n \)

  Here \( \eta(\sigma : n' : m, \sigma : n) = (n', n) \in \mathbb{E} \).

\( \Box \)

**Definition 3.8.3.** Let \( \sigma \in \Sigma \), and \( n, n', m \in \mathbb{N} \). We define \( \tilde{\eta} : \Sigma \times \Sigma \to \mathbb{E} \) as:

\[
\tilde{\eta}([\ell], [n]) = (\bot, n) \quad \text{if} \quad n \in N_{\text{entry}}
\]

\[
\tilde{\eta}(\sigma : n, \sigma : n : n') = (n, n') \quad \text{if} \quad \ell(n) = \text{call} \quad \text{and} \quad n \rightarrow n'
\]

\[
\tilde{\eta}(\sigma : n, \sigma : n') = (n, n') \quad \text{if} \quad \ell(n) = \text{check}(P) \quad \text{and} \quad n \rightarrow n'
\]

\[
\tilde{\eta}(\sigma : n : m, \sigma : n') = (m, n') \quad \text{if} \quad \ell(m) = \text{return} \quad \text{and} \quad m \leftarrow n'
\]

In any other case, \( \tilde{\eta} \) is left undefined.
3.8. TRAVERSABLE PATHS

Definition 3.8.4. A concrete path $\pi = \langle n_0, \ldots, n_k \rangle$ is traversable if a derivation:

$$\underline{\varepsilon} \triangleright \sigma_0 \triangleright \sigma_1 \triangleright \cdots \triangleright \sigma_k$$

exists such that $\tilde{\eta}(\underline{\varepsilon}, \sigma_0) = (\bot_N, n_0)$, and:

$$\forall i \in 0..k-1. \tilde{\eta}(\sigma_i, \sigma_{i+1}) = (n_i, n_{i+1})$$

We denote with $\tilde{\Pi}^e$ the set of all traversable paths, and with $\tilde{\Pi}_n^e$ the set of all traversable paths leading to $n$ (clearly, only entry paths can be traversable). We write $\tilde{\Pi}^e(G)$ and $\tilde{\Pi}_n^e(G)$ when we want to make clear that the control flow graph under consideration is $G$.

Lemma 3.8.5. Let $\pi$ be a concrete entry path. Then $\pi$ is traversable iff:

$$\forall i \in 0..k-1. \tilde{\chi}(\langle n_0, \ldots, n_i \rangle) \triangleright \tilde{\chi}(\langle n_0, \ldots, n_{i+1} \rangle)$$

(3.16)

Proof. For the $\triangleright$ part, define $\sigma_i = \tilde{\chi}(\langle n_0, \ldots, n_i \rangle)$ for $i \in 0..k$. Then, by (3.16), we have that $\sigma_i \triangleright \sigma_{i+1}$ for each $i \in 0..k-1$. Now we prove that $\tilde{\eta}(\sigma_i, \sigma_{i+1}) = (n_i, n_{i+1})$, for each $i \in 0..k-1$. We proceed by cases on the last rule used to derive $\tilde{\chi}(\langle n_0, \ldots, n_{i+1} \rangle)$:

- case [call]:

$$\langle n_0, \ldots, n_i \rangle \rightarrow_{\tilde{\chi}} \sigma : n_i \rightarrow n_{i+1}$$

$$\langle n_0, \ldots, n_{i+1} \rangle \rightarrow_{\tilde{\chi}} \sigma : n_i \rightarrow n_{i+1}$$

Here, $\varepsilon(n_i) = \text{call}$ implies $\tilde{\eta}(\sigma : n_i, \sigma : n_i : n_{i+1}) = (n_i, n_{i+1})$.

- case [transfer]:

$$\langle n_0, \ldots, n_i \rangle \rightarrow_{\tilde{\chi}} \sigma : n_i \rightarrow n_{i+1}$$

$$\langle n_0, \ldots, n_{i+1} \rangle \rightarrow_{\tilde{\chi}} \sigma : n_i \rightarrow n_{i+1}$$

Here, $\varepsilon(n_i) = \text{check}(P)$ for some permission $P$, hence $\tilde{\eta}(\sigma : n_i, \sigma : n_i : n_{i+1}) = (n_i, n_{i+1})$.

- case [return]:

$$\langle n_0, \ldots, n_i \rangle \rightarrow_{\tilde{\chi}} \sigma : m : n_i \rightarrow n_{i+1}$$

$$\langle n_0, \ldots, n_{i+1} \rangle \rightarrow_{\tilde{\chi}} \sigma : n_i \rightarrow n_{i+1}$$

Here, $\varepsilon(n_i) = \text{return}$ implies $\tilde{\eta}(\sigma : m : n_i, \sigma : n_i : n_{i+1}) = (n_i, n_{i+1})$.

Moreover, we have $\tilde{\eta}(\underline{\varepsilon}, \sigma_0) = (\bot_N, n_0)$ because $\tilde{\chi}(\langle n_0 \rangle) = [n_0]$ and $n_0 \in N_{entry}$: therefore, the requirements in definition 3.8.4 are fulfilled, and the concrete path $\pi$ is traversable.

For the only if part, assume $\sigma_0 \triangleright \cdots \triangleright \sigma_k$ is a derivation such that $\tilde{\eta}(\underline{\varepsilon}, \sigma_0) = (\bot_N, n_0)$ and $\tilde{\eta}(\sigma_i, \sigma_{i+1}) = (n_i, n_{i+1})$ for each $i \in 0..k-1$. By mathematical induction on the length of $\pi$, we prove that $\sigma_i = \tilde{\chi}(\langle n_0, \ldots, n_i \rangle)$ for each $i \in 0..k-1$. For the base case, if $i = 0$, then $\tilde{\chi}(\langle n_0 \rangle) = [n_0]$: this does the work, because $\tilde{\eta}(\underline{\varepsilon}, \sigma_0) = (\bot_N, n_0)$ implies $\sigma_0 = [n_0]$. For the inductive case, we assume the statement is true for an arbitrary $i \in 0..k-2$; then, we show it also holds for $i + 1$. We proceed by case analysis on the edge $(n_i, n_{i+1})$:
• case [call]:
\[\langle n_0, \ldots, n_i \rangle \rightarrow \chi \sigma : n_i \quad n_i \rightarrow n_{i+1}\]
\[\langle n_0, \ldots, n_{i+1} \rangle \rightarrow \chi \sigma : n_{i+1}\]

By the inductive hypothesis, we have \(\sigma_i = \sigma : n_i\). Since \(\tilde{\eta}(\sigma_i, \sigma_{i+1}) = n_i \rightarrow n_{i+1}\), by definition 3.8.3 it must be \(\sigma_{i+1} = \sigma : n_{i+1}\).

• case [transfer]:
\[\langle n_0, \ldots, n_i \rangle \rightarrow \chi \sigma : n_i \quad n_i \rightarrow \rightarrow n_{i+1}\]
\[\langle n_0, \ldots, n_{i+1} \rangle \rightarrow \chi \sigma : n_{i+1}\]

By the inductive hypothesis, we have \(\sigma_i = \sigma : n_i\). Since \(\tilde{\eta}(\sigma_i, \sigma_{i+1}) = n_i \rightarrow n_{i+1}\), by definition 3.8.3 it must be \(\sigma_{i+1} = \sigma : n_{i+1}\).

• case [return]:
\[\langle n_0, \ldots, n_i \rangle \rightarrow \chi \sigma : m : n_i \quad n_i \leftarrow n_{i+1}\]
\[\langle n_0, \ldots, n_{i+1} \rangle \rightarrow \chi \sigma : n_{i+1}\]

By the inductive hypothesis, \(\sigma_i = \sigma : m : n_i\). Since \(\tilde{\eta}(\sigma_i, \sigma_{i+1}) = n_i \leftarrow n_{i+1}\), by definition 3.8.3 it must be \(\sigma_{i+1} = \sigma : n_{i+1}\).

\[\square\]

**Theorem 3.8.6.** Let \(\tilde{\pi}\) be a traversable path. Then, \(\tilde{\pi}\) is valid.

*Proof.* Let \(\tilde{\pi} = \langle n_0, \ldots, n_k \rangle\), and \(n_j \leftarrow n_{j+1}\) for some \(j < k\) (otherwise \(\tilde{\pi}\) is clearly valid). Since \(\tilde{\pi}\) is traversable, by lemma 3.8.5 it follows that:

\[\tilde{\chi}(\langle n_0, \ldots, n_j \rangle) > \tilde{\chi}(\langle n_0, \ldots, n_{j+1} \rangle)\]  \hspace{1cm} (3.17)

Now, the last rule used to derive \(\tilde{\chi}(\langle n_0, \ldots, n_{j+1} \rangle)\) give rise to the following transition:

\[\tilde{\chi}(\langle n_0, \ldots, n_{j+1} \rangle) = \sigma : m : n_{j+1}\]

By lemma 3.6.6, it must be \(m = n_i\) for \(i = 0 \ldots j\). Then, by lemma 3.6.13, it follows that \(n_i \equiv \tilde{\pi} n_{i-1}\). Moreover, the transition (3.17) takes the form:

\[\ell(n_i) = \text{return} \quad n_i \rightarrow n_{i+1}\]

Therefore, we also have \(n_i \rightarrow n_{i+1}\), which actually proves that \(\tilde{\pi}\) is valid. \[\square\]
Counterexample 3.8.7. Not all valid paths are traversable.

Proof. Consider the control flow graph in Fig. 3.7. The concrete path \( \pi = (n_0, n_1, n_2, n_3) \) is valid, because \( n_0 \xrightarrow{\pi} n_2 \) and \( n_0 \rightarrow n_3 \). However, \( \pi \) is not traversable: in fact, no transition can occur on state \([n_0, n_1]\), because \([n_0, n_1] \not\in JDK[P']\).
Chapter 4

Data Flow Analysis

Data Flow Analysis (DFA) is a static technique for predicting safe and computable approximations to the set of values that the objects of a program may assume during its execution. These approximations are then used to analyze properties of programs in a safe manner: if a property holds at static time, then it will always hold at run-time. The vice-versa may not be true: the analysis may “err on the safe side”. The properties are then interpreted in such a way that an analysis remains correct even when it produces a larger property than ideally possible. This corresponds to producing a valid inference in a program logic for partial correctness. However, DFA and other static program analysis techniques are generally more efficient than program verification, and for that reason more approximate, because the focus is on the fully automatic processing of large programs.

Data flow analysis techniques have been widely used since the early days of optimizing compilers, and they still represent an active field of study in computer science. The first successful effort in endowing DFA with a solid theoretical basis is due to Kildall [Kil73], whose work, subsequently refined by Kam and Ullman [KU76, KU77], still constitutes the approach to DFA which is mainly practiced nowadays.

In what follows, we will give a brief introduction to DFA, using the notation and terminology of [NNH99]. However, our approach will be slightly different from those mentioned above, in order to better fit with our program model and the analyses we will build upon it.
4.1 Basic definitions

In the most general setting, a data flow analysis problem is just a combination of a control flow graph \( G \), a property space \( \mathcal{L} \), which describes the data flow information associated to \( G \) by the analysis, a safety test \( \phi : \mathcal{L} \to \text{Bool} \), which characterize the solutions of the problem, and a partial order \( \sqsubseteq \), which tells when a solution is more precise than another. \(^1\)

The fixed point approach to data flow analysis states that the solutions for a data flow problem are the fixed points of a given transfer function \( f : \mathcal{L} \to \mathcal{L} \), that is \( l = f(l) \) implies \( \phi(l) \) is true. Therefore, according to the informal meaning given to \( \sqsubseteq \), the most precise solution for a data flow problem is just the least fixed point of its transfer function.

Transfer functions actually specify abstract versions of the semantics of control flow graphs. By the moment, we do not require that transfer functions are defined compositionally (i.e. the abstract semantics of a graph is constructed from the abstract semantics of its nodes).

The other approach to data flow analysis is the meet over all paths approach. Roughly, it consists of two steps: first, the abstract semantics of each traversable path is computed; then, the results of the previous step are mixed to obtain the abstract semantics of the whole graph. Since, due to the presence of loops, it may happen that an infinite number of paths must be taken into account, the meet over all paths approach is in general not effective.

In what follows, we will mainly focus on the fixed point approach: this choice is motivated by the fact that, under given conditions, this approach will always find solutions for data flow problems, while maintaining the same accuracy of the meet over all paths approach (see e.g. theorem 4.3.7).

**Definition 4.1.1.** A data flow analysis is a triple \( \langle \mathcal{L}, \mathcal{F}, \mu \rangle \) where:

- the partial order \( \mathcal{L} = \langle \mathcal{L}, \sqsubseteq \rangle \) is a property space;
- \( \mathcal{F} \subseteq \{ f \mid f : \mathcal{L} \to \mathcal{L} \} \) is a family of transfer functions;
- \( \mu \) is a mapping from control flow graphs to transfer functions.

\(^1\)In what follows, we establish that \( l_0 \sqsubseteq l_1 \) means "\( l_0 \) is more precise than \( l_1 \)."
4.1. BASIC DEFINITIONS

According to the informal definition sketched above, a control flow graph $G$, together with a data flow analysis $\langle \mathcal{L}, \mathcal{T}, \mu \rangle$, instantiates a data flow problem $\langle G, \mathcal{L}, \mu(G) \rangle$. We say that $l \in \mathcal{L}$ is a solution for the problem $\text{DFP} = \langle G, \mathcal{L}, f \rangle$ (for brevity, a DFP-solution), if $l \in \text{fix}(f)$, where $\text{fix}(f) = \{ l \in \mathcal{L} \mid l = f(l) \}$.

In the general case, the specification of a data flow analysis per se is not worth troubling about: actually, it is not even known whether a data flow problem has a solution or not. Therefore, in what follows we will put constraints on the structure of the property space and the transfer functions, in such a way that, for any control flow graph, a solution indeed exists and it is (efficiently) computable.

The following definition establishes sufficient conditions for which a DFA fulfills the first requirement, and part of the second one.

**Definition 4.1.2.** A data flow analysis is monotone if:

- the property space satisfies the *ascending chain condition*, i.e. any increasing chain $l_0 \subseteq l_1 \subseteq \cdots \subseteq l_k \subseteq \cdots$ of elements of $\mathcal{L}$ eventually stabilises, that is:

  $$\exists k_0 \in \mathbb{N}. \forall k \geq k_0. l_k = l_{k_0}$$

  Moreover, we require $\mathcal{L}$ to be endowed with a bottom element $\bot_{\mathcal{L}}$.

- $\mathcal{T}$ is a family of monotone functions, that is:

  $$l \subseteq l' \implies f(l) \subseteq f(l')$$

  for any $f \in \mathcal{T}$ and $l, l' \in \mathcal{L}$.

It is evident that any instance of a monotone data flow analysis always admits a solution. In fact, by the ascending chain condition, the chain:

$$\bot_{\mathcal{L}} \subseteq f(\bot_{\mathcal{L}}) \subseteq f^2(\bot_{\mathcal{L}}) \subseteq \cdots \subseteq f^k(\bot_{\mathcal{L}}) \subseteq \cdots$$

eventually stabilises to the least fixed point of $f$.

Although this suggests an algorithm which actually computes the least solution for any monotone data flow problem, we are not able to give any estimate of its computational complexity: this happens because no assumptions about the structure of the property space and the transfer function were made.
4.2 Data flow frameworks

By looking at archetypical examples of data flow analyses [ASU86, Muc97, NNH99], it turns out that most of them are based on property spaces and families of transfer functions of the form:

\[ \mathcal{L} = \mathbb{N} \cup \{ \bot \} \rightarrow \mathcal{L}_N \quad \mathcal{F} = \mathbb{E}_p \rightarrow \mathcal{F}_E \]  

(4.1)

where \( \mathcal{L}_N \) is a local property space and \( \mathcal{F}_E \subseteq \{ f \mid f : \mathcal{L}_N \rightarrow \mathcal{L}_N \} \) is a family of local transfer functions. Actually, this means that data flow information is homogeneously distributed among nodes; each edge is then associated with a transfer function, telling how the information propagates through the graph. Moreover, a confluence operator \( \bigsqcup : \mathcal{L}_N \rightarrow \mathcal{L}_N \) is needed, in order to specify how different guesses \( l_1(n), \ldots, l_k(n) \) for the analysis at node \( n \) are combined into a single guess \( l(n) = \bigsqcup \{ l_i(n) \mid i \in 1..k \} \).

Without loss of generality, in what follows we will only consider forward analyses, where the information propagates in the same direction as the control flow. Backward analyses only require minor adjustments (e.g. changing the direction of edges) to be handled by the iteration-based algorithms described below. On the other hand, different kinds of algorithms (e.g. structure-based) would require substantial effort to be adapted to the backward case.

Definition 4.2.1. A data flow framework is a quadruple \( \langle \mathcal{L}_N, \mathcal{F}_E, \mu_E, \iota \rangle \), where:

- \( \mathcal{L}_N = \langle \mathcal{L}_N, \sqcup, \bot_{\mathcal{L}_N} \rangle \) is a join semi-lattice, i.e. a non-empty set with a binary join operator \( \sqcup : \mathcal{L}_N \times \mathcal{L}_N \rightarrow \mathcal{L}_N \), which is idempotent, commutative and associative. Moreover, \( \mathcal{L}_N \) is required to contain an unit element \( \bot_{\mathcal{L}_N} \) such that, for any \( l \in \mathcal{L}_N \):
  \[ \bot_{\mathcal{L}_N} \sqcup l = l = l \sqcup \bot_{\mathcal{L}_N} \]

- \( \mathcal{F}_E \subseteq \{ f \mid f : \mathcal{L}_N \rightarrow \mathcal{L}_N \} \) is closed under composition, and contains the identity function \( id_{\mathcal{L}_N} = _{def} \lambda x : \mathcal{L}_N. x \);

- \( \mu_E \) is a mapping from edges to local transfer functions;

- \( \iota \in \mathcal{L}_N \) is the solution for the isolated entry node \( \bot_N \).
4.2. DATA FLOW FRAMEWORKS

**Theorem 4.2.2.** A data flow framework is a data flow analysis.

*Proof.* The global property space $\mathcal{L}$ and the family $\mathcal{F}$ of global transfer functions can be reconstructed from $\mathcal{L}_N$ and $\mathcal{F}_E$ as shown by equation (4.1). The global mapping $\mu$ is:

$$
\mu(G) = \lambda e : E_p, \mu_E(e)
$$

To complete the construction, it suffices to observe that the join operator $\sqcup$ actually induces a partial order $\sqsubseteq$ on $\mathcal{L}$. This is made evident by defining

$$
1 \sqsubseteq 1' \text{ if and only if } 1 \sqcap 1' = 1'
$$

Again, given a control flow graph $G$ and a data flow framework $\langle \mathcal{L}_N, \mathcal{F}_E, \mu_E, \iota \rangle$, we can instantiate a problem $\langle G, \mathcal{L}_N, \mathcal{F}_E, \mu_E, \iota \rangle$, where:

$$
f_E = \{ f_{(m,n)} : \mathcal{L}_N \rightarrow \mathcal{L}_N \mid (m, n) \in E_p \land f_{(m,n)} = \mu_E(m, n) \}
$$

is the set of local transfer functions associated with the edges of the graph.

**Definition 4.2.3.** A data flow framework $\langle \mathcal{L}_N, \mathcal{F}_E, \mu_E, \iota \rangle$ is monotone if $\mathcal{L}_N$ satisfies the ascending chain condition, and all transfer functions in $\mathcal{F}_E$ are monotone (with respect to the partial order induced by the join operator).

**Theorem 4.2.4.** A monotone framework is a monotone data flow analysis.

*Proof.* To prove that the ascending chain condition is satisfied by the global property space, consider the increasing chain of elements of $\mathcal{L}$:

$$
l_0 \subseteq l_1 \subseteq \cdots \subseteq l_k \subseteq \cdots
$$

By the construction in (4.1), the global property space $\mathcal{L}$ is pointwise ordered by:

$$
l \subseteq l' \text{ if and only if } \forall n \in N \cup \{ \perp_N \}, \ l(n) \subseteq l'(n)
$$

Since the ascending chain condition holds for $\mathcal{L}_N$, we have that:

$$
\forall n \in N, \exists k_n \in N, \forall k \geq k_n, l_k(n) = l_{k_n}(n)
$$

Then, if we choose $k_0 = \max \{ k_n \mid n \in N \}$, we obtain $l_k(n) = l_{k_0}(n)$ for any $k \geq k_0$ and $n \in N$. Therefore, we also have $l_k = l_{k_0}$ for any $k \geq k_0$. To conclude, we prove that the global transfer function is monotone. By monotonicity of local transfer functions, we have:

$$
l \subseteq l' \Rightarrow \forall m \in N, \ l(m) \subseteq l'(m)
$$

$$
\Rightarrow \forall (m,n) \in E_p, \ f_{(m,n)}[(l(m))] \subseteq f_{(m,n)}[(l'(m))]
$$

$$
\Rightarrow \bigsqcup_{(m,n) \in E_p} f_{(m,n)}[(l(m))] \subseteq \bigsqcup_{(m,n) \in E_p} f_{(m,n)}[(l'(m))]
$$

$$
\Rightarrow f(l) \subseteq f(l')
$$

**Definition 4.2.5.** A monotone framework is distributive if:

$$
f(l \sqcup l') = f(l) \sqcup f(l')
$$

for each transfer function $f \in \mathcal{F}_E$ and $l, l' \in \mathcal{L}_N$. 
4.3 Solutions and their properties

So far, we have not specified what the solutions for an instance of a data flow framework are. According to the two approaches to data flow analysis presented above, we distinguish between two classes of solutions.

The fixed point strategy iteratively approximates the least solution of a system of equations, which specifies the relation between the solution at node \( n \) and the solutions at its predecessors (i.e., those \( m \) such that \( (m, n) \in E_p \)).

**Definition 4.3.1.** A fixed point solution for an instance \( \langle G, \mathcal{L}_N, f_E, \iota \rangle \) of a data flow framework is a solution of the equation system \( \text{DFA}\overline{-}(G) \):

\[
\text{DFA}(n) = \begin{cases} 
\iota & \text{if } n = \bot_N \\
\bigsqcup_{(m, n) \in E_p} f(m, n)(\text{DFA}(m)) & \text{otherwise}
\end{cases}
\]

We write \( \iota \models \text{DFA}\overline{-}(G) \) when \( \iota \in \mathcal{L} \) is solution to the equation system. The MFP solution is then defined as the least solution to \( \text{DFA}\overline{-}(G) \), i.e.:

\[
\text{MFP} = \bigsqcap \{ \iota \in \mathcal{L} \mid \iota \models \text{DFA}\overline{-}(G) \}
\]

The global transfer function \( f \) is reconstructed from the set \( f_E \) of local transfer functions as follows:

\[
f(\iota) = \lambda n. \text{ if } n = \bot_N \text{ then } \iota \text{ else } \bigsqcup_{(m, n) \in E_p} f(m, n)(\iota(m))
\]

The meet over all paths strategy directly mimics possible program executions: for each node \( n \) in the control flow graph, the solution at \( n \) is obtained by joining the abstract semantics of all traversable paths reaching \( n \). Less accurate solutions are obtained by considering approximations to the set of traversable paths (e.g., valid paths or concrete paths).

In order to formally specify this class of solutions, we first need to define the transfer function of a path:

**Definition 4.3.2.** The transfer function of a path \( \pi \) (either abstract or concrete) is defined as:

\[
f_{\pi} = \begin{cases} 
\text{id}_{\mathcal{L}_N} & \text{if } \pi = \langle \rangle \\
f_{(n_k, \ldots, n_0)} \circ \cdots \circ f_{(n_0, n_1)} \circ f_{(n, \bot_N)} & \text{if } \pi = \langle n_0, \ldots, n_k \rangle
\end{cases}
\]
4.3. SOLUTIONS AND THEIR PROPERTIES

**Definition 4.3.3.** The *meet over all traversable paths* (MTP) solution for an instance $\langle G, \mathcal{L}_N, f_t, \iota \rangle$ of a data flow framework is defined as:

$$\text{MTP}(n) = \bigsqcup_{\pi \in \Pi_n} f_{t}(\iota)$$

If we agree the empty path reaches the entry node $\perp_N$ (i.e. $\langle \rangle \in \Pi_{\perp_N}$), then we yield the correct solution at $\perp_N$, i.e. $\text{MTP}(\perp_N) = \iota d_{\perp_N}(\iota) = \iota$.

Since, in our program model, it is always possible to statically test whether a path is traversable or not (in the sense of definition 3.8.4), the procedure of ruling out non-traversable paths can be substituted by an appropriate definition of the transfer functions. This gives rise to the following definition:

**Definition 4.3.4.** The *meet over all valid paths* (MVP) solution for an instance $\langle G, \mathcal{L}_N, f_t, \iota \rangle$ of a data flow framework is defined as:

$$\text{MVP}(n) = \bigsqcup_{\pi \in \Pi_n} f_{\pi}(\iota)$$

However, the MFP solution if often a rather coarse approximation of the MVP solution. In fact, by definition 4.3.1, the MFP solution propagates the information at the exit of a method to all of its callers, regardless of the calling sequence (i.e. it is *context insensitive*). More accurate approximations can be obtained by considering fixed point solutions over different property spaces. Context sensitive analyses are subject of investigation in [SP81].

At best, the MFP solution can aim at approximating the *meet over all paths* solution, which guesses the analysis at node $n$ by combining the pieces of data flow information collected through all paths reaching $n$.

