Università degli Studi di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

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

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 σ, 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 $\sigma \rhd \sigma'$. The operational semantics incorporates a specification of the Java stack inspection mechanism: this is done through a predicate JDK. The predicate is true for a state σ and a permission P, when P is granted to σ according to the security policy: in this case, we write $\sigma \vdash \mathsf{JDK}(\mathsf{P})$. Otherwise, if P is denied to σ, the predicate is false, and we write $\sigma \nvdash \mathsf{JDK}(\mathsf{P})$.

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 $\mathsf{P} \in \gamma(\mathfrak{n})$ (resp. $\mathsf{P} \in \delta(\mathfrak{n})$). Then, whenever there is a derivation $[] \rhd \cdots \rhd \sigma : \mathfrak{n}$, it always happens $\sigma : \mathfrak{n} \vdash \mathsf{JDK}(\mathsf{P})$ (resp. $\sigma : \mathfrak{n} \nvdash \mathsf{JDK}(\mathsf{P})$). 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 $\mathfrak{m}$ such that $\mathsf{P} \in \gamma(\mathfrak{m})$ or $\mathsf{P} \in \delta(\mathfrak{m})$. 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 operational 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 constraints, 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 approach 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 accuracy. 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 extensions 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:

```
grant  signedBy signer_1,...,signer_k codeBase origin {
       permission name_1 target_1, action_1
           ...
       permission name_h target_h, action_h
}
```

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.

```
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`.

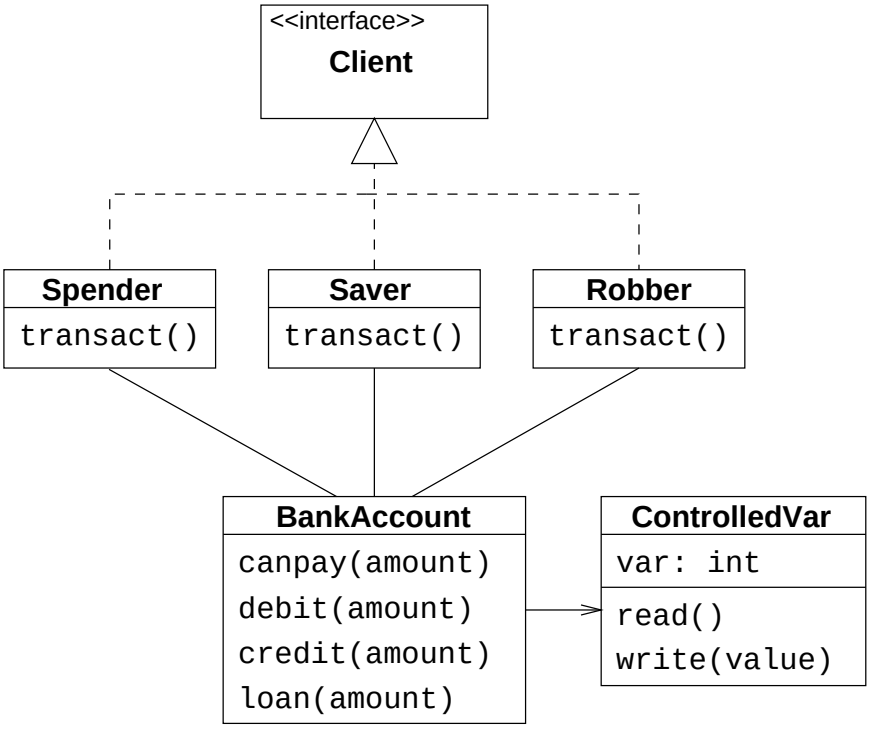

Figure 2.1: UML class diagram for the e-commerce application.

```
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);
    }
}
```

Class	Protection Domain	Permissions
`Spender` `Saver`	Client	`canpay`, `credit`, `debit`
`Robber`	Unknown	
`BankAccount`	Bank	`canpay`, `credit`, `debit`, `loan`, `read`, `write`
`ControlledVar` `Ecommerce`	System	`AllPermission`

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.

Bigliardi 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 Mètayer 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 Mètayer and Thorn [JLT98b] make one of the first attempts to apply static tecniques 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.

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.

Fournet 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 estabilish 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 tecnique 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 $\mathsf{G} = \langle \mathsf{N}, \mathsf{E}, \mathsf{N}_{entry} \rangle$, where:

- N is the set of nodes, including a distinguished element $\bot_{\mathsf{N}}$. Each node $n \in \mathsf{N} \setminus \{ \bot_{\mathsf{N}} \}$ 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 $\bot_{\mathsf{N}}$ plays the technical role of a single, isolated entry point.

- $\mathsf{E} = \mathsf{E}_{call} \uplus \mathsf{E}_{trans} \uplus \mathsf{E}_{entry} \subseteq \mathsf{N} \times (\mathsf{N} \setminus \{ \bot_{\mathsf{N}} \})$ is the set of edges. Edges are split into *call edges* $n \longrightarrow n' \in \mathsf{E}_{call}$, modelling interprocedural flow, and *transfer edges* $n \dashrightarrow n' \in \mathsf{E}_{trans}$, which instead correspond to intraprocedural flow. Moreover, we have the set of *entry edges* $\bullet\!\!\longrightarrow n \in \mathsf{E}_{entry}$,

containing all pairs $(\bot_{\mathsf{N}}, \mathfrak{n})$ for $\mathfrak{n} \in \mathsf{N}_{entry}$. The $\bot_{\mathsf{N}}$ element only appears in entry edges. Since the flow from a return node $\mathfrak{m}$ to next instruction $\mathfrak{n}'$ of the respective call node $\mathfrak{n}$ is *not* explicitly represented by an element $(\mathfrak{m}, \mathfrak{n}') \in \mathsf{E}$, we indicate E as the set of *abstract* edges (cf. def. 3.5.17).

- $\mathsf{N}_{entry} \subseteq \mathsf{N} \setminus \{\bot_{\mathsf{N}}\}$ 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.

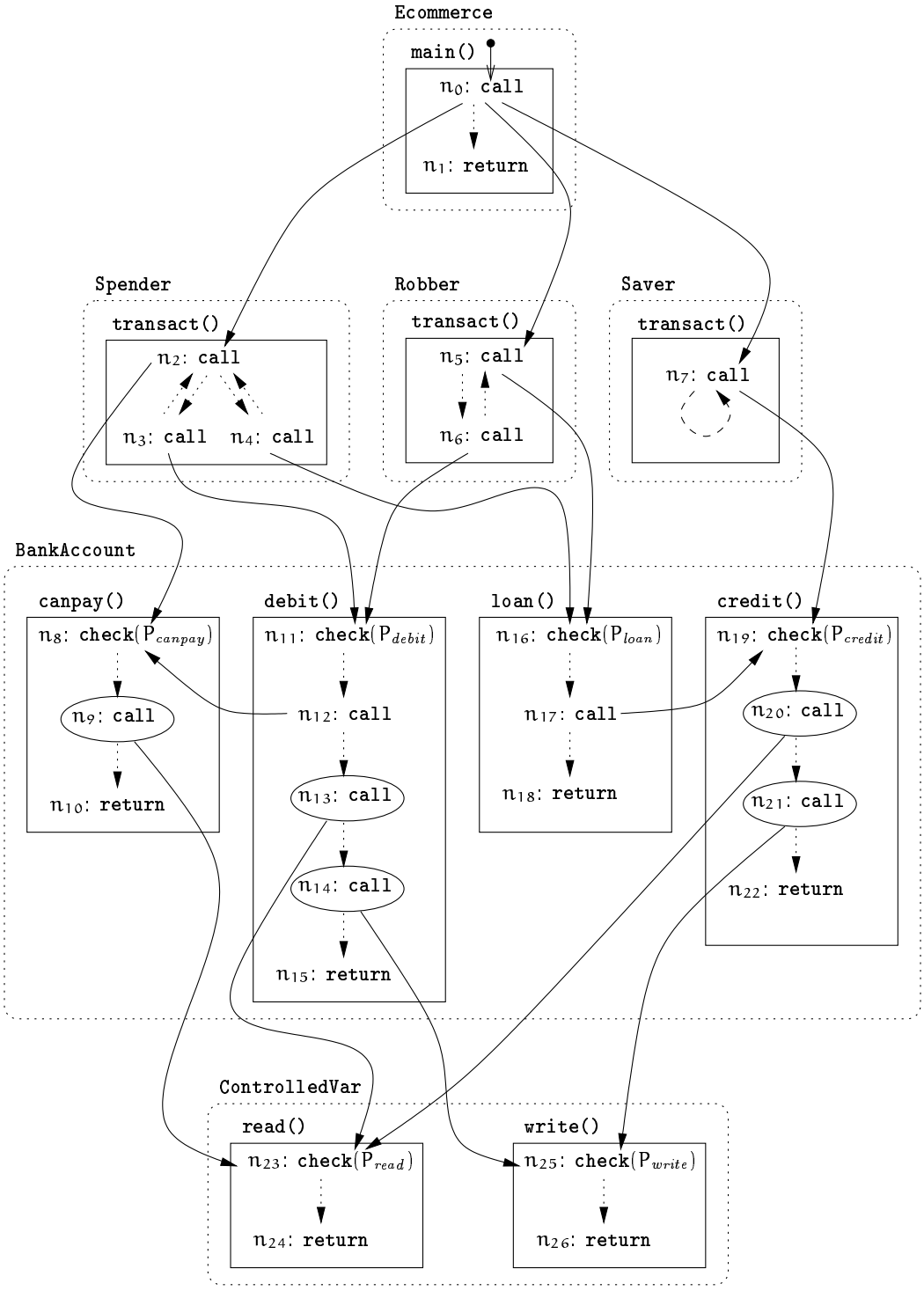

Figure 3.1: Control flow graph for the e-commerce application.

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:

$$\mathsf{E}_{call} \cap \mathsf{E}_{trans} = \varnothing$$

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.:

$$\ell(n) = \texttt{call} \implies \exists n' \in \mathsf{N}.\, n \longrightarrow n'$$

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.:

$$\ell(n) = \texttt{check}(\mathsf{P}) \ \wedge \ (n, n') \in \mathsf{E} \implies n \dashrightarrow n'$$

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.:

$$\ell(\mathtt{n}) = \texttt{return} \quad \Longrightarrow \quad \neg\exists \mathtt{n}' \in \mathsf{N}.\ (\mathtt{n}, \mathtt{n}') \in \mathsf{E}$$

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.

$$\frac{n \in \mathsf{N}_{entry}}{[] \rhd [n]} \quad [\rhd_{\varnothing}]$$

$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow n'}{\sigma : n \rhd \sigma : n : n'} \quad [\rhd_{call}]$$

$$\frac{\ell(n) = \texttt{check}(\mathsf{P}) \quad \sigma : n \vdash \mathsf{JDK}(\mathsf{P}) \quad n \dashrightarrow n'}{\sigma : n \rhd \sigma : n'} \quad [\rhd_{check}]$$

$$\frac{\ell(m) = \texttt{return} \quad n \dashrightarrow n'}{\sigma : n : m \rhd \sigma : n'} \quad [\rhd_{return}]$$

Table 3.1: Operational semantics of control flow graphs.

3.2 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 $\mathsf{N} \setminus \{\perp_{\mathsf{N}}\}$, i.e.:

$$\Sigma \quad = \quad (\mathsf{N} \setminus \{\perp_{\mathsf{N}}\})^*$$

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 $n \in \mathsf{N} \setminus \{\perp_{\mathsf{N}}\}$, then $\sigma : n$ is the concatenation between σ and $[n]$.

The transition relation $\rhd \subseteq \Sigma \times \Sigma$ is defined in Table 3.1 (the JDK predicate is defined in Table 3.3). We write $\sigma \rhd \sigma'$ when the execution can lead from σ to σ' 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:

$$\frac{}{\mathsf{G} \vdash []} \qquad \frac{\mathsf{G} \vdash \sigma \quad \sigma \rhd \sigma'}{\mathsf{G} \vdash \sigma'}$$

We say that a state σ is G-reachable iff $\mathsf{G} \vdash \sigma$. We say that a node n is G-reachable iff $\mathsf{G} \vdash \sigma : n$ for some state σ.

n	$Perm(n)$
$n_2 - n_4, n_7$	$\{ \mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit} \}$
$n_5 - n_6$	$\varnothing$
$n_8 - n_{22}$	$\{ \mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}, \mathsf{P}_{loan}, \mathsf{P}_{read}, \mathsf{P}_{write} \}$
$n_0 - n_1, n_{23} - n_{26}$	**Permission**

Table 3.2: Permissions for the e-commerce application.

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 $n \in \mathsf{N} \setminus \{ \bot_{\mathsf{N}} \}$ with the following additional information:

- $Perm(n)$, the set of permissions associated with n.
- $Priv(n)$, 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 **Permission** the set of permissions which actually occur in the control flow graph of interest:

$$\textbf{Permission} \;=\; \bigcup_{n \in \mathsf{N}} Perm(n) \;\cup\; \bigcup_{n \in \mathsf{N}} \{ \mathsf{P} \mid \ell(n) = \mathtt{check}(\mathsf{P}) \}$$

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.:

$$n \dashrightarrow n' \quad \Longrightarrow \quad Perm(n) = Perm(n')$$

Well-formedness constraint 6. Only call nodes can be privileged, i.e.:

$$\mathit{Priv}(\mathfrak{n}) \quad \Longrightarrow \quad \ell(\mathfrak{n}) = \texttt{call}$$

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

CHECK-PERMISSION(P, σ)

```
1  while σ ≠ NIL do
2     n ← POP(σ)
3     if P ∉ Perm(n)
4        then throw  "access control exception"
5     if Priv(n)
6        then return
```

Figure 3.2: The stack inspection algorithm.