**Definition 4.3.5.** The *meet over all paths* (MOP) solution for an instance $\langle G, \mathcal{L}_N, f_t, \iota \rangle$ of a data flow framework is defined as:

$$\text{MOP}(n) = \bigsqcup_{\pi \in \Pi_n \cup \Pi_n} f_{\pi}(\iota)$$

It can be shown [KU77] that the MOP solution is not always computable.
CHAPTER 4. DATA FLOW ANALYSIS

Theorem 4.3.6. Consider the MTP, MVP, MOP and MFP solutions to an instance of a monotone framework. We have:

\[
\text{MTP} \subseteq \text{MVP} \subseteq \text{MOP} \subseteq \text{MFP}
\]

Proof. First of all, note that all solutions coincide on the isolated entry node \( \perp_N \), i.e. \( \text{MTP}(\perp_N) = \text{MVP}(\perp_N) = \text{MOP}(\perp_N) = \text{MFP}(\perp_N) = t \). Then, from now on we will always consider \( n \neq \perp_N \). For the first inequality, observe that, by theorem 3.8.6, any traversable path is also a valid path, i.e. \( \vec{P}_n \subseteq \vec{P}_n \). Then, since \( X \subseteq Y \implies \bigcup X \subseteq Y \), we have:

\[
\text{MTP}(n) = \bigcup \{ f_n(t) \subseteq \bigcup f_n(t) = \text{MVP}(n) \}
\]

For the second inequality, it suffices to observe that, by definition 3.7.1, any valid path is also a concrete path. Then \( \vec{P}_n \subseteq \vec{P}_n \subseteq \vec{P}_n \cup \vec{P}_n \), and:

\[
\text{MVP}(n) = \bigcup \{ f_n(t) \subseteq \bigcup f_n(t) = \text{MOP}(n) \}
\]

For the last inequality, we need the following auxiliary definition:

\[
\text{MOP}_k(n) = \bigcup \{ f_n(t) \subseteq \bigcup f_n(t) = \text{MOP}(n) \}
\]

(4.2)

For any \( k \in \mathbb{N} \), \( \text{MOP}_k \) is an approximation of the MOP solution, where only paths whose length is less than \( k \) are considered. Clearly, \( \text{MOP}_k \subseteq \text{MOP} \) for any \( k \). Moreover, we have:

\[
\text{MOP}(n) = \bigcup_{k \in \mathbb{N}} \text{MOP}_k(n)
\]

(4.3)

In fact, consider the increasing chain:

\[
\text{MOP}_0(n) \subseteq \text{MOP}_1(n) \subseteq \ldots \subseteq \text{MOP}_k(n) \subseteq \ldots
\]

By the ascending chain condition, there is a \( k_0 \in \mathbb{N} \) such that \( \text{MOP}_k(n) = \text{MOP}_{k_0}(n) \) for any \( k \geq k_0 \). This means that, to yield the MOP solution at node \( n \), it suffices to look at paths whose length is less than \( k \). Then, \( \text{MOP}(n) = \text{MOP}_{k_0}(n) = \bigcup_{k \in \mathbb{N}} \text{MOP}_k(n) \). Now, equation (4.3) enables us to prove the inclusion \( \text{MOP} \subseteq \text{MFP} \) by showing that:

\[
\forall k \in \mathbb{N} \text{, } \text{MOP}_k(n) \subseteq \text{MFP}(n)
\]

We proceed by mathematical induction on the length \( k \). For the base case, \( \text{MOP}_0(n) \) is the least upper bound over an empty set of paths: by convention, this equals to \( \perp_{\mathcal{N}} \), and
4.3. SOLUTIONS AND THEIR PROPERTIES

Clearly $\bot_{\mathcal{L}_N} \sqsubseteq \text{MFP}(n)$. For the inductive case, assume $\text{MOP}_k(n) \sqsubseteq \text{MFP}(n)$. Then:

\[
\text{MFP}(n) = f(\text{MFP}(n)) = \bigcup_{(m,n) \in \mathcal{E}_\rho} f_{m,n}(\text{MFP}(m)) \]

by def. 4.3.1

\[
\subseteq \bigcup_{(m,n) \in \mathcal{E}_\rho} f_{m,n}(\text{MOP}_k(m)) \]

by ind. hyp., monot. $f$

\[
= \bigcup_{(m,n) \in \mathcal{E}_\rho} \left( \bigcup_{\pi \in \Pi_m \cup \Pi_m} f_{m,n}(f_{\pi(t)}) \right) \]

by def. $\text{MOP}_k$

\[
= \bigcup_{(m,n) \in \mathcal{E}_\rho} \left( \bigcup_{\pi \in \Pi_m \cup \Pi_m \mid |\pi| < k} f_{m,n}(f_{\pi(t)}) \right) \cup \bigcup_{\pi \in \Pi_m \cup \Pi_m \mid |\pi| < k} f_{m,n}(f_{\pi(t)}) \]

by associ. $\sqcup$

\[
\subseteq \bigcup_{(m,n) \in \mathcal{E}_\rho} \left( \bigcup_{\pi \in \Pi_m \cup \Pi_m \mid |\pi| < k} f_{m,n}(f_{\pi(t)}) \right) \cup \bigcup_{\pi \in \Pi_m \cup \Pi_m \mid |\pi| < k} f_{m,n}(f_{\pi(t)}) \]

as $\mathcal{E}_\rho \subseteq \mathcal{E}_\rho$

\[
= \bigcup_{\pi' \in \Pi_{n+1} \mid \pi' \leq k+1} f_{\pi'(t)} \cup \bigcup_{\pi' \in \Pi_{n+1} \mid \pi' \leq k+1} f_{\pi'(t)} \]

by def. $f_{\pi}$

\[
= \bigcup_{\pi' \in \Pi_{n+1} \cup \Pi_{n+1} \mid \pi' \leq k+1} f_{\pi'(t)} \]

by associ. $\sqcup$

\[
= \text{MOP}_{k+1}(n) \]

by def. $\text{MOP}_k$

\[
\square
\]

It can be proved [KU77] that no algorithm can compute the MOP solution for all monotone data flow frameworks; however, for the class of distributive frameworks, the following coincidence theorem holds:

**Theorem 4.3.7.** The MOP and MFP solutions coincide for any distributive framework, provided the graph is connected (i.e. $\Pi_n \neq \emptyset$ for each $n \in \mathbb{N}$), and one of the following conditions holds:

\[
f_{m \rightarrow n} = \bot_{\mathcal{L}_n} \quad \text{for each } m, n \in \mathbb{N} \quad (4.4a)
\]

\[
f_{m \leftarrow n} = \bot_{\mathcal{L}_n} \quad \text{for each } m, n \in \mathbb{N} \text{ such that } \ell(m) = \text{call} \quad (4.4b)
\]

**Proof.** First of all observe that, if $f$ is distributive and $X$ is non-empty, then:

\[
f(\bigcup_{l \in X} X) = \bigcup_{l \in X} f(l) \quad (4.5)
\]
Now, assume that condition (4.4a) is satisfied. Then, it follows that:

\[ f_\pi(t) \subseteq f_{\alpha(\bar{\pi})}(t) \]

for any concrete path \( \bar{\pi} \). Since, with the above, the transfer function of a concrete path is always more precise than the transfer function of an abstract path, we also have that:

\[ \bigsqcup_{\pi \in P_n} f_\pi(t) \subseteq \bigsqcup_{\pi \in \bar{P}_n} f_\pi(t) \quad (4.6) \]

Next we calculate:

\[
\begin{align*}
\text{MOP}[m] &= \bigsqcup_{\pi \in P_m \cup \bar{P}_m} f_\pi(t) \quad \text{by def. 4.3.5} \\
&= \bigsqcup_{\pi \in P_m} f_\pi(t) \cup \bigsqcup_{\pi \in \bar{P}_m} f_\pi(t) \quad \text{by assoc. } \cup \\
&= \bigsqcup_{\pi \in P_m} f_\pi(t) \quad \text{by (4.6)} \\
&= \bigsqcup_{(m,n) \in E_p} \left( \bigsqcup_{\pi \in P_m} f_{(m,n)}(f_\pi(t)) \right) \quad \text{by def. } f_\pi \\
&= \bigsqcup_{(m,n) \in E_p} \left( \bigsqcup_{\pi \in P_m} f_{(m,n)} \left( \bigsqcup_{\pi \in \bar{P}_m} f_\pi(t) \right) \right) \quad \text{by (4.5)} \\
&= \bigsqcup_{(m,n) \in E_p} \left( \bigsqcup_{\pi \in P_m} f_{(m,n)} \left( \bigsqcup_{\pi \in \bar{P}_m} f_\pi(t) \right) \right) \quad \text{by (4.4a)} \\
&= \bigsqcup_{(m,n) \in E_p} \left( \bigsqcup_{\pi \in P_m \cup \bar{P}_m} f_\pi(t) \right) \quad \text{by (4.6)} \\
&= \bigsqcup_{(m,n) \in E_p} f_{(m,n)} \left( \bigsqcup_{\pi \in P_m \cup \bar{P}_m} f_\pi(t) \right) \quad \text{by assoc. } \cup \\
&= \bigsqcup_{(m,n) \in E_p} f_{(m,n)}(\text{MOP}[m]) \quad \text{by def. 4.3.5}
\end{align*}
\]

Therefore, MOP is a solution to the data flow equations. Since, by definition 4.3.1, MFP is the least solution to such equations, we have MFP \( \subseteq \) MOP. On the other hand, by theorem 4.3.6 it follows MOP \( \subseteq \) MFP; hence the equality is proved.

Similar arguments can be used to prove that the MOP and MFP solutions coincide if the hypothesis (4.4b) is satisfied.

Context-sensitive versions of this theorem can be found in [SP81] and [KS92].

The next section presents a polynomial-time iterative algorithm that computes the MFP solution to the instance of monotone framework given as input. Other classes of algorithms for data flow analysis are described in [MR90].
4.4 The Worklist-Iteration algorithm

**Algorithm 2:** Worklist-Iteration solution for monotone frameworks

*Input:* an instance of a monotone framework DFP = \( (G, \mathcal{L}_N, f_\mathcal{L}, \ell) \).

*Output:* the MFP solution for DFP.

**WORKLIST-ITERATION\((G, \mathcal{L}_N, f_\mathcal{L}, \ell)\)**

1. \( W \leftarrow \text{NIL} \)
2. \( \text{Analysis}[\bot_N] \leftarrow \ell \)
3. \( \text{for each } (m, n) \text{ in } E_\mathcal{L} \text{ do} \)
4. \( W \leftarrow \text{CONS}((m, n), W) \)
5. \( \text{for each } n \text{ in } N \text{ do} \)
6. \( \text{Analysis}[n] \leftarrow \bot_N \)
7. \( \text{while } W \neq \text{NIL} \text{ do} \)
8. \( (m, n) \leftarrow \text{HEAD}(W) \)
9. \( W \leftarrow \text{TAIL}(W) \)
10. \( \text{if } f_{(m, n)}(\text{Analysis}[m]) \not\subseteq \text{Analysis}[n] \)
11. \( \text{then } \text{Analysis}[n] \leftarrow \text{Analysis}[n] \cup f_{(m, n)}(\text{Analysis}[m]) \)
12. \( \text{for each } (n, n') \text{ in } E_\mathcal{L} \text{ do} \)
13. \( W \leftarrow \text{CONS}((n, n'), W) \)

**Theorem 4.4.1.** The worklist-iteration algorithm always terminates, and it computes the MFP solution to the instance of monotone framework given as input. More precisely, for each \( n \in N \cup \{ \bot_N \} \), we have:

\[
\text{MFP}(n) = \text{Analysis}[n]
\]

*Proof.* We first carry out the proof of termination of the algorithm. The bounded for loops at lines 3-4 and 5-6 trivially terminate: actually, they are executed \(|E_\mathcal{L}| + |N|\) times respectively. The worklist \( W \) contains \(|E_\mathcal{L}|\) elements when the while loop at lines 7-13 starts. Each iteration of the while loop removes an element from the worklist (line 9), and may add up to \(|E_\mathcal{L}|\) new elements (lines 12-13).

An edge \((n, n')\) can be added only if \( f_{(m, n)}(\text{Analysis}[m]) \not\subseteq \text{Analysis}[n] \) for some node \( m \), or they are incomparable (line 10). In both cases, the new value assigned to \( \text{Analysis}[n] \) at line 11 is strictly greater than its previous one. In fact, the inequality:

\[
\text{Analysis}[n] \not\subseteq \text{Analysis}[n] \cup f_{(m, n)}(\text{Analysis}[m])
\]

is implied by the more general result:

\[
I' \not\subseteq I \implies I \cup (I \cup I') = (I \cup I')
\]

The non-strict inequality \( I \cup I \subseteq I \cup I' \) is always true, because, by definition of \( \subseteq \), this is equivalent to \( I \cup (I \cup I') = (I \cup I') \), which holds by associativity and idempotency of \( \cup \).
Strictness is proved by contradiction. Assume \( l = l \cup l' \). By definition of \( \subseteq \), this is the same as saying \( l' \subseteq l \), which clearly contradicts hypothesis \( l' \not\subseteq l \).

Now, any strictly increasing chain must be finite. To see why, suppose, by contradiction, that \( l_0 \subseteq l_1 \subseteq \cdots \subseteq l_k \subseteq \cdots \) is an infinite strictly increasing chain. Then, since \( l_i \subseteq l_{i+1} \) clearly implies \( l_i \subseteq l_{i+1} \) for any \( i \in \mathbb{N} \), by the ascending chain condition we find an index \( k_0 \in \mathbb{N} \) such that \( \forall k \geq k_0, l_k = l_{k+1} \), which contradicts our hypothesis about strictness. Therefore, each edge can be inserted into the worklist only a finite number of times: this causes the worklist to be eventually exhausted, and the algorithm to terminate (note that we have not used the fact that the analysis is monotone).

The proof of the correctness result is split in two parts: first, we prove that, for each node \( n \), the inclusion relation \( \text{Analysis}[n] \subseteq \text{MFP}(n) \) is invariant for the \textbf{while} loop; then, we will conclude by showing that, on termination of the loop, also the converse inequality \( \text{MFP}[n] \subseteq \text{Analysis}[n] \) will hold.

For the first part, observe that, after the initialisation for loop at lines 5-6, we have \( \text{Analysis}[\bot_N] = i = \text{MFP}(\bot_N) \), and \( \text{Analysis}[n] = \bot_N \subseteq \text{MFP}(n) \) for each \( n \neq \bot_N \). Therefore, the inclusion holds when the while loop is entered. Next, we show that the inclusion is preserved after each iteration of the loop. When the condition of the if statement at line 10 is false, the array Analysis is left unchanged. Otherwise, only the value of \( \text{Analysis}[n] \) is updated, and we have:

\[
\text{Analysis}[n] \leftarrow \text{Analysis}[n] \cup f_{(m,n)}[\text{Analysis}[m]]
\]

by the assignment at line 11

\[
\subseteq \text{MFP}(n) \cup f_{(m,n)}[\text{MFP}(m)]
\]

by hyp. \( \text{Analysis}[n] \subseteq \text{MFP}(n) \)

\[
\subseteq \text{MFP}(n) \cup \bigcup_{(m,n) \in E_r} f_{(m,n)}[\text{MFP}(m)]
\]

by monotonicity of \( f_{(m,n)} \)

\[
\subseteq \text{MFP}(n) \cup \text{MFP}(n)
\]

by def. \( \subseteq \) and \( \cup \)

\[
= \text{MFP}(n)
\]

by def. \( \text{MFP} \)

For the second part, note that the inequality:

\[
\forall (m,n) \in E_p, \text{Analysis}[n] \supseteq f_{(m,n)}[\text{Analysis}[m]]
\]  

(4.7)

is ensured at the exit of the \textbf{while} loop. In fact, whenever the condition of the if statement at line 10 is false for some \( (m,n) \in E_p \), the assignment at line 11 manages to establish the inequality \( \text{Analysis}[n] \supseteq f_{(m,n)}[\text{Analysis}[m]] \), because:

\[
\text{Analysis}[n] \cup f_{(m,n)}[\text{Analysis}[m]] \supseteq f_{(m,n)}[\text{Analysis}[m]]
\]

trivially follows by definition of \( \cup \) and \( \supseteq \). Actually, after the value of \( \text{Analysis}[n] \) has been updated, it may happen that \( \text{Analysis}[n'] \supseteq f_{(m,n')}[\text{Analysis}[n]] \) for some \( (n,n') \in E_p \). However, once the assignment at line 11 has been performed, the inner \textbf{for} loop at lines 12-13 takes care of inserting all edges \( (n,n') \in E_p \) in the worklist: then, the inequality \( \text{Analysis}[n'] \supseteq f_{(m,n')}[\text{Analysis}[n]] \) will eventually be re-established in a subsequent iteration of the while. Hence, the inequality (4.7) is satisfied at the exit of the loop. By definition of \( \bigcup \), from (4.7) it follows that, for any \( n \in \mathbb{N} \):

\[
\text{Analysis}[n] \supseteq \bigcup_{(m,n) \in E_r} f_{(m,n)}[\text{Analysis}[m]]
\]

Then, \( \text{Analysis} \) turns out to be a \textit{prefixed point} of the global transfer function \( f \). By the Knaster-Tarski theorem on minimum fixed points, monotonicity of \( f \) implies that MFP is just the least prefixed point of \( f \). Thus:

\[
\text{Analysis}[n] \supseteq \text{MFP}(n)
\]
Chapter 5

Static analyses

In the previous chapters, we have seen that stack inspection plays a crucial role in the Java security model. We have studied an abstract representation of Java programs, which includes a formalization of stack inspection.

Based on this information, in this chapter we develop two families of data flow analyses, which aim at discovering what permissions are granted/denied to code in all possible executions.

We prove that all the analyses are safe, in the sense that they never predict program behaviours that do not correspond to any actual execution; then, we show how these analyses can be used to optimize the Java bytecode.
\section{The Denied Permissions Analysis}

We say that a permission \( P \) is \textit{denied} to a state \( \sigma \) if \( \sigma \not
\models \text{JDK}(P) \). Purpose of the Denied Permissions Analysis (DP for short) is that of finding, for each node of a given control flow graph, which permissions are \textit{certainly} denied to any state reaching that node. We require the analysis to be \textit{safe}, in the sense that it never claims that a permission \( P \) is denied to a node \( n \) when an execution \([ \| ] \triangleright \cdots \triangleright \sigma : n \) exists such that \( \sigma : n \models \text{JDK}(P) \).

\textbf{Definition 5.1.1.} The sets of permissions denied, respectively, to a state \( \sigma \) and to a node \( n \), are defined as:

\[
\Delta(\sigma) = \begin{cases} \{ P \in \text{Permission} \mid \sigma \not\models \text{JDK}(P) \} & \text{if } G \vdash \sigma \\ \emptyset & \text{otherwise} \end{cases}
\]

\[
\Delta(n) = \bigcap_{G \vdash \sigma : n} \Delta(\sigma : n)
\]

Given any control flow graph \( G \), a DP analysis is specified by means of an equation system \( \text{DP}^- (G) \). We say that \( \delta \) is a DP-solution if \( \delta \models \text{DP}^- (G) \).

\textbf{Definition 5.1.2 (DP-soundness).} A DP-solution \( \delta \) is \textit{sound} if:

\[
P \in \delta(n) \implies \forall \sigma \in \Sigma. G \vdash \sigma : n \implies \sigma : n \not\models \text{JDK}(P)
\]

\textbf{Lemma 5.1.3.} Let \( \delta \) be a sound DP-solution. Then, for each node \( n \in \mathbb{N} \):

\[
\delta(n) \subseteq \Delta(n)
\]

\textit{Proof.} Let \( P \in \delta(n) \), where \( \delta \) is a sound DP-solution. If \( n \) is not \( G \)-reachable, then \( \Delta(n) \) is the intersection of an empty collection of permission sets, which, by definition, equals to the universal set \textbf{Permission}. Therefore, \( P \) is trivially contained in \( \Delta(n) \).

Otherwise, if \( n \) is \( G \)-reachable, by definition 5.1.2 we have that \( \sigma : n \not\models \text{JDK}(P) \) for any \( G \)-reachable state \( \sigma : n \). Actually, by definition 5.1.1, this means that \( P \in \Delta(\sigma : n) \) for any state \( \sigma : n \) such that \( G \vdash \sigma : n \), that is:

\[
P \in \bigcap_{G \vdash \sigma : n} \Delta(\sigma : n)
\]

\textbf{Definition 5.1.4.} A DP-solution \( \delta \) is \textit{non-trivial} if:

\[
P \not\in \text{Perm}(n) \implies P \in \delta(n)
\]
5.1. THE DENIED PERMISSIONS ANALYSIS

**Definition 5.1.5 (DP-completeness).** A DP-solution $\delta$ is *complete* if:

$$P \notin \delta(n) \implies \exists \sigma \in \Sigma. G \vdash \sigma : n \land \sigma : n \vdash \text{JDK}(P)$$

**Lemma 5.1.6.** Let $\delta$ be a complete DP-solution. Then, for each $n \in \mathbb{N}$:

$$\delta(n) \supseteq \Delta(n)$$

**Proof.** Let $\delta$ be a complete DP-solution, and $P \in \Delta(n)$. If $n$ is not $G$-reachable, then $\Delta(n)$ is the intersection of an empty collection of permission sets, which, by definition, equals to the universal set Permission. Therefore, the inclusion relation holds if also $\delta(n)$ equals to Permission. By contradiction, assume $P \notin \delta(n)$ for some permission $P$. Then, by definition 5.1.5, at least one $G$-reachable state $\sigma : n$ must exist. This clearly contradicts our hypothesis that $n$ is not $G$-reachable.

Otherwise, if $n$ is $G$-reachable, by definition 5.1.1 and assumption $P \in \Delta(n)$, we have that $\sigma : n \not\vdash \text{JDK}(P)$ for any $G$-reachable state $\sigma : n$. By contradiction, assume $P \notin \delta(n)$: then, by definition 5.1.5, at least one $G$-reachable state $\sigma : n$ exists such that $\sigma : n \vdash \text{JDK}(P)$. This raises a contradiction with hypothesis $\sigma : n \not\vdash \text{JDK}(P)$.

**Example 5.1.7.** Consider the e-commerce application of Fig. 3.1. The optimal DP-solution is shown in Table 5.1. Since the value of analysis does not matter for unreachable nodes, they are tagged with *unreach*.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\delta(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_0$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$n_1$</td>
<td>$\text{unreach}$</td>
</tr>
<tr>
<td>$n_2 \rightarrow n_4$</td>
<td>${P_{loan}, P_{read}, P_{write}}$</td>
</tr>
<tr>
<td>$n_5$</td>
<td>${P_{canpay}, P_{credit}, P_{debit}, P_{loan}, P_{read}, P_{write}}$</td>
</tr>
<tr>
<td>$n_6$</td>
<td>$\text{unreach}$</td>
</tr>
<tr>
<td>$n_7 \rightarrow n_8$</td>
<td>${P_{loan}, P_{read}, P_{write}}$</td>
</tr>
<tr>
<td>$n_9$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$n_{10} \rightarrow n_{12}$</td>
<td>${P_{loan}, P_{read}, P_{write}}$</td>
</tr>
<tr>
<td>$n_{13} \rightarrow n_{14}$</td>
<td>$\text{unreach}$</td>
</tr>
<tr>
<td>$n_{15} \rightarrow n_{16}$</td>
<td>${P_{loan}, P_{read}, P_{write}}$</td>
</tr>
<tr>
<td>$n_{17} \rightarrow n_{18}$</td>
<td>$\text{unreach}$</td>
</tr>
<tr>
<td>$n_{19}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$n_{20} \rightarrow n_{21}$</td>
<td>${P_{loan}, P_{read}, P_{write}}$</td>
</tr>
<tr>
<td>$n_{22}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$n_{23} \rightarrow n_{26}$</td>
<td>${P_{loan}, P_{read}, P_{write}}$</td>
</tr>
</tbody>
</table>

**Table 5.1:** The optimal DP-solution for the e-commerce application.
5.2 The Granted Permissions Analysis

We say that a permission $P$ is granted to a state $\sigma$ if $\sigma \vdash \text{JDK}(P)$. The Granted Permissions Analysis (GP for short) will give, for each program node, a safe approximation of the set of permissions that are certainly granted to any state reaching that node.

**Definition 5.2.1.** The set of permissions granted, respectively, to a state $\sigma$ and to a node $n$, are defined as:

$$
\Gamma(\sigma) = \begin{cases} 
\{P \in \text{Permission} \mid \sigma \vdash \text{JDK}(P)\} & \text{if } G \vdash \sigma \\
\emptyset & \text{otherwise}
\end{cases}
$$

$$
\Gamma(n) = \bigcap_{G \vdash \sigma : n} \Gamma(\sigma : n)
$$

Given any control flow graph $G$, a GP analysis is specified by means of an equation system $\text{GP}^=(G)$. We say that $\gamma$ is a GP-solution if $\gamma \models \text{GP}^=(G)$.

**Definition 5.2.2 (GP-soundness).** A GP-solution $\gamma$ is sound if:

$$
P \in \gamma(n) \implies \forall \sigma \in \Sigma. G \vdash \sigma : n \implies \sigma : n \vdash \text{JDK}(P)
$$

**Lemma 5.2.3.** Let $\gamma$ be a sound GP-solution. Then, for each node $n \in N$:

$$
\gamma(n) \subseteq \Gamma(n)
$$

**Proof.** Let $P \in \gamma(n)$, where $\gamma$ is a sound GP-solution. If $n$ is not $G$-reachable, then $\Gamma(n)$ is the intersection of an empty collection of permission sets, which, by definition, equals to the universal set Permission. Therefore, $P$ is trivially contained in $\Gamma(n)$.

Otherwise, if $n$ is $G$-reachable, by definition 5.2.2 we have that $\sigma : n \vdash \text{JDK}(P)$ for any $G$-reachable state $\sigma : n$. Actually, by definition 5.2.1, this means that $P \in \Gamma(\sigma : n)$ for any state $\sigma : n$ such that $G \vdash \sigma : n$, that is:

$$
P \in \bigcap_{G \vdash \sigma : n} \Gamma(\sigma : n)
$$

**Definition 5.2.4.** A GP-solution $\gamma$ is non-trivial if:

$$
P \in \text{Perm}(n) \land \text{Priv}(n) \implies P \in \gamma(n)
$$

**Definition 5.2.5 (GP-completeness).** A GP-solution $\gamma$ is complete if:

$$
P \notin \gamma(n) \implies \exists \sigma \in \Sigma. G \vdash \sigma : n \land \sigma : n \nvdash \text{JDK}(P)$$
5.2. THE GRANTED PERMISSIONS ANALYSIS

Lemma 5.2.6. Let $\gamma$ be a complete GP-solution. Then, for each $n \in \mathbb{N}$:

$$\gamma(n) \supseteq \Gamma(n)$$

Proof. Let $\gamma$ be a complete GP-solution, and $P \in \Gamma(n)$. If $n$ is not G-reachable, then $\Gamma[n]$ is the intersection of an empty collection of permission sets, which, by definition, equals to the universal set Permission. Therefore, the inclusion relation holds if also $\gamma(n)$ equals to Permission. By contradiction, assume $P \notin \gamma(n)$ for some permission $P$. Then, by definition 5.2.5, at least one G-reachable state $\sigma : n$ must exist. This clearly contradicts our hypothesis that $n$ is not G-reachable.