$$\frac{}{[] \vdash \mathsf{JDK}(\mathsf{P})} \quad [\mathsf{JDK}_{\varnothing}]$$

$$\frac{\mathsf{P} \in \mathit{Perm}(\mathfrak{n}) \quad \sigma \vdash \mathsf{JDK}(\mathsf{P})}{\sigma : \mathfrak{n} \vdash \mathsf{JDK}(\mathsf{P})} \quad [\mathsf{JDK}_{\prec}]$$

$$\frac{\mathsf{P} \in \mathit{Perm}(\mathfrak{n}) \quad \mathit{Priv}(\mathfrak{n})}{\sigma : \mathfrak{n} \vdash \mathsf{JDK}(\mathsf{P})} \quad [\mathsf{JDK}_{\mathit{Priv}}]$$

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 σ and node $\mathfrak{n}$:

$$\mathsf{P} \notin \mathit{Perm}(\mathfrak{n}) \implies \sigma : \mathfrak{n} \nvdash \mathsf{JDK}(\mathsf{P})$$

- 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:

$$[] = \sigma_0 \rhd \sigma_1 \rhd \cdots \rhd \sigma_k = \sigma : n$$

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 $k = 1$, $n \in N_{entry}$ and $\sigma = []$: 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 $\sigma_{k-1} \rhd \sigma_k$, we have:

- case [*call*]:
$$\frac{\ell(n') = \texttt{call} \quad n' \longrightarrow n}{\sigma' : n' \rhd \sigma' : n' : n} \qquad \text{where } \sigma_{k-1} = \sigma' : n'$$
Here, the i we are looking for is just $k-1$.
- case [*check*]:
$$\frac{\ell(n') = \texttt{check(P)} \quad \sigma : n' \vdash \mathsf{JDK(P)} \quad n' \dashrightarrow n}{\sigma : n' \rhd \sigma : n}$$
By the inductive hypothesis, $\exists i \in 0..k-2.\ \sigma_i = \sigma$.
- case [*return*]:
$$\frac{\ell(m) = \texttt{return} \quad n' \dashrightarrow n}{\sigma : n' : m \rhd \sigma : n}$$
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 $\sigma_i = \sigma$.

□

Corollary 3.4.2.

$$G \vdash \sigma : n \quad \Longrightarrow \quad G \vdash \sigma$$

Proof. Let $\sigma : n$ be a G-reachable state, i.e. $G \vdash \sigma : n$. By definition, this means that a derivation $\sigma_0 \rhd \cdots \rhd \sigma_k$ exists, with $\sigma_0 = []$ 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, $\sigma_0 \rhd \cdots \rhd \sigma_i$ is a derivation for σ: thus, $G \vdash \sigma$. □

Lemma 3.4.3. Let:

$$[] = \sigma_0 \rhd \sigma_1 \rhd \cdots \rhd \sigma_k = \sigma : n : m$$

be a derivation, and define:

$$i^\star \ = \ \max\{\, i \in 0..k-1 \mid \sigma_i = \sigma : n \,\}$$

Then:

$$\forall i \in i^\star..k.\ \exists \tau_i \in N^*.\ \sigma_i = \sigma : n : \tau_i$$

Proof. By lemma 3.4.1, we know that there is at least one index i such that $\sigma_i = \sigma : n$, hence $i^\star$ 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 $n \in N_{entry}$, $\ell(n) = \texttt{call}$ and $n \longrightarrow m$: here $i^\star = 1$ and we define $\tau_1 = []$. For the inductive case, we proceed by case analysis on the transition $\sigma_{k-1} \rhd \sigma_k$, yielding:

- case [*call*]:

$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow m}{\sigma : n \rhd \sigma : n : m} \qquad \text{where } \sigma_{k-1} = \sigma : n$$

  Here, by definition of $i^\star$, we have $i^\star = k-1$, and the lemma holds by letting $\tau_{i^\star} = []$.

- case [*check*]:

$$\frac{\ell(m') = \texttt{check}(P) \quad \sigma : n : m' \vdash \mathsf{JDK}(P) \quad m' \dashrightarrow m}{\sigma : n : m' \rhd \sigma : n : m}$$

  By the inductive hypothesis, $\forall i \in i^\star..k-1.\ \exists \tau_i \in N^*.\ \sigma_i = \sigma : n : \tau_i$.

- case [*return*]:

$$\frac{\ell(n') = \texttt{return} \quad m' \dashrightarrow m}{\sigma : n : m' : n' \rhd \sigma : n : m}$$

  Define $j^\star = \max\{\, j \in 0..k-2 \mid \sigma_j = \sigma : n : m' \,\}$. By the inductive hypothesis, $\forall i \in j^\star..k-1.\ \exists \tau'_i \in N^*.\ \sigma_i = \sigma : n : m' : \tau'_i$. Applying again the inductive hypothesis, we have that: $\forall i \in i^\star..j^\star - 1.\ \exists \tau''_i \in N^*.\ \sigma_i = \sigma : n : \tau''_i$. Then we define:

$$\tau_i = \begin{cases} \tau''_i & \text{if } i^\star \le i < j^\star \\ m' : \tau'_i & \text{if } j^\star \le i < k \end{cases}$$

□

Lemma 3.4.4.

$$G \vdash \sigma : n : m \quad \Longrightarrow \quad \ell(n) = \texttt{call}$$

Proof. Consider an arbitrary derivation for $\sigma : n : m$, say: $\sigma_0 \rhd \cdots \rhd \sigma_k$, where $\sigma_0 = []$ and $\sigma_k = \sigma : n : m$. By lemma 3.4.3, we know that, choosing $i^\star = \max\{\, i \in 0..k-1 \mid \sigma_i = \sigma : n \,\}$, we have $\forall i \in i^\star..k.\ \exists \tau_i \in N^*.\ \sigma_i = \sigma : n : \tau_i$. By contradiction, assume $\ell(n) \neq \texttt{call}$. This leaves the following cases for the transition $\sigma_{i^\star} \rhd \sigma_{i^\star+1}$:

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

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

□

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 $G = \langle N, E, N_{entry} \rangle$ 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 Π_{n_0,n_k} the set of all abstract paths from n_0 to n_k, and with Π 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 Π_n the set of all abstract entry paths leading to n, and with Π_{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.

$$G \vdash \sigma : n \quad \Longrightarrow \quad \Pi_n \neq \varnothing$$

Proof. The proof is carried out by induction on the derivation used to estabilish $G \vdash \sigma : n$. For the base case, if $\sigma = []$ 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*]:

$$\frac{\ell(n') = \texttt{call}\ \ n' \longrightarrow n}{\sigma' : n' \ \triangleright\ \sigma' : n' : n} \qquad \text{where } \sigma = \sigma' : n'$$

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

- case [*check*]:

$$\frac{\ell(n') = \texttt{check}(P) \quad \sigma : n' \vdash \mathsf{JDK}(P) \quad n' \dashrightarrow n}{\sigma : n' \rhd \sigma : n}$$

By the inductive hypothesis, $G \vdash \sigma : n'$ implies $\Pi_{n'} \neq \varnothing$. Again, take $\pi \in \Pi_{n'}$. Here $n' \dashrightarrow n$, hence $\pi : n \in \Pi_n$.

- case [*return*]:

$$\frac{\ell(m) = \texttt{return} \quad n' \dashrightarrow n}{\sigma : n' : m \rhd \sigma : n}$$

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

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

Therefore, we can apply the inductive hypothesis to $\sigma : n'$, obtaining $\Pi_{n'} \neq \varnothing$. Again, if we take $\pi \in \Pi_{n'}$, then $n' \dashrightarrow n$ implies $\pi : n \in \Pi_n$.

□

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

$$w(n, n') = \begin{cases} 1 & \text{if } n \longrightarrow n' \\ 0 & \text{if } n \dashrightarrow n' \end{cases}$$

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:

$$\begin{aligned} w(\langle\rangle) &= 0 \\ w(\langle n_0, \ldots, n_k \rangle) &= 1 + \sum_{i=0}^{k-1} w(n_i, n_{i+1}) \end{aligned}$$

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:

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

Proof. Since π_0 and π_1 are non-empty, by definition 3.5.4 we have:

$$\begin{aligned} w(\pi_0) + w(\pi_1) &= \left(1 + \sum_{i=0}^{h-1} w(n_i, n_{i+1})\right) + \left(1 + \sum_{i=h+1}^{k-1} w(n_i, n_{i+1})\right) \\ &= \left(1 + \sum_{i=0}^{k-1} w(n_i, n_{i+1})\right) + 1 - w(n_h, n_{h+1}) \\ &= w(\pi) + 1 - w(n_h, n_{h+1}) \end{aligned}$$

□

Definition 3.5.6. Let π be an abstract path, $n, n' \in N$, and $\sigma \in \Sigma$. We define the transition system $\chi = \langle \Pi \cup \Sigma, \Sigma, \rightarrow_\chi \rangle$ by the following rules:

$$\frac{n \in N_{entry}}{\langle n \rangle \rightarrow_\chi [n]} \qquad [\chi_{entry}]$$

$$\frac{\pi : n' \rightarrow_\chi \sigma \quad n' \longrightarrow n}{\pi : n' : n \rightarrow_\chi \sigma : n} \qquad [\chi_{call}]$$

$$\frac{\pi : n' \rightarrow_\chi \sigma : n' \quad n' \dashrightarrow n}{\pi : n' : n \rightarrow_\chi \sigma : n} \qquad [\chi_{trans}]$$

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

$$\pi \rightarrow_\chi [m_0, \ldots, m_h]$$

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

$$\forall i \in 0..h.\ m_i = n_{\varphi(i)} \quad \wedge \quad \varphi(h) = k \tag{3.1}$$

Proof. The proof is carried out by mathematical induction on the length of the abstract path π. For the base case, if $\pi = \langle n_0 \rangle$ and $n_0 \in N_{entry}$, then it must be $\pi \rightarrow_\chi [n_0]$, because only the first rule of χ 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*]:

  $$\frac{\langle n_0, \ldots, n_{k-1} \rangle \rightarrow_\chi \sigma \quad n_{k-1} \longrightarrow n_k}{\langle n_0, \ldots, n_k \rangle \rightarrow_\chi \sigma : n_k}$$

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

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

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

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

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

- case [*transfer*]:

$$\frac{\langle n_0, \ldots, n_{k-1}\rangle \rightarrow_\chi \sigma : n_{k-1} \quad n_{k-1} \dashrightarrow n_k}{\langle n_0, \ldots, n_k\rangle \rightarrow_\chi \sigma : n_k}$$

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, φ is strictly increasing on $0..|\sigma|-1$, because, inside that interval, it coincides with φ'. 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, φ is strictly increasing inside the whole domain $0..|\sigma|$. Since we also have $n_k = n_{\varphi(|\sigma|)}$, we can conclude that φ satisfies (3.1).

□

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 φ is the function defined in lemma 3.5.7 and $i \in 0..h$, we have:

$$\langle n_0, \ldots, n_{\varphi(i)}\rangle \rightarrow_\chi [m_0, \ldots, m_i] \tag{3.2}$$

Proof. The proof is carried out by mathematical induction on the length of the abstract path π. 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 $n_0 \in N_{entry}$, then, by the first rule of χ, 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*]:

$$\frac{\langle n_0, \ldots, n_{k-1}\rangle \rightarrow_\chi \sigma \quad n_{k-1} \longrightarrow n_k}{\langle n_0, \ldots, n_k\rangle \rightarrow_\chi \sigma : n_k}$$

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

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

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

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

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*]:

$$\frac{\langle n_0, \ldots, n_{k-1}\rangle \rightarrow_\chi \sigma : n_{k-1} \quad n_{k-1} \dashrightarrow n_k}{\langle n_0, \ldots, n_k\rangle \rightarrow_\chi \sigma : n_k}$$

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

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

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:

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

□

Lemma 3.5.9. The evaluation of χ is deterministic, i.e.:

$$\pi \rightarrow_\chi \sigma \ \wedge\ \pi \rightarrow_\chi \sigma' \quad \Longrightarrow \quad \sigma = \sigma'$$

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 π. For the base case, let $\pi = \langle n\rangle$, with $n \in N_{entry}$. If $\pi \rightarrow_\chi \sigma$ and $\pi \rightarrow_\chi \sigma'$, then it must be $\sigma = \sigma' = [n]$, because only the first rule of χ 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:

$$\frac{\pi' : n' \rightarrow_\chi \sigma \quad n' \longrightarrow n}{\pi' : n' : n \rightarrow_\chi \sigma : n} \qquad \frac{\pi' : n' \rightarrow_\chi \sigma' \quad n' \longrightarrow n}{\pi' : n' : n \rightarrow_\chi \sigma' : n}$$

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

- case [*transfer*]: consider the two transitions:

$$\frac{\pi' : n' \rightarrow_\chi \sigma : n' \quad n' \dashrightarrow n}{\pi' : n' : n \rightarrow_\chi \sigma : n} \qquad \frac{\pi' : n' \rightarrow_\chi \sigma' : n' \quad n' \dashrightarrow n}{\pi' : n' : n \rightarrow_\chi \sigma' : n}$$

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

□

Lemma 3.5.10. The evaluation of χ is total on abstract entry paths, i.e.:

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

Proof. We prove the stronger result:

$$\pi \in \Pi_n \quad \Longrightarrow \quad \exists \sigma \in \Sigma.\ \pi \rightarrow_\chi \sigma : n$$

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

$$\pi' : n' \in \Pi_{n'} \quad \Longrightarrow \quad \exists \sigma' \in \Sigma.\ \pi' : n' \rightarrow_\chi \sigma' : n'$$

Now we proceed by case analysis on the edge (n', n):

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

$$\frac{\pi' : n' \rightarrow_\chi \sigma' : n' \quad n' \longrightarrow n}{\pi' : n' : n \rightarrow_\chi \sigma' : n' : n}$$

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

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

$$\frac{\pi' : n' \rightarrow_\chi \sigma' : n' \quad n' \dashrightarrow n}{\pi' : n' : n \rightarrow_\chi \sigma' : n}$$

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

□

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

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

Theorem 3.5.11.