Otherwise, if $n$ is G-reachable, by definition 5.2.1 and assumption $P \in \Gamma[n]$, we have that $\sigma : n \vdash JDK(P)$ for any G-reachable state $\sigma : n$. By contradiction, assume $P \notin \gamma(n)$: then, by definition 5.2.5, at least one G-reachable state $\sigma : n$ exists such that $\sigma : n \not\vdash JDK(P)$. This raises a contradiction with hypothesis $\sigma : n \vdash JDK(P)$. \hfill \qed

Theorem 5.2.7. Let $\delta$ a sound DP-solution and $\gamma$ a sound GP-solution. Then, for any G-reachable node $n \in \mathbb{N}$:

$$\gamma(n) \subseteq \delta(n)$$

Proof. Assume $P \in \gamma(n)$, but also $P \notin \delta(n)$. This is a contradiction, because, if $G \vdash \sigma : n$, then DP-soundness ensures $\sigma : n \not\vdash JDK(P)$, while GP-soundness states $\sigma : n \vdash JDK(P)$. \hfill \qed

Example 5.2.8. The optimal GP-solution for example 3.1.2 is in Table 5.2.

<table>
<thead>
<tr>
<th>n</th>
<th>$\gamma(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_0$</td>
<td>${P_{\text{canPay}}, P_{\text{credit}}, P_{\text{debts}}, P_{\text{loan}}, P_{\text{read}}, P_{\text{write}}}$ unreach</td>
</tr>
<tr>
<td>$n_1$</td>
<td>${P_{\text{canPay}}, P_{\text{credit}}}$</td>
</tr>
<tr>
<td>$n_2 - n_4$</td>
<td>${P_{\text{canPay}}, P_{\text{credit}}, P_{\text{debts}}}$</td>
</tr>
<tr>
<td>$n_5$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$n_6$</td>
<td>unreach</td>
</tr>
<tr>
<td>$n_7 - n_8$</td>
<td>${P_{\text{canPay}}, P_{\text{credit}}, P_{\text{debts}}}$</td>
</tr>
<tr>
<td>$n_9$</td>
<td>${P_{\text{canPay}}, P_{\text{credit}}, P_{\text{debts}}, P_{\text{loan}}, P_{\text{read}}, P_{\text{write}}}$</td>
</tr>
<tr>
<td>$n_{10} - n_{12}$</td>
<td>${P_{\text{canPay}}, P_{\text{credit}}, P_{\text{debts}}}$</td>
</tr>
<tr>
<td>$n_{13} - n_{14}$</td>
<td>${P_{\text{canPay}}, P_{\text{credit}}, P_{\text{debts}}, P_{\text{loan}}, P_{\text{read}}, P_{\text{write}}}$</td>
</tr>
<tr>
<td>$n_{15}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$n_{16}$</td>
<td>${P_{\text{canPay}}, P_{\text{credit}}, P_{\text{debts}}}$</td>
</tr>
<tr>
<td>$n_{17} - n_{18}$</td>
<td>${P_{\text{canPay}}, P_{\text{credit}}, P_{\text{debts}}}$</td>
</tr>
<tr>
<td>$n_{19}$</td>
<td>${P_{\text{canPay}}, P_{\text{credit}}, P_{\text{debts}}}$</td>
</tr>
<tr>
<td>$n_{20} - n_{21}$</td>
<td>${P_{\text{canPay}}, P_{\text{credit}}, P_{\text{debts}}, P_{\text{loan}}, P_{\text{read}}, P_{\text{write}}}$</td>
</tr>
<tr>
<td>$n_{22}$</td>
<td>${P_{\text{canPay}}, P_{\text{credit}}, P_{\text{debts}}}$</td>
</tr>
<tr>
<td>$n_{23} - n_{26}$</td>
<td>${P_{\text{canPay}}, P_{\text{credit}}, P_{\text{debts}}, P_{\text{loan}}, P_{\text{read}}, P_{\text{write}}}$</td>
</tr>
</tbody>
</table>

Table 5.2: The optimal GP-solution for the e-commerce application.
5.3 The DP\(^0\) Analysis

\[
\overline{\text{DP}\_\text{in}}(n) = \bigcup_{(m,n) \in E} \overline{\text{DP}\_\text{out}}(m,n)
\]

\[
\overline{\text{DP}\_\text{out}}(m,n) =
\begin{cases} 
\text{Perm}(n) & \text{if } \bullet \rightarrow n \\
\overline{\text{DP}\_\text{call}}(m) \cap \text{Perm}(n) & \text{if } m \rightarrow n \\
\overline{\text{DP}\_\text{trans}}(m) & \text{if } m \rightarrow n
\end{cases}
\]

\[
\overline{\text{DP}\_\text{call}}(n) =
\begin{cases} 
\text{Perm}(n) & \text{if } \text{Priv}(n) \\
\overline{\text{DP}\_\text{in}}(n) & \text{otherwise}
\end{cases}
\]

\[
\overline{\text{DP}\_\text{trans}}(n) =
\begin{cases} 
\emptyset & \text{if } \text{kill}^0(n) \\
\overline{\text{DP}\_\text{in}}(n) \cup \{P\} & \text{if } \ell(n) = \text{check}(P) \text{ and } \neg \text{kill}^0(n) \\
\overline{\text{DP}\_\text{in}}(n) & \text{otherwise}
\end{cases}
\]

\[
\text{kill}^0(n) = \text{def } \ell(n) = \text{check}(P) \text{ and } P \notin \text{Perm}(n)
\]

Table 5.3: The DP\(^0\) Analysis.

The DP\(^0\) analysis is defined in Table 5.3 by means of the complemented analysis \(\overline{\text{DP}\_n}\) w.r.t. **Permission**. The intuition follows on how a solution is built.

- the permissions non-denied at the entry of a node are the union of those (non-denied) at the exit of all its callers.
- a call node generates non-denied permissions only if it is privileged; otherwise, it propagates the non-denied permissions at its entry.
- a check node \(n\) enforcing a permission \(P\) kills all the permissions at its entry if \(P \notin \text{Perm}(n)\); otherwise, it propagates the non-denied permissions at its entry, and generates the permission it enforces.
- return nodes have no outgoing edges, so they are irrelevant here.
- permissions non-denied at node \(n\) always belong to \(\text{Perm}(n)\).
5.3. THE DP^0 ANALYSIS

Lemma 5.3.1. Let \( \langle \delta_{in}, \delta_{out}, \delta_{call}, \delta_{trans} \rangle \models \text{DP}^0(G) \). Then:

\[
P \in \delta_{in}(n) \cap \text{Perm}(n) \implies \sigma \not\vdash \text{JDK}(P)
\]

for any G-reachable state \( \sigma : n \).

Proof. The proof is carried out by contradiction, assuming \( \sigma \vdash \text{JDK}(P) \). Then, we proceed by induction on the derivation used to establish \( G \vdash \sigma : n \). The base case corresponds to our single axiom:

\[
\frac{n \in \text{N}_{\text{entry}}}{[] \vdash [n]}
\]

For \( n \in \text{N}_{\text{entry}} \), we have that \( P \in \delta_{in}(n) \implies P \in \delta_{out}(\downarrow N, n) \). Then \( P \notin \text{Perm}(n) \), and it cannot be, at the same time, \( P \in \text{Perm}(n) \). Hence, the premises of the lemma are never satisfied, and the implication trivially holds. For the inductive case, we proceed by case analysis on the last rule used to derive \( \sigma : n \), yielding:

- case [call]:

\[
\frac{\ell([n']) = \text{call} \quad n' \rightarrow n}{\sigma' : n' \vdash \sigma' : n' : n \quad \text{where} \ \sigma = \sigma' : n'}
\]

By definition of \( \text{DP}^0_{in} \), \( P \in \delta_{in}[n] \implies P \in \delta_{out}[n', n] \), that is \( P \in \delta_{call}[n', n] \) or \( P \notin \text{Perm}(n) \). The latter option is prevented by our assumptions about \( P \), then only the former one is considered. If \( \neg \text{Priv}(n') \), then \( P \in \delta_{in}[n'] \); assumption \( \sigma' : n' \vdash \text{JDK}(P) \) imposes \( P \in \text{Perm}(n') \), hence we can apply the inductive hypothesis to deduce \( \sigma' \not\vdash \text{JDK}(P) \). This prevents the JDK rule to be applicable, and a contradiction arises with assumption \( \sigma' : n' \vdash \text{JDK}(P) \). Otherwise, if \( \text{Priv}(n') \), then it should hold \( P \notin \text{Perm}(n') \), contradicting again assumption \( \sigma' : n' \vdash \text{JDK}(P) \).

- case [check]:

\[
\frac{\ell([n']) = \text{check}(P') \quad \sigma : n' \vdash \text{JDK}(P') \quad n' \rightarrow n}{\sigma : n' \vdash \sigma : n}
\]

Here \( P \in \delta_{in}[n] \implies P \in \delta_{trans}[n'] \) and, by the premises of the \( \sigma_{\text{check}} \) rule, we also have \( P' \in \text{Perm}(n') \). Then, it must be \( P \in \delta_{in}[n'] \), and the inductive hypothesis can be applied to obtain a contradiction \( \sigma \not\vdash \text{JDK}(P) \).

- case [return]:

\[
\frac{\ell([m]) = \text{return} \quad n' \rightarrow m}{\sigma : n' : m \vdash \sigma : n}
\]

Here, \( P \in \delta_{in}[n] \implies P \in \delta_{trans}[n'] \). By lemma 3.4.4, it must be \( \ell([n']) = \text{call} \), and again it turns out that \( P \in \delta_{in}[n'] \). Since both \( n \) and \( n' \) carry the same permissions, we also have \( P \in \text{Perm}(n) \implies P \in \text{Perm}(n') \). By lemma 3.4.1, any derivation of \( \sigma : n' : m \) is of the form:

\[
[] \vdash \cdots \vdash \sigma : n' : m
\]

Therefore, we can apply the inductive hypothesis to \( \sigma : n' \), and a contradiction arises by noticing that the inductive hypothesis yields \( \sigma \not\vdash \text{JDK}(P) \).
Theorem 5.3.2. Let \( \langle \delta_{in}, \delta_{out}, \delta_{call}, \delta_{trans} \rangle \models DP^0(G) \). For any \( n \in N \), define:

\[
\delta(n) = \delta_{call}(n)
\]

Then \( \delta \) is a sound \( DP^0 \)-solution.

Proof. Let \( \sigma : n \) be a \( G \)-reachable state, and \( p \in \delta_{call}(n) \). By contradiction, assume \( \sigma : n \vdash JDK[p] \). This clearly implies \( p \notin Perm(n) \); otherwise, neither the \( JDK_{\bot} \) nor the \( JDK_{Prev} \) rules are applicable. Then we have to distinguish between two cases:

- if \( \neg P_iv(n) \), then \( p \in \delta_{in}(n) \), and by lemma 5.3.1 it follows \( \sigma \vdash JDK[p] \). This prevents the \( JDK_{\bot} \) rule to be applicable, hence a contradiction arises with our assumption \( \sigma : n \vdash JDK[p] \).

- if \( P_iv(n) \), then \( p \notin Perm(n) \). Again, this contradicts \( \sigma : n \vdash JDK[p] \).

\[\square\]

Example 5.3.3. Consider the e-commerce application of Fig. 3.1. By direct computation of the analysis at \( n_2 \), we obtain:

\[
\delta_{in}(n_2) = \delta_{out}(n_0, n_2) \cup \delta_{out}(n_3, n_2) \cup \delta_{out}(n_4, n_2) \\
= (\delta_{call}(n_0) \cap Perm(n_2)) \cup \delta_{trans}(n_3) \cup \delta_{trans}(n_4) \\
= (\delta_{in}(n_0) \cap Perm(n_2)) \cup \delta_{in}(n_3) \cup \delta_{in}(n_4) \\
= (Perm(n_0) \cap Perm(n_2)) \cup \delta_{out}(n_2, n_3) \cup \delta_{out}(n_2, n_4) \\
= \{ \text{pay}, \text{credit}, \text{debit} \} \cup \delta_{trans}(n_2) \\
= \{ \text{pay}, \text{credit}, \text{debit} \} \cup \delta_{in}(n_2)
\]

This recursive equation is satisfied by any superset of \( \{ \text{pay}, \text{credit}, \text{debit} \} \): since the property space for \( DP^0 \) is partially ordered by \( \subseteq \), this is just the MFP solution at \( n_2 \). Actually, this is the most accurate DP-sound solution: in fact, no permissions can be granted to an execution stack having \( n_2 \) as top element, outside those associated to the protection domain of \( n_2 \) (Client).

\[
\delta_{in}(n_5) = \delta_{out}(n_0, n_5) \cup \delta_{out}(n_6, n_5) = (\delta_{call}(n_0) \cap \emptyset) \cup \delta_{trans}(n_6) \\
= \delta_{in}(n_6) = \delta_{out}(n_5, n_6) = \delta_{trans}(n_5) = \delta_{in}(n_5)
\]

This recursive equation is satisfied by any set, so the MFP solution at \( n_5 \) is \( \emptyset \). Again, this is the most accurate DP-sound solution: in fact, no permissions
can be granted to an execution stack having \( n_5 \) as top element, because the protection domain of \( n_5 \) (Unknown) does not carry any permission.

\[
\delta_{in}(n_{16}) = \delta_{out}(n_4, n_{16}) \cup \delta_{out}(n_5, n_{16}) = \delta_{call}(n_4) \cup \delta_{call}(n_5) = \delta_{in}(n_4) \cup \delta_{in}(n_5) = \delta_{out}(n_2, n_4) \cup \delta_{in}(n_5) = \delta_{in}(n_2) \cup \delta_{in}(n_5)
\]

By the previous computations, we have that this equation is satisfied by any superset of \( \{ \text{P\_canpay}, \text{P\_credit}, \text{P\_debit}\} \). Therefore, taking the complement of the MFP solution, it follows that all the permissions except \( \{ \text{P\_canpay}, \text{P\_credit}, \text{P\_debit}\} \) belong to \( \delta(n_{16}) \). Actually, \( \text{P\_loan} \in \delta(n_{16}) \), too: then, it is not necessary to perform stack inspection in order to enforce the security check at \( n_{16} \), because, by the soundness result for the DP\(^0\) analysis, we know it will always fail.

The full MFP solution to DP\(^0\) for the e-commerce example is in Table A.1.

**Theorem 5.3.4.** DP\(^0\) is a monotone data flow analysis.

*Proof.* In order to minimize the notational effort, we only work on the complemented analysis \( \overline{\text{DP}\_P} \), since its monotonicity clearly implies monotonicity of DP\(^0\). The global property space for the complemented analysis is the pair \( \mathcal{L} = (\mathcal{L}, \subseteq) \), where:

\[
\mathcal{L} = \mathcal{L}_{in} \times \mathcal{L}_{out} \times \mathcal{L}_{call} \times \mathcal{L}_{trans}
\]

The sets \( \mathcal{L}_{in}, \mathcal{L}_{call}, \mathcal{L}_{trans} \) are total function spaces from \( N \) to \( \mathcal{P}(\text{Permission}) \), while \( \mathcal{L}_{out} \) is a total function space from \( E \) to \( \mathcal{P}(\text{Permission}) \). In symbols:

\[
\mathcal{L}_{in}, \mathcal{L}_{call}, \mathcal{L}_{trans} = N \rightarrow \mathcal{P}(\text{Permission})
\]

\[
\mathcal{L}_{out} = E \rightarrow \mathcal{P}(\text{Permission})
\]

Since, for any control flow graph, each of the sets \( N, E \) and \( \text{Permission} \) is finite, then also \( \mathcal{L}_{in}, \mathcal{L}_{call}, \mathcal{L}_{trans} \) and \( \mathcal{L}_{out} \) are finite, with cardinality:

\[
|\mathcal{L}_{in}| = |\mathcal{L}_{call}| = |\mathcal{L}_{trans}| = 2^{|N| \cdot |\text{Permission}|}
\]

\[
|\mathcal{L}_{out}| = 2^{|E| \cdot |\text{Permission}|}
\]

Assuming that \( \mathcal{P}(\text{Permission}) \) is partially ordered by the subset relation \( \subseteq \), a standard construction equips each of these spaces with a pointwise order. More precisely, we have:

\[
\mathcal{L}_{in} \subseteq \mathcal{L}'_{in} \quad \forall n \in N, \mathcal{L}_{in}(n) \subseteq \mathcal{L}'_{in}(n)
\]

\[
\mathcal{L}_{out} \subseteq \mathcal{L}'_{out} \quad \forall (m, n) \in E, \mathcal{L}_{out}(m, n) \subseteq \mathcal{L}'_{out}(m, n)
\]

\[
\mathcal{L}_{call} \subseteq \mathcal{L}'_{call} \quad \forall n \in N, \mathcal{L}_{call}(n) \subseteq \mathcal{L}'_{call}(n)
\]

\[
\mathcal{L}_{trans} \subseteq \mathcal{L}'_{trans} \quad \forall n \in N, \mathcal{L}_{trans}(n) \subseteq \mathcal{L}'_{trans}(n)
\]

With the above, our function spaces \( \mathcal{L}_{in}, \mathcal{L}_{out}, \mathcal{L}_{call}, \mathcal{L}_{trans} \) turn out to be finite partial orders. Since \( \mathcal{L} \) is their cartesian product, it follows that \( \mathcal{L} \) is a finite partial order, too; even more so, it fulfills the ascending chain condition. Actually, \( \mathcal{L} \) has cardinality:

\[
|\mathcal{L}| = 2^{(3|N| + |E|) \cdot |\text{Permission}|}
\]
The bottom element of \( \mathcal{L} \) is \( \bot_{\mathcal{L}} = (\bot_{\mathcal{L}_N}, \bot_{\mathcal{L}_{call}}, \bot_{\mathcal{L}_{return}}, \bot_{\mathcal{L}_{var}}, \bot_{\mathcal{L}_{def}}) \), where:

\[
\bot_{\mathcal{L}_N}, \bot_{\mathcal{L}_{call}}, \bot_{\mathcal{L}_{return}} = \bot_{N \to \mathcal{P}(Permission)} = \lambda n : N, \emptyset
\]

\[
\bot_{\mathcal{L}_var} = \bot_{E \to \mathcal{P}(Permission)} = \lambda (m, n) : E, \emptyset
\]

If we prove that, for any control flow graph, the global transfer function it is associated with is monotone, we can take \( \mathcal{F} \) to be the space of monotone functions over \( \mathcal{L} \). Now, given a control flow graph \( G \), the transfer function \( f = \mu(G) \) is defined by the equation system in Table 5.3 as a quadruple:

\[
f = (f_{in}, f_{out}, f_{call}, f_{trans})
\]

By definition of monotonicity, we have to prove that, for any \( l, l' \in \mathcal{L} \):

\[
l \subseteq l' \implies f(l) \subseteq f(l')
\]

So, let’s assume \( l \subseteq l' \). By definition of the \( \subseteq \) relation, we have to deal with four cases:

- \( f_{in}(l) \subseteq f_{in}(l') \). Let \( n \in N \). We have:

\[
f_{in}(l)(n) = \bigcup_{(m, n) \in E} l_{out}(m, n)
\]

\[
f_{in}(l')(n) = \bigcup_{(m, n) \in E} l'_{out}(m, n)
\]

Here hypothesis \( l \subseteq l' \) ensures that, for any \( (m, n) \in E \), it is \( l_{out}(m, n) \subseteq l'_{out}(m, n) \), and the inequality is preserved by the union operator on sets.

- \( f_{out}(l) \subseteq f_{out}(l') \). Here we have to distinguish between the three kinds of edges. Let \( (m, n) \in E \). If \( \rightarrow \), then \( m = \bot_N \), and the inequality trivially holds, because:

\[
f_{out}(l)(m, n) = \text{Perm}(n) = f_{out}(l')(m, n),
\]

Otherwise, if \( m \rightarrow n \), then:

\[
f_{out}(l)(m, n) = l_{call}(m) \cap \text{Perm}(n)
\]

\[
f_{out}(l')(m, n) = l'_{call}(m) \cap \text{Perm}(n)
\]

Here the inequality holds, as \( l \subseteq l' \) ensures \( l_{call}(m) \subseteq l'_{call}(m) \) for any node \( m \in N \). Otherwise, if \( m \rightarrow n \):

\[
f_{out}(l)(m, n) = l_{trans}(m)
\]

\[
f_{out}(l')(m, n) = l'_{trans}(m)
\]

Again, \( f \) preserves the inequality, because \( l \subseteq l' \) ensures \( l_{trans}(m) \subseteq l'_{trans}(m) \).

- \( f_{call}(l) \subseteq f_{call}(l') \). There two cases, according to \( n \) being privileged or not. In the first case, we have:

\[
f_{call}(l)(n) = \text{Perm}(n) = f_{call}(l')(n)
\]

and the inequality trivially holds. In the other case, we have:

\[
f_{call}(l)(n) = l_{in}(n)
\]

\[
f_{call}(l')(n) = l'_{in}(n)
\]

Hypothesis \( l \subseteq l' \) implies \( l_{in}(n) \subseteq l'_{in}(n) \), hence the inequality is preserved.
5.3. THE DP⁰ ANALYSIS

- \( f_{\text{trans}}(l) \subseteq f_{\text{trans}}(l') \). There are three cases. If \( \text{kill}^0(n) \), then \( l(n) = \text{check}(P) \) but \( P \not\in \text{Perm}(n) \). Thus:
  \[
  f_{\text{trans}}(l)(n) = \emptyset = f_{\text{trans}}(l')(n)
  \]
  Otherwise, if \( l(n) = \text{check}(P) \) and \( P \in \text{Perm}(n) \), then:
  \[
  f_{\text{trans}}(l)(n) = l_\text{in}(n) \cup \{P\} \\
  f_{\text{trans}}(l')(n) = l'_\text{in}(n) \cup \{P\}
  \]
  Here \( l \subseteq l' \implies l_\text{in}(n) \subseteq l'_\text{in}(n) \), and this suffices for the \( \subseteq \) relation to be preserved.
  The last case (\( n \) is a call) is just as above, after having discarded the singleton \( \{P\} \).

\[\square\]

Although we have proved that the DP⁰ analysis admits solutions for each of its instances, we are not given any effective method to compute them yet. Therefore, we now show that the data flow analysis \( \text{DP}^0 = \langle \mathcal{L}, \mathcal{F}, \mu \rangle \) constructed above can be turned into a distributive framework \( \langle \mathcal{L}_N, \mathcal{F}_E, \mu_E, \iota \rangle \).

This enables us to use a standard worklist-iteration algorithm to efficiently solve any instance of the DP⁰ analysis.

**Theorem 5.3.5.** \( \text{DP}^0 \) is a distributive data flow framework.

**Proof.** Again, for simplicity, we consider the complemented analysis \( \overline{\text{DP}^0} \). We have to build a quadruple \( \langle \mathcal{L}_N, \mathcal{F}_E, \mu_E, \iota \rangle \) satisfying the requirements in definitions 4.21 and 4.25.

The local property space is a triple \( \langle \mathcal{L}_N, \cup, \bot \rangle \), where \( \mathcal{L}_N = \mathcal{P}[^\text{Permission}] \), the join operator \( \cup \) is the binary union on sets, and \( \bot \) is the empty set (which is an unit for \( \cup \), since \( \cup \emptyset = \emptyset \)). Since \( \cup \) is idempotent, commutative and associative, this definition actually makes \( \mathcal{L}_N \) a join semi-lattice. By the fact that \( \mathcal{F}_E \) is finite for any control flow graph, it clearly follows that \( \mathcal{L}_N \) satisfies the ascending chain condition.

Moreover, a standard construction equips \( \mathcal{L}_N \) with a complete lattice structure:

\[
\mathcal{L}_N = \langle \mathcal{L}_N, \subseteq, \bigcup, \bigcap, \bot, \top \rangle
\]

where the partial order \( \subseteq \) is the inclusion relation \( \subseteq \) on sets, and each subset \( X \) of \( \mathcal{L}_N \) has least upper bound \( \bigsqcup X = \bigsqcup X \) and greatest lower bound \( \bigsqcap X = \bigsqcap X \). Furthermore, the bottom element is \( \bot = \emptyset \), and the top element is \( \top = \bigsqcup \mathcal{L}_N = \mathcal{L}_N = \mathcal{L}_N \).

Given a control flow graph \( G \), the set of local transfer functions \( f_E = \mu_E(G) \) is defined in Table 5.4. Again, if each \( f \in f_E \) is distributive regardless of \( G \), we can take \( f_E \) to be the space of distributive functions over \( \mathcal{L}_N \). By definition of distributivity, we have for prove that, for any \( l, l' \in \mathcal{L}_N \) and \( (m, n) \in E_p \):

\[
f_{(m, n)}(l \cup l') = f_{(m, n)}(l) \cup f_{(m, n)}(l')
\]

We have to deal with four cases:

- **case [entry]:** if \( \bullet \rightarrow n \), then \( f_{\rightarrow n}(l) = \text{Perm}(n) \) for any \( l \in \mathcal{L}_N \). Thus:
  \[
f_{\rightarrow n}(l \cup l') = \text{Perm}(n) = f_{\rightarrow n}(l) \cup f_{\rightarrow n}(l')
  \]
  follows by idempotency of \( \cup \).
\[
\begin{align*}
    f_{\rightarrow n}(l) &= Perm(n) \\
    f_{m \rightarrow n}(l) &= Perm(n) \cap \begin{cases} 
        Perm(m) & \text{if } Priv(m) \\
        l & \text{otherwise} 
    \end{cases} \\
    f_{m \rightarrow n}(l) &= \begin{cases} 
        \emptyset & \text{if } kill^0(l, m) \\
        l \cup \{P\} & \text{if } f(m) = \text{check}(P) \text{ and } \neg kill^0(l, m) \\
        l & \text{otherwise} 
    \end{cases} \\
    kill^0(l, n) &= \text{def } f(n) = \text{check}(P) \text{ and } P \notin Perm(n)
\end{align*}
\]

Table 5.4: Local transfer functions for the DP^0 Analysis.