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

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

- case [*call*]:

$$\frac{\ell(n') = \texttt{call} \quad n' \longrightarrow n}{\sigma' : n' \rhd \sigma' : n' : n} \quad \text{where } \sigma = \sigma' : n'$$

  By the inductive hypothesis applied to $\sigma' : n'$, we have:

$$\exists \pi \in \Pi_{n'} . \chi(\pi) = \sigma' : n'$$

  Since $n' \longrightarrow n$, the path $\pi : n$ is in Π_n, and $\chi(\pi : n) = \sigma' : n' : n$.

- case [*check*]:

$$\frac{\ell(n') = \texttt{check}(P) \quad \sigma : n' \vdash \mathsf{JDK}(P) \quad n' \dashrightarrow n}{\sigma : n' \rhd \sigma : n}$$

  By the inductive hypothesis applied to $\sigma : n'$, we have:

$$\exists \pi \in \Pi_{n'} . \chi(\pi) = \sigma : n'$$

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

- case [*return*]:

$$\frac{\ell(m) = \texttt{return} \quad n' \dashrightarrow n}{\sigma : n' : m \rhd \sigma : n}$$

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

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

  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, $n' \dashrightarrow n$ implies $\pi : n \in \Pi_n$, and $\chi(\pi : n) = \sigma : n$.

□

Theorem 3.5.12. Let π be an abstract entry path. Then:

$$\left|\chi(\pi)\right| \quad = \quad w(\pi)$$

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

- case [*call*]:

$$\frac{\chi(\pi' : n') = \sigma \quad n' \longrightarrow n}{\chi(\pi' : n' : n) = \sigma : n} \quad \text{where } \pi = \pi' : n' : n$$

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

$$\left|\chi(\pi' : n' : n)\right| \; = \; |\sigma : n| \; = \; |\sigma| + 1 \; = \; w(\pi' : n') + 1 \; = \; w(\pi' : n' : n)$$

- case [*transfer*]:

$$\frac{\chi(\pi' : n') = \sigma : n' \quad n' \dashrightarrow n}{\chi(\pi' : n' : n) = \sigma : n} \qquad \text{where } \pi = \pi' : n' : n$$

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

$$\big|\chi(\pi' : n' : n)\big| \;=\; |\sigma : n| \;=\; |\sigma : n'| \;=\; w(\pi' : n') \;=\; w(\pi' : n' : n)$$

□

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:

$$n \in N_{exit} \quad \Longrightarrow \quad \forall \pi \in \Pi_n.\ w(\pi) = 1$$

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:

$$G \vdash \sigma : n \quad \Longrightarrow \quad \neg \exists \sigma' \in \Sigma.\ \sigma : n \rhd \sigma'$$

Proof. By contradiction, assume $\sigma : n \rhd \sigma'$ for some $\sigma' \in \Sigma$. Then, since $\ell(n) = \texttt{return}$, only the $\rhd_{return}$ rule is applicable, and we obtain:

$$\frac{\ell(n) = \texttt{return} \quad m \dashrightarrow n'}{\sigma'' : m : n \rhd \sigma'' : n'} \qquad \text{where } \sigma = \sigma'' : m \text{ and } \sigma' = \sigma'' : n'$$

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:

$$1 \;=\; w(\pi) \;=\; |\sigma'' : m : n| \;=\; |\sigma''| + 2$$

□

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

$$\rho(n, n') \;=\; \{\, m \in N \mid \ell(m) = \texttt{return} \;\wedge\; \exists \pi \in \Pi_{n',m}.\; w(\pi) = 1 \,\}$$

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

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

Lemma 3.5.16.

$$G \vdash \sigma : n : m \;\wedge\; \ell(m) = \texttt{return} \quad\Longrightarrow\quad m \in \rho(n)$$

Proof. We prove the stronger result:

$$G \vdash \sigma : n : m \quad\Longrightarrow\quad \exists n' \in N,\; \pi \in \Pi_{n',m}.\; n \longrightarrow n' \;\wedge\; w(\pi) = 1$$

The proof is carried out by induction on the derivation used to estabilish $G \vdash \sigma : n : m$. For the base case, we have $\sigma = []$, $n \in N_{entry}$, $\ell(n) = \texttt{call}$ and $n \longrightarrow m$: 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*]:
$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow m}{\sigma : n \;\triangleright\; \sigma : n : m} \qquad \text{where } \sigma_{k-1} = \sigma : n$$
Here $n \longrightarrow m$ and $\langle m \rangle$ is clearly an abstract path from m to itself having unit weight.
- case [*check*]:
$$\frac{\ell(m') = \texttt{check(P)} \quad \sigma : n : m' \vdash \mathsf{JDK(P)} \quad m' \dashrightarrow m}{\sigma : n : m' \;\triangleright\; \sigma : n : m}$$
By the inductive hypothesis:
$$\exists n' \in N,\; \pi' \in \Pi_{n',m'}.\; n \longrightarrow n' \;\wedge\; w(\pi') = 1$$
Then the path $\pi = \pi' : m$ is in $\Pi_{n',m}$, and $w(\pi) = w(\pi') = 1$ as $m' \dashrightarrow m$.
- case [*return*]:
$$\frac{\ell(m'') = \texttt{return} \quad m' \dashrightarrow m}{\sigma : n : m' : m'' \;\triangleright\; \sigma : n : m}$$
By lemma 3.4.1, any derivation of $\sigma : n : m' : m''$ is of the form:
$$[] \;\triangleright\; \cdots \;\triangleright\; \sigma : n : m' \;\triangleright\; \cdots \;\triangleright\; \sigma : n : m' : m''$$
Therefore, we can apply the inductive hypothesis to $\sigma : n : m'$, obtaining:
$$\exists n' \in N,\; \pi' \in \Pi_{n',m'}.\; n \longrightarrow n' \;\wedge\; w(\pi') = 1$$
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 $\ell(m) = \texttt{return}$ and by definition of $\rho(n)$. □

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 $n \longrightarrow n'$ for some n) is connected to a return node by a path with unit weight:

$$n \longrightarrow n' \quad \Longrightarrow \quad \rho(n, n') \neq \varnothing$$

Definition 3.5.17. We say that $m \hookrightarrow n'$ is a *return edge* iff:

$$\exists n \in N.\ m \in \rho(n) \ \wedge\ n \dashrightarrow n'$$

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

Lemma 3.5.18.

$$m \in N_{exit} \ \wedge\ \exists n \in N.\ m \in \rho(n) \quad \Longrightarrow \quad \Pi_n = \varnothing$$

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

$$\exists \pi' \in \Pi_{n',m}.\ w(\pi') = 1 \tag{3.3}$$

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

$$\begin{aligned}
1 &= w(\pi) && \text{by w.f. constraint 7} \\
&= w(\pi'') + w(\pi') + w(n, n') - 1 && \text{by lemma 3.5.5} \\
&= w(\pi'') + w(\pi') + 1 - 1 && \text{as } n \longrightarrow n' \\
&= w(\pi'') + 1 && \text{by (3.3)}
\end{aligned}$$

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

3.6 Concrete paths

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

$$\begin{array}{lll} \forall i \in 0..k-1. & \ell(n_i) = \texttt{call} & \wedge \quad n_i \longrightarrow n_{i+1}, \text{ or} \\ & \ell(n_i) = \texttt{check}(P) & \wedge \quad n_i \dashrightarrow n_{i+1}, \text{ or} \\ & \ell(n_i) = \texttt{return} & \wedge \quad n_i \hookrightarrow n_{i+1} \end{array}$$

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

Lemma 3.6.2.

$$G \vdash \sigma : n \quad \Longrightarrow \quad \widetilde{\Pi}_n \neq \varnothing$$

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

- case [*call*]:
$$\frac{\ell(n') = \texttt{call} \quad n' \longrightarrow n}{\sigma' : n' \rhd \sigma' : n' : n} \qquad \text{where } \sigma = \sigma' : n'$$
By the inductive hypothesis, $G \vdash \sigma : n'$ implies $\widetilde{\Pi}_{n'} \neq \varnothing$. So, take $\tilde{\pi} \in \widetilde{\Pi}_{n'}$. Since $\ell(n') = \texttt{call}$ and $n' \longrightarrow n$, the path $\tilde{\pi} : n$ is in $\widetilde{\Pi}_n$.

- case [*check*]:
$$\frac{\ell(n') = \texttt{check}(P) \quad \sigma : n' \vdash \mathsf{JDK}(P) \quad n' \dashrightarrow n}{\sigma : n' \rhd \sigma : n}$$
By the inductive hypothesis, $G \vdash \sigma : n'$ implies $\widetilde{\Pi}_{n'} \neq \varnothing$. Again, take $\tilde{\pi} \in \widetilde{\Pi}_{n'}$. Here $\ell(n') = \texttt{check}(P)$ and $n' \dashrightarrow n$, hence $\tilde{\pi} : n \in \widetilde{\Pi}_n$.

- case [*return*]:
$$\frac{\ell(m) = \texttt{return} \quad n' \dashrightarrow n}{\sigma : n' : m \rhd \sigma : n}$$
By the inductive hypothesis, $G \vdash \sigma : n' : m$ implies $\widetilde{\Pi}_m \neq \varnothing$. Moreover, by lemma 3.5.16, $G \vdash \sigma : n' : m$ and $\ell(m) = \texttt{return}$ imply $m \in \rho(n')$. Now, take $\tilde{\pi} \in \widetilde{\Pi}_m$. From $m \in \rho(n')$ and $n' \dashrightarrow n$ we deduce $m \hookrightarrow n$, hence $\tilde{\pi} : n \in \widetilde{\Pi}_n$.

□

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

$$\begin{aligned}\tilde{w}(\langle\rangle) &= 0\\ \tilde{w}(\langle n_0,\ldots,n_k\rangle) &= 1+\sum_{i=0}^{k-1} w(n_i,n_{i+1})\end{aligned}$$

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:

$$\begin{aligned}\tilde{w}(\tilde{\pi}_0)+\tilde{w}(\tilde{\pi}_1) &= \left(1+\sum_{i=0}^{h-1}\tilde{w}(n_i,n_{i+1})\right)+\left(1+\sum_{i=h+1}^{k-1}\tilde{w}(n_i,n_{i+1})\right)\\ &= \left(1+\sum_{i=0}^{k-1}\tilde{w}(n_i,n_{i+1})\right)+1-w(n_h,n_{h+1})\\ &= \tilde{w}(\tilde{\pi})+1-w(n_h,n_{h+1})\end{aligned}$$

□

Definition 3.6.5. Let $\tilde{\pi}$ be a concrete path, $n,n',m\in N$, and $\sigma\in\Sigma$. We define the transition system $\tilde{\chi}=\langle\widetilde{\Pi}\cup\Sigma,\Sigma,\rightarrow_{\tilde{\chi}}\rangle$ by the following rules:

$$\frac{n\in N_{entry}}{\langle n\rangle\rightarrow_\chi[n]}\qquad[\tilde{\chi}_{entry}]$$

$$\frac{\tilde{\pi}:n'\rightarrow_\chi\sigma\quad n'\longrightarrow n}{\tilde{\pi}:n':n\rightarrow_\chi\sigma:n}\qquad[\tilde{\chi}_{call}]$$

$$\frac{\tilde{\pi}:n'\rightarrow_\chi\sigma:n'\quad n'\dashrightarrow n}{\tilde{\pi}:n':n\rightarrow_\chi\sigma:n}\qquad[\tilde{\chi}_{trans}]$$

$$\frac{\tilde{\pi}:m\rightarrow_\chi\sigma:n':m\quad m\hookrightarrow n}{\tilde{\pi}:m:n\rightarrow_\chi\sigma:n}\qquad[\tilde{\chi}_{return}]$$

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:

$$\forall i\in 0..h.\ m_i=n_{\varphi(i)}\quad\wedge\quad\varphi(h)=k \tag{3.4}$$

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

- case [*return*]:
$$\frac{\langle n_0, \ldots, n_{k-1}\rangle \rightarrow_{\tilde{\chi}} \sigma : m : n_{k-1} \quad n_{k-1} \hookrightarrow n_k}{\langle n_0, \ldots, n_k\rangle \rightarrow_{\tilde{\chi}} \sigma : n_k}$$
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, φ is strictly increasing on $0..|\sigma|-1$, because, inside that interval, it coincides with φ'. 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, φ is strictly increasing inside the whole domain $0..|\sigma|$. Since we also have $n_k = n_{\varphi(|\sigma|)}$, we can conclude that φ satisfies (3.4).