- case \textit{[call]}: if \( m \rightarrow n \), we have two sub-cases, depending on \( m \) being privileged or not. In the first case, 
  \( f_{m \rightarrow n}(l) = Perm(n) \cap Perm(m) \) for any \( l \in L_N \). Then:
  \[ f_{m \rightarrow n}(l \cup l') = Perm(n) \cap Perm(m) = f_{m \rightarrow n}(l) \cup f_{m \rightarrow n}(l') \]
  follows by idempotency of \( \cup \). Otherwise, if \( m \) is not privileged:
  \[ f_{m \rightarrow n}(l \cup l') = Perm(n) \cap (l \cup l') = (Perm(n) \cap l) \cup (Perm(n) \cap l') = f_{m \rightarrow n}(l) \cup f_{m \rightarrow n}(l') \]
  follows by distributivity of \( \cap \) over \( \cup \).

- case \textit{[transfer]}: if \( m \rightarrow n \), we have three sub-cases, because the predicate \( kill^0 \) only depends on \( m \). If \( kill^0(l, m) \) then \( f(m) = \text{check}(P) \) but \( P \notin Perm(m) \). Therefore:
  \[ f_{m \rightarrow n}(l \cup l') = \emptyset = f_{m \rightarrow n}(l) \cup f_{m \rightarrow n}(l') \]
  follows by idempotency of \( \cup \). Otherwise, if \( f(m) = \text{check}(P) \) and \( P \notin Perm(m) \):
  \[ f_{m \rightarrow n}(l \cup l') = (l \cup l') \cup \{P\} = (l \cup \{P\}) \cup (l' \cup \{P\}) = f_{m \rightarrow n}(l) \cup f_{m \rightarrow n}(l') \]
  follows by associativity and idempotency of \( \cup \). The last case (\( m \) is not a check node) is just as above, after having discarded the singleton \( \{P\} \).

- case \textit{[return]}: if \( m \leftarrow n \), then \( f_{m \rightarrow n}(l) = \emptyset \) for any \( l \in L_N \). Therefore:
  \[ f_{m \rightarrow n}(l \cup l') = \emptyset = f_{m \rightarrow n}(l) \cup f_{m \rightarrow n}(l') \]
  follows by idempotency of \( \cup \).
5.3. THE $\text{DP}^0$ ANALYSIS

\[\text{Lemma 5.3.6.}\] Let \(\langle \delta_{\text{in}}, \delta_{\text{out}}, \delta_{\text{call}}, \delta_{\text{trans}} \rangle\) be the MFP solution to $\text{DP}^0$. Then:

\[P \in \overline{\delta}_{\text{in}}(n) \implies P \in \text{Perm}(n)\]

for any \(n \in \mathbb{N}\).

\[\text{Proof.}\] Let \(f\) be the global transfer function for the complemented analysis $\overline{\text{DP}^0}$. Then we know that the MFP solution to $\overline{\text{DP}^0}$ is the least upper bound of the (finite) chain:

\[\perp_{\mathcal{L}} \subseteq f(\perp_{\mathcal{L}}) \subseteq f^2(\perp_{\mathcal{L}}) \subseteq \cdots \subseteq f^k(\perp_{\mathcal{L}}) = f^{k+1}(\perp_{\mathcal{L}})\]

Now let \(l = f^i(\perp_{\mathcal{L}})\) for some index \(i \in \mathbb{N}\). If we define:

\[\phi(l) \equiv \forall n \in \mathbb{N}, P \in l_{\text{in}}(n) \implies P \in \text{Perm}(n)\]

then we can use a straightforward mathematical induction to prove \(\phi\) is true on the least $\overline{\text{DP}^0}$-solution. The base case asks for \(\phi(\perp_{\mathcal{L}})\): this is trivially true, since \(\perp_{\mathcal{L}}(n) = \emptyset\) for any node \(n \in \mathbb{N}\). For the inductive case, assume \(\phi(l)\) is true: we prove that this actually implies \(\phi(f(l))\). Take \(P \in f_{\text{in}}(l)(n)\). Then, we have:

\[P \in \bigcup_{(m,n) \in E} l_{\text{out}}(m,n)\]

Now, in order to prove that \(\forall (m,n) \in E, P \in l_{\text{out}}(m,n) \implies P \in \text{Perm}(n)\), we have to distinguish between the three kinds of (abstract) edges:

- if \(\cdot \rightarrow n\), then \(P \in \text{Perm}(n)\), and the statement clearly holds.
- if \(m \rightarrow n\), by monotonicity of \(f\) and definition of \(f_{\text{out}}\) we have:

\[l_{\text{out}}(m,n) \subseteq f_{\text{out}}(l)(m,n) = l_{\text{out}}(m) \cap \text{Perm}(n)\]

hence \(P \in \text{Perm}(n)\).
- if \(m \rightarrow n\), then:

\[l_{\text{out}}(m,n) \subseteq f_{\text{out}}(l)(m,n) = l_{\text{trans}}(m) \subseteq f_{\text{trans}}(l)(m)\]

In this case we have just two possibilities: \(P \in l_{\text{in}}(m)\), or \(P = P'\) if \(f(m) = \text{check}(P')\). In the first case, the hypothesis \(\phi(l)\) implies \(P \in \text{Perm}(m)\), and we can deduce \(P \in \text{Perm}(n)\) by the fact that both \(m\) and \(n\) lie in the same protection domain. The second case can only happen if \(P' \in \text{Perm}(m)\), and the same argument used above still works.

\[\square\]

\[\text{Lemma 5.3.7.}\] The MFP solution to $\text{DP}^0$ is non-trivial.

\[\text{Proof.}\] Let \(\delta\) be the greatest $\text{DP}^0$-solution for \(G\) and \(n\) a node. Since, by duality, \(\overline{\delta}\) is the least $\overline{\text{DP}^0}$-solution, lemma 5.3.6 tells \(P \in \overline{\delta}_{\text{in}}(n) \implies P \in \text{Perm}(n)\). This indeed implies non-triviality of \(\delta\), as \(P \in \overline{\delta}_{\text{out}}(n) \implies P \in \overline{\delta}_{\text{in}}(n) \lor P \in \text{Perm}(n)\).

\[\square\]
Counterexample 5.3.8. Not all DP₀-solutions are non-trivial.

Proof. Consider the control flow graph in Fig. 5.1. By direct computation, we have:

\[
\begin{align*}
\delta_{in}(n_1) &= \delta_{out}(n_0, n_1) \cup \delta_{out}(n_2, n_1) = (\delta_{call}(n_0) \cap \{P\}) \cup \delta_{trans}(n_2) \\
&= (\delta_{in}(n_0) \cap \{P\}) \cup \delta_{in}(n_2) = (\{P\} \cap \{P\}) \cup \delta_{out}(n_1, n_2) \\
&= \{P\} \cup \delta_{trans}(n_1) = \{P\} \cup \delta_{in}(n_1)
\end{align*}
\]

Note that the equation \(\delta_{in}(n_1) = \{P\} \cup \delta_{in}(n_1)\) is solved by any superset of \(\{P\}\). Define:

\[
\begin{align*}
\delta(n_0) &= \{P\} \\
\delta(n_1) &= \{P, P'\} \quad \text{for } i = 1, 3
\end{align*}
\]

Then, \(\delta\) is a trivial fixed point solution to DP₀: in fact, \(P' \notin \text{Perm}(n_1)\) but \(P' \notin \delta(n_1)\). □
5.4 The $\text{GP}^0$ Analysis

\[
\begin{align*}
\text{GP}^0_{\text{in}}(n) &= \bigcap_{(m,n) \in E} \text{GP}^0_{\text{out}}(m,n) \\
\text{GP}^0_{\text{out}}(m,n) &= \begin{cases} 
\text{Perm}(n) & \text{if } m \rightarrow n \\
\text{GP}^0_{\text{call}}(m) \cap \text{Perm}(n) & \text{if } m \rightarrow n \\
\text{GP}^0_{\text{trans}}(m) & \text{if } m \rightarrow n
\end{cases} \\
\text{GP}^0_{\text{call}}(n) &= \begin{cases} 
\text{Perm}(n) & \text{if } \text{Priv}(n) \\
\text{GP}^0_{\text{in}}(n) & \text{otherwise}
\end{cases} \\
\text{GP}^0_{\text{trans}}(n) &= \begin{cases} 
\emptyset & \text{if } \text{kill}^0(n) \\
\text{GP}^0_{\text{in}}(n) \cup \{P\} & \text{if } ([n] = \text{check}(P) \text{ and } \neg \text{kill}^0(n)} \\
\text{GP}^0_{\text{in}}(n) & \text{otherwise}
\end{cases}
\end{align*}
\]

Table 5.5: The $\text{GP}^0$ Analysis.

The $\text{GP}^0$ analysis is defined in Table 5.5 and explained below.

- the permissions granted at the entry of a node are those granted at the exit of all its callers.

- call nodes generate granted permissions only if they are privileged; otherwise they propagate those at their entry points.

- a check node $n$ enforcing a permission $P$ kills all the permissions at its entry if $P \notin \text{Perm}(n)$; otherwise, it propagates the granted permissions at its entry, and generates the permission it enforces.

- return nodes have no outgoing edges, so they are irrelevant here.

- permissions granted at node $n$ always belong to $\text{Perm}(n)$. 

Lemma 5.4.1. Let \(<\gamma_{in}, \gamma_{out}, \gamma_{call}, \gamma_{trans}> \models GP^0(G)\). Then:

\[ P \in \gamma_{in}(n) \implies \sigma \vdash JDK(P) \land P \in Perm(n) \]

for any \(G\)-reachable state \(\sigma : n\).

Proof. We proceed by induction on the derivation used to establish \(G \vdash \sigma : n\). The base case corresponds to our single axiom:

\[ n \in N_{entry} \]

\[ \vdash [n] \]

Here rule JDK\(_{out}\) ensures that \([n] \models JDK(P)\) for any permission \(P\), and \(P \in Perm(n)\) is due to the fact that \(P \in \gamma_{in}(n) \implies P \in \gamma_{out}(\downarrow N, n) = Perm(n)\) when \(n \in N_{entry}\). For the inductive case, we proceed by case analysis on the last rule used to derive \(\sigma : n\), yielding:

- **case [call]:**

  \[ \ell([n']) = \text{call} \implies n' \rightarrow n \]

  where \(\sigma = \sigma' : n'\)

  By definition of \(GP^0_{\text{in}}\), \(P \in \gamma_{in}(n) \implies P \in \gamma_{out}(\downarrow N, n)\), that is \(P \in \gamma_{call}(n')\) and \(P \in Perm(n)\). If \(- \text{Priv}(n')\), then \(P \in \gamma_{in}(n)\), so we can apply the inductive hypothesis to deduce \(\sigma' \vdash JDK(P) \land P \in Perm(n')\): then rule JDK\(_{out}\) ensures \(\sigma' : n' \vdash JDK(P)\). Otherwise, if \(\text{Priv}(n')\), then \(P \in Perm(n')\): in this case, \(\sigma' : n' \vdash JDK(P)\) is ensured by rule JDK\(_{priv}\).

- **case [check]:**

  \[ \ell([n']) = \text{check}(P') \implies n' \rightarrow n \]

  \(\sigma : n' \vdash \sigma : n\)

  Here \(P \in \gamma_{in}(n) \implies P \in \gamma_{trans}(n')\) and, by the premises of the \(D_{\text{check}}\) rule, we also have \(P' \in Perm(n')\). There are two cases: \(P \in \gamma_{in}(n')\) or \(P = P'\). In the first case, we can apply the inductive hypothesis to deduce \(\sigma' \vdash JDK(P') \land P \in Perm(n')\). Observe that, by the protection domain constraint, \(n' \rightarrow n\) implies \(Perm(n') = Perm(n)\), thus we can deduce \(P \in Perm(n)\). In the second case, \(\sigma : n' \vdash JDK(P)\) is assumed in the premises of the \(D_{\text{check}}\) rule. As check nodes are not privileged, this can be derived only by rule JDK\(_{out}\), whose premises are just \(\sigma \vdash JDK(P) \land P \in Perm(n')\).

- **case [return]:**

  \[ \ell([m]) = \text{return} \implies n' \rightarrow n \]

  \(\sigma : n' : m \vdash \sigma : n\)

  Here \(P \in \gamma_{in}(n) \implies P \in \gamma_{trans}(n')\). As, by lemma 3.4.4, it must be \(\ell([n']) = \text{call}\), then \(P \in \gamma_{in}(n')\). By lemma 3.4.1, any derivation of \(\sigma : n' : m\) is of the form:

  \[ \vdash \cdots \vdash \sigma : n' \vdash \cdots \vdash \sigma : n' : m \]

  Therefore, we can apply the inductive hypothesis to \(\sigma : n'\), obtaining \(\sigma \vdash JDK(P) \land P \in Perm(n')\). Again, this implies \(P \in Perm(n)\), since both \(n\) and \(n'\) lie in the same protection domain.

\(\square\)
Theorem 5.4.2. Let \( \langle \gamma_{in}, \gamma_{out}, \gamma_{call}, \gamma_{trans} \rangle \models \text{GP}^0(G) \). For any \( n \in \mathbb{N} \), define:

\[
\gamma(n) = \gamma_{call}(n)
\]

Then \( \gamma \) is a sound \( \text{GP}^0 \)-solution.

Proof. Let \( \sigma : n \) be a \( G \)-reachable state, and \( P \in \gamma_{call}(n) \). We have to distinguish between two cases:

- if \( \neg \text{Priv}(n) \), then \( P \in \gamma_{in}(n) \). Lemma 5.4.1 tells \( \sigma \vdash \text{JDK}(P) \) and \( P \in \text{Perm}(n) \); but these are exactly the premises of rule \( \text{JDK} \), therefore \( \sigma : n \vdash \text{JDK}(P) \).
- if \( \text{Priv}(n) \), then \( P \in \text{Perm}(n) \). Here, \( \sigma : n \vdash \text{JDK}(P) \) is ensured by rule \( \text{JDK}_{\text{Priv}} \).

\[ \square \]

Example 5.4.3. Consider the e-commerce application of Fig. 3.1. By direct computation of the analysis at \( n_{23} \), we obtain:

\[
\gamma_{in}(n_{23}) = \gamma_{out}(n_9, n_{23}) \cap \gamma_{out}(n_{13}, n_{23}) \cap \gamma_{out}(n_{20}, n_{23}) = (\gamma_{call}(n_9) \cap \gamma_{call}(n_{13}) \cap \gamma_{call}(n_{20})) \cap \text{Perm}(n_{23}) = \text{Perm}(n_9) \cap \text{Perm}(n_{13}) \cap \text{Perm}(n_{20}) = \{P_{\text{canpay}}, P_{\text{credit}}, P_{\text{debit}}, P_{\text{loan}}, P_{\text{read}}, P_{\text{write}}\}
\]

This equation admits an unique solution, i.e. the set of permissions associated with the protection domain Bank. Since \( P_{\text{read}} \in \gamma(n_{23}) \), the soundness result for the \( \text{GP}^0 \) analysis ensures that the security check at \( n_{23} \) will always succeed.

An analogous result holds for the analysis at node \( n_{25} \):

\[
\gamma_{in}(n_{25}) = \gamma_{out}(n_{14}, n_{25}) \cap \gamma_{out}(n_{21}, n_{25}) = (\gamma_{call}(n_{14}) \cap \gamma_{call}(n_{21})) \cap \text{Perm}(n_{25}) = \text{Perm}(n_{14}) \cap \text{Perm}(n_{21}) = \{P_{\text{canpay}}, P_{\text{credit}}, P_{\text{debit}}, P_{\text{loan}}, P_{\text{read}}, P_{\text{write}}\}
\]

The full MFP solution to \( \text{GP}^0 \) for the e-commerce example is in Table A.1.

Theorem 5.4.4. \( \text{GP}^0 \) is a distributive data flow framework.

Proof. The proof is exactly the same as the proof of theorem 5.3.5, because the local transfer functions for the \( \text{GP}^0 \) analysis are just those defined in Table 5.4. The local property space is dual to the space for the \( \text{GP}^0 \) analysis; now \( \sqcap \) is the binary intersection on sets, while \( \sqcup \in \mathbb{N} \) is the universal set \text{Permission}. \[ \square \]
Lemma 5.4.5. Let \( \langle \gamma_{\text{in}}, \gamma_{\text{out}}, \gamma_{\text{call}}, \gamma_{\text{trans}} \rangle \models \text{GP}^0(G) \). Then, for any \( n \in \mathbb{N} \):

\[
P \in \gamma_{\text{in}}(n) \quad \implies \quad P \in \text{Perm}(n)
\]

Proof. Let \( \delta = \langle \delta_{\text{in}}, \delta_{\text{out}}, \delta_{\text{call}}, \delta_{\text{trans}} \rangle \) be the MFP solution to \( \text{DP}^0 \), and assume \( P \in \gamma_{\text{in}}(n) \). As \( \gamma_{\text{in}}(n) \subseteq \gamma_{\text{call}}(n) \) regardless \( n \) being privileged or not, theorem 5.2.7 ensures \( P \in \delta_{\text{call}}(n) \). Then, since \( \delta \) is non-trivial by lemma 5.3.7, we deduce \( P \in \text{Perm}(n) \).

Lemma 5.4.6. Any \( \text{GP}^0 \)-solution is non-trivial.

Proof. Let \( \langle \gamma_{\text{in}}, \gamma_{\text{out}}, \gamma_{\text{call}}, \gamma_{\text{trans}} \rangle \models \text{GP}^0(G) \) Then, for any privileged node \( n \), we have exactly \( \gamma_{\text{call}}(n) = \text{Perm}(n) \).
5.5 The DP\(^1\) Analysis

\[
\begin{align*}
\overline{\text{DP}}^\text{in}(n) &= \bigcup_{(m,n) \in \mathcal{E}} \overline{\text{DP}}^\text{out}(m,n) \\
\overline{\text{DP}}^\text{out}(m,n) &= \begin{cases} 
\text{Perm}(n) & \text{if } \bullet \rightarrow n \\
\overline{\text{DP}}^\text{call}(m) \cap \text{Perm}(n) & \text{if } m \rightarrow n \\
\overline{\text{DP}}^\text{trans}(m) & \text{if } m \dashrightarrow n
\end{cases} \\
\overline{\text{DP}}^\text{call}(n) &= \begin{cases} 
\text{Perm}(n) & \text{if } \text{Priv}(n) \\
\overline{\text{DP}}^\text{in}(n) & \text{otherwise}
\end{cases} \\
\overline{\text{DP}}^\text{trans}(n) &= \begin{cases} 
\emptyset & \text{if } \text{kill}^1(n) \\
\bigcup_{(m,n) \in \mathcal{E}} \overline{\text{DP}}^\text{out}(m,n) & \text{if } \ell(n) = \text{check}(P) \text{ and } \neg \text{kill}^1(n) \\
\overline{\text{DP}}^\text{trans}(m,n) & \text{if } (m,n) \subseteq \mathcal{E} \\
\overline{\text{DP}}^\text{in}(n) & \text{otherwise}
\end{cases} \\
\text{kill}^1(n) &= \text{def } \ell(n) = \text{check}(P) \text{ and } P \notin \overline{\text{DP}}^\text{in}(n)
\end{align*}
\]

Table 5.6: The DP\(^1\) Analysis.

The DP\(^1\) analysis is defined in Table 5.6. In only differs from the DP\(^0\) analysis in the the set of non-denied permissions propagated through check nodes.

In the DP\(^0\) analysis, a check node \(n\) propagates all the permissions at its entry, unless the permission it enforces does not belong to \(\text{Perm}(n)\). This is a main source of degradation, as example 5.5.1 shows.

The DP\(^1\) analysis let a check node propagate the permissions of any caller that may pass the check. As we will see, this leads to strictly more accurate solutions than those achievable by the DP\(^0\) analysis.
**Example 5.5.1.** Consider the control flow graph in Fig. 5.2. By direct computation of the $\text{DP}^0$ analysis at $n_4$, we obtain:

$$
\delta_{in}(n_4) = \delta_{trans}(n_3) = \delta_{in}(n_3)
= \delta_{out}(n_0, n_3) \cup \delta_{out}(n_1, n_3) \cup \delta_{out}(n_2, n_3)
= \delta_{in}(n_0) \cup \delta_{in}(n_1) \cup \delta_{in}(n_2)
= \{P_0, P_1\} \cup \{P_1, P_2\} \cup \{P_0, P_3\} = \{P_0, P_1, P_2, P_3\}
$$

Observe that, since $P_3 \notin \delta^0(n_4)$, this solution is imprecise, because it does not predict that the check at $n_4$ will always fail. Next, we calculate:

$$
\delta_{out}(n_0, n_3) = \delta_{in}(n_0) = \{P_0, P_1\}
\delta_{out}(n_1, n_3) = \delta_{in}(n_1) = \{P_1, P_2\}
\delta_{out}(n_2, n_3) = \delta_{in}(n_2) = \{P_0, P_3\}
$$

Then, according to the $\text{DP}^1$ analysis, the permissions non-denied to $n_4$ are:

$$
\delta_{in}(n_4) = \delta_{trans}(n_3) = \bigcup_{(m, n_3) \in E} \{\delta_{out}(m, n_3) \mid P_1 \in \delta_{out}(m, n_3)\}
= \delta_{out}(n_0, n_3) \cup \delta_{out}(n_1, n_3) = \{P_0, P_1, P_2\}
$$

This shows that the $\text{DP}^1$ analysis is strictly more accurate than the $\text{DP}^0$. Theorem 5.5.3 will ensure that $\text{DP}^1$ enjoys the soundness property, too.
Lemma 5.5.2. Let \( \langle \delta_{\text{in}}, \delta_{\text{out}}, \delta_{\text{call}}, \delta_{\text{trans}} \rangle \models \text{DP}^1(G) \), and:

\[
\mathbb{L} = \sigma_0 \triangleright \sigma_1 \triangleright \cdots \triangleright \sigma_k = \sigma : \mathbb{N}
\]

be a derivation. Then:

\[
P \in \delta_{\text{out}}(\eta(\sigma_{k-1}, \sigma_k)) \cap \text{Perm}(\mathbb{N}) \implies \sigma \not\in \text{JDK}(P)
\]

Proof. The proof is carried out by contradiction, assuming \( \sigma \vdash \text{JDK}(P) \). Then, we proceed by induction on the length of derivation \( \mathbb{L} \triangleright \cdots \triangleright \sigma_{k-1} \triangleright \sigma : \mathbb{N} \). The base case corresponds to our single axiom:

\[
\frac{n \in N_{\text{entry}}}{\mathbb{L} \triangleright [n]}
\]

For \( n \in N_{\text{entry}} \), we have defined \( \eta([\mathbb{L}], [n]) = (\bot_{\mathbb{N}}, n) \). As \( P \in \delta_{\text{out}}(\bot_{\mathbb{N}}, n) \implies P \notin \text{Perm}(\mathbb{N}) \), \( P \) is prevented from being in \( \text{Perm}(\mathbb{N}) \). Hence, the premises of the lemma are never satisfied, and the implication trivially holds. For the inductive case, we proceed by case analysis on the rule used to derive \( \sigma_{k-1} \triangleright \sigma : \mathbb{N} \), yielding:

- **case [call]:**
  \[
  \frac{\ell([n]) = \text{call} \quad [n'] \rightarrow n}{\sigma' : [n'] \triangleright \sigma' : n' : n} \quad \text{where } \sigma_{k-1} = \sigma = \sigma' : n'
  \]

Here \( \eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma' : n', \sigma' : n' : n) = (n', n) \in E_{\text{call}} \), hence \( P \in \delta_{\text{out}}(n', n) \implies P \in \delta_{\text{call}}(n') \) or \( P \notin \text{Perm}(n) \). The latter option is prevented by our assumptions about \( P \), then only the former one is considered. If \( \neg \text{Priv}(n') \), then \( P \in \delta_{\text{in}}(n') \), and, using lemma 3.8.2, we obtain \( P \in \delta_{\text{out}}(\eta(\sigma_{k-2}, \sigma' : n')) \). Now, assumption \( \sigma' : n' \vdash \text{JDK}(P) \) implies \( P \in \text{Perm}(n') \), so we can apply the inductive hypothesis to deduce \( \sigma' \not\vdash \text{JDK}(P) \). This prevents the \text{JDK}_\triangleright \) rule to be applicable, and a contradiction arises with assumption \( \sigma' : n' \vdash \text{JDK}(P) \). Otherwise, if \( \text{Priv}(n') \), then it should hold \( P \notin \text{Perm}(n') \), contradicting again assumption \( \sigma' : n' \vdash \text{JDK}(P) \).

- **case [check]:**
  \[
  \frac{\ell([n]) = \text{check}(P') \quad \sigma : n' \vdash \text{JDK}(P') \quad n' \rightarrow n}{\sigma : n' \triangleright \sigma : n} \]

Here \( \eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma : n', \sigma : n) = (n', n) \in E_{\text{trans}} \), then \( P \in \delta_{\text{out}}(n', n) \implies P \in \delta_{\text{trans}}(n') \). As both \( n \) and \( n' \) lie in the same protection domain, assumption \( P \in \text{Perm}(n) \) also ensures \( P \in \text{Perm}(n') \). Now, it can be shown that:

\[
P \in \bigcap_{(m, n') \in E} \delta_{\text{out}}(m, n') \quad \text{(5.1)}
\]

As check nodes are not privileged, by rule \text{JDK}_\triangleright \) we know that \( \sigma : n' \vdash \text{JDK}(P') \) can only be true if \( \sigma \vdash \text{JDK}(P') \). Then, by applying contrapositively the inductive hypothesis, we deduce that \( P' \not\in \delta_{\text{out}}(\eta(\sigma_{k-2}, \sigma_{k-1})) \cap \text{Perm}(n') \). Now, if \( P' \in \text{Perm}(n') \) by equation \text{(5.1)}, then \( P \in \delta_{\text{out}}(\eta(\sigma_{k-2}, \sigma_{k-1})) \). Applying again the inductive hypothesis, we finally obtain a contradiction \( \sigma \not\vdash \text{JDK}(P) \).
• case [return]:

$$\ell(m) = \text{return } n' \to n$$

Here, $$\eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma : n' : m, \sigma : n) = (n', n) \in \mathcal{E}_{\text{trans}}:$$ thus, $$P \in \delta_{\text{out}}(n', n) \implies P \in \delta_{\text{trans}}(n').$$ By lemma 3.4.4, it must be $$\ell(n') = \text{call},$$ then it turns out that $$P \in \delta_{\text{in}}(n').$$ Now, lemma 3.4.1 states that $$\exists k \geq 2. \sigma_i = \sigma : n' :$$ by lemma 3.8.2, we then have that $$P \in \delta_{\text{out}}[\eta(\sigma_{i-1}, \sigma : n')]$$, and, again, assumption $$P \in \text{Perm}(n)$$ implies $$P \in \text{Perm}(n')$$ as both $$n$$ and $$n'$$ carry the same permissions. Hence, the inductive hypothesis can be applied to yield $$\sigma \not\vdash \text{JDK}(P).$$

Theorem 5.5.3. Let $$\langle \delta_{\text{in}}, \delta_{\text{out}}, \delta_{\text{call}}, \delta_{\text{trans}} \rangle \models \text{DP}^1(G).$$ For any $$n \in \mathbb{N},$$ define:

$$\delta(n) = \delta_{\text{call}}(n)$$

Then $$\delta$$ is a sound $$\text{DP}^1$$-solution.