□

Lemma 3.6.7. 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, if φ is the function defined in lemma 3.6.6, and $i \in 0..h$, we have:
$$\langle n_0, \ldots, n_{\varphi(i)}\rangle \rightarrow_{\tilde{\chi}} [m_0, \ldots, m_i] \tag{3.5}$$

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

- case [*return*]:
$$\frac{\langle n_0, \ldots, n_{k-1}\rangle \rightarrow_{\tilde{\chi}} \sigma : m : n_{k-1} \quad n_{k-1} \hookrightarrow n_k}{\langle n_0, \ldots, n_k\rangle \rightarrow_{\tilde{\chi}} \sigma : n_k}$$
Let $\sigma : m : n_{k-1} = [m'_0, \ldots, m'_{h'}]$, with $m'_{h'-1} = m$ and $m'_{h'} = n_{k-1}$. 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:
$$\forall i \in 0..h'.\ \langle n_0, \ldots, n_{\varphi'(i)}\rangle \rightarrow_{\chi} [m'_0, \ldots, m'_i]$$

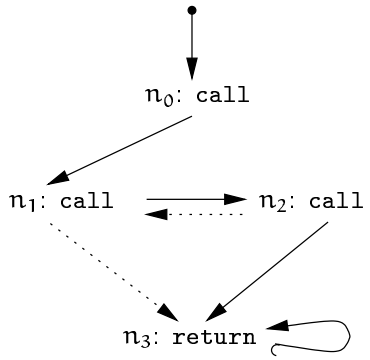

Figure 3.3: Control flow graph for counterexample 3.6.9

Following the construction of lemma 3.6.6, we define $\varphi : 0..h \to 0..k$ 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:

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

□

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

$$\tilde{\pi} \to_{\tilde{\chi}} \sigma \;\wedge\; \tilde{\pi} \to_{\tilde{\chi}} \sigma' \quad\Longrightarrow\quad \sigma = \sigma'$$

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

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

- case [*return*]: consider the two transitions:

$$\frac{\tilde{\pi}' : n' \to_{\tilde{\chi}} \sigma : m : n' \quad n' \hookrightarrow\ n}{\tilde{\pi}' : n' : n \to_{\tilde{\chi}} \sigma : n} \qquad \frac{\tilde{\pi}' : n' \to_{\tilde{\chi}} \sigma' : m' : n' \quad n' \hookrightarrow\ n}{\tilde{\pi}' : n' : n \to_{\tilde{\chi}} \sigma' : n}$$

  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 $n_0 \in N_{entry}$ and $n_i \longrightarrow n_{i+1}$ for $i \in 0..2$, by applying the $\tilde{\chi}_{entry}$ rule once and $\tilde{\chi}_{call}$ three times, we obtain:

$$\langle n_0, n_1, n_2, n_3 \rangle \rightarrow_{\tilde{\chi}} [n_0, n_1, n_2, n_3]$$

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

$$\frac{\langle n_0, n_1, n_2, n_3 \rangle \rightarrow_{\tilde{\chi}} [n_0, n_1, n_2, n_3] \quad n_3 \hookrightarrow n_3}{\langle n_0, n_1, n_2, n_3, n_3 \rangle \rightarrow_{\tilde{\chi}} [n_0, n_1, n_3]}$$

It is evident that, since $n_3 \hookrightarrow n_3$ 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 $\tilde{\chi}_{return}$ rule is applied, we derive a state whose length is an unit shorter: eventually, the state will become of unit length, and the $\tilde{\chi}_{return}$ rule will no more be applicable. In our case, to attain this aim it suffices to repeat the foregoing process two times more:

$$\frac{\langle n_0, n_1, n_2, n_3, n_3 \rangle \rightarrow_{\tilde{\chi}} [n_0, n_1, n_3] \quad n_3 \hookrightarrow n_3}{\langle n_0, n_1, n_2, n_3, n_3, n_3 \rangle \rightarrow_{\tilde{\chi}} [n_0, n_3]}$$

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

$$\frac{\langle n_0, n_1, n_2, n_3, n_3, n_3 \rangle \rightarrow_{\tilde{\chi}} [n_0, n_3] \quad n_3 \hookrightarrow n_3}{\langle n_0, n_1, n_2, n_3, n_3, n_3, n_3 \rangle \rightarrow_{\tilde{\chi}} [n_3]}$$

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 $\langle n_0, n_1, n_2, n_3, n_3, n_3, n_3 \rangle$. □

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

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

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

$$\left|\tilde{\chi}(\tilde{\pi})\right| \quad = \quad \tilde{w}(\tilde{\pi})$$

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

- case [*call*]:

$$\frac{\tilde{\chi}(\tilde{\pi}' : n') = \sigma \quad n' \longrightarrow n}{\tilde{\chi}(\tilde{\pi}' : n' : n) = \sigma : n} \qquad \text{where } \tilde{\pi} = \tilde{\pi}' : n' : n$$

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

$$\left|\tilde{\chi}(\tilde{\pi}' : n' : n)\right| \;=\; |\sigma : n| \;=\; |\sigma| + 1 \;=\; \tilde{w}(\tilde{\pi}' : n') + 1 \;=\; \tilde{w}(\tilde{\pi}' : n' : n)$$

- case [*transfer*]:

$$\frac{\tilde{\chi}(\tilde{\pi}' : n') = \sigma : n' \quad n' \dashrightarrow n}{\tilde{\chi}(\tilde{\pi}' : n' : n) = \sigma : n} \qquad \text{where } \tilde{\pi} = \tilde{\pi}' : n' : n$$

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

$$\left|\tilde{\chi}(\tilde{\pi}' : n' : n)\right| \;=\; |\sigma : n| \;=\; |\sigma : n'| \;=\; \tilde{w}(\tilde{\pi}' : n') \;=\; \tilde{w}(\tilde{\pi}' : n' : n)$$

- case [*return*]:

$$\frac{\tilde{\chi}(\tilde{\pi}' : m) = \sigma : n' : m \quad m \hookrightarrow n}{\tilde{\chi}(\tilde{\pi}' : m : n) = \sigma : n} \qquad \text{where } \tilde{\pi} = \tilde{\pi}' : m : n$$

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

$$\left|\tilde{\chi}(\tilde{\pi}' : m : n)\right| \;=\; |\sigma : n| \;=\; |\sigma : n' : m| - 1 \;=\; \tilde{w}(\tilde{\pi}' : m) - 1 \;=\; \tilde{w}(\tilde{\pi}' : m : n)$$

□

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

$$\forall i' < j.\ \ell(n_{i'}) = \texttt{call} \;\wedge\; \tilde{w}(\langle n_{i'+1}, \ldots, n_j \rangle) = 1 \implies i' \le i$$

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:

$$n_i \rightleftarrows_{\tilde{\pi}} n_j \;\wedge\; n_{i'} \rightleftarrows_{\tilde{\pi}} n_j \implies i = i'$$
$$n_i \rightleftarrows_{\tilde{\pi}} n_j \;\wedge\; n_i \rightleftarrows_{\tilde{\pi}} n_{j'} \implies j = j'$$

Proof. For the first part, hypotheses $n_i \rightleftarrows_{\tilde{\pi}} n_j$ and $n_{i'} \rightleftarrows_{\tilde{\pi}} n_j$ imply:

$$\tilde{w}(\langle n_{i+1}, \ldots, n_j \rangle) = 1 = \tilde{w}(\langle n_{i'+1}, \ldots, n_j \rangle) \implies i \le i' \wedge i' \le i \implies i = i'$$

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

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

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

$$\sum_{y=j+1}^{j'-1} w(n_y, n_{y+1}) \;=\; 1 \tag{3.6}$$

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.:

$$|f(x+1) - f(x)| \;\leq\; 1 \tag{3.7}$$

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:

$$x^\star \;=\; \max\{\, x \in j+1..j' \mid f(x) = 1 \,\} \tag{3.8}$$

Observe that $x^\star$ is well-defined, as $f(j+1) = 1$ by (3.6). Besides, it must be $f(x^\star+1) = 0$. By condition (3.7), we know that $f(x^\star+1)$ can only assume three values, i.e. 0, 1 or 2. The second option is prevented, as it contradicts $x^\star$ 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 $\exists x \in x^\star+1..j'-1.\ f(x) = 1$. Again, this would violate definition (3.8). Now, by definition of f, we have:

$$f(x^\star) \;=\; f(x^\star+1) + w(n_{x^\star}, n_{x^\star+1})$$

Therefore $w(n_{x^\star}, n_{x^\star+1}) = 1$, and $n_{x^\star} \longrightarrow n_{x^\star+1}$ as only call edges have unit weight. Actually, we have proved that $\ell(n_{x^\star}) = \texttt{call}$ and $\tilde{w}(\langle n_{x^\star+1}, \ldots, n_{j'}\rangle) = 1$: by definition 3.6.11, this is in contradiction with hypothesis $n_i \rightleftarrows_{\tilde{\pi}} n'_j$, because $x^\star > j > i$. □

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

$$\tilde{\chi}(\tilde{\pi}) = \sigma : n_i : n_j \quad\Longrightarrow\quad n_i \rightleftarrows_{\tilde{\pi}} n_j$$

Proof. We prove the stronger result:

$$\begin{aligned}
\tilde{\chi}(\tilde{\pi}) = \sigma : n_i : n_j \quad &\Longrightarrow \quad n_i \longrightarrow n_{i+1} && \text{(3.9a)}\\
\wedge \quad &\tilde{w}(\langle n_{i+1}, \ldots, n_j\rangle) = 1 && \text{(3.9b)}\\
\wedge \quad &\forall i' > i.\ \ell(n_{i'}) = \texttt{call} \implies \tilde{w}(\langle n_{i'+1}, \ldots, n_j\rangle) < 1 && \text{(3.9c)}
\end{aligned}$$

By definition 3.6.11 and hypothesis $\ell(n_j) = \texttt{return}$, this clearly implies $n_i \rightleftarrows_{\tilde{\pi}} n_j$. The proof of the stronger statement is carried out by mathematical induction on the length of the concrete path. For the base case, if $\sigma = []$, $n_i \in N_{entry}$, $\ell(n_i) = \texttt{call}$ and $n_i \longrightarrow n_j$, then $\tilde{w}(\langle n_{i+1}\rangle) = 1$, 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 $\tilde{\chi}(\tilde{\pi}) = \sigma : n_i : n_j$, yielding:

- case [*call*]:

$$\frac{\tilde{\chi}(\langle n_0, \ldots, n_{j-1}\rangle) = \sigma : n_{j-1} \quad n_{j-1} \longrightarrow n_j}{\tilde{\chi}(\langle n_0, \ldots, n_j\rangle) = \sigma : n_{j-1} : n_j}$$

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

- case [*check*]:

$$\frac{\tilde{\chi}(\langle n_0, \ldots, n_{j-1}\rangle) = \sigma : n_i : n_{j-1} \quad n_{j-1} \dashrightarrow n_j}{\tilde{\chi}(\langle n_0, \ldots, n_j\rangle) = \sigma : n_i : n_j}$$

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

$$n_i \longrightarrow n_{i+1} \tag{3.10a}$$
$$\tilde{w}(\langle n_{i+1}, \ldots, n_{j-1}\rangle) = 1 \tag{3.10b}$$
$$\forall i' > i.\ \ell(n_{i'}) = \texttt{call} \implies \tilde{w}(\langle n_{i'+1}, \ldots, n_{j-1}\rangle) < 1 \tag{3.10c}$$

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

$$\begin{aligned}
\tilde{w}(\langle n_{i+1}, \ldots, n_j\rangle) &= \tilde{w}(\langle n_{i+1}, \ldots, n_{j-1}\rangle) + w(n_{j-1}, n_j) && \text{by def. } \tilde{w}\\
&= \tilde{w}(\langle n_{i+1}, \ldots, n_{j-1}\rangle) && \text{as } n_{j-1} \dashrightarrow n_j\\
&= 1 && \text{by (3.10b)}
\end{aligned}$$

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

$$\begin{aligned}
\tilde{w}(\langle n_{i'+1}, \ldots, n_j\rangle) &= \tilde{w}(\langle n_{i'+1}, \ldots, n_{j-1}\rangle) + w(n_{j-1}, n_j) && \text{by def. } \tilde{w}\\
&= \tilde{w}(\langle n_{i'+1}, \ldots, n_{j-1}\rangle) && \text{as } n_{j-1} \dashrightarrow n_j\\
&< 1 && \text{by (3.10c)}
\end{aligned}$$

- case [*return*]:

$$\frac{\tilde{\chi}(\langle n_0, \ldots, n_{j-1}\rangle) = \sigma : n_i : m : n_{j-1} \quad n_{j-1} \hookrightarrow n_j}{\tilde{\chi}(\langle n_0, \ldots, n_j\rangle) = \sigma : n_i : n_j}$$

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 $\tilde{\chi}(\langle n_0, \ldots, n_{j-1}\rangle) = \sigma' : n_h : n_{j-1}$ (where $\sigma' = \sigma : n_i$), we have:

$$n_h \longrightarrow n_{h+1} \tag{3.11a}$$
$$\tilde{w}(\langle n_{h+1}, \ldots, n_{j-1}\rangle) = 1 \tag{3.11b}$$
$$\forall i' > h.\ \ell(n_{i'}) = \texttt{call} \implies \tilde{w}(\langle n_{i'+1}, \ldots, n_{j-1}\rangle) < 1 \tag{3.11c}$$

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

$$n_i \longrightarrow n_{i+1} \tag{3.12a}$$
$$\tilde{w}(\langle n_{i+1}, \ldots, n_h\rangle) = 1 \tag{3.12b}$$
$$\forall i' > i.\ \ell(n_{i'}) = \texttt{call} \implies \tilde{w}(\langle n_{i'+1}, \ldots, n_h\rangle) < 1 \tag{3.12c}$$

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

$$\begin{aligned}
\tilde{w}(\langle n_{i+1}, \ldots, n_j\rangle) &= \tilde{w}(\langle n_{i+1}, \ldots, n_{j-1}\rangle) + w(n_{j-1}, n_j) && \text{by def. } \tilde{w}\\
&= \tilde{w}(\langle n_{i+1}, \ldots, n_{j-1}\rangle) - 1 && \text{as } n_{j-1} \hookrightarrow n_j\\
&= \tilde{w}(\langle n_{i+1}, \ldots, n_h\rangle) + \tilde{w}(\langle n_{h+1}, \ldots, n_{j-1}\rangle)\\
&\quad + w(n_h, n_{h+1}) - 2 && \text{by lemma 3.6.4}\\
&= \tilde{w}(\langle n_{i+1}, \ldots, n_h\rangle) + \tilde{w}(\langle n_{h+1}, \ldots, n_{j-1}\rangle) - 1 && \text{by (3.11a)}\\
&= \tilde{w}(\langle n_{i+1}, \ldots, n_h\rangle) + 1 - 1 && \text{by (3.11b)}\\
&= 1 + 1 - 1 && \text{by (3.12b)}\\
&= 1
\end{aligned}$$

To prove (3.9c), let $i' > i$ such that $\ell(n_{i'}) = \texttt{call}$. If $i' < h$, then:

$$\begin{aligned}
\tilde{w}(\langle n_{i'+1}, \ldots, n_j \rangle) &= \tilde{w}(\langle n_{i'+1}, \ldots, n_{j-1} \rangle) + w(n_{j-1}, n_j) && \text{by def. } \tilde{w} \\
&= \tilde{w}(\langle n_{i'+1}, \ldots, n_{j-1} \rangle) - 1 && \text{as } n_{j-1} \hookrightarrow n_j \\
&= \tilde{w}(\langle n_{i'+1}, \ldots, n_h \rangle) + \tilde{w}(\langle n_{h+1}, \ldots, n_{j-1} \rangle) \\
&\quad + w(n_h, n_{h+1}) - 2 && \text{by lemma 3.6.4} \\
&= \tilde{w}(\langle n_{i'+1}, \ldots, n_h \rangle) + \tilde{w}(\langle n_{h+1}, \ldots, n_{j-1} \rangle) - 1 && \text{by (3.11a)} \\
&= \tilde{w}(\langle n_{i'+1}, \ldots, n_h \rangle) + 1 - 1 && \text{by (3.11b)} \\
&< 1 + 1 - 1 && \text{by (3.12c)} \\
&= 1
\end{aligned}$$

Otherwise, if $i' > h$, we obtain:

$$\begin{aligned}
\tilde{w}(\langle n_{i'+1}, \ldots, n_j \rangle) &= \tilde{w}(\langle n_{i'+1}, \ldots, n_{j-1} \rangle) + w(n_{j-1}, n_j) && \text{by def. } \tilde{w} \\
&= \tilde{w}(\langle n_{i'+1}, \ldots, n_{j-1} \rangle) - 1 && \text{as } n_{j-1} \hookrightarrow n_j \\
&< 1 - 1 && \text{by (3.11c)} \\
&< 1
\end{aligned}$$

□

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

$$\frac{k = 0}{\tilde{\pi} \rightarrow_\alpha \langle n_0 \rangle}$$

$$\frac{\langle n_0, \ldots, n_{k-1} \rangle \rightarrow_\alpha \pi \quad (n_{k-1}, n_k) \in E}{\tilde{\pi} \rightarrow_\alpha \pi : n_k}$$

$$\frac{\langle n_0, \ldots, n_{i^\star} \rangle \rightarrow_\alpha \pi \quad n_{i^\star} \rightleftarrows_{\tilde{\pi}} n_{k-1} \quad n_{i^\star} \dashrightarrow n_k}{\tilde{\pi} \rightarrow_\alpha \pi : n_k}$$

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:

$$\forall i \in 0..h.\ m_i = n_{\varphi(i)} \quad \wedge \quad \varphi(h) = k \tag{3.13}$$

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 α 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:

- cases [*call*, *transfer*]:

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

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

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

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

- case [*return*]:

$$\frac{\langle n_0, \ldots, n_{i^*}\rangle \rightarrow_\alpha \pi \quad n_{i^*} \rightleftarrows_{\tilde{\pi}} n_{k-1} \quad n_{i^*} \dashrightarrow n_k}{\langle n_0, \ldots, n_k\rangle \rightarrow_\alpha \pi : n_k}$$

  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, φ is strictly increasing on $0..|\pi|-1$, because, inside that interval, it coincides with φ'. 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, φ is strictly increasing inside the whole domain $0..|\pi|$. Since we also have $n_k = n_{\varphi(|\pi|)}$, we can conclude that φ satisfies (3.13).