Proof. Let $$\sigma : n$$ be a $$G$$-reachable state, and $$P \in \delta_{\text{call}}(n).$$ By contradiction, assume $$\sigma : n \not\vdash \text{JDK}(P).$$ This clearly implies $$P \in \text{Perm}(n):$$ otherwise, neither the $$\text{JDK}_{\text{rev}}$$ nor the $$\text{JDK}_{\text{prev}}$$ rules are applicable. Then we have to distinguish between two cases:

- if $$\neg \text{Priv}(n),$$ then $$P \in \delta_{\text{in}}(n).$$ Consider a derivation $$[] \leadsto \cdots \leadsto \sigma_{k-1} \leadsto \sigma : n.$$ By definition of $$\delta_{\text{in}}$$ we then have $$P \in \delta_{\text{out}}[\eta(\sigma_{k-1}, \sigma : n)],$$ and by lemma 5.5.2 it follows $$\sigma \not\vdash \text{JDK}(P).$$ Since the $$\text{JDK}_{\text{rev}}$$ rule is not applicable in this case, a contradiction arises with our assumption $$\sigma : n \vdash \text{JDK}(P).$$

- if $$\text{Priv}(n),$$ then $$P \not\in \text{Perm}(n).$$ Again, this contradicts $$\sigma : n \vdash \text{JDK}(P).$$

Example 5.5.4. Consider the e-commerce application of Fig. 3.1. Proceeding as in example 5.3.3, for the MFP solution at $$n_2$$ and $$n_5$$ we obtain:

$$\overline{\delta}_{\text{in}}(n_2) = \{\text{P\_cangpay}, \text{P\_credit}, \text{P\_debit}\}$$

$$\overline{\delta}_{\text{in}}(n_5) = \emptyset$$

Now, by direct computation of the analysis at edges $$(n_3, n_{11})$$ and $$(n_6, n_{11})$$:

$$\overline{\delta}_{\text{out}}(n_3, n_{11}) = \overline{\delta}_{\text{call}}(n_3) \cap \text{Perm}(n_{11}) = \overline{\delta}_{\text{in}}(n_3) \cap \text{Perm}(n_{11})$$

$$= \overline{\delta}_{\text{out}}(n_2, n_3) \cap \text{Perm}(n_{11}) = \overline{\delta}_{\text{trans}}(n_2) \cap \text{Perm}(n_{11})$$

$$= \overline{\delta}_{\text{in}}(n_2) \cap \text{Perm}(n_{11}) = \{\text{P\_cangpay}, \text{P\_credit}, \text{P\_debit}\}$$

$$\overline{\delta}_{\text{out}}(n_6, n_{11}) = \overline{\delta}_{\text{call}}(n_6) \cap \text{Perm}(n_{11}) = \overline{\delta}_{\text{in}}(n_6) \cap \text{Perm}(n_{11})$$

$$= \overline{\delta}_{\text{out}}(n_5, n_6) \cap \text{Perm}(n_{11}) = \overline{\delta}_{\text{trans}}(n_5) \cap \text{Perm}(n_{11})$$

$$= \overline{\delta}_{\text{in}}(n_5) \cap \text{Perm}(n_{11}) = \emptyset$$
5.5. THE DP¹ ANALYSIS

This shows that the security check at n₁₁ may be passed by executions reaching n₁₁ through (n₃, n₁₁), whereas the check will always fail for any execution flowing through (n₆, n₁₁). This is ensured by by lemma 5.5.2, because:

\[ P_{\text{debit}} \in \delta_{\text{out}}(n₆, n₁₁) \cap \text{Perm}(n₁₁) \implies \sigma : n₆ \nless \text{JDK}(P_{\text{debit}}) \]

for any derivation \( || \triangleright \cdots \triangleright \sigma : n₆ \triangleright \sigma : n₆ : n₁₁ \), and this clearly implies that \( \sigma : n₆ : n₁₁ \nless \text{JDK}(P_{\text{debit}}) \).

The full MFP solution to DP¹ for the e-commerce example is in Table A.1.

**Theorem 5.5.5.** DP¹ is a monotone data flow analysis.

*Proof.* The property space for \( \text{DP}^¹ \) is exactly the same as that for the \( \text{DP}^³ \) analysis. Let \( f \) be the \( \text{DP}^¹ \) transfer function, and \( l \subseteq l' \). The proof of monotonicity for \( f \) differs from the relative proof for the \( \text{DP}^³ \) analysis in just one case, i.e. \( f_{\text{trans}}(l) \subseteq f_{\text{trans}}(l') \) when \( l[n] = \text{check}(P) \) and \( \text{kill}^¹[n] \):

\[
\begin{align*}
  f_{\text{trans}}(l)[n] &= \bigcup_{(m,n) \in E} l_{\text{out}}(m,n) & \text{by def. } f_{\text{trans}} \\
  \subseteq \bigcup_{(m,n) \in E} l_{\text{out}}(m,n) & \text{by hyp. } l_{\text{out}} \subseteq l'_{\text{out}} \\
  \subseteq \bigcup_{(m,n) \in E} l'_{\text{out}}(m,n) & \text{by hyp. } l_{\text{out}} \subseteq l'_{\text{out}} \\
  = f_{\text{trans}}(l')[n] & \text{by def. } f_{\text{trans}}
\end{align*}
\]

\( \square \)

**Theorem 5.5.6.** DP¹ is a monotone data flow framework.

*Proof.* Again, it is worth considering the complemented analysis \( \overline{\text{DP}^¹} \). The local property space is a triple \( \langle \mathcal{L}_N, \sqcup, \sqcap \rangle \), where:

\[ \mathcal{L}_N = \mathcal{P}(\text{Permission}) \times \mathcal{P}(\text{Permission}) \]

The two components of the product represent the data flow information at the entry and at the exit of a node, respectively. The join operator \( \sqcup \) is defined in a coordinate-wise fashion:

\[ \langle l₀, l₁ \rangle \sqcup \langle l₀', l₁' \rangle = \langle l₀ \sqcup l₀', l₁ \sqcup l₁' \rangle \]

The bottom element is \( \bot_{\mathcal{L}_N} = (\varnothing, \varnothing) \). Since \( \sqcup \) is idempotent, commutative and associative, this definition actually makes \( \mathcal{L}_N \) a join semi-lattice. By the fact that Permission is finite for any control flow graph, it clearly follows that \( \mathcal{L}_N \) satisfies the ascending chain condition.

Again, a standard construction equips \( \mathcal{L}_N \) with a complete lattice structure:

\[ \mathcal{L}_N = \langle \mathcal{L}_N, \sqcup, \sqcap, \bot_{\mathcal{L}_N}, \top_{\mathcal{L}_N} \rangle \]
Table 5.7: Local transfer functions for the DP1 Analysis.

The partial order \( \sqsubseteq \) is the determined coordinatewise:

\[
(l_0, l_1) \sqsubseteq (l'_0, l'_1) \quad \text{iff} \quad l_0 \subseteq l'_0 \text{ and } l_1 \subseteq l'_1
\]

and the other components of the lattice are defined in the obvious manner.

Given a control flow graph \( G \), the set of local transfer functions \( f_e = \mu_e(G) \) is defined in Table 5.7. If each \( f \in \mathscr{F}_e \) is monotone regardless of \( G \), we can take \( \mathscr{F}_e \) to be the space of monotone functions over \( \mathbb{L}_n \). So, let \( (l_0, l_1) \sqsubseteq (l'_0, l'_1) \) be two arbitrary elements of \( \mathbb{L}_n \). By definition of monotonicity, we have for prove that, for any \( (m, n) \in \mathbb{E}_p \):

\[
f_{(m, n)}((l_0, l_1)) \sqsubseteq f_{(m, n)}((l'_0, l'_1))
\]

For the \( f_{(m, n)}^0 \) component, we have to deal with four cases:

- **case [entry]:** if \( \bullet \rightarrow n \), then \( f_{(m, n)}^0(l) = \text{Perm}(n) \) for any \( l \in \mathbb{L}_n \). Thus:
  \[
f_{(m, n)}^0((l_0, l_1)) = \text{Perm}(n) = f_{(m, n)}^0((l'_0, l'_1))
\]

- **case [call]:** if \( m \rightarrow n \), we have two cases, depending on \( m \) being privileged or not. In the first case, \( f_{(m, n)}^0(l) = \text{Perm}(n) \cap \text{Perm}(m) \) for any \( l \in \mathbb{L}_n \). Then:
  \[
f_{(m, n)}^0((l_0, l_1)) = \text{Perm}(n) \cap \text{Perm}(m) = f_{(m, n)}^0((l'_0, l'_1))
\]

Otherwise, if \( m \) is not privileged:

\[
f_{(m, n)}^0((l_0, l_1)) = \text{Perm}(n) \cap l_1 \sqsubseteq \text{Perm}(n) \cap l'_1 = f_{(m, n)}^0((l'_0, l'_1))
\]

follows by hypothesis \( l_1 \sqsubseteq l'_1 \).
5.5. THE DP\(^1\) ANALYSIS

- case [transfer]: if \(m \rightarrow n\), by hypothesis \(l_1 \subseteq l'_1\), we have:
  \[ f^0_{m \rightarrow n}((l_0, l_1)) = l_1 \subseteq l'_1 = f^0_{m \rightarrow n}((l^*_0, l'_1)) \]

- case [return]: if \(m \leftarrow n\), then \(f^0_{m \rightarrow n}(l) = \emptyset\) for any \(l \in \mathcal{L}_N\). Therefore:
  \[ f^0_{m \rightarrow n}((l_0, l_1)) = \emptyset = f^0_{m \rightarrow n}((l^*_0, l'_1)) \]

For the \(f^1_{(m,n)}\) component, let \(\overline{l}_0 = f^0_{m,n}((l_0, l_1))\) and \(\overline{l}_0' = f^0_{m,n}((l^*_0, l'_1))\). Observe that, by definition of \(\text{kill}^1\), we have, for any \(l, l' \in \mathcal{P}([\text{Permission}])\):
  \[ l \subseteq l' \land \text{kill}^1(l', n) \Rightarrow \text{kill}^1(l, n) \]

By monotonicity of \(f^0\), we know \(\overline{l}_0 \subseteq \overline{l}_0'\); therefore, \(\text{kill}^1(\overline{l}_0', n) \Rightarrow \text{kill}^1(\overline{l}_0, n)\), and we have to deal with the following cases:

- if \(\text{kill}^1(\overline{l}_0', n)\), then it must be \(\text{kill}^1(\overline{l}_0, n)\), too. Thus:
  \[ f^1_{(m,n)}((l_0, l_1)) = \emptyset = f^1_{(m,n)}((l^*_0, l'_1)) \]

- if \(\text{kill}^1(\overline{l}_0, n)\), but \(\neg \text{kill}^1(\overline{l}_0', n)\), then:
  \[ f^1_{(m,n)}((l_0, l_1)) = \emptyset \subseteq \overline{l}_0 = f^1_{(m,n)}((l^*_0, l'_1)) \]

- otherwise, if \(\neg \text{kill}^1(\overline{l}_0, n)\), then:
  \[ f^1_{(m,n)}((l_0, l_1)) = \overline{l}_0 \subseteq \overline{l}_0' = f^1_{(m,n)}((l^*_0, l'_1)) \]

\[\square\]

**Counterexample 5.5.7.** The DP\(^1\) analysis is *not* distributive.

**Proof.** By theorem 4.3.7, it suffices to show that the MOP and MFP solutions do not agree for some instance of the DP\(^1\) analysis. Consider the control flow graph in Fig. 5.3. First, we compute the MFP solution. At the exit of node \(n_2\), we have:

\[
\begin{align*}
\overline{\delta}_{\text{out}}(n_2, n_3) &= \overline{\delta}_{\text{out}}(n_2) = \overline{\delta}_{\text{in}}(n_2) \\
&= \overline{\delta}_{\text{out}}(n_0, n_2) \cup \overline{\delta}_{\text{out}}(n_1, n_2) \\
&= (\overline{\delta}_{\text{out}}(n_0) \cap \{P_0, P_1\}) \cup (\overline{\delta}_{\text{out}}(n_1) \cap \{P_0, P_1\}) \\
&= (\overline{\delta}_{\text{in}}(n_0) \cup \overline{\delta}_{\text{in}}(n_1)) \cap \{P_0, P_1\} \\
&= (\{P_0\} \cup \{P_1\}) \cap \{P_0, P_1\} = \{P_0, P_1\}
\end{align*}
\]

Since \(P_0 \in \overline{\delta}_{\text{out}}(n_2, n_3)\), then \(\neg \text{kill}^1(n_3)\), and:

\[\text{MFP}(n_4) = \overline{\delta}_{\text{in}}(n_4) = \overline{\delta}_{\text{out}}(n_3, n_4) = \overline{\delta}_{\text{trans}}(n_3) = \{P_0, P_1\}\]

For the MOP solution, we have two (abstract) paths leading to \(n_4\), i.e. \(\gamma_0 = (n_0, n_2, n_3, n_4)\) and \(\gamma_1 = (n_1, n_2, n_3, n_4)\). Given a path \((n_0, \ldots, n_k)\), we write \(i \mapsto n_0 \mapsto n_1 \mapsto \cdots \mapsto n_k \mapsto l_k\) for the sequence \((i, f_{n_0}(i), \ldots, f_{n_k}(i), l_k)\). Then the values of \(f_{\gamma_0}(i)\) and \(f_{\gamma_1}(i)\) are accumulated along the paths \(\gamma_0\) and \(\gamma_1\) as follows:

\[
\begin{align*}
i &\mapsto n_0 \quad \{P_0\} &\mapsto n_2 \quad \{P_0\} &\mapsto n_3 \quad \{P_0\} \\
i &\mapsto n_1 \quad \{P_1\} &\mapsto n_2 \quad \{P_1\} &\mapsto n_3 \quad \emptyset
\end{align*}
\]
Thus, the MOP solution at node $n_4$ is:

$$\text{MOP}(n_4) = f_{n_2}(i) \cup f_{n_1}(i) = \{P_0\} \cup \emptyset = \{P_0\}$$

Since $\text{MFP}(n_4) \neq \text{MOP}(n_4)$, the $D_P^1$ analysis cannot be distributive.

**Theorem 5.5.8.** Let $\delta^0$ and $\delta^1$ be the MFP solutions for the $D_P^0$ and $D_P^1$ analyses, respectively. Then $\delta^1$ is more accurate than $\delta^0$, i.e.:

$$\delta^1 \subseteq \delta^0$$

**Proof.** Let $f$ and $g$ be the global transfer functions for the complemented analyses $\overline{D_P^0}$ and $\overline{D_P^1}$, respectively. We know that the MFP solutions for $\overline{D_P^0}$ and $\overline{D_P^1}$ are the least upper bounds of the (finite) chains:

$$\bot \subseteq f(\bot) \subseteq f^2(\bot) \subseteq \cdots \subseteq f^{k}(\bot) = f^{k+1}(\bot)$$

$$\bot \subseteq g(\bot) \subseteq g^2(\bot) \subseteq \cdots \subseteq g^{h}(\bot) = g^{h+1}(\bot)$$

Now define:

$$\phi(l^0, l^1) \equiv l^0 \sqsubseteq l^1$$

The base case $\phi(\bot, \bot)$ is true, as $\bot = \bot = \bot = \bot$ for any $n,m \in N$, and clearly $\emptyset \subseteq \emptyset$. For the inductive case, we will prove:

$$l^0 = f(\bot) \land l^1 = g(\bot) \land \phi(l^0, l^1) \implies \phi(f^2(l^0), g(l^1))$$

This clearly implies that $g^h(l^1) \subseteq f^{2k}(l^0) = f^k(l^0)$. Note that we are allowed to prove just $\phi(f(l^0), g(l^1))$ whenever possible. In fact, by monotonicity of $f$, we have:

$$f^2(l^0) = f(f(l^0)) \sqsubseteq f(l^0)$$

provided $l^0 = f(\bot)$. Now, assume $\phi(l^0, l^1)$. By definition of the $\sqsubseteq$ relation, we have to provide a proof for each of the following four cases:
5.5. THE DP^1 ANALYSIS

- \( g_{in}(l^1) \subseteq f_{in}(l^0) \). Here we have:
  \[
  f_{in}(l^0)[n] = \bigcup_{(m,n) \in E} l_{out}^0(m,n)
  \]
  \[
  g_{in}(l^1)[n] = \bigcup_{(m,n) \in E} l_{out}^1(m,n)
  \]
  Here hypothesis \( \phi(l^0, l^1) \) ensures \( l_{out}^0(m,n) \supseteq l_{out}^1(m,n) \) for any \((m,n) \in E\), and the inequality is preserved by the union operator on sets.

- \( g_{out}(l^1) \subseteq f_{out}(l^0) \). Here we have to distinguish between the three kinds of edges. If \( \bullet \rightarrow n \), then:
  \[
  f_{out}(l^0)[m,n] = \text{Perm}(n) = f_{out}(l^1)[m,n]
  \]
  If \( m \rightarrow n \), then:
  \[
  f_{out}(l^0)[m,n] = l_{cal}(m) \cap \text{Perm}(n)
  \]
  \[
  g_{out}(l^1)[m,n] = l_{cal}(m) \cap \text{Perm}(n)
  \]
  and the inequality holds, because \( l^0 \supseteq l^1 \) ensures \( l_{cal}^0(m) \supseteq l_{cal}^1(m) \) for any node \( m \in N \). Otherwise, if \( m \rightarrow n \): 
  \[
  f_{out}(l^0)[m,n] = l_{\text{trans}}^0(m)
  \]
  \[
  g_{out}(l^1)[m,n] = l_{\text{trans}}^1(m)
  \]
  Again the inequality is preserved, because \( l^0 \supseteq l^1 \) ensures \( l_{\text{trans}}^0(m) \supseteq l_{\text{trans}}^1(m) \).

- \( g_{cal}(l^1) \subseteq f_{cal}(l^0) \). Here there are two cases, according to \( n \) being privileged or not. In the first case, we have:
  \[
  f_{cal}(l^0)[n] = \text{Perm}(n) = g_{cal}(l^1)[n]
  \]
  and the inequality trivially holds. In the other case, we have:
  \[
  f_{cal}(l^0)[n] = l_{in}^0(n)
  \]
  \[
  g_{cal}(l^1)[n] = l_{in}^1(n)
  \]
  Assumption \( l^0 \supseteq l^1 \) implies \( l_{in}^0(n) \supseteq l_{in}^1(n) \), hence the inequality is preserved.

- \( g_{trans}(l^1) \subseteq f_{trans}(l^0) \). The most interesting case is \( l(n) = \text{check}(P) \) and \( \neg \text{kill}^1(n) \). In such case, lemma 5.5.9 ensures \( P \in \text{Perm}(n) \), hence we have also \( \neg \text{kill}^0(n) \). Thus:
  \[
  g_{trans}(l^1)[n] = \bigcup_{(m,n) \in E} l_{out}^1(m,n)
  \]
  by def. \( g_{trans} \)
  \[
  = \bigcup_{(m,n) \in E} l_{out}^1(m,n) \cup \{ P \}
  \]
  by a property of \( \bigcup \)
  \[
  \subseteq \bigcup_{(m,n) \in E} l_{out}^0(m,n) \cup \{ P \}
  \]
  by a property of \( \bigcup \)
  \[
  \subseteq \bigcup_{(m,n) \in E} l_{out}^0(m,n) \cup \{ P \}
  \]
  by hypothesis \( l_{out}^0 \supseteq l_{out}^1 \)
  \[
  = f_{in}(l^0)[n] \cup \{ P \}
  \]
  by def. \( f_{in} \)
  \[
  = f_{trans}(f(l^0))[n]
  \]
  by def. \( f_{trans} \)
Otherwise, if \( \ell(n) = \text{check}(P) \) and \( \text{kill}^1(n) \), we have \( P \notin \text{kill}^1(n) \). Here we have to distinguish between two cases, i.e. \( P \in \text{Perm}(n) \) or not. In the first case, we have:

\[
\text{trans}(P)[n] = \text{kill}^0(n) \cup \{ P \} \supset \emptyset = \text{trans}(l^1)[n]
\]

If \( P \notin \text{Perm}(n) \), then \( \text{kill}^0(n) \) holds too. Thus:

\[
\text{trans}(P)[n] = \emptyset = \text{trans}(l^1)[n]
\]

The last case \( \ell(n) = \text{call} \) involves \( \text{trans}(l^0)[n] = \text{kill}^0(n) \) and \( \text{trans}(l^1)[n] = \text{kill}^1(n) \), hence assumption \( \Phi(l^0, l^1) \) does the job.

\[\square\]

**Lemma 5.5.9.** Let \( \langle \delta_{\text{in}}, \delta_{\text{out}}, \delta_{\text{call}}, \delta_{\text{trans}} \rangle \) be the MFP solution to \( \text{DP}^1 \). Then:

\[
P \in \overline{\delta}_{\text{in}}(n) \implies P \in \text{Perm}(n)
\]

for any \( n \in \mathbb{N} \).

*Proof.* The proof is similar to the proof of theorem 5.3.6. \[\square\]

**Lemma 5.5.10.** The MFP solution to \( \text{DP}^1 \) is non-trivial.

*Proof.* The result easily follows from 5.5.9, as it is shown in 5.3.7. \[\square\]
5.6 The $\text{GP}^1$ Analysis

\[
\text{GP}^1_{\text{in}}(n) = \bigcap_{(m,n) \in \mathcal{E}} \text{GP}^1_{\text{out}}(m,n)
\]

\[
\text{GP}^1_{\text{out}}(m,n) = \begin{cases} 
\text{Perm}(n) & \text{if } m \rightarrow n \\
\text{GP}^1_{\text{out}}(m) \cap \text{Perm}(n) & \text{if } m \rightarrow \rightarrow n \\
\text{GP}^1_{\text{trans}}(m) & \text{if } m \rightarrow \leftarrow n
\end{cases}
\]

\[
\text{GP}^1_{\text{out}}(n) = \begin{cases} 
\text{Perm}(n) & \text{if } \text{Priv}(n) \\
\text{GP}^1_{\text{in}}(n) & \text{otherwise}
\end{cases}
\]

\[
\text{GP}^1_{\text{trans}}(n) = \begin{cases} 
\emptyset & \text{if } \text{kill}^1(n) \\
\bigcap_{(m,n) \in \mathcal{E}} \text{GP}^1_{\text{out}}(m,n) \cup \left\{ P \right\} & \text{if } \ell(n) = \text{check}(P) \text{ and } \neg \text{kill}^1(n) \\
\text{GP}^1_{\text{in}}(n) & \text{otherwise}
\end{cases}
\]

Table 5.8: The $\text{GP}^1$ Analysis.

The $\text{GP}^1$ analysis is defined in Table 5.8. In only differs from the $\text{GP}^0$ analysis in the the set of granted permissions propagated through check nodes.

In the $\text{GP}^0$ analysis, a check node $n$ propagates all the permissions at its entry, unless the permission it enforces does not belong to $\text{Perm}(n)$. This is a main source of degradation, as example 5.6.1 shows.