□

Lemma 3.6.16. The evaluation of α is deterministic, i.e.:

$$\tilde{\pi} \rightarrow_\alpha \pi \ \wedge \ \tilde{\pi} \rightarrow_\alpha \pi' \quad \Longrightarrow \quad \pi = \pi'$$

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

Proof. The proof is carried out by mathematical induction on the length of the concrete path $\tilde{\pi}$. For the base case, let $\tilde{\pi} = \langle n_0\rangle$. If $\tilde{\pi} \rightarrow_\alpha \pi$ and $\tilde{\pi} \rightarrow_\alpha \pi'$, then it must be $\pi = \pi' = \langle n_0\rangle$, because only the first rule of α is applicable. For the inductive case, let $\tilde{\pi} = \langle n_0, \ldots, n_k\rangle$. We proceed by case analysis on the last edge of the concrete path:

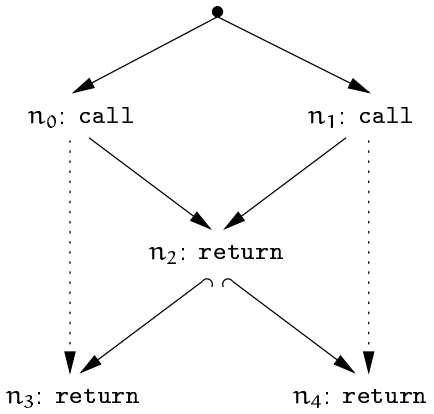

Figure 3.4: Control flow graph for counterexample 3.6.17

- cases [*call*, *transfer*]: consider the two transitions:

$$\frac{\langle n_0, \ldots, n_{k-1}\rangle \rightarrow_\alpha \pi \quad (n', n) \in E}{\langle n_0, \ldots, n_k\rangle \rightarrow_\alpha \pi : n_k} \qquad \frac{\langle n_0, \ldots, n_{k-1}\rangle \rightarrow_\alpha \pi' \quad (n', n) \in E}{\langle n_0, \ldots, n_k\rangle \rightarrow_\alpha \pi' : n_k}$$

  By the inductive hypothesis, we have $\pi = \pi'$, then $\pi : n_k = \pi' : n_k$ holds, too.

- case [*return*]: consider the two transitions:

$$\frac{\langle n_0, \ldots, n_{i^*}\rangle \rightarrow_\alpha \pi \quad n_{i^*} \rightleftarrows_{\tilde{\pi}} n_{k-1} \quad n_{i^*} \dashrightarrow n_k}{\langle n_0, \ldots, n_k\rangle \rightarrow_\alpha \pi : n_k}$$

$$\frac{\langle n_0, \ldots, n_{i^*}\rangle \rightarrow_\alpha \pi' \quad n_{i^*} \rightleftarrows_{\tilde{\pi}} n_{k-1} \quad n_{i^*} \dashrightarrow n_k}{\langle n_0, \ldots, n_k\rangle \rightarrow_\alpha \pi' : n_k}$$

  By the inductive hypothesis, we have $\pi = \pi'$, which clearly implies $\pi : n_k = \pi' : n_k$.

□

Counterexample 3.6.17. The evaluation of α 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 $\tilde{\pi} = \langle n_0, n_2, n_4\rangle$, then $\alpha(\langle n_0, n_2\rangle)$ is defined, but $\alpha(\tilde{\pi})$ is not, because $n_0 \rightleftarrows_{\tilde{\pi}} n_2$ but $n_0 \not\dashrightarrow n_4$. □

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

$$\alpha(\tilde{\pi}) \quad = \quad \pi \qquad\qquad \text{if } \tilde{\pi} \rightarrow_\alpha \pi$$

Lemma 3.6.18. Let $\tilde{\pi}$ be a concrete path, such that $\alpha(\tilde{\pi})$ is defined. Then:

$$\tilde{\pi} \in \widetilde{\Pi}_{n_0,n_k} \quad \Longrightarrow \quad \alpha(\tilde{\pi}) \in \Pi_{n_0,n_k}$$

for any $n_0, n_k \in \mathsf{N}$.

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

- cases [*call*, *transfer*]:

$$\frac{\alpha(\langle n_0, \ldots, n_{k-1} \rangle) = \pi \quad (n_{k-1}, n_k) \in \mathsf{E}}{\alpha(\langle n_0, \ldots, n_k \rangle) = \pi : n_k}$$

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

- case [*return*]:

$$\frac{\alpha(\langle n_0, \ldots, n_{i^*} \rangle) = \pi \quad n_{i^*} \rightleftarrows_{\tilde{\pi}} n_{k-1} \quad n_{i^*} \dashrightarrow n_k}{\alpha(\langle n_0, \ldots, n_k \rangle) = \pi : n_k}$$

  By the inductive hypothesis, we have $\pi = \alpha(\langle n_0, \ldots, n_{i^*} \rangle) \in \Pi_{n_0,n_{i^*}}$. Therefore, by premise $n_{i^*} \dashrightarrow n_k$, it follows $\pi : n_k \in \Pi_{n_0,n_k}$.

□

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

$$\tilde{w}(\tilde{\pi}) \quad = \quad w(\alpha(\tilde{\pi}))$$

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

- cases [*call*, *transfer*]:

$$\frac{\alpha(\langle n_0, \ldots, n_{k-1} \rangle) = \pi \quad (n_{k-1}, n_k) \in \mathsf{E}}{\alpha(\langle n_0, \ldots, n_k \rangle) = \pi : n_k}$$

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

$$\begin{aligned} \tilde{w}(\langle n_0, \ldots, n_k \rangle) &= \tilde{w}(\langle n_0, \ldots, n_{k-1} \rangle) + w(n_{k-1}, n_k) && \text{by def. } \tilde{w} \\ &= w(\pi) + w(n_{k-1}, n_k) && \text{by ind. hyp.} \\ &= w(\pi : n_k) && \text{by def. } w \end{aligned}$$

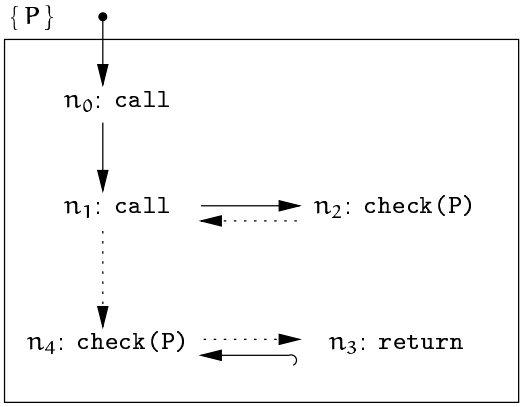

Figure 3.5: Control flow graph for counterexample 3.6.20

- case [*return*]:

$$\frac{\alpha(\langle n_0, \ldots, n_{i^*}\rangle) = \pi \quad n_{i^*} \rightleftarrows_{\tilde{\pi}} n_{k-1} \quad n_{i^*} \dashrightarrow n_k}{\alpha(\langle n_0, \ldots, n_k\rangle) = \pi : n_k}$$

By the inductive hypothesis, we have $\tilde{w}(\langle n_0, \ldots, n_{i^*}\rangle) = w(\pi)$. Moreover, by lemma 3.6.18, $\langle n_0, \ldots, n_{i^*}\rangle \in \widetilde{\Pi}_{n_0, n_{i^*}}$ implies $\pi \in \Pi_{n_0, n_{i^*}}$. Then:

$$\begin{aligned}
\tilde{w}(\langle n_0, \ldots, n_k\rangle) &= \tilde{w}(\langle n_0, \ldots, n_{k-1}\rangle) + w(n_{k-1}, n_k) && \text{by def. } \tilde{w} \\
&= \tilde{w}(\langle n_0, \ldots, n_{k-1}\rangle) - 1 && \text{as } n_{k-1} \hookrightarrow n_k \\
&= \tilde{w}(\langle n_0, \ldots, n_{i^*}\rangle) + \tilde{w}(\langle n_{i^*+1}, \ldots, n_{k-1}\rangle) \\
&\quad + w(n_{i^*}, n_{i^*+1}) - 2 && \text{by lemma 3.6.4} \\
&= \tilde{w}(\langle n_0, \ldots, n_{i^*}\rangle) + w(n_{i^*}, n_{i^*+1}) - 1 && \text{as } n_{i^*} \rightleftarrows_{\tilde{\pi}} n_{k-1} \\
&= \tilde{w}(\langle n_0, \ldots, n_{i^*}\rangle) && \text{as } n_{i^*} \longrightarrow n_{i^*+1} \\
&= w(\pi) && \text{by ind. hyp.} \\
&= w(\pi) + w(n_{i^*}, n_k) && \text{as } n_{i^*} \dashrightarrow n_k \\
&= w(\pi : n_k) && \text{by def. } w
\end{aligned}$$

□

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

$$\neg\exists \tilde{\pi} \in \widetilde{\Pi}.\ \pi = \alpha(\tilde{\pi})$$

Proof. To see why, consider the control flow graph in Fig. 3.5: the return edge $n_3 \hookrightarrow n_4$ follows by the fact that $w(\langle n_2, n_1, n_4, n_3\rangle) = 1$ implies $n_3 \in \rho(n_1, n_2)$, and $n_1 \dashrightarrow n_4$. Now, any concrete (non-empty) path $\tilde{\pi}$ leaving from n_1 is of the form:

$$n_1 : \langle n_2, n_1\rangle^* \quad | \quad \langle n_1, n_2\rangle^+$$

Therefore, the abstract path $\pi = \langle n_1, n_4\rangle$ does not belong to the image of α, because, by lemma 3.6.18, for this to be possible it should be $\tilde{\pi} \in \widetilde{\Pi}_{n_1, n_4}$. □

3.7 Valid paths

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

$$n_j \hookrightarrow n_{j+1} \implies \exists i \in 0..k.\ n_i \rightleftarrows_{\tilde{\pi}} n_j \ \wedge\ n_i \dashrightarrow n_{j+1}$$

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

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

Proof. This result clearly follows by the fact that $n_i \rightleftarrows_{\tilde{\pi}} n_j$ implies $i < j$. □

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

Proof. Let $\tilde{\pi}' = \langle n_i, \ldots, n_j \rangle$, and choose an index $j' \in i..j-1$ such that $n_{j'} \hookrightarrow n_{j'+1}$. If $j' = j$ or no return node exists between i and j, we are done. Otherwise, for $x \in i+1..j$, 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 $n_i \rightleftarrows_{\tilde{\pi}} n_j$. Similarly to the proof of lemma 3.6.12, $g(x)$ is a step function which ensures the existence of intermediate values. Now define:

$$G = \{ x \in i+1..j \mid g(x) < 0 \}$$

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

$$\sum_{y=i}^{j'-1} w(n_y, n_{y+1}) = 1 + g(j') > 0 \tag{3.14}$$

Now, for $x \in i..j'$, 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:

$$F \;=\; \{\, x \in i..j' \mid f(x) = 1 \,\}$$