The $\text{GP}^1$ analysis let a check node propagate both the permission it enforces and the permissions granted to all the callers that may pass the check. Any sound DP-analysis can be used to this purpose. As we will see, this leads to strictly more accurate solutions than those achievable by the $\text{GP}^0$ analysis.
Example 5.6.1. Consider the control flow graph in Fig. 5.4. By direct computation of the $\text{GP}^0$ analysis at $n_4$, we obtain:

$$\gamma_{in}^0(n_4) = \gamma_{trans}^0(n_3) = \gamma_{in}^0(n_3) \cup \{P_1\}$$

$$= (\gamma_{out}^0(n_0,n_3) \cap \gamma_{out}^0(n_1,n_3) \cap \gamma_{out}^0(n_2,n_3)) \cup \{P_1\}$$

$$= (\gamma_{in}^0(n_0) \cap \gamma_{in}^0(n_1) \cap \gamma_{in}^0(n_2)) \cup \{P_1\}$$

$$= ([P_0,P_1,P_2] \cap [P_0,P_3] \cap [P_1,P_2,P_3]) \cup \{P_1\} = \{P_1\}$$

Observe that, since $P_2 \notin \gamma^0(n_4)$, this solution is imprecise, because it does not predict that the check at $n_4$ will always succeed. By example 5.5.1, we have:

$$\delta_{out}^I(n_0,n_3) = \delta_{in}^I(n_0) = \{P_0,P_1\}$$

$$\delta_{out}^I(n_1,n_3) = \delta_{in}^I(n_1) = \{P_1,P_2\}$$

$$\delta_{out}^I(n_2,n_3) = \delta_{in}^I(n_2) = \{P_0,P_3\}$$

Then, according to the $\text{GP}^1$ analysis, the permissions granted to $n_4$ are:

$$\gamma_{in}^1(n_4) = \gamma_{in}^1_{imm}(n_3) = \bigcap_{(m,n_3) \in \mathcal{E}} \{\gamma_{out}^1(m,n_3) \mid P_1 \in \delta_{out}^I(m,n_3)\} \cup \{P_1\}$$

$$= (\gamma_{out}^1(n_0,n_3) \cap \gamma_{out}^1(n_2,n_3)) \cup \{P_1\} = \{P_1,P_2\}$$

This shows that the $\text{GP}^1$ analysis is strictly more accurate than the $\text{GP}^0$. Theorem 5.6.3 will ensure that $\text{GP}^1$ enjoys the soundness property, too.
Lemma 5.6.2. Let \( \langle \gamma_{in}, \gamma_{out}, \gamma_{call}, \gamma_{trans} \rangle \models \text{GP}^1(G) \), and:

\[
[] = \sigma_0 \triangleright \sigma_1 \triangleright \cdots \triangleright \sigma_k = \sigma : n
\]

be a derivation. Then:

\[
P \in \gamma_{out}(\eta(\sigma_{k-1}, \sigma_k)) \implies \sigma \vdash \text{JKD}(P) \land P \in \text{Perm}(n)
\]

Proof. We proceed by induction on the length of derivation \( [] \triangleright \cdots \triangleright \sigma_{k-1} \triangleright \sigma : n \). The base case corresponds to our single axiom:

\[
n \in N_{entry} \implies [] \triangleright [n]
\]

For \( n \in N_{entry} \), we have defined \( \eta([], [n]) = (\bot_N, n) \), so \( P \in \gamma_{out}(\bot_N, n) \implies P \in \text{Perm}(n) \), and \( \sigma = [] \vdash \text{JKD}(P) \) by rule JKD\(_a\). For the inductive case, we proceed by case analysis on the rule used to derive \( \sigma_{k-1} \triangleright \sigma : n \), yielding:

- case [call]:

\[
\begin{align*}
\ell(n') & = \text{call} \quad n' \rightarrow n \\
\sigma' : n' & \triangleright \sigma' : n' : n \\
\text{where } & \sigma_{k-1} = \sigma = \sigma' : n'
\end{align*}
\]

Here \( \eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma' : n', \sigma' : n' : n) = [n', n] \in E_{\text{call}} \), hence \( P \in \gamma_{out}[n', n] \implies P \in \gamma_{call}[n'] \) and \( P \in \text{Perm}(n) \). If \( \neg \text{Priv}(n') \), then \( P \in \gamma_{in}[n'] \), and, using lemma 3.8.2, we obtain \( P \in \gamma_{out}(\eta(\sigma_{k-2}, \sigma_{k-1})) \). Then we can apply the inductive hypothesis to deduce \( \sigma' \vdash \text{JKD}(P) \) and \( P \in \text{Perm}(n') \), thus obtaining \( \sigma' : n' \vdash \text{JKD}(P) \) by rule JKD\(_a\). Otherwise, if \( \text{Priv}(n') \), then \( P \in \text{Perm}(n') \): in this case, \( \sigma' : n' \vdash \text{JKD}(P) \) is ensured by rule JKD\(_{priv}\).

- case [check]:

\[
\begin{align*}
\ell(n') & = \text{check}(P') \quad \sigma : n' \vdash \text{JKD}(P') \quad n' \rightarrow n \\
\sigma : n' & \triangleright \sigma : n
\end{align*}
\]

Here \( \eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma : n', \sigma : n) = [n', n] \in E_{\text{trans}} \), then \( P \in \gamma_{out}[n', n] \implies P \in \gamma_{trans}[n'] \). Now, it can be shown that:

\[
P \in \bigcap_{(m, n') \in E \land \{P' \in \delta_{\text{trans}}(m, n') \}} \gamma_{out}[m, n'] \cup \{P'\} \quad (5.2)
\]

If \( P = P' \), then \( \sigma : n' \vdash \text{JKD}(P) \) follows by the premises of the \( \triangleright \text{check} \) rule, and this obviously implies \( P \in \text{Perm}(n') \). As check nodes are not privileged, \( \sigma \vdash \text{JKD}(P) \) follows from rule JKD\(_a\), and \( P \in \text{Perm}(n) \) from the fact that \( n \) and \( n' \) carry the same permissions. Otherwise, if \( P \neq P' \), premise \( \sigma : n' \vdash \text{JKD}(P') \) implies \( P \in \text{Perm}(n') \), and rule JKD\(_a\) also states \( \sigma \vdash \text{JKD}(P') \). Then, by applying contrapositively lemma 5.5.2, it turns out that \( P' \notin \delta_{\text{out}}[\eta(\sigma_{k-2}, \sigma_{k-1})] \). Therefore, by equation (5.2), it is indeed \( P \in \gamma_{out}(\eta(\sigma_{k-2}, \sigma_{k-1})) \), and, applying the inductive hypothesis, we finally obtain \( \sigma \vdash \text{JKD}(P) \).
case \texttt{return}:

\[
\ell([m]) = \texttt{return } n' \rightarrow n
\]

Here, \(\eta[s_k-1, s_k] = \eta[s : n' : m, s : n] = (n', n) \in E_{\text{trans}}\) thus, \(P \in \gamma_{\text{out}}(n', n) \Rightarrow P \in \gamma_{\text{trans}}(n')\). By lemma 3.4.4, it must be \(\ell(n') = \texttt{call}\), then it turns out that \(P \in \gamma_{\text{in}}(n')\). Now, lemma 3.4.1 states that \(\exists i \in 1..k-2, s_i = s : n'\), and by lemma 3.8.2, we then have that \(P \in \gamma_{\text{out}}[\eta[s_i-1, s_i : n']]\). Therefore we can apply the inductive hypothesis, obtaining \(\sigma \vdash \text{JDK}[P]\) and \(P \in \text{Perm}(n')\): again, \(P \in \text{Perm}(n') \Rightarrow P \in \text{Perm}(n)\), as both \(n\) and \(n'\) lie in the same protection domain.

\[\square\]

**Theorem 5.6.3.** Let \(\langle \gamma_{\text{in}}, \gamma_{\text{out}}, \gamma_{\text{call}}, \gamma_{\text{trans}} \rangle \models \text{GP}^1(G)\). For any \(n \in N\), define:

\[
\gamma(n) = \gamma_{\text{call}}(n)
\]

Then \(\gamma\) is a sound \(\text{GP}^1\)-solution.

**Proof.** Let \(\gamma \models \text{GP}^1(G)\), \(\sigma : n\) a G-reachable state, and \(P \in \gamma_{\text{call}}(n)\). We have to distinguish between two cases:

- if \(\neg \text{Priv}(n)\), then \(P \in \gamma_{\text{in}}(n)\). Consider a derivation \(\langle \! \langle \! \rightarrow \cdots \rightarrow s_{k-1} \rightarrow s : n \rangle \! \rangle\). By definition of \(\text{GP}^1_{\text{in}}\), we then have \(P \in \gamma_{\text{out}}[\eta[s_{k-1}, s : n]]\), and by lemma 5.6.2 it follows \(\sigma \vdash \text{JDK}[P]\) and \(P \in \text{Perm}(n)\). But these are exactly the premises of rule \(\text{JDK}_{\text{in}}\), therefore \(\sigma : n \vdash \text{JDK}[P]\).

- if \(\text{Priv}(n)\), then \(P \in \gamma_{\text{in}}(n)\) or \(P \in \text{Perm}(n)\). In the former case, we showed above that \(\sigma : n \vdash \text{JDK}[P]\); in the latter, this is ensured by rule \(\text{JDK}_{\text{priv}}\).

\[\square\]

**Example 5.6.4.** Consider the e-commerce application of Fig. 3.1. By direct computation of the analysis at \(n_2\), we obtain:

\[
\gamma_{\text{in}}(n_2) = \gamma_{\text{out}}(n_0, n_2) \cap \gamma_{\text{out}}(n_3, n_2) \cap \gamma_{\text{out}}(n_4, n_2)
\]

\[
= (\gamma_{\text{call}}(n_0) \cap \text{Perm}(n_2)) \cap \gamma_{\text{trans}}(n_3) \cap \gamma_{\text{trans}}(n_4)
\]

\[
= \gamma_{\text{in}}(n_0) \cap \text{Perm}(n_2) \cap \gamma_{\text{in}}(n_3) \cap \gamma_{\text{in}}(n_4)
\]

\[
= \text{Perm}(n_0) \cap \text{Perm}(n_2) \cap \gamma_{\text{out}}(n_2, n_3) \cap \gamma_{\text{out}}(n_2, n_4)
\]

\[
= \{P_{\text{canpay}}, P_{\text{credit}}, P_{\text{debit}}\} \cap \gamma_{\text{trans}}(n_2)
\]

\[
= \{P_{\text{canpay}}, P_{\text{credit}}, P_{\text{debit}}\} \cap \gamma_{\text{in}}(n_2)
\]

This recursive equation is satisfied by any subset of \(\{P_{\text{canpay}}, P_{\text{credit}}, P_{\text{debit}}\}\): since, regardless of some technical details, the property space for \(\text{GP}^1\) can be seen as partially ordered by \(\supseteq\), this is just the MFP solution at \(n_2\).
The set of permissions granted at the exit of $n_{11}$ is:

$$
\gamma_{trans}(n_{11}) = \bigcap_{(m, n_{11}) \in E} \gamma_{out}(m, n_{11}) \cup \{P_{debit}\}
$$

Now, the only edges leading to $n_{11}$ are $(n_3, n_{11})$ and $(n_6, n_{11})$. By example 5.5.4, we have that $P_{debit} \in \delta_{out}(n_6, n_{11})$, and $P_{debit} \notin \delta_{out}(n_3, n_{11})$. Then:

$$
\gamma_{trans}(n_{11}) = \gamma_{out}(n_3, n_{11}) \cup \{P_{debit}\} = \gamma_{call}(n_3) \cup \{P_{debit}\}
$$

$$
= \gamma_{in}(n_3) \cup \{P_{debit}\} = \gamma_{out}(n_2, n_3) \cup \{P_{debit}\}
$$

$$
= \gamma_{trans}(n_2) \cup \{P_{debit}\} = \gamma_{in}(n_2) \cup \{P_{debit}\}
$$

$$
= \{P_{canpay}, P_{credit}, P_{debit}\}
$$

Therefore, the set of permissions granted at the entry of node $n_8$ is:

$$
\gamma_{in}(n_8) = \gamma_{out}(n_2, n_8) \cap \gamma_{out}(n_{12}, n_8) = \gamma_{call}(n_2) \cap \gamma_{call}(n_{12})
$$

$$
= \gamma_{in}(n_2) \cap \gamma_{in}(n_{12}) = \gamma_{in}(n_2) \cap \gamma_{trans}(n_{11})
$$

$$
= \{P_{canpay}, P_{credit}, P_{debit}\}
$$

Since $P_{canpay} \in \gamma(n_8)$, the soundness result for the GP$^1$ analysis ensures that the security check at $n_8$ will always succeed. Note that this information was not discovered by the GP$^0$ analysis.

The full MFP solution to GP$^1$ for the e-commerce example is in Table A.1.

**Theorem 5.6.5.** GP$^1$ is a monotone data flow framework.

**Proof.** The proof closely resembles the one of theorem 5.5.6, hence it is only sketched here.

The set of local transfer functions is defined in Table 5.9. A solution $\delta$ to the DP$^1$ analysis is used to specify how the information flows from the entry to the exit of nodes.

The local property space is dual to the space for the DP$^1$ analysis: set union is replaced by set intersection, and the order relation $\subseteq$ is replaced with $\supseteq$. □

**Theorem 5.6.6.** Let $\gamma^0$ and $\gamma^1$ be the MFP solutions for the GP$^0$ and GP$^1$ analyses, respectively. Then $\gamma^1$ is more accurate than $\gamma^0$, i.e.:

$$
\gamma^1 \sqsubseteq \gamma^0
$$

**Proof.** The proof is similar to the one of theorem 5.5.8, then it is not carried out here. □

**Corollary 5.6.7.** Any GP$^1$-solution is non-trivial.
\[
\begin{align*}
    f_{m,n}(\langle l_0, l_1 \rangle) &= (f_{m,n}^0, f_{m,n}^1)(\langle l_0, l_1 \rangle) \\
    f_{m \rightarrow n}^0(\langle l_0, l_1 \rangle) &= \text{Perm}(n) \\
    f_{m \leftarrow n}^0(\langle l_0, l_1 \rangle) &= \text{Perm}(n) \cap \begin{cases} 
        \text{Perm}(m) & \text{if } \text{Priv}(m) \\
        l_1 & \text{otherwise}
    \end{cases} \\
    f_{m \rightarrow n}^1(\langle l_0, l_1 \rangle) &= l_1 \\
    f_{m \leftarrow n}^1(\langle l_0, l_1 \rangle) &= \emptyset \\
    f_{m,n}^1(\langle l_0, l_1 \rangle) &= \begin{cases} 
        \emptyset & \text{if } \text{kill}^1(\delta(n), n) \\
        f_{m,n}^0(\langle l_0, l_1 \rangle) & \text{otherwise}
    \end{cases} \\
    \text{kill}^1(l, n) &= \text{def } \ell(n) = \text{check}(P) \text{ and } P \notin l
\end{align*}
\]

Table 5.9: Local transfer functions for the GP^1 Analysis.
5.7 The DP^2 Analysis

\[
\begin{align*}
\text{DP^2}_{\text{in}}(n) &= \begin{cases} 
unreach & \text{if } \forall (m, n) \in E, \text{DP^2}_{\text{out}}(m, n) = unreach \\
\bigcup_{(m, n) \in E} \text{DP^2}_{\text{out}}(m, n) & \text{otherwise}
\end{cases} \\
\text{DP^2}_{\text{out}}(m, n) &= \begin{cases} 
unreach & \text{if } \text{DP^2}_{\text{in}}(m) = unreach \\
\text{Perm}(n) & \text{if } m \rightarrow n \\
\text{DP^2}_{\text{call}}(m) \cap \text{Perm}(n) & \text{if } m \rightarrow\rightarrow n \\
\text{DP^2}_{\text{trans}}(m) & \text{if } m \rightarrow\rightarrow n
\end{cases} \\
\text{DP^2}_{\text{call}}(n) &= \begin{cases} 
\text{Perm}(n) & \text{if } \text{Priv}(n) \\
\text{DP^2}_{\text{in}}(n) & \text{otherwise}
\end{cases} \\
\text{DP^2}_{\text{trans}}(n) &= \begin{cases} 
unreach & \text{if } \text{DP^2}_{\text{in}}(n) = unreach \text{ or kill}^2(n) \\
\bigcup_{(m, n) \in E} \text{DP^2}_{\text{out}}(m, n) & \text{if } \ell(n) = \text{check}(P) \text{ and } \neg \text{kill}^2(n) \\
\text{DP^2}_{\text{in}}(n) & \text{otherwise}
\end{cases} \\
\text{kill}^2(n) &= \text{def } \ell(n) = \text{check}(P) \text{ and } P \notin \text{DP^2}_{\text{in}}(n), \text{ or } \\
& \quad \ell(n) = \text{call} \text{ and } \forall m \in p(n), \text{DP^2}_{\text{in}}(m) = unreach
\end{align*}
\]

Table 5.10: The DP^2 Analysis.

The DP^2 analysis, defined in Table 5.10, improves the DP^1 analysis by introducing a special value unreach to represent unreachable nodes.

Contrary to the previous analyses, the DP^2 analysis prevents unreachable nodes from propagating permissions. As we will see, this leads to strictly more accurate solutions than those achievable by the DP^1 analysis.
Three kinds of unreachable nodes are detected by the DP² analysis. Let \( n \) be a node, and \( m \) one of its callers, i.e. \((m, n) \in E\). We say that \( n \) is DP-unreachable from \( m \) if one of the following conditions holds:

- \( m \) is DP-unreachable;
- \( m \) is a check against a permission \( P \), and none of its callers has \( P \) among its non-denied permissions;
- \( m \) is a call, \( m \rightarrow n \), and all possible return points of \( m \), i.e. all nodes in \( \rho(m) \), are DP-unreachable.

Then, we say that \( n \) is DP-unreachable if \( n \) is DP-unreachable from each of its callers. However, this definition does not fully characterize the set of unreachable nodes (i.e. nodes \( n \) for which no derivation \( I \vdash \cdots \vdash \sigma : n \) exists), as pointed out by example 5.7.1.

**Example 5.7.1.** Consider the control flow graph in Fig. 5.5, where we have \( \rho(n_0) = \{ n_3 \} = \rho(n_2) \). According to the informal definition given above, \( n_5 \) is not DP-unreachable, because the check at \( n_4 \) can be passed by caller \( n_0 \). Consequently, also \( n_3 \) in not DP-unreachable. However, \( n_3 \) is unreachable indeed: in fact, the only concrete path leading to \( n_3 \) is \( \langle n_2, n_4, n_5, n_3 \rangle \), which is not traversable because \( [n_2, n_4] \not\in JDK(P_0) \).
Lemma 5.7.2. Let \( \langle \delta_{in}, \delta_{out}, \delta_{call}, \delta_{trans} \rangle \models \text{DP}^2(G) \), and:

\[
[] = \sigma_0 \triangleright \sigma_1 \triangleright \cdots \triangleright \sigma_k = \sigma : n
\]

be a derivation. Then both the following statements hold:

\[
\begin{align*}
\delta_{out}(\eta(\sigma_k, n)) & \neq \text{unreach} & (5.3a) \\
P \in \delta_{out}(\eta(\sigma_k, n)) \cap \text{Perm}(n) & \implies \sigma \not\models \text{JK}(P) & (5.3b)
\end{align*}
\]

Proof. The proof is carried out by contradiction: we assume that \( \sigma \models \text{JK}(P) \) whenever \( P \in \delta_{out}(\eta(\sigma_k, n)) \cap \text{Perm}(n) \). Then, we proceed by induction on the length of derivation \( [] \triangleright \cdots \triangleright \sigma_k \triangleright \sigma : n \). The base case corresponds to our single axiom:

\[
[] \triangleright [n]
\]

For \( n \in N_{entry} \), we have defined \( \eta([n], n) = ([N], n) \), hence \( \delta_{out}([N], n) = \text{Perm}(n) \). Then (5.3a) is true because \( \text{Perm}(n) \neq \text{unreach} \), and (5.3b) trivially holds, as \( P \in \delta_{out}([N], n) \implies P \notin \text{Perm}(n) \), and the premises of the implication are never satisfied. For the inductive case, we proceed by case analysis on the rule used to derive \( \sigma_k \triangleright \sigma : n \), yielding:

- **case [call]:**

  \[
  \ell([n]) = \text{call} \quad n' \rightarrow n \quad \text{where} \quad \sigma_{k-1} = \sigma = \sigma' : n'
  \]

  Here \( \eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma' : n', \sigma' : n : n) = [n', n] \in E_{call} \). By the inductive hypothesis, it must be \( \delta_{out}(\eta(\sigma_{k-2}, \sigma' : n')) \neq \text{unreach} \), hence \( \delta_{in}(n') \neq \text{unreach} \): this actually implies \( \delta_{out}(n', n) \neq \text{unreach} \), so the (5.3a) is established.

  For the (5.3b), \( P \in \delta_{out}(n', n) \implies P \in \delta_{call}(n') \) or \( P \notin \text{Perm}(n') \). The latter option is prevented by our assumptions about \( P \), then only the former one is considered. If \( \neg \text{Priv}(n') \), then \( P \in \delta_{in}(n') \), and, using lemma 3.8.2, we obtain \( P \in \delta_{out}(\eta(\sigma_{k-2}, \sigma_{k-1})) \). Note that assumption \( \sigma' : n' \models \text{JK}(P) \) imposes \( P \in \text{Perm}(n') \), so we can apply the inductive hypothesis to deduce \( \sigma' \not\models \text{JK}(P) \). This prevents the JK rule to be applicable, and a contradiction arises with assumption \( \sigma' : n' \models \text{JK}(P) \). Otherwise, if \( \text{Priv}(n') \), then it should hold \( P \notin \text{Perm}(n') \), contradicting again assumption \( \sigma' : n' \models \text{JK}(P) \).

- **case [check]:**

  \[
  \ell([n]) = \text{check}(P') \quad \sigma : n' \models \text{JK}(P') \quad n' \rightarrow n
  \]

  Here \( \eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma : n', \sigma : n) = [n', n] \in E_{trans} \). By the inductive hypothesis, it must be \( \delta_{out}(\eta(\sigma_{k-2}, \sigma : n')) \neq \text{unreach} \), hence \( \delta_{in}(n') \neq \text{unreach} \). Therefore, \( \delta_{trans}(n') \) can be unreach only if \( P' \in \delta_{in}(n') \), i.e. \( \forall [v, n', n] \in E, P' \in \delta_{out}(v, n') \). If this happens, by lemma 3.8.2 we also have \( P' \in \delta_{out}(\eta(\sigma_{k-2}, \sigma : n')) \), and premise \( \sigma : n' \models \text{JK}(P') \) implies \( P' \in \text{Perm}(n') \). Now the inductive hypothesis can be applied to obtain \( \sigma \not\models \text{JK}(P') \): since \( n' \) is not privileged, this is a contradiction. Then it must be \( \delta_{trans}(n') \neq \text{unreach} \), and the (5.3a) is established, because this clearly implies \( \delta_{out}(n', n) \neq \text{unreach} \).
Now, for the (5.3b), we have that \( P \in \delta_{\text{out}}(n', n) \implies P \in \delta_{\text{trans}}(n') \). As both \( n \) and \( n' \) lie in the same protection domain, assumption \( P \in \text{Perm}(n) \) also ensures \( P \in \text{Perm}(n') \). Now, it can be shown that:

\[
P \in \bigcap_{(m, n') \in \mathcal{E}} \delta_{\text{out}}(m, n') \quad (5.4)
\]

As check nodes are not privileged, by rule JDK, we know that \( \sigma : n' \vdash \text{JD}(P') \) can only be true if \( \sigma \vdash \text{JD}(P) \). Then, by applying contrapositively the inductive hypothesis, we deduce that \( P' \notin \delta_{\text{out}}([n(\sigma_{k-2}, \sigma_{k-1})] \cap \text{Perm}(n')) \). Now, premise \( \sigma : n' \vdash \text{JD}(P'/') \) requires \( P' \in \text{Perm}(n') \) therefore, it is indeed \( P' \notin \delta_{\text{out}}([n(\sigma_{k-2}, \sigma_{k-1})]) \). By equation (5.4), this implies \( P \in \delta_{\text{out}}([n(\sigma_{k-2}, \sigma_{k-1})]) \), and, applying again the inductive hypothesis, we finally obtain a contradiction \( \sigma \not\vdash \text{JD}(P) \).

- case [return]:

\[
\ell(m) = \text{return} \quad n' \rightarrow n
\]

Here, \( \eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma : n' : m, \sigma : n) = [n', n] \in \mathcal{E}_{\text{trans}} \). By the inductive hypothesis, it must be \( \delta_{\text{out}}([n(\sigma_{k-2}, \sigma_{k-1})]) \neq \text{unreach} \), hence \( \delta_{\text{in}}(m) \neq \text{unreach} \). Now, lemma 3.4.1 states that there is an index \( i < k \) such that \( \sigma_i = \sigma : n' \): hence, the inductive hypothesis is also applicable to the derivation \( \sigma_0 \triangleright \cdots \triangleright \sigma_i \), yielding \( \delta_{\text{out}}([n(\sigma_{i-1}, \sigma_i : n')]) \neq \text{unreach} \) and \( \delta_{\text{in}}(m) \neq \text{unreach} \). By lemma 3.4.4, it must be \( \ell([n']) = \text{call} \), then the only case leading to \( \delta_{\text{trans}}([n']) = \text{unreach} \) is when \( \forall m' \in P(n') \). \( \delta_{\text{in}}(m') = \text{unreach} \): indeed, this cannot actually happen, as lemma 3.5.16 implies \( m \in P(n') \) and we already know \( \delta_{\text{in}}(m) \neq \text{unreach} \). This proves the (5.3a).

For the (5.3b), we have that \( P \in \delta_{\text{out}}([n', n]) \implies P \in \delta_{\text{trans}}([n']) \). By lemma 3.4.4, it must be \( \ell([n']) = \text{call} \), then it turns out that \( P \in \delta_{\text{in}}([n']) \). Now, lemma 3.4.1 states that \( \exists i \in 1, k-2 : \sigma_i = \sigma : n' \) by lemma 3.8.2, we then have that \( P \in \delta_{\text{out}}([n(\sigma_{i-1}, \sigma_i : n')] \); and, again, assumption \( P \in \text{Perm}(n) \) implies \( P \in \text{Perm}(n') \) as both \( n \) and \( n' \) carry the same permissions. Hence, the inductive hypothesis can be applied to yield \( \sigma \not\vdash \text{JD}(P) \).

\[\square\]

**Theorem 5.7.3.** Let \( \langle \delta_{\text{in}}, \delta_{\text{out}}, \delta_{\text{call}}, \delta_{\text{trans}} \rangle \models \text{DP}^2(G) \). For any \( n \in \mathbb{N} \), define:

\[
\delta(n) = \begin{cases} 
\varnothing & \text{if } \delta_{\text{in}}(n) = \text{unreach} \\
\delta_{\text{call}}(n) & \text{otherwise}
\end{cases}
\]

Then \( \delta \) is a sound \( \text{DP}^2 \)-solution.