By existence of intermediate values, we have that $\exists x \in i+1..j'-1.\ f(x) = 1$, 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 $\exists x \in i'+1..j'-1.\ f(x) = 1$. 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, $n_{i'} \longrightarrow n_{i'+1}$. Actually, we have proved that $\ell(n_{i'}) = \texttt{call}$ and $\tilde{w}(\langle n_{i'+1}, \ldots, n_{j'}\rangle) = 1$. Moreover, we have that $\tilde{w}(\langle n_{i''+1}, \ldots, n_{j'}\rangle) \neq 1$ for any $i'' > i'$. By definition 3.6.11, this indeed suffices to state $n_{i'} \rightleftarrows_{\tilde{\pi}'} n_{j'}$. On the other hand, by definition 3.7.1, $n_{i'} \dashrightarrow n_{j'+1}$ immediately follows by the fact that $\tilde{\pi}$ is valid and hypothesis $n_{j'} \hookrightarrow n_{j'+1}$. Then, also $\tilde{\pi}'$ is valid. □

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:

$$n_i \rightleftarrows_{\tilde{\pi}} n_j \quad\Longrightarrow\quad \tilde{\chi}(\tilde{\pi}) = \sigma : n_i : n_j$$

Proof. Assume $n_i \rightleftarrows_{\tilde{\pi}} n_j$. We prove the stronger statement:

$$\forall i' > i.\ \exists \tau \in \Sigma.\ \tilde{\chi}(\langle n_0, \ldots, n_{i'}\rangle) = \sigma : n_i : \tau : n_{i'} \;\wedge\; |\tau| = \tilde{w}(\langle n_{i+1}, \ldots, n_{i'}\rangle) - 1 \qquad (3.15)$$

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

$$\tilde{\chi}(\langle n_0, \ldots, n_j\rangle) = \sigma : n_i : \tau : n_j \;\wedge\; |\tau| = \tilde{w}(\langle n_{i+1}, \ldots, n_j\rangle) - 1$$

By definition 3.6.11, $n_i \rightleftarrows_{\tilde{\pi}} n_j$ implies $\tilde{w}(\langle n_{i+1}, \ldots, n_j\rangle) = 1$, hence $|\tau| = 0$. Actually, this can only happen if $\tau = []$: therefore, $\tilde{\chi}(\langle n_0, \ldots, n_j\rangle) = \sigma : n_i : n_j$. 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:

$$\frac{\tilde{\chi}(\langle n_0, \ldots, n_i\rangle) = \sigma : n_i \quad n_i \longrightarrow n_{i+1}}{\tilde{\chi}(\langle n_0, \ldots, n_{i+1}\rangle) = \sigma : n_i : n_{i+1}}$$

If we define $\tau = []$ we are done, because $|[]| = 0 = \tilde{w}(\langle n_{i+1}\rangle) - 1$. For the inductive case, we assume the statement is true for an arbitrary $i' \in i+1..j-1$; 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:

- case [*call*]: by the inductive hypothesis, we have $\tilde{\chi}(\langle n_0, \ldots, n_{i'} \rangle) = \sigma : n_i : \tau' : n_{i'}$, with $|\tau'| = \tilde{w}(\langle n_{i+1}, \ldots, n_{i'} \rangle) - 1$. Hence, the following transition occurs:

$$\frac{\tilde{\chi}(\langle n_0, \ldots, n_{i'} \rangle) = \sigma : n_i : \tau' : n_{i'} \quad n_{i'} \longrightarrow n_{i'+1}}{\tilde{\chi}(\langle n_0, \ldots, n_{i'+1} \rangle) = \sigma : n_i : \tau' : n_{i'} : n_{i'+1}}$$

  If we define $\tau = \tau' : n_{i'}$, then $\tilde{\chi}(\langle n_0, \ldots, n_{i'+1} \rangle) = \sigma : n_i : \tau : n_{i'+1}$, and:

$$|\tau| = |\tau'| + 1 = \tilde{w}(\langle n_{i+1}, \ldots, n_{i'} \rangle) - 1 + 1 = \tilde{w}(\langle n_{i+1}, \ldots, n_{i'+1} \rangle) - 1$$

- case [*check*]: by the inductive hypothesis, we have $\tilde{\chi}(\langle n_0, \ldots, n_{i'} \rangle) = \sigma : n_i : \tau' : n_{i'}$, with $|\tau'| = \tilde{w}(\langle n_{i+1}, \ldots, n_{i'} \rangle) - 1$. Hence, the following transition occurs:

$$\frac{\tilde{\chi}(\langle n_0, \ldots, n_{i'} \rangle) = \sigma : n_i : \tau' : n_{i'} \quad n_{i'} \dashrightarrow n_{i'+1}}{\tilde{\chi}(\langle n_0, \ldots, n_{i'+1} \rangle) = \sigma : n_i : \tau' : n_{i'+1}}$$

  If we define $\tau = \tau'$, then $\tilde{\chi}(\langle n_0, \ldots, n_{i'+1} \rangle) = \sigma : n_i : \tau : n_{i'+1}$, and:

$$|\tau| = |\tau'| = \tilde{w}(\langle n_{i+1}, \ldots, n_{i'} \rangle) - 1 = \tilde{w}(\langle n_{i+1}, \ldots, n_{i'+1} \rangle) - 1$$

- case [*return*]: by the inductive hypothesis, $\tilde{\chi}(\langle n_0, \ldots, n_{i'} \rangle) = \sigma : n_i : \tau' : n_{i'}$, with $|\tau'| = \tilde{w}(\langle n_{i+1}, \ldots, n_{i'} \rangle) - 1$. Now, since $\tilde{\pi}$ is a valid path and $n_i \rightleftarrows_{\tilde{\pi}} n_j$, by the proof of lemma 3.7.3 it turns out that $\tilde{w}(\langle n_{i+1}, \ldots, n_{i'+1} \rangle) \geq 1$. Since $n_{i'} \hookrightarrow n_{i'+1}$, we then have $\tilde{w}(\langle n_{i+1}, \ldots, n_{i'} \rangle) > 1$. Therefore $|\tau'| > 0$, so we can write $\tau' = \tau'' : n_{i''}$ for some $\tau'' \in \Sigma$ and $i'' \in i + 1..i' - 1$. The following transition occurs:

$$\frac{\tilde{\chi}(\langle n_0, \ldots, n_{i'} \rangle) = \sigma : n_i : \tau'' : n_{i''} : n_{i'} \quad n_{i'} \hookrightarrow n_{i'+1}}{\tilde{\chi}(\langle n_0, \ldots, n_{i'+1} \rangle) = \sigma : n_i : \tau'' : n_{i'+1}}$$

  If we define $\tau = \tau''$, then $\tilde{\chi}(\langle n_0, \ldots, n_{i'+1} \rangle) = \sigma : n_i : \tau : n_{i'+1}$, and:

$$|\tau| = |\tau''| = |\tau'| - 1 = \tilde{w}(\langle n_{i+1}, \ldots, n_{i'} \rangle) - 1 - 1 = \tilde{w}(\langle n_{i+1}, \ldots, n_{i'+1} \rangle) - 1$$

□

Lemma 3.7.5. The evaluation of $\tilde{\chi}$ is total on valid entry paths, i.e.:

$$\tilde{\pi} \in \widetilde{\Pi}^{v}_{entry} \quad \Longrightarrow \quad \exists \sigma \in \Sigma.\ \tilde{\pi} \rightarrow_{\tilde{\chi}} \sigma$$

Proof. We prove the stronger result:

$$\tilde{\pi} \in \widetilde{\Pi}^{v}_{n} \quad \Longrightarrow \quad \exists \sigma \in \Sigma.\ \tilde{\pi} \rightarrow_{\tilde{\chi}} \sigma : n$$

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

$$\tilde{\pi}' : n' \in \widetilde{\Pi}_{n'} \quad \Longrightarrow \quad \exists \sigma' \in \Sigma.\ \tilde{\pi}' : n' \rightarrow_{\tilde{\chi}} \sigma' : n'$$

Now we proceed by case analysis on the edge (n', n):

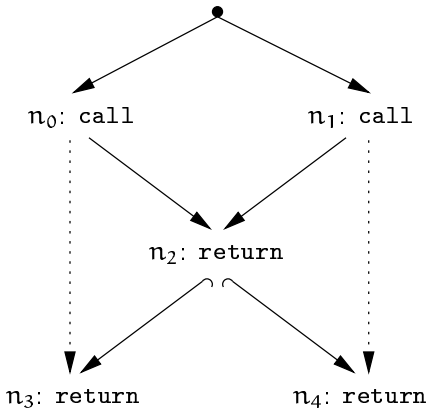

Figure 3.6: Control flow graph for counterexample 3.7.6

- case [*call*]: if $n' \longrightarrow n$, then the $\tilde{\chi}_{call}$ rule is applicable, and we obtain:

$$\frac{\tilde{\pi}' : n' \rightarrow_{\tilde{\chi}} \sigma' : n' \quad n' \longrightarrow n}{\tilde{\pi}' : n' : n \rightarrow_{\tilde{\chi}} \sigma' : n' : n}$$

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

- case [*transfer*]: if $n' \dashrightarrow n$, we can apply the $\tilde{\chi}_{trans}$ rule, yielding:

$$\frac{\tilde{\pi}' : n' \rightarrow_{\tilde{\chi}} \sigma' : n' \quad n' \dashrightarrow n}{\tilde{\pi}' : n' : n \rightarrow_{\tilde{\chi}} \sigma' : n}$$

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

- case [*return*]: if $n' \hookrightarrow n$, write $\tilde{\pi} = \langle n_0, \ldots, n_j \rangle$, with $n' = n_j$ and $n = n_{j+1}$ for some $j \geq 0$. By the inductive hypothesis, we have that:

$$\langle n_0, \ldots, n_j \rangle \rightarrow_{\tilde{\chi}} \sigma' : n_j$$

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

$$\langle n_0, \ldots, n_i \rangle \rightarrow_{\tilde{\chi}} \sigma'' : n_i$$

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

$$\frac{\tilde{\pi}' : n' \rightarrow_{\tilde{\chi}} \sigma'' : n_i : n' \quad n' \hookrightarrow n}{\tilde{\pi}' : n' : n \rightarrow_{\tilde{\chi}} \sigma'' : n}$$

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

□

Counterexample 3.7.6. Validity is not necessary for $\tilde{\chi}$ to be defined.

Proof. To see why, consider the control flow graph in Fig. 3.6, and take the concrete path $\tilde{\pi} = \langle n_0, n_2, n_4 \rangle$. Now, this path is *not* valid, because, with regard to the return edge $n_2 \hookrightarrow n_4$, we have $n_0 \rightleftarrows_{\tilde{\pi}} n_2$ but $n_0 \not\dashrightarrow n_4$. However, $\tilde{\chi}(\tilde{\pi})$ is defined, because $\langle n_0, n_2 \rangle \rightarrow_{\tilde{\chi}} [n_0, n_2]$ by $\tilde{\chi}_{entry}$ and $\tilde{\chi}_{call}$, and $\langle n_0, n_2, n_4 \rangle \rightarrow_{\tilde{\chi}} [n_4]$ by $\tilde{\chi}_{return}$. □

Theorem 3.7.7.

$$\mathsf{G} \vdash \sigma : n \quad \Longrightarrow \quad \exists \tilde{\pi} \in \widetilde{\Pi}^{\upsilon}_{n}.\ \tilde{\chi}(\tilde{\pi}) = \sigma : n$$

Proof. The proof is carried out by induction on the derivation used to estabilish $\mathsf{G} \vdash \sigma : n$. For the base case, if $\sigma = []$ and $n \in \mathsf{N}_{entry}$, then $\langle n \rangle \in \widetilde{\Pi}^{\upsilon}_{n}$ and $\tilde{\chi}(\langle n \rangle) = [n]$. For the inductive case, we proceed by case analysis on the last rule used to derive $\mathsf{G} \vdash \sigma : n$, yielding:

- case [*call*]:

$$\frac{\ell(n') = \texttt{call} \quad n' \longrightarrow n}{\sigma' : n' \ \rhd\ \sigma' : n' : n} \qquad \text{where } \sigma = \sigma' : n'$$

  By the inductive hypothesis applied to $\sigma' : n'$, we have:

$$\exists \tilde{\pi} \in \widetilde{\Pi}^{\upsilon}_{n'}.\ \tilde{\chi}(\tilde{\pi}) = \sigma' : n'$$

  Since $\ell(n') = \texttt{call}$ and $n' \longrightarrow n$, the path $\tilde{\pi} : n$ is in $\widetilde{\Pi}^{\upsilon}_{n}$, and:

$$\frac{\tilde{\chi}(\tilde{\pi}) = \sigma' : n' \quad n' \longrightarrow n}{\tilde{\chi}(\tilde{\pi} : n) = \sigma' : n' : n}$$

- case [*check*]:

$$\frac{\ell(n') = \texttt{check}(\mathsf{P}) \quad \sigma : n' \vdash \mathsf{JDK}(\mathsf{P}) \quad n' \dashrightarrow n}{\sigma : n' \ \rhd\ \sigma : n}$$

  By the inductive hypothesis applied to $\sigma : n'$, we have:

$$\exists \tilde{\pi} \in \widetilde{\Pi}^{\upsilon}_{n'}.\ \tilde{\chi}(\tilde{\pi}) = \sigma : n'$$

  Here we have $\ell(n') = \texttt{check}(\mathsf{P})$ and $n' \dashrightarrow n$, hence $\tilde{\pi} : n \in \widetilde{\Pi}^{\upsilon}_{n}$, and:

$$\frac{\tilde{\chi}(\tilde{\pi}) = \sigma : n' \quad n' \dashrightarrow n}{\tilde{\chi}(\tilde{\pi} : n) = \sigma : n}$$

- case [*return*]:

$$\frac{\ell(m) = \texttt{return} \quad n' \dashrightarrow n}{\sigma : n' : m \ \rhd\ \sigma : n}$$

  By the inductive hypothesis applied to $\sigma : n' : m$, we have:

$$\exists \tilde{\pi} \in \widetilde{\Pi}^{\upsilon}_{m}.\ \tilde{\chi}(\tilde{\pi}) = \sigma : n' : m$$

  By lemma 3.5.16, we have $m \in \rho(n')$, hence $n' \dashrightarrow n$ implies that $m \hookrightarrow n$ is a return edge. Now, lemma 3.6.13 ensures $n' \rightleftarrows_{\tilde{\pi}} m$, then the path $\tilde{\pi} : n$ is still valid. Therefore, $\tilde{\pi} : n \in \widetilde{\Pi}^{\upsilon}_{n}$, and:

$$\frac{\tilde{\chi}(\tilde{\pi}) = \sigma : n' : m \quad m \hookrightarrow n}{\tilde{\chi}(\tilde{\pi} : n) = \sigma : n}$$

□

Lemma 3.7.8. The evaluation of α is total on valid paths, i.e.:

$$\tilde{\pi} \in \widetilde{\Pi}^{v} \quad \Longrightarrow \quad \exists \pi \in \Pi.\ \tilde{\pi} \rightarrow_{\alpha} \pi$$

Proof. The proof is carried out by mathematical induction on the length of the valid path. For the base case, if $\tilde{\pi} = \langle n \rangle \in \widetilde{\Pi}^{v}$, then $\tilde{\pi} \rightarrow_{\alpha} \langle n \rangle$ by the first rule of α. For the inductive case, let $\tilde{\pi} = \langle n_0, \ldots, n_k \rangle$. 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 $\pi \in \Pi_{n_0, n_{k-1}}$. Then, the following transition occurs:

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

- case [*return*]: assume $n_{k-1} \hookrightarrow n_k$. Since $\tilde{\pi}$ is a valid path, it must be $n_{i^*} \rightleftarrows_{\tilde{\pi}} n_{k-1}$ and $n_{i^*} \dashrightarrow n_k$ for some $i^* < k - 1$. 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 π such that $\langle n_0, \ldots, n_{i^*} \rangle \rightarrow_{\alpha} \pi$. Therefore, the following transition occurs:

$$\frac{\alpha(\langle n_0, \ldots, n_{i^*} \rangle) = \pi \quad n_{i^*} \rightleftarrows_{\tilde{\pi}} n_{k-1} \quad n_{i^*} \dashrightarrow n_k}{\alpha(\langle n_0, \ldots, n_k \rangle) = \pi : n_k}$$

□

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

$$\tilde{\chi}(\tilde{\pi}) \quad = \quad \chi(\alpha(\tilde{\pi}))$$

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

- 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 $\tilde{\chi}(\langle n_0, \ldots, n_{k-1} \rangle)$ and $\alpha(\langle n_0, \ldots, n_{k-1} \rangle)$ are defined. By definition of $\tilde{\chi}$ and α, we have:

$$\frac{\tilde{\chi}(\langle n_0, \ldots, n_{k-1} \rangle) = \sigma \quad n_{k-1} \longrightarrow n_k}{\tilde{\chi}(\langle n_0, \ldots, n_k \rangle) = \sigma : n_k}$$

$$\frac{\alpha(\langle n_0, \ldots, n_{k-1} \rangle) = \pi \quad n_{k-1} \longrightarrow n_k}{\alpha(\langle n_0, \ldots, n_k \rangle) = \pi : n_k}$$

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

$$\frac{\chi(\pi) = \sigma \quad n_{k-1} \longrightarrow n_k}{\chi(\pi : n_k) = \sigma : n_k}$$

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

$$\frac{\tilde{\chi}(\langle n_0, \ldots, n_{k-1} \rangle) = \sigma : n_{k-1} \quad n_{k-1} \dashrightarrow n_k}{\tilde{\chi}(\langle n_0, \ldots, n_k \rangle) = \sigma : n_k}$$

$$\frac{\alpha(\langle n_0, \ldots, n_{k-1} \rangle) = \pi \quad n_{k-1} \dashrightarrow n_k}{\alpha(\langle n_0, \ldots, n_k \rangle) = \pi : n_k}$$

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

$$\frac{\chi(\pi) = \sigma : n_{k-1} \quad n_{k-1} \dashrightarrow n_k}{\chi(\pi : n_k) = \sigma : n_k}$$

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

$$\frac{\tilde{\chi}(\langle n_0, \ldots, n_{k-1} \rangle) = \sigma : n_i : n_{k-1} \quad n_{k-1} \hookrightarrow n_k}{\tilde{\chi}(\langle n_0, \ldots, n_k \rangle) = \sigma : n_k}$$

$$\frac{\alpha(\langle n_0, \ldots, n_{i^*} \rangle) = \pi \quad n_{i^*} \rightleftarrows_{\tilde{\pi}} n_{k-1} \quad n_{i^*} \dashrightarrow n_k}{\alpha(\langle n_0, \ldots, n_k \rangle) = \pi : n_k}$$

  Actually, $\langle n_0, \ldots, n_{i^*} \rangle$ is a valid entry path, too: hence, by lemma 3.7.5, we have $\tilde{\chi}(\langle n_0, \ldots, n_{i^*} \rangle) = \sigma' : n_{i^*}$ for some state $\sigma' \in \Sigma$. Then, by hypothesis $n_{i^*} \rightleftarrows_{\tilde{\pi}} n_{k-1}$, lemma 3.7.4 ensures that:

$$\tilde{\chi}(\langle n_0, \ldots, n_{i^*}, \ldots, n_{k-1} \rangle) = \sigma' : n_{i^*} : n_{k-1}$$

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

$$\frac{\chi(\pi) = \sigma : n_{i^*} \quad n_{i^*} \dashrightarrow n_k}{\chi(\pi : n_k) = \sigma : n_k}$$

□

Lemma 3.7.10. Let $\tilde{\pi}$ be a valid path. Then:

$$n \rightleftarrows_{\tilde{\pi}} m \quad \Longrightarrow \quad m \in \rho(n)$$

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

$$w(\pi) \;=\; w(\pi^\star) - w(n_i, n_{i+1}) \;=\; \tilde{w}(\tilde{\pi}^\star) - 1 \;=\; 1$$

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

3.8 Traversable paths

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

$$\begin{array}{lcll} \eta([],[n]) & = & (\bot_{\mathsf{N}}, n) & \text{if } n \in \mathsf{N}_{entry} \\ \eta(\sigma : n, \sigma : n : n') & = & (n, n') & \text{if } \ell(n) = \texttt{call} \text{ and } n \longrightarrow n' \\ \eta(\sigma : n, \sigma : n') & = & (n, n') & \text{if } \ell(n) = \texttt{check}(\mathsf{P}) \text{ and } n \dashrightarrow n' \\ \eta(\sigma : n : m, \sigma : n') & = & (n, n') & \text{if } \ell(m) = \texttt{return} \text{ and } n \dashrightarrow n' \end{array}$$

In any other case, η is left undefined.

Lemma 3.8.2.

$$\sigma' \rhd \sigma : n \implies \exists (m, n) \in \mathsf{E}.\ \eta(\sigma', \sigma : n) = (m, n)$$

Proof. If $\sigma' = []$, $n \in \mathsf{N}_{entry}$ and $\sigma = [n]$, then $(\bot_{\mathsf{N}}, n) \in \mathsf{E}_{entry}$ and $\eta([],[n]) = (\bot_{\mathsf{N}}, n)$. Otherwise, we proceed by case analysis on the rule used in transition $\sigma' \rhd \sigma : n$, yielding:

- case [*call*]:
$$\frac{\ell(n') = \texttt{call} \quad n' \longrightarrow n}{\sigma' : n' \rhd \sigma' : n' : n} \quad \text{where } \sigma = \sigma' : n'$$
Here $\eta(\sigma' : n', \sigma' : n' : n) = (n', n) \in \mathsf{E}$.
- case [*check*]:
$$\frac{\ell(n') = \texttt{check}(\mathsf{P}) \quad \sigma : n' \vdash \mathsf{JDK}(\mathsf{P}) \quad n' \dashrightarrow n}{\sigma : n' \rhd \sigma : n}$$
Here $\eta(\sigma : n', \sigma : n) = (n', n) \in \mathsf{E}$.
- case [*return*]:
$$\frac{\ell(m) = \texttt{return} \quad n' \dashrightarrow n}{\sigma : n' : m \rhd \sigma : n}$$
Here $\eta(\sigma : n' : m, \sigma : n) = (n', n) \in \mathsf{E}$.

□

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

$$\begin{array}{lcll} \tilde{\eta}([],[n]) & = & (\bot_{\mathsf{N}}, n) & \text{if } n \in \mathsf{N}_{entry} \\ \tilde{\eta}(\sigma : n, \sigma : n : n') & = & (n, n') & \text{if } \ell(n) = \texttt{call} \text{ and } n \longrightarrow n' \\ \tilde{\eta}(\sigma : n, \sigma : n') & = & (n, n') & \text{if } \ell(n) = \texttt{check}(\mathsf{P}) \text{ and } n \dashrightarrow n' \\ \tilde{\eta}(\sigma : n : m, \sigma : n') & = & (m, n') & \text{if } \ell(m) = \texttt{return} \text{ and } m \hookrightarrow n' \end{array}$$

In any other case, $\tilde{\eta}$ is left undefined.

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

$$[] \rhd \sigma_0 \rhd \sigma_1 \rhd \cdots \rhd \sigma_k$$

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

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

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

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

$$\forall i \in 0..k-1.\ \tilde{\chi}(\langle n_0, \ldots, n_i \rangle) \rhd \tilde{\chi}(\langle n_0, \ldots, n_{i+1} \rangle) \tag{3.16}$$

Proof. For the *if* 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 $\sigma_i \rhd \sigma_{i+1}$ 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*]:
$$\frac{\langle n_0, \ldots, n_i \rangle \to_{\tilde{\chi}} \sigma : n_i \quad n_i \longrightarrow n_{i+1}}{\langle n_0, \ldots, n_{i+1} \rangle \to_{\tilde{\chi}} \sigma : n_i : n_{i+1}}$$
Here, $\ell(n_i) = \texttt{call}$ implies $\tilde{\eta}(\sigma : n_i, \sigma : n_i : n_{i+1}) = (n_i, n_{i+1})$.
- case [*transfer*]:
$$\frac{\langle n_0, \ldots, n_i \rangle \to_{\tilde{\chi}} \sigma : n_i \quad n_i \dashrightarrow n_{i+1}}{\langle n_0, \ldots, n_{i+1} \rangle \to_{\tilde{\chi}} \sigma : n_{i+1}}$$
Here, $\ell(n_i) = \texttt{check}(P)$ for some permission P, hence $\tilde{\eta}(\sigma : n_i, \sigma : n_{i+1}) = (n_i, n_{i+1})$.
- case [*return*]:
$$\frac{\langle n_0, \ldots, n_i \rangle \to_{\tilde{\chi}} \sigma : m : n_i \quad n_i \hookrightarrow n_{i+1}}{\langle n_0, \ldots, n_{i+1} \rangle \to_{\tilde{\chi}} \sigma : n_{i+1}}$$
Here, $\ell(n_i) = \texttt{return}$ implies $\tilde{\eta}(\sigma : m : n_i, \sigma : n_{i+1}) = (n_i, n_{i+1})$.

Moreover, we have $\tilde{\eta}([], \sigma_0) = (\bot_N, n_0)$ 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 $\tilde{\pi}$ is traversable.

For the *only if* part, assume $\sigma_0 \rhd \cdots \rhd \sigma_k$ is a derivation such that $\tilde{\eta}([], \sigma_0) = (\bot_N, n_0)$ 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 $\tilde{\pi}$, 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 $\tilde{\eta}([], \sigma_0) = (\bot_N, n_0)$ 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*]:

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

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

- case [*transfer*]:

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

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

- case [*return*]:

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

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

□

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 $n_j \hookrightarrow n_{j+1}$ for some $j < k$ (otherwise $\tilde{\pi}$ is clearly valid). Since $\tilde{\pi}$ is traversable, by lemma 3.8.5 it follows that:

$$\tilde{\chi}(\langle n_0, \ldots, n_j \rangle) \rhd \tilde{\chi}(\langle n_0, \ldots, n_{j+1} \rangle) \tag{3.17}$$

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

$$\frac{\tilde{\chi}(\langle n_0, \ldots, n_j \rangle) = \sigma : m : n_j \quad n_j \hookrightarrow n_{j+1}}{\tilde{\chi}(\langle n_0, \ldots, n_{j+1} \rangle) = \sigma : n_{j+1}}$$

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

$$\frac{\ell(n_j) = \texttt{return} \quad n_i \dashrightarrow n_{j+1}}{\sigma : n_i : n_j \rhd \sigma : n_{j+1}}$$

Therefore, we also have $n_i \dashrightarrow n_{j+1}$, which actually proves that $\tilde{\pi}$ is valid. □

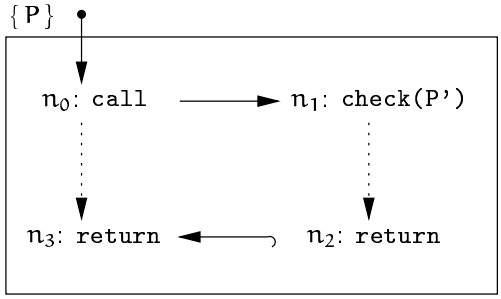

Figure 3.7: Control flow graph for counterexample 3.8.7

Counterexample 3.8.7. Not all valid paths are traversable.