*Proof.* The proof is exactly the same as the one of theorem 5.5.3. \[\square\]
Lemma 5.7.4. Let $\langle \delta_{in}, \delta_{out}, \delta_{call}, \delta_{trans} \rangle \models DP^2(G)$. Then, for any $n \in N$:

$$\delta_{in}(n) = unreach \implies -\exists \sigma \in \Sigma. G \vdash \sigma : n$$

Proof. By contradiction, assume $\delta_{in}(n) = unreach$ and $n$ is G-reachable, i.e. a derivation

$$[] = \sigma_0 \triangleright \sigma_1 \triangleright \cdots \triangleright \sigma_k = \sigma : n$$

exists. Now, lemma 5.7.2 ensures that $\delta_{out}(\eta(\sigma_{k-1}, \sigma : n)) \neq unreach$. Then, we have a contradiction, as this would imply $\delta_{in}(n) \neq unreach$. □

Example 5.7.5. Consider the e-commerce application of Fig. 3.1. Proceeding as in example 5.3.3, for the MFP solution at $n_2$ and $n_5$ we obtain:

$$\bar{\delta}_{in}(n_2) = \{ \text{pay, credit, debit} \}$$

$$\bar{\delta}_{in}(n_5) = \emptyset$$

By direct computation of the analysis at $n_{16}$, we have:

$$\bar{\delta}_{in}(n_{16}) = \bar{\delta}_{out}(n_{16}) \cup \bar{\delta}_{out}(n_5, n_{16})$$

$$= (\bar{\delta}_{call}(n_4) \cup \bar{\delta}_{call}(n_5)) \cap \text{Perm}(n_{16})$$

$$= (\bar{\delta}_{in}(n_4) \cup \bar{\delta}_{in}(n_5)) \cap \text{Perm}(n_{16})$$

$$= (\bar{\delta}_{out}(n_2, n_4) \cup \emptyset) \cap \text{Perm}(n_{16})$$

$$= \bar{\delta}_{trans}(n_2) \cap \text{Perm}(n_{16})$$

$$= \bar{\delta}_{in}(n_2) \cap \text{Perm}(n_{16})$$

$$= \{ \text{pay, credit, debit} \}$$

Since $\ell(n_{16}) = \text{check(ploan)}$ and $\text{ploan} \notin \bar{\delta}_{in}(n_{16})$, it turns out that kill$^2(n_{16})$. Thus, $\bar{\delta}_{trans}(n_{16}) = unreach$, and $\bar{\delta}_{in}(n_{17}) = unreach$, too. By lemma 5.7.4, this shows that node $n_{17}$ is unreachable.

Moreover, we have kill$^2(n_5)$, because $\bar{\delta}_{in}(n_{18}) = unreach$ and $\rho(n_5) = [n_{18}]$. Then $\bar{\delta}_{in}(n_6) = unreach$, and lemma 5.7.4 tells that $n_6$ is unreachable, too.

Note that weaker information is obtained by the DP$^1$ analysis: in fact, it just predicts that all permissions are denied to $n_5$ and $n_{17}$.

The full MFP solution to DP$^2$ for the e-commerce example is in Table A.1.
Theorem 5.7.6. DP² is a monotone data flow framework.

Proof. Consider the complemented analysis DP². The local property space is a product:

\[ \mathcal{L}_N = \mathcal{P}[\text{Permission}] \times \mathcal{P}[\text{Permission}] \times \mathcal{O} \]

where O is the two-element cpo \( \bot \subseteq \top \). Similarly to the data flow framework constructed for the DP¹ analysis, the first two components of the product represent the data flow information at the entry and at the exit of a node. The last component tells whether the node is reachable (\( \top \)) or not (\( \bot \)). The join operator \( \sqcup \) is defined in a coordinatewise fashion:

\[ \langle l_0, l_1, l_2 \rangle \sqcup \langle l'_0, l'_1, l'_2 \rangle = \langle l_0 \sqcup l'_0, l_1 \sqcup l'_1, l_2 \sqcup l'_2 \rangle \]

where \( \bot \sqcup \bot = \bot \) and \( \top \sqcup \top = \top \) for any \( l \in \mathcal{O} \). The bottom element is \( \bot_{\mathcal{L}_N} = \langle \emptyset, \emptyset, \bot \rangle \). Since both the join operators of \( \mathcal{P}[\text{Permission}] \) and \( \mathcal{O} \) are idempotent, commutative and associative, this definition actually makes \( \mathcal{L}_N \) a join semi-lattice. By the fact that Permission is finite for any control flow graph, it clearly follows that \( \mathcal{L}_N \) satisfies the ascending chain condition. Again, a standard construction equips \( \mathcal{L}_N \) with a complete lattice structure:

\[ \mathcal{L}_N = \langle \mathcal{L}_N, \subseteq, \bigcup, \bigcap, \bot_{\mathcal{L}_N}, \top_{\mathcal{L}_N} \rangle \]

The partial order \( \subseteq \) is the determined coordinatewise:

\[ \langle l_0, l_1, l_2 \rangle \subseteq \langle l'_0, l'_1, l'_2 \rangle \quad \text{iff} \quad l_0 \subseteq l'_0 \quad \text{and} \quad l_1 \subseteq l'_1 \quad \text{and} \quad l_2 \subseteq l'_2 \]

and the other components of the lattice are defined in the obvious manner. The solution for the isolated entry node is defined as:

\[ t = \langle \emptyset, \emptyset, \top \rangle \]

Given a control flow graph \( G \), the set of local transfer functions \( f_E = \mu_E(G) \) is defined in Table 5.11. If each \( f \in f_E \) is monotone regardless of \( G \), we can take \( \mathcal{F}_E \) to be the space of monotone functions over \( \mathcal{L}_N \). So, let \( \langle l_0, l_1, l_2 \rangle \subseteq \langle l'_0, l'_1, l'_2 \rangle \) be two arbitrary elements of \( \mathcal{L}_N \). By definition of monotonicity, we have for prove that, for any \( \langle m, n \rangle \in E_p \):

\[ f_{\langle m, n \rangle}(\langle l_0, l_1, l_2 \rangle) \subseteq f_{\langle m, n \rangle}(\langle l'_0, l'_1, l'_2 \rangle) \]

Consider the case \( l_2 = \bot \), first. We have:

\[ f_{\langle m, n \rangle}^0(\langle l_0, l_1, \bot \rangle) = f_{\langle m, n \rangle}^1(\langle l_0, l_1, \bot \rangle) = f_{\langle m, n \rangle}^2(\langle l_0, l_1, \bot \rangle) = \emptyset \]

Then \( f_{\langle m, n \rangle}(\langle l_0, l_1, \bot \rangle) = \bot_{\mathcal{L}_N} \subseteq f_{\langle m, n \rangle}(\langle l'_0, l'_1, l'_2 \rangle) \), regardless of \( l'_0, l'_1 \) and \( l'_2 \).

Otherwise, let \( l_2 = \top \): since we have assumed \( l_2 \subseteq l'_2 \), it must be \( l'_2 = \top \), too. Under these hypotheses, the proof of monotonicity for the components \( f_{\langle m, n \rangle}^0 \) and \( f_{\langle m, n \rangle}^1 \) is identical to the relative proof for the DP¹ analysis (Theorem 5.5.6).

For the \( f_{\langle m, n \rangle}^2 \) component, we have to deal with the following cases:

- if \( \ell(m) = \text{call} \) and \( m \rightarrow n \), then:
  \[ f_{\langle m, n \rangle}^2(\langle l_0, l_1, \top \rangle) = \bot = f_{\langle m, n \rangle}^1(\langle l'_0, l'_1, \top \rangle) \]

- if \( \ell(m) = \text{check}(P) \) and \( P \not\in l'_1 \), then \( P \not\in l_0 \), because \( l_0 \subseteq l'_0 \) by hypothesis. Thus:
  \[ f_{\langle m, n \rangle}^2(\langle l_0, l_1, \top \rangle) = \bot = f_{\langle m, n \rangle}^1(\langle l'_0, l'_1, \top \rangle) \]
5.7. THE DP² ANALYSIS

\[
\begin{align*}
f_{m,n}(\langle l_0, l_1, l_2 \rangle) & = \langle f^0_{\langle l_0, l_1, l_2 \rangle}, f^1_{\langle l_0, l_1, l_2 \rangle}, f^2_{\langle l_0, l_1, l_2 \rangle} \rangle \\
f^0_{m,n}(\langle l_0, l_1, \bot \rangle) & = \varnothing \\
f^0_{m\rightarrow n}(\langle l_0, l_1, T \rangle) & = \text{Perm}(n) \\
f^0_{m\rightarrow n}(\langle l_0, l_1, \bot \rangle) & = \text{Perm}(n) \cap \begin{cases} 
\text{Perm}(m) & \text{if } \text{Priv}(m) \\
l_1 & \text{otherwise}
\end{cases} \\
f^0_{m\rightarrow n}(\langle l_0, l_1, T \rangle) & = l_1 \\
f^0_{m\rightarrow n}(\langle l_0, l_1, \bot \rangle) & = \varnothing \\
f^1_{m,n}(\langle l_0, l_1, T \rangle) & = \begin{cases} 
\varnothing & \text{if } \text{kill}^2(\langle l_0', n \rangle) \\
l_0' & \text{otherwise}
\end{cases} \\
f^2_{m,n}(\langle l_0, l_1, \bot \rangle) & = \bot \\
f^2_{m,n}(\langle l_0, l_1, T \rangle) & = \begin{cases} 
\bot & \text{if } l[m] = \text{call} \text{ and } m \rightarrow n, \\
T & \text{if } l[m] = \text{check}(P) \text{ and } P \not\in l_0 \\
T & \text{otherwise}
\end{cases}
\end{align*}
\]

\[\text{kill}^2(\langle l, n \rangle) \overset{\text{def}}{=} l[n] = \text{check}(P) \text{ and } P \not\in l\]

Table 5.11: Local transfer functions for the DP² Analysis.

- if \( l[m] = \text{check}(P) \) and \( P \in l_0' \), but \( P \not\in l_0 \), then:
  \[
f^2_{m,n}(\langle l_0, l_1, T \rangle) = \bot \subseteq T = f^2_{m,n}(\langle l_0', l_1', T \rangle)
\]

- in the remaining cases, we have:
  \[
f^2_{m,n}(\langle l_0, l_1, T \rangle) = T = f^2_{m,n}(\langle l_0', l_1', T \rangle)
\]

\[\Box\]
Theorem 5.7.7. Let $\delta^1$ and $\delta^2$ be the MFP solutions for the DP$^1$ and DP$^2$ analyses, respectively. Then $\delta^2$ is more accurate than $\delta^1$, i.e.:

$$\delta^2 \subseteq \delta^1$$

Proof. Let $f_E$ and $g_E$ be the sets of local transfer functions for the complemented analyses DP$^1$ and DP$^2$, respectively, as defined in tables 5.7 and 5.11. Moreover, let $L^1_N$ and $L^2_N$ the local property spaces for DP$^1$ and DP$^2$.

By looking at the proofs of theorems 5.5.6 and 5.7.6, it follows that, for each element in $L^1_N$, there is a counterpart in $L^2_N$ carrying the same intuitive meaning. More precisely, if $\langle l_0, l_1 \rangle \in L^1_N$, then the element $\langle l_0, l_1, \top \rangle \in L^2_N$ carries the same information relative to the set of permissions non-denied at the entry ($l_0$) and at the exit ($l_1$) of the node. Besides, the $\top$ component means that the node is reachable: this information is implicit in $L^1_N$, because the DP$^1$ analysis does not distinguish between reachable and unreachable nodes.

Now, assume we can prove that:

$$\forall (m,n) \in E, \ g(m,n)(\langle l_0, l_1, \top \rangle) \subseteq (f(m,n)(\langle l'_0, l'_1 \rangle), \top) \quad (5.5)$$

for any $\langle l'_0, l'_1 \rangle \in L^1_N$ and $\langle l_0, l_1, \top \rangle \in L^2_N$ such that $l_0 \subseteq l'_0$ and $l_1 \subseteq l'_1$. Since the inclusion relation is preserved by greatest lower bounds, the global transfer functions satisfy:

$$g(l) \subseteq (f(l'), \top)$$

for each $l = \lambda n. \ (l_0(n), l_1(n), \top)$ and $l' = \lambda n. \ (l'_0(n), l'_1(n))$ such that:

$$\forall n \in N. \ l_0(n) \subseteq l'_0(n) \land l_1(n) \subseteq l'_1(n)$$

Since the MFP solutions for DP$^1$ and DP$^2$ are the least upper bound of the (finite) chains:

$$\perp_{L^1} \subseteq f(\perp_{L^1}) \subseteq f^2(\perp_{L^1}) \subseteq \cdots \subseteq f^k(\perp_{L^1}) = f^{k+1}(\perp_{L^1})$$

$$\perp_{L^2} \subseteq g(\perp_{L^2}) \subseteq g^2(\perp_{L^2}) \subseteq \cdots \subseteq g^h(\perp_{L^2}) = g^{h+1}(\perp_{L^2})$$

and, by (5.5), $g^i(\perp_{L^2}) \subseteq f^i(\perp_{L^1})$ for any $i \in N$, we eventually conclude that $\delta^2 \subseteq \delta^1$.

Therefore, we proceed with the proof of (5.5). For each $(m,n) \in E$, let $g^0_{(m,n)}$, $g^1_{(m,n)}$, $g^2_{(m,n)}$ be the components of $g_{(m,n)}$, and $f^0_{(m,n)}$, $f^1_{(m,n)}$, $f^2_{(m,n)}$ those of $f_{(m,n)}$.

Since $g^i_{(m,n)}(\langle l_0, l_1, \top \rangle) \subseteq \top$ trivially follows by definition of the cpo $O$, it suffices to show that the inequality is satisfied by the first two components of $f$ and $g$. The proof is carried out by cases on the edge $(m,n)$:

- if $\bullet \rightarrow n$, then:

$$g^0_{m \rightarrow n}(\langle l_0, l_1, \top \rangle) = Perm(n) = f^0_{m \rightarrow n}(\langle l'_0, l'_1 \rangle)$$

- if $m \rightarrow n$, there are two cases. If $m$ is privileged then:

$$g^0_{m \rightarrow n}(\langle l_0, l_1, \top \rangle) = Perm(m) = f^0_{m \rightarrow n}(\langle l'_0, l'_1 \rangle)$$

and the inequality trivially holds. In the other case, we have:

$$g^0_{m \rightarrow n}(\langle l_0, l_1, \top \rangle) = l_1 \subseteq l'_1 = f^0_{m \rightarrow n}(\langle l'_0, l'_1 \rangle)$$
5.7. THE DP² ANALYSIS

- if \( m \rightarrow n \), then:
  \[
  g^0_{m \rightarrow n}(\langle l_0, l_1, \top \rangle) = l_1 \subseteq l'_1 = f^0_{m \rightarrow n}(\langle l'_0, l'_1 \rangle)
  \]

- if \( m \leftarrow n \), then:
  \[
  g^0_{m \leftarrow n}(\langle l_0, l_1, \top \rangle) = \emptyset = f^0_{m \leftarrow n}(\langle l'_0, l'_1 \rangle)
  \]

For the second component, let \( \widehat{l}_0 = g^0_{(m,n)}(\langle l_0, l_1, \top \rangle) \) and \( \widehat{l}_0 = f^0_{(m,n)}(\langle l'_0, l'_1 \rangle) \). Observe that, by definition of \( \text{kill}^1 \) and \( \text{kill}^2 \), we have, for any \( l, l' \in \mathcal{P}[\text{Permission}] \):

\[
l \subseteq l' \land \text{kill}^1(l', n) \implies \text{kill}^2(l, n)
\]

By the part of the proof relative to \( g^0_{(m,n)} \) and \( f^0_{(m,n)} \), we know \( \widehat{l}_0 \subseteq \widehat{l}_0 \): therefore, \( \text{kill}^1(\widehat{l}_0, n) \implies \text{kill}^2(\widehat{l}_0, n) \), and we have to deal with the following cases:

- if \( \text{kill}^1(\widehat{l}_0, n) \), then \( \text{kill}^2(\widehat{l}_0, n) \), and:
  \[
  g^1_{(m,n)}(\langle l_0, l_1, \top \rangle) = \emptyset = f^1_{(m,n)}(\langle l'_0, l'_1 \rangle)
  \]

- if \( \text{kill}^2(\widehat{l}_0, n) \) but \( \neg\text{kill}^1(\widehat{l}_0, n) \):
  \[
  g^1_{(m,n)}(\langle l_0, l_1, \top \rangle) = \emptyset \subseteq \widehat{l}_0 = f^1_{(m,n)}(\langle l'_0, l'_1 \rangle)
  \]

- in any remaining cases, we have:
  \[
  g^1_{(m,n)}(\langle l_0, l_1, \top \rangle) = \widehat{l}_0 \subseteq \widehat{l}_0 = f^1_{(m,n)}(\langle l'_0, l'_1 \rangle)
  \]

\[\square\]

**Lemma 5.7.8.** Let \( \langle \delta_{\text{in}}, \delta_{\text{out}}, \delta_{\text{call}}, \delta_{\text{trans}} \rangle \) be the MFP solution to DP². Then:

\[
\mathcal{P} \in \mathfrak{F}_{\text{in}}(n) \implies \mathcal{P} \in \text{Perm}(n)
\]

for any \( n \in \mathbb{N} \).

**Proof.** The proof is similar to the proof of theorem 5.3.6. \( \square \)

**Lemma 5.7.9.** The MFP solution to DP² is non-trivial.

**Proof.** The result easily follows from 5.7.8, as it is shown in 5.3.7. \( \square \)
Counterexample 5.7.10. The DP$^2$ analysis is not complete.

Proof. Consider the control flow graph $G$ in Fig. 5.6, and let $\delta$ the MFP solution to DP$^2(G)$. By direct computation of the analysis (with some shortcuts), we obtain:

\[
\begin{align*}
\overline{\delta}_m(n_2) &= \overline{\delta}_{out}(n_0, n_2) \cup \overline{\delta}_{out}(n_1, n_2) = \overline{\delta}_{call}(n_0) \cup \overline{\delta}_{call}(n_1) \\
&= \{P_0, P_1\} \cup \{P_1, P_2\} = \{P_0, P_1, P_2\} \\
\overline{\delta}_m(n_4) &= \overline{\delta}_{out}(n_2, n_4) = \overline{\delta}_{out}(n_2, n_4) \cap \{P_0, P_2\} \\
&= \{P_0, P_1, P_2\} \cap \{P_0, P_2\} = \{P_0, P_2\} \\
\overline{\delta}_m(n_5) &= \overline{\delta}_{out}(n_4, n_5) = \overline{\delta}_{trans}(n_4) = \overline{\delta}_{out}(n_2, n_4) = \{P_0, P_2\} \\
\overline{\delta}_m(n_3) &= \overline{\delta}_{out}(n_2, n_3) = \overline{\delta}_{trans}(n_2) = \overline{\delta}_m(n_2) = \{P_0, P_1, P_2\}
\end{align*}
\]

Observe that, in order to estimate the analysis at $n_3$, we have used the fact that $n_5 \in p(n_2)$ and $\overline{\delta}_m(n_3) \neq unreach$: this means that the return node at $n_5$ is reachable, and the information can flow from the call at $n_2$ to $n_3$, where the control transfers after the return.

Now, it is easy to show that the computed solution is sound. However, according to definition 5.1.5, this solution is not complete. To see why, consider node $n_3$. Actually, there are only two (concrete) paths leading to $n_3$, that is $\vec{p}_0 = \langle n_0, n_2, n_4, n_5, n_3 \rangle$ and $\vec{p}_1 = \langle n_1, n_2, n_4, n_5, n_3 \rangle$. Only the first of these paths is traversable, through the derivation:

\[
[\top] \triangleright [n_0] \triangleright [n_0, n_2] \triangleright [n_0, n_2, n_4] \triangleright [n_0, n_2, n_5] \triangleright [n_0, n_3]
\]

On the contrary, the other path is not traversable. In fact, the derivation:

\[
[\top] \triangleright [n_1] \triangleright [n_1, n_2] \triangleright [n_1, n_2, n_4]
\]

is blocked, because $[n_1, n_2, n_4] \notin JDK(P_2)$. Now, we have that $P_2$ is denied to all executions leading to $n_3$: in fact, $\chi(\vec{p}_0) = [n_0, n_3] \notin JDK(P_2)$. Since, as we showed above, $P_2 \notin \delta(n_3)$, we deduce that the DP$^2$ analysis is not complete. \(\Box\)
5.8 The GP² Analysis

\[
\begin{align*}
\text{GP}^2_{\text{in}}(n) & = \begin{cases} 
\text{unreach} & \text{if } \forall (m,n) \in E. \overline{\text{DP}}^2_{\text{out}}(m,n) = \text{unreach} \\
\bigcap_{(m,n) \in E} \text{GP}^2_{\text{out}}(m,n) & \text{otherwise}
\end{cases} \\
\text{GP}^2_{\text{out}}(m,n) & = \begin{cases} 
\text{unreach} & \text{if } \text{GP}^2_{\text{in}}(m) = \text{unreach} \\
\text{Perm}(n) & \text{if } \leftarrow \rightarrow n \\
\text{GP}^2_{\text{out}}(m) \cap \text{Perm}(n) & \text{if } m \rightarrow n \\
\text{GP}^2_{\text{trans}}(m) & \text{if } m \rightarrow n
\end{cases} \\
\text{GP}^2_{\text{cat}}(n) & = \begin{cases} 
\text{unreach} & \text{if } \text{GP}^2_{\text{in}}(n) = \text{unreach} \\
\text{Perm}(n) & \text{if } \text{Priv}(n) \\
\text{GP}^2_{\text{in}}(n) & \text{otherwise}
\end{cases} \\
\text{GP}^2_{\text{trans}}(n) & = \begin{cases} 
\text{unreach} & \text{if } \text{GP}^2_{\text{in}}(n) = \text{unreach} \text{ or } \text{kill}^2(n) \\
\bigcap_{(m,n) \in E} \text{GP}^2_{\text{out}}(m,n) \cup \{P\} & \text{if } \ell(n) = \text{check}(P) \text{ and } \neg \text{kill}^2(n) \\
\text{GP}^2_{\text{in}}(n) & \text{otherwise}
\end{cases}
\end{align*}
\]

Table 5.12: The GP² Analysis.

The GP² analysis, defined in Table 5.12, refines the GP¹ analysis by taking into account the information about unreachable nodes computed by the DP² analysis. As we will see, this gives rise to more accurate (but still incomplete) solutions than those achievable by the GP¹ analysis.
Lemma 5.8.1. Let \( \langle \gamma_{in}, \gamma_{out}, \gamma_{call}, \gamma_{trans} \rangle = GP^2(G) \), and:

\[
\lambda = \sigma_0 \triangleright \sigma_1 \triangleright \cdots \triangleright \sigma_k = \sigma : n
\]

be a derivation. Then both the following statements hold:

\[
\gamma_{out}(\eta(\sigma_{k-1}, \sigma_k)) \neq \text{unreach} \tag{5.6a}
\]

\[
P \in \gamma_{out}(\eta(\sigma_{k-1}, \sigma_k)) \implies \sigma \vdash JDK(P) \land P \in \text{Perm}(n) \tag{5.6b}
\]

Proof. We proceed by induction on the length of derivation \( \lambda \triangleright \cdots \triangleright \sigma_{k-1} \triangleright \sigma : n \). The base case corresponds to our single axiom:

\[
\frac{n \in N_{entry}}{\lambda \triangleright [n]}
\]

For \( n \in N_{entry} \), we have defined \( \eta([\lambda], [n]) = [\bot_n, n] \), so \( P \in \gamma_{out}(\bot_n, n) \implies P \in \text{Perm}(n) \).

Then (5.6a) is true because \( \text{Perm}(n) \neq \text{unreach} \), and (5.6b) also holds because, by rule \( JDK_{\phi} \), \( \sigma = \bot \vdash JDK(P) \). For the inductive case, we proceed by case analysis on the rule used to derive \( \sigma_{k-1} \triangleright \sigma : n \), yielding:

- **case [call]:**

\[
\frac{\ell([n']) = \text{call} \quad n' \rightarrow n}{\sigma' : n' \triangleright \sigma' : n' : n} \quad \text{where} \quad \sigma_{k-1} = \sigma = \sigma' : n'
\]

Here \( \eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma' : n', \sigma' : n' : n) = [n', n] \in E_{call} \). By the inductive hypothesis, it must be \( \gamma_{out}(\eta(\sigma_{k-2}, \sigma' : n')) \neq \text{unreach} \), hence \( \gamma_{in}(n') \neq \text{unreach} \). This actually implies \( \gamma_{out}(n', n) \neq \text{unreach} \), so the (5.6a) is established.

For the (5.6b), observe that \( P \in \gamma_{out}(n', n) \implies P \in \gamma_{out}(n') \) and \( P \in \text{Perm}(n) \). If \( \neg \text{Priv}(n') \), then \( P \in \gamma_{in}(n') \), and, using lemma 3.8.2, we obtain \( P \in \gamma_{out}(\eta(\sigma_{k-2}, \sigma_{k-1})) \).

Then we can apply the inductive hypothesis to deduce \( \sigma' : n' \vdash JDK(P) \) and \( P \in \text{Perm}(n') \), thus obtaining \( \sigma' : n' \vdash JDK(P) \) by rule \( JDK_{\phi} \). Otherwise, if \( \text{Priv}(n') \), then \( P \in \text{Perm}(n') \); in this case, \( \sigma' : n' \vdash JDK(P) \) is ensured by rule \( JDK_{\text{prim}} \).

- **case [check]:**

\[
\frac{\ell([n']) = \text{check}(P') \quad \sigma : n' \vdash JDK(P')} {\sigma : n' \triangleright \sigma : n} \quad \text{where} \quad \sigma_{k-1} = \sigma = \sigma' : n'
\]

Here \( \eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma : n', \sigma : n) = [n', n] \in E_{trans} \). By the inductive hypothesis, it must be \( \gamma_{out}(\eta(\sigma_{k-2}, \sigma' : n')) \neq \text{unreach} \), hence \( \gamma_{in}(n') \neq \text{unreach} \). Therefore, \( \gamma_{trans}(n') \) can be \( \text{unreach} \) only if \( P' \in \delta_{in}(n') \), i.e. \( \forall (m, n') \in E. P' \in \delta_{out}(m, n') \). If this happens, by lemma 3.8.2 we also have \( P' \in \delta_{out}(\eta(\sigma_{k-2}, \sigma : n')) \). Since check nodes are not privileged, by the premise \( \sigma : n' \vdash JDK(P') \) we deduce both \( \sigma : n' \vdash JDK(P') \) and \( P' \in \text{Perm}(n') \). Since, by (5.3b), it is also \( \sigma \vdash JDK(P') \), we have a contradiction. This proves the (5.6a).

For the (5.6b), note that \( P \in \gamma_{out}(n', n) \implies P \in \gamma_{trans}(n') \). It can be shown that:

\[
P \in \bigcap_{(m, n') \in E} \gamma_{out}(m, n') \cup \{P'\} \tag{5.7}
\]
5.8. THE GP² ANALYSIS

If $P = P'$, then $\sigma : n' \vdash JDK(P)$ follows by the premises of the $\triangleright$ check rule, and this obviously implies $P \in Perm(n')$. As check nodes are not privileged, $\sigma \vdash JDK(P)$ follows from rule $ JDK_{\alpha}$, and $P \in Perm(n)$ from the fact that $n$ and $n'$ carry the same permissions. Otherwise, if $P \neq P'$, premise $\sigma : n' \vdash JDK(P')$ implies $P \in Perm(n')$, and rule $ JDK_{\alpha}$ also states $\sigma \vdash JDK(P')$. Then, by applying contrapositively lemma 5.7.2, it turns out that $P' \notin \delta_{out}(\eta(\sigma_{k-2}, \sigma_{k-1}))$. By equation (5.7), it follows $P \in \gamma_{out}(\eta(\sigma_{k-2}, \sigma_{k-1}))$: then, by the inductive hypothesis, we finally obtain $\sigma \vdash JDK(P)$.

- case $\text{return}$:

$$\ell(\eta) = \text{return } n' \rightarrow n$$

$$\sigma : n', m \triangleright \sigma : n$$

Here, $\eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma : n', m, \sigma : n) = (n', n) \in E_{\text{trans}}$. By the inductive hypothesis, it must be $\gamma_{out}(\eta(\sigma_{k-2}, \sigma' : n')) \neq \text{unreach}$, hence $\gamma_{in}(\eta) \neq \text{unreach}$.

Now, lemma 3.4.1 states that there is an index $i < k$ such that $\sigma_i = \sigma : n'$: then, the inductive hypothesis is also applicable to the derivation $\sigma_0 \triangleright \cdots \triangleright \sigma_i$, yielding $\gamma_{out}(\eta(\sigma_{i-1}, \sigma' : n')) \neq \text{unreach}$ and $\gamma_{in}(n') \neq \text{unreach}$. By lemma 3.4.4, it must be $\ell(n') = \text{call}$, then $\gamma_{\text{trans}}(n') = \text{unreach}$ only occurs if $\gamma_{in}(n') = \text{unreach}$ for each $m' \in \rho(n')$. However, this cannot happen, because lemma 3.5.16 implies $m \in \rho(n')$, and we already know that $\gamma_{in}(\eta) \neq \text{unreach}$. This proves (5.6a).

For (5.6b), observe that $P \in \gamma_{out}(n', n) \Rightarrow P \in \gamma_{\text{trans}}(n')$. By lemma 3.4.4, it must be $\ell(n') = \text{call}$, then it turns out that $P \in \gamma_{in}(n')$. Now, lemma 3.4.1 states that $\exists i \in 1..k-2\sigma_i = \sigma : n'$, and by lemma 3.8.2, we then have that $P \in \gamma_{out}(\eta(\sigma_{i-1}, \sigma : n'))$. Therefore we can apply the inductive hypothesis, obtaining $\sigma \vdash JDK(P)$ and $P \in Perm(n')$: again, $P \in Perm(n') \Rightarrow P \in Perm(n)$, as both $n$ and $n'$ lie in the same protection domain.

\[\square\]

**Theorem 5.8.2.** Let $\langle \gamma_{in}, \gamma_{out}, \gamma_{\text{call}}, \gamma_{\text{trans}} \rangle \models GP²(G)$. For any $n \in N$, define:

$$\gamma(n) = \begin{cases} \emptyset & \text{if } \gamma_{in}(n) = \text{unreach} \\ \gamma_{\text{call}}(n) & \text{otherwise} \end{cases}$$

Then $\gamma$ is a sound GP²-solution.

**Proof.** The proof is exactly the same as the one of theorem 5.6.3. $\square$

**Lemma 5.8.3.** Let $\langle \gamma_{in}, \gamma_{out}, \gamma_{\text{call}}, \gamma_{\text{trans}} \rangle \models GP²(G)$. Then, for any $n \in N$:

$$\gamma_{in}(n) = \text{unreach} \implies \neg \exists \sigma \in \Sigma. G \vdash \sigma : n$$

**Proof.** This proof is exactly the same as the one of lemma 5.7.4. $\square$

**Example 5.8.4.** Consider the e-commerce application of Fig. 3.1. By direct computation of the analysis at $n_7$, we obtain:

$$\gamma_{in}(n_7) = \gamma_{out}(n_0, n_7) \cap \gamma_{out}(n_7, n_7)$$

$$= (\gamma_{\text{call}}(n_0) \cap Perm(n_7)) \cap \gamma_{\text{trans}}(n_7)$$

$$= \{P_{\text{compay}}, P_{\text{credit}}, P_{\text{debit}}\} \cap \gamma_{in}(n_7)$$
This recursive equation is satisfied by any subset of \{P_{\text{canpay}}, P_{\text{credit}}, P_{\text{debit}}\}: since, regardless of some technical details, the property space for GP^2 can be seen as partially ordered by \( \supseteq \), this is just the MFP solution at \( n_7 \).

The set of permissions granted at the entry of \( n_{19} \) is:

\[
\gamma_{in}(n_{19}) = \bigcap_{\{(m,n_{19})\in E \}} \gamma_{out}(m,n_{19}) \setminus \delta_{out}(m,n_{19}) \neq \text{unreach}
\]

Now, the only edges leading to \( n_{19} \) are \((n_7,n_{19})\) and \((n_{17},n_{19})\). Since, by example 5.7.5, \( \delta_{in}(n_7) = \text{unreach} \), it follows that:

\[
\gamma_{in}(n_{19}) = \gamma_{out}(n_7,n_{19}) = \gamma_{call}(n_7) \cap \text{Perm}(n_{19}) \\
= \gamma_{in}(n_7) \cap \text{Perm}(n_{19}) = \{ P_{\text{canpay}}, P_{\text{credit}}, P_{\text{debit}} \}
\]

Since \( P_{\text{credit}} \in \gamma(n_{19}) \), the soundness result for the GP^2 analysis ensures that the security check at \( n_{19} \) will always succeed. Note that this information was not discovered by the GP^1 analysis.

The set of permissions granted at the entry of \( n_{11} \) is:

\[
\gamma_{in}(n_{11}) = \bigcap_{\{(m,n_{11})\in E \}} \gamma_{out}(m,n_{11}) \setminus \delta_{out}(m,n_{11}) \neq \text{unreach}
\]

The only edges leading to \( n_{11} \) are \((n_3,n_{11})\) and \((n_6,n_{11})\). By example 5.7.5, we have that \( \delta_{in}(n_6) = \text{unreach} \): then, \( \delta_{out}(n_6,n_{11}) = \text{unreach} \), too. Proceeding as in example 5.6.4, we find that \( \gamma_{in}(n_2) = \{ P_{\text{canpay}}, P_{\text{credit}}, P_{\text{debit}} \} \). Therefore:

\[
\gamma_{in}(n_{11}) = \gamma_{out}(n_3,n_{11}) = \gamma_{call}(n_3) \cap \text{Perm}(n_{11}) \\
= \gamma_{in}(n_3) \cap \text{Perm}(n_{11}) = \gamma_{out}(n_2,n_3) \cap \text{Perm}(n_{11}) \\
= \gamma_{trans}(n_2) \cap \text{Perm}(n_{11}) = \gamma_{in}(n_2) \cap \text{Perm}(n_{11}) \\
= \{ P_{\text{canpay}}, P_{\text{credit}}, P_{\text{debit}} \}
\]

Since \( P_{\text{debit}} \in \gamma(n_{11}) \), the soundness result for the GP^2 analysis ensures that the security check at \( n_{11} \) will always succeed. Again, observe that this information was not discovered by the GP^1 analysis.

The full MFP solution to GP^2 for the e-commerce example is in Table A.1.
Counterexample 5.8.5. The GP² analysis is not complete.

Proof. Consider a slight variation of the e-commerce example of Fig. 3.1, where the only difference is that permission \( P_{loan} \) is now granted to the protection domain Client. The set of permissions not denied at the exit of \( n_{16} \) is:

\[
\delta_{\text{trans}}(n_{16}) = \bigcup_{P_{loan} \in \delta_{\text{out}}(m, n_{16})} \delta_{\text{out}}(m, n_{16})
\]

Now, by direct computation of the DP² analysis at \([n_4, n_{16}]\), we have:

\[
\delta_{\text{out}}(n_4, n_{16}) = \delta_{\text{cat}}(n_4) \cap \text{Perm}(n_{16}) = \delta_{\text{in}}(n_4) \cap \text{Perm}(n_{16})
\]

\[
= \delta_{\text{out}}(n_2, n_4) \cap \text{Perm}(n_{16}) = \delta_{\text{trans}}(n_2) \cap \text{Perm}(n_{16})
\]

\[
= \delta_{\text{in}}(n_2) \cap \text{Perm}(n_{16}) = \{ P_{\text{bank}}, P_{\text{credit}}, P_{\text{debit}}, P_{\text{loan}} \}
\]

On the other hand, from example 5.7.5 we know that \( \delta_{\text{out}}(n_5, n_{16}) = \emptyset \). Moreover, since \( P_{\text{loan}} \in \delta_{\text{in}}(n_{16}) \), it follows that \( \delta_{\text{in}}(n_{17}) \neq \text{unreach} \). Thus:

\[
\delta_{\text{in}}(n_{18}) = \delta_{\text{out}}(n_{17}, n_{18}) = \delta_{\text{trans}}(n_{17}) = \delta_{\text{in}}(n_{17})
\]

\[
= \delta_{\text{out}}(n_{16}, n_{17}) = \delta_{\text{trans}}(n_{16}) = \delta_{\text{out}}(n_4, n_{16})
\]

Since \( n_{18} \in \rho(n_5) \) and \( \delta_{\text{in}}(n_{18}) \neq \text{unreach} \), we have \( \neg \text{kill}^2(n_5) \), although node \( n_6 \) is actually unreachable. Then \( \gamma_{\text{in}}(n_6) = \gamma_{\text{in}}(n_5) \), and:

\[
\gamma_{\text{in}}(n_{11}) = \gamma_{\text{out}}(n_3, n_{11}) \cap \gamma_{\text{out}}(n_6, n_{11}) = \gamma_{\text{cat}}(n_3) \cap \gamma_{\text{cat}}(n_6) \cap \text{Perm}(n_{11})
\]

\[
= \gamma_{\text{in}}(n_3) \cap \gamma_{\text{in}}(n_6) \cap \text{Perm}(n_{11}) = \gamma_{\text{in}}(n_3) \cap \gamma_{\text{in}}(n_5) \cap \text{Perm}(n_{11})
\]

\[
= \gamma_{\text{in}}(n_3) \cap \emptyset \cap \text{Perm}(n_{11}) = \emptyset
\]

Therefore, it seems that no permission is granted to node \( n_{11} \), and the security check enforced by that node is indeed necessary. By looking more closely the set of executions that may reach \( n_{11} \), we find that this is not true: in fact, the only reachable state having \( n_{11} \) as top node is \( n_0 : n_3 : n_{11} \vdash \text{JDK}(P_{\text{debit}}) \). Thus, the GP² analysis is not complete. □

Theorem 5.8.6. GP² is a monotone data flow framework.

Proof. The proof closely resembles the one of theorem 5.7.6, hence it is only sketched here.

The set of local transfer functions is defined in Table 5.13. A solution \( \delta \) to the DP² analysis is used to specify how the information flows from the entry to the exit of nodes.

The local property space is dual to the space for the DP² analysis: set union is replaced by set intersection, and the order relation \( \subseteq \) is replaced with \( \supseteq \). The last component of the property space, that is the cpo \( O \), is left unchanged. □

Theorem 5.8.7. Let \( \gamma^1 \) and \( \gamma^2 \) be the MFP solutions for the GP¹ and GP² analyses, respectively. Then \( \gamma^2 \) is more accurate than \( \gamma^1 \), i.e.:

\[
\gamma^2 \sqsubseteq \gamma^1
\]

Proof. The proof is similar to the one of theorem 5.7.7, then it is not carried out here. □

Corollary 5.8.8. Any GP²-solution is non-trivial.
$f_{(m,n)}((l_0, l_1, l_2)) = \langle f_{(m,n)}^0, f_{(m,n)}^1, f_{(m,n)}^2 \rangle((l_0, l_1, l_2))$

$f_{(m,n)}^0((l_0, l_1, \perp)) = \emptyset$

$f_{(m,n)}^0((l_0, l_1, \top)) = \text{Perm}(n)$

$f_{(m,n)}^0((l_0, l_1, \top)) = \text{Perm}(n) \cap \begin{cases} \text{Perm}(m) & \text{if } \text{Priv}(m) \\ l_1 & \text{otherwise} \end{cases}$

$f_{(m,n)}^0((l_0, l_1, \top)) = l_1$

$f_{(m,n)}^0((l_0, l_1, \top)) = \emptyset$

$f_{(m,n)}^0((l_0, l_1, \perp)) = \emptyset$

$f_{(m,n)}^1((l_0, l_1, \top)) = \begin{cases} \emptyset & \text{if } \text{kill}^2[\delta(n), n] \\ f_{(m,n)}^2((l_0, l_1, \top)) & \text{otherwise} \end{cases}$

$f_{(m,n)}^2((l_0, l_1, \top)) = \bot$

$f_{(m,n)}^2((l_0, l_1, \top)) = \begin{cases} \bot & \text{if } \ell(m) = \text{call and } m \rightarrow n, \\ \text{or } \ell(m) = \text{check}(P) \text{ and } P \notin \delta(n) \\ \top & \text{otherwise} \end{cases}$

Table 5.13: Local transfer functions for the GP² Analysis.
5.9 Optimized stack inspection

The correctness results proved in the previous sections shed light on a possible optimization of the stack inspection algorithm. First of all, class files must be enriched with the results of the GP and DP analyses: this is done by mapping fragments of bytecode to nodes, and each node \( n \) to the pair \( \langle \delta(n), \gamma(n) \rangle \). We require that both \( \delta \) and \( \gamma \) are sound and non-trivial analyses.

When an access control decision has to be made against a permission \( P \), the call stack (with nodes in place of protection domains) is examined top-down as follows. Assume \( n \) to be the currently scanned node. If \( P \in \delta(n) \), then an \texttt{AccessControlException} is thrown. Otherwise, if \( P \in \gamma(n) \) the algorithm succeeds. If neither case occurs, the search goes on.

This optimized stack inspection algorithm is shown if Fig. 5.7. A formal specification is given in Fig. 5.14, and it is proved to yield the same results of the standard JDK.

\textbf{Algorithm 3:} Optimized stack inspection

\texttt{CHECK-PERMISSION}(P, \sigma, \delta, \gamma)

1 \hspace{1em} \textbf{while} \ \sigma \neq \text{NIL} \ \textbf{do}
2 \hspace{1em} \ n \leftarrow \text{POP}(\sigma)
3 \hspace{1em} \textbf{if} \ P \in \delta(n)
4 \hspace{1em} \textbf{then throw} \ \text{"access control exception"}
5 \hspace{1em} \textbf{if} \ P \in \gamma(n)
6 \hspace{1em} \textbf{then return}

Figure 5.7: The optimized stack inspection algorithm.
Table 5.14: Specification of the optimized access control policy.

\[
\frac{\emptyset \vdash JDK^* (P)}{[JDK^*]}
\]

\[
\frac{P \notin \delta(n) \quad \sigma \vdash JDK^* (P)}{\sigma : n \vdash JDK^* (P)}
\]

\[
\frac{P \in \gamma(n)}{\sigma : n \vdash JDK^* (P)}
\]

**Theorem 5.9.1 (Correctness of JDK*).** Let G be a control flow graph and \( \sigma \) a reachable state. Then, for any permission \( P \):

\[
\sigma \vdash JDK (P) \iff \sigma \vdash JDK^* (P)
\]

**Proof.** The proof for the forward implication is carried out by induction on the derivation \( \sigma \vdash JDK (P) \). Case analysis on the last applied rule yields:

- **case [JDK**\(\emptyset\)**]: here \( \sigma = \emptyset \), and \( \emptyset \vdash JDK^* (P) \) is deduced by rule JDK\(\emptyset\).
- **case [JDK**\(\sigma\)**]: here \( \sigma = \sigma' : n \), and the premises of the rule ensure \( P \in Perm(n) \) and \( \sigma' \vdash JDK (P) \). Applying the inductive hypothesis, we deduce \( \sigma' \vdash JDK^* (P) \). Now assume, by contradiction, that \( P \in \delta (n) \). Since \( \delta \) is DP-sound, this would imply that \( \sigma' : n \notin JDK^* (P) \), contradicting our assumption \( \sigma' : n \vdash JDK (P) \). Therefore, it must be \( P \notin \delta (n) \), and the rule JDK\(\emptyset\) can be applied to obtain \( \sigma' : n \vdash JDK^* (P) \).
- **case [JDK**\(\text{Priv}\)**]: here \( \sigma = \sigma' : n \), and the premises of the rule state \( P \in Perm(n) \) and \( \text{Priv}(n) \). Since \( \gamma \) is non-trivial, this implies that \( P \in \gamma(n) \), and the JDK\(\emptyset\) rule then yields \( \sigma' : n \vdash JDK^* (P) \).

For the backward implication, assume \( \sigma \vdash JDK^* (P) \). Again, we proceed by induction on the depth of the call stack. Case analysis on the last applied rule gives:

- **case [JDK**\(\emptyset\)**]: here \( \sigma = \emptyset \), and \( \emptyset \vdash JDK (P) \) is deduced by rule JDK\(\emptyset\).
- **case [JDK**\(\sigma\)**]: here \( \sigma = \sigma' : n \), and the premises of the rule say that \( P \notin \delta (n) \) and \( \sigma' \vdash JDK (P) \). By applying the inductive hypothesis, we obtain \( \sigma' \vdash JDK^* (P) \). Moreover, as \( \delta \) is non-trivial, we know that \( P \in Perm(n) \); then the rule JDK\(\emptyset\) can be used to deduce \( \sigma' : n \vdash JDK^* (P) \).
- **case [JDK**\(\text{Perm}\)**]: here \( \sigma = \sigma' : n \), and by premise we have \( P \in \gamma(n) \). Since \( \gamma \) is GP-sound, this indeed implies \( \sigma' : n \vdash JDK (P) \).

\( \square \)
Chapter 6

Conclusions

In this thesis we developed two families of Data Flow Analyses for the Java bytecode. The Granted Permissions Analysis computes a safe approximation of the set of permissions which are always granted to bytecode at run-time. Specularly, the Denied Permissions Analysis approximates the set of permissions which are always denied. The analyses provide us with the basis for reducing the run-time overhead due to stack inspection.

Here, we focussed on Java bytecode: however, the same techniques can be applied to programming languages whose security architectures rely on stack inspection to enforce the dynamic check of permissions (e.g. C# [Wil00]).

The extension of our proposal to the full access control policy requires the control flow graph construction algorithm to single out the program points where new threads can be generated. This step seems to be the hard part of the job. Indeed, we feel that our analyses only require slight modifications.

Our program model does not handle the dynamic linking features of Java. Actually, the whole program is available prior the construction of its control flow graph. The extension of our approach to cope with dynamic linking requires substantial efforts. The first step consists in linking dynamically the relevant graphs. Then the available solutions for the various program fragments have to be combined. Some preliminary work on data flow analysis taking care of dynamic linking can be found in [RRL99, SBC00].
Appendix A

An e-commerce example

grant codeBase "file:{java.home}/lib/ext" {
    permission java.security.AllPermission;
};

grant signedBy "Bank" {
    permission BankPermission "canpay";
    permission BankPermission "debit";
    permission BankPermission "credit";
    permission BankPermission "loan";
    permission VarPermission "read";
    permission VarPermission "write";
};

grant signedBy "Client" {
    permission BankPermission "canpay";
    permission BankPermission "debit";
    permission BankPermission "credit";
};

public class Ecommerce {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000);

        new Spender(account).start();
        new Saver(account).start();
        new Robber(account).start();
    }
}
public abstract class Client extends Thread {
    public abstract void transact();

    public void run() {
        while(true) transact();
    }
}

public class Spender extends Client {
    private BankAccount account;
    private final int AMOUNT = 10;

    public Spender(BankAccount account) {
        this.account = account;
    }

    public void transact() {
        if (account.canPay(AMOUNT)) account.debit(AMOUNT);
        else account.loan(AMOUNT);
    }
}

public class Saver extends Client {
    private BankAccount account;
    private final int AMOUNT = 10;

    public Saver(BankAccount account) {
        this.account = account;
    }

    public void transact() {
        account.credit(AMOUNT);
    }
}

public class Robber extends Client {
    private BankAccount account;
    private final int AMOUNT = 9999;

    public Robber(BankAccount account) {
        this.account = account;
    }

    public void transact() {
        account.loan(AMOUNT);
        account.debit(AMOUNT);
    }
}
import java.security.*;

public class BankAccount {
    private Permission canpay = new BankPermission("canpay", null);
    private Permission debit = new BankPermission("debit", null);
    private Permission credit = new BankPermission("credit", null);
    private Permission loan = new BankPermission("loan", null);
    private ControlledVar balance;

    public BankAccount(int initBalance) {
        balance = new ControlledVar(initBalance);
    }

    public boolean canpay(final int amount) {
        AccessController.checkPermission(canpay);
        Object res = AccessController.doPrivileged(new PrivilegedAction() {
            public Object run() {
                return new Boolean(balance.read() > amount);
            }
        });
        return ((Boolean) res).booleanValue();
    }

    public void debit(final int amount) {
        AccessController.checkPermission(debit);
        if (this.canpay(amount)) {
            AccessController.doPrivileged(new PrivilegedAction() {
                public Object run() {
                    balance.write(balance.read() - amount);
                    return null;
                }
            });
        }
    }

    public void credit(final int amount) {
        AccessController.checkPermission(credit);
        AccessController.doPrivileged(new PrivilegedAction() {
            public Object run() {
                balance.write(balance.read() + amount);
                return null;
            }
        });
    }

    public void loan(final int amount) {
        AccessController.checkPermission(loan);
        credit(amount);
    }
}
import java.security.*;

public class ControlledVar {
    private Permission read = new VarPermission("read", null);
    private Permission write = new VarPermission("write", null);

    private int var;

    public ControlledVar(int initValue) {
        var = initValue;
    }

    public void write(int newValue) {
        AccessController.checkPermission(write);
        var = newValue;
    }

    public int read() {
        AccessController.checkPermission(read);
        return var;
    }
}
<table>
<thead>
<tr>
<th>n</th>
<th>$\delta^0[n]$</th>
<th>$\gamma^0[n]$</th>
<th>$\delta^1[n]$</th>
<th>$\gamma^1[n]$</th>
<th>$\delta^2[n]$</th>
<th>$\gamma^2[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_0$</td>
<td>Permission</td>
<td></td>
<td>Permission</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_2$</td>
<td>${ P_{\text{can pay}}, P_{\text{credit}}, P_{\text{debit}} }$</td>
<td>${ P_{\text{can pay}}, P_{\text{credit}}, P_{\text{debit}} }$</td>
<td>${ P_{\text{can pay}}, P_{\text{credit}}, P_{\text{debit}} }$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_5$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>$n_6$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_7$</td>
<td>${ P_{\text{can pay}}, P_{\text{credit}}, P_{\text{debit}} }$</td>
<td>${ P_{\text{can pay}}, P_{\text{credit}}, P_{\text{debit}} }$</td>
<td>${ P_{\text{can pay}}, P_{\text{credit}}, P_{\text{debit}} }$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_8$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_9$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_{10}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_{11}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_{12}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_{13}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_{14}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_{15}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_{16}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_{17}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_{18}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_{19}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_{20}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_{21}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_{22}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_{23}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_{24}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_{25}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_{26}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A.1: Solutions to the analysis for the e-commerce application.
Index of symbols

\(\mathbb{N}, 23\)
\(\bot, 23\)
\(\ell, 23\)
\(\mathbb{E}, 23\)
\(\mathbb{E}_{\text{call}}, 23\)
\(\mathbb{E}_{\text{trans}}, 23\)
\(\bullet \rightarrow \bullet, 23\)
\(\mathbb{E}_{\text{entry}}, 23\)
\(\mathbb{N}_{\text{entry}}, 24\)
\(\ast, 28\)
\(\Sigma, 28\)
\([\;], 28\)
\(\vdash, 28\)
\(\triangleright, 28\)
\(\trianglerighteq, 28\)
\(\text{Perm}(), 29\)
\(\text{Priv}(), 29\)
\textbf{Permission}, 29
\(\text{JDK}, 31\)
\(\langle\;\rangle, 34\)
\(\Pi, 34\)
\(\Pi_{\text{no,mk}}, 34\)
\(\Pi_{\text{mk}}, 34\)
\(\Pi_{\text{entry}}, 34\)
\(w, 35\)
\(\varphi, 36\)
\(\chi, 39\)
\(\mathbb{N}_{\text{exit}}, 41\)
\(\rho, 42\)
\(\tilde{\Pi}, 44\)
\(\tilde{\Pi}_{\text{no,mk}}, 44\)
\(\tilde{\Pi}_{\text{mk}}, 44\)
\(\tilde{\Pi}_{\text{entry}}, 44\)
\(\tilde{w}, 45\)
\(\tilde{\chi}, 48\)
\(\tilde{\pi}, 49\)
\(\rightarrow_{\pi}, 52\)
\(\alpha, 54\)
\(\tilde{\Pi}_{\nu}, 57\)
\(\tilde{\Pi}_{\text{no,mk}}^\nu, 57\)
\(\tilde{\Pi}_{\text{mk}}^\nu, 57\)
\(\tilde{\Pi}_{\text{entry}}^\nu, 57\)
\(\eta, 64\)
\(\bar{\eta}, 64\)
\(\tilde{\Pi}_{\nu}, 65\)
\(\tilde{\Pi}_{\text{mk}}^\nu, 65\)
\(\mathcal{L}, 70\)
\(\phi, 70\)
\(\subseteq, 70\)
\(f, 70\)
\(\mathcal{F}, 70\)
\(\mu, 70\)
\(\text{fix}, 71\)
\(\bot_{\mathcal{L}}, 71\)
\(\mathcal{L}_{\mathbb{N}}, 72\)
\(\mathcal{F}_{\mathbb{E}}, 72\)
\(\mu_{\mathbb{E}}, 72\)
\(\iota, 72\)
\(\bot_{\mathcal{L}_{\mathbb{N}}}, 72\)
\(\sqcup, 72\)
\(f_{\mathbb{E}}, 73\)
\(\models, 74\)
\(f_{\pi}, 74\)
\(\Delta, 82\)
\(\delta, 82\)
\(\Gamma, 84\)
\(\gamma, 84\)
Bibliography