Proof. Consider the control flow graph in Fig. 3.7. The concrete path $\tilde{\pi} = \langle n_0, n_1, n_2, n_3 \rangle$ is valid, because $n_0 \rightleftarrows_{\tilde{\pi}} n_2$ and $n_0 \dashrightarrow n_3$. However, $\tilde{\pi}$ is not traversable: in fact, no transition can occur on state $[n_0, n_1]$, because $[n_0, n_1] \not\models \mathsf{JDK}(\mathtt{P}')$. □

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* $\phi : \mathcal{L} \to \mathbf{Bool}$, 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 mantaining 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;
- μ is a mapping from control flow graphs to transfer functions.

[1] In what follows, we estabilish that $l_0 \sqsubseteq l_1$ means "l_0 is *more precise* than l_1".

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

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 estabilishes 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 $\mathsf{l}_0 \sqsubseteq \mathsf{l}_1 \sqsubseteq \cdots \sqsubseteq \mathsf{l}_k \sqsubseteq \cdots$ of elements of $\mathcal{L}$ *eventually stabilises*, that is:
$$\exists \mathsf{k}_0 \in \mathbb{N}.\ \forall \mathsf{k} \geq \mathsf{k}_0.\ \mathsf{l}_\mathsf{k} = \mathsf{l}_{\mathsf{k}_0}$$
Moreover, we require $\mathcal{L}$ to be endowed with a bottom element $\bot_\mathcal{L}$.

- $\mathcal{F}$ is a family of *monotone* functions, that is:
$$\mathsf{l} \sqsubseteq \mathsf{l}' \quad \Longrightarrow \quad f(\mathsf{l}) \sqsubseteq f(\mathsf{l}')$$
for any $f \in \mathcal{F}$ and $\mathsf{l}, \mathsf{l}' \in \mathcal{L}$.

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:

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

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:

$$\mathcal{L} \;=\; N \cup \{\bot_N\} \to \mathcal{L}_N \qquad\qquad \mathcal{F} \;=\; E_\rho \to \mathcal{F}_E \tag{4.1}$$

where $\mathcal{L}_N$ is a *local property space* and $\mathcal{F}_E \subseteq \{f \mid f : \mathcal{L}_N \to \mathcal{L}_N\}$ 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 $\bigsqcup : \mathcal{L}_N^* \to \mathcal{L}_N$ 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 $\sqcup : \mathcal{L}_N \times \mathcal{L}_N \to \mathcal{L}_N$, 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 $l \in \mathcal{L}_N$:

$$\bot_{\mathcal{L}_N} \sqcup l \;=\; l \;=\; l \sqcup \bot_{\mathcal{L}_N}$$

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

- μ_E is a mapping from edges to local transfer functions;

- $\iota \in \mathcal{L}_N$ is the solution for the isolated entry node $\bot_N$.

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 μ is:

$$\mu(G) \;=\; \lambda e : E_\rho .\, \mu_E(e)$$

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:

$$l \sqsubseteq l' \qquad \text{iff} \quad l \sqcup l' = l' \qquad \square$$

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 $\langle G, \mathcal{L}_N, f_E, \iota \rangle$, where:

$$f_E \;=\; \{ f_{(m,n)} : \mathcal{L}_N \to \mathcal{L}_N \mid (m,n) \in E_\rho \;\wedge\; f_{(m,n)} = \mu_E(m,n) \}$$

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 $\mathcal{L}$:

$$l_0 \sqsubseteq l_1 \sqsubseteq \cdots \sqsubseteq l_k \sqsubseteq \cdots$$

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

$$l \sqsubseteq l' \qquad \text{iff} \quad \forall n \in N \cup \{ \bot_N \}.\; l(n) \sqsubseteq l'(n)$$

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

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

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:

$$\begin{aligned}
l \sqsubseteq l' \;&\Longrightarrow\; \forall m \in N.\; l(m) \sqsubseteq l'(m) \\
&\Longrightarrow\; \forall (m,n) \in E_\rho .\; f_{(m,n)}(l(m)) \sqsubseteq f_{(m,n)}(l'(m)) \\
&\Longrightarrow\; \bigsqcup_{(m,n)\in E_\rho} f_{(m,n)}(l(m)) \;\sqsubseteq\; \bigsqcup_{(m,n)\in E_\rho} f_{(m,n)}(l'(m)) \\
&\Longrightarrow\; f(l) \sqsubseteq f(l')
\end{aligned}$$

$\square$

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 $(m, n) \in E_\rho$).

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 $DFA^=(G)$:

$$DFA(n) = \begin{cases} \iota & \text{if } n = \bot_N \\ \bigsqcup_{(m,n)\in E_\rho} f_{(m,n)}(DFA(m)) & \text{otherwise} \end{cases}$$

We write $l \models DFA^=(G)$ when $l \in \mathcal{L}$ is solution to the equation system. The MFP solution is then defined as the *least* solution to $DFA^=(G)$, i.e.:

$$MFP \;=\; \bigsqcap \{ l \in \mathcal{L} \mid l \models DFA^=(G) \}$$

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

$$f(l) \;=\; \lambda n.\ \text{if } n = \bot_N \text{ then } \iota \text{ else } \bigsqcup_{(m,n)\in E_\rho} f_{(m,n)}(l(m))$$

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 π (either abstract or concrete) is defined as:

$$f_\pi \;=\; \begin{cases} id_{\mathcal{L}_N} & \text{if } \pi = \langle\rangle \\ f_{(n_{k-1},n_k)} \circ \cdots \circ f_{(n_0,n_1)} \circ f_{(\bot_N,n_0)} & \text{if } \pi = \langle n_0, \cdots, n_k \rangle \end{cases}$$

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

$$\mathrm{MTP}(n) \;=\; \bigsqcup_{\tilde{\pi} \in \tilde{\Pi}^{\tau}_{n}} f_{\tilde{\pi}}(\iota)$$

If we agree the empty path reaches the entry node $\bot_N$ (i.e. $\langle\rangle \in \Pi_{\bot_N}$), then we yield the correct solution at $\bot_N$, i.e. $\mathrm{MTP}(\bot_N) = id_{\mathcal{L}_N}(\iota) = \iota$.

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 $\langle G, \mathcal{L}_N, f_E, \iota \rangle$ of a data flow framework is defined as:

$$\mathrm{MVP}(n) \;=\; \bigsqcup_{\tilde{\pi} \in \tilde{\Pi}^{\upsilon}_{n}} f_{\tilde{\pi}}(\iota)$$

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 $\langle G, \mathcal{L}_N, f_E, \iota \rangle$ of a data flow framework is defined as:

$$\mathrm{MOP}(n) \;=\; \bigsqcup_{\pi \in \Pi_n \cup \tilde{\Pi}_n} f_{\pi}(\iota)$$

It can be shown [KU77] that the MOP solution is not always computable.

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

$$\mathrm{MTP} \sqsubseteq \mathrm{MVP} \sqsubseteq \mathrm{MOP} \sqsubseteq \mathrm{MFP}$$

Proof. First of all, note that all solutions coincide on the isolated entry node $\bot_N$, i.e. $\mathrm{MTP}(\bot_N) = \mathrm{MVP}(\bot_N) = \mathrm{MOP}(\bot_N) = \mathrm{MFP}(\bot_N) = \iota$. Then, from now on we will always consider $n \neq \bot_N$. For the first inequality, observe that, by theorem 3.8.6, any traversable path is also a valid path, i.e. $\widetilde{\Pi}^{\tau}_n \subseteq \widetilde{\Pi}^{\upsilon}_n$. Then, since $X \subseteq Y \implies \bigsqcup X \sqsubseteq \bigsqcup Y$, we have:

$$\mathrm{MTP}(n) = \bigsqcup_{\tilde{\pi} \in \widetilde{\Pi}^{\tau}_n} f_{\pi}(\iota) \sqsubseteq \bigsqcup_{\tilde{\pi} \in \widetilde{\Pi}^{\upsilon}_n} f_{\pi}(\iota) = \mathrm{MVP}(n)$$

For the second inequality, it suffices to observe that, by definition 3.7.1, any valid path is also a concrete path. Then $\widetilde{\Pi}^{\upsilon}_n \subseteq \widetilde{\Pi}_n \subseteq \Pi_n \cup \widetilde{\Pi}_n$, and:

$$\mathrm{MVP}(n) = \bigsqcup_{\tilde{\pi} \in \widetilde{\Pi}^{\upsilon}_n} f_{\pi}(\iota) \sqsubseteq \bigsqcup_{\pi \in \Pi_n \cup \widetilde{\Pi}_n} f_{\pi}(\iota) = \mathrm{MOP}(n)$$

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

$$\mathrm{MOP}_k(n) = \bigsqcup_{\substack{\pi \in \Pi_n \cup \widetilde{\Pi}_n \\ |\pi| < k}} f_{\pi}(\iota) \tag{4.2}$$

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

$$\mathrm{MOP}(n) = \bigsqcup_{k \in \mathbb{N}} \mathrm{MOP}_k(n) \tag{4.3}$$

In fact, consider the increasing chain:

$$\mathrm{MOP}_0(n) \sqsubseteq \mathrm{MOP}_1(n) \sqsubseteq \ldots \sqsubseteq \mathrm{MOP}_k(n) \sqsubseteq \ldots$$

By the ascending chain condition, there is a $k_0 \in \mathbb{N}$ such that $\mathrm{MOP}_k(n) = \mathrm{MOP}_{k_0}(n)$ 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, $\mathrm{MOP}(n) = \mathrm{MOP}_{k_0}(n) = \bigsqcup_{k \in \mathbb{N}} \mathrm{MOP}_k(n)$. Now, equation (4.3) enables us to prove the inclusion $\mathrm{MOP} \sqsubseteq \mathrm{MFP}$ by showing that:

$$\forall k \in \mathbb{N}.\ \mathrm{MOP}_k(n) \sqsubseteq \mathrm{MFP}(n)$$

We proceed by mathematical induction on the length k. For the base case, $\mathrm{MOP}_0(n)$ is the least upper bound over an empty set of paths: by convention, this equals to $\bot_{\mathcal{L}_N}$, and

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

$$
\begin{aligned}
\mathrm{MFP}(n) = \quad & f(\mathrm{MFP}(n)) && \text{by def. 4.3.1}\\
= \quad & \bigsqcup_{(m,n)\in E_\rho} f_{(m,n)}(\mathrm{MFP}(m)) && \text{by def. } f\\
\sqsupseteq \quad & \bigsqcup_{(m,n)\in E_\rho} f_{(m,n)}(\mathrm{MOP}_k(m)) && \text{by ind. hyp., monot. } f\\
= \quad & \bigsqcup_{(m,n)\in E_\rho} f_{(m,n)}\Big(\bigsqcup_{\substack{\pi \in \Pi_m \cup \widetilde{\Pi}_m\\ |\pi|<k}} f_\pi(\iota)\Big) && \text{by def. } \mathrm{MOP}_k\\
\sqsupseteq \quad & \bigsqcup_{(m,n)\in E_\rho} \Big(\bigsqcup_{\substack{\pi \in \Pi_m \cup \widetilde{\Pi}_m\\ |\pi|<k}} f_{(m,n)}(f_\pi(\iota))\Big) && \text{by monot. } f\\
= \quad & \bigsqcup_{(m,n)\in E_\rho} \Big(\bigsqcup_{\substack{\pi \in \Pi_m\\ |\pi|<k}} f_{(m,n)}(f_\pi(\iota)) \;\sqcup\; \bigsqcup_{\substack{\pi \in \widetilde{\Pi}_m\\ |\pi|<k}} f_{(m,n)}(f_\pi(\iota))\Big) && \text{by assoc. } \sqcup\\
\sqsupseteq \quad & \bigsqcup_{(m,n)\in E} \Big(\bigsqcup_{\substack{\pi \in \Pi_m\\ |\pi|<k}} f_{(m,n)}(f_\pi(\iota))\Big) \;\sqcup\; \bigsqcup_{(m,n)\in \tilde{E}} \Big(\bigsqcup_{\substack{\pi \in \widetilde{\Pi}_m\\ |\pi|<k}} f_{(m,n)}(f_\pi(\iota))\Big) && \text{as } E, \tilde{E} \subseteq E_\rho\\
= \quad & \bigsqcup_{\substack{\pi' \in \Pi_n\\ |\pi'|<k+1}} f_{\pi'}(\iota)) \;\sqcup\; \bigsqcup_{\substack{\pi' \in \widetilde{\Pi}_n\\ |\pi'|<k+1}} f_{\pi'}(\iota) && \text{by def. } f_\pi\\
= \quad & \bigsqcup_{\substack{\pi' \in \Pi_n \cup \widetilde{\Pi}_n\\ |\pi'|<k+1}} f_{\pi'}(\iota) && \text{by assoc. } \sqcup\\
= \quad & \mathrm{MOP}_{k+1}(n) && \text{by def. } \mathrm{MOP}_k
\end{aligned}
$$

□

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. $\Pi_n \neq \varnothing$ for each $n \in N$), and one of the following conditions holds:

$$f_{m \hookrightarrow n} = \bot_{\mathcal{L}_N} \quad \text{for each } m, n \in N \tag{4.4a}$$

$$f_{m \dashrightarrow n} = \bot_{\mathcal{L}_N} \quad \text{for each } m, n \in N \text{ such that } \ell(m) = \texttt{call} \tag{4.4b}$$

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

$$f\Big(\bigsqcup X\Big) = \bigsqcup_{l \in X} f(l) \tag{4.5}$$

Now, assume that condition (4.4a) is satisfied. Then, it follows that:

$$f_{\tilde{\pi}}(\iota) \;\sqsubseteq\; f_{\alpha(\tilde{\pi})}(\iota)$$

for any concrete path $\tilde{\pi}$. 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:

$$\bigsqcup_{\tilde{\pi}\in\widetilde{\Pi}_n} f_{\tilde{\pi}}(\iota) \;\sqsubseteq\; \bigsqcup_{\pi\in\Pi_n} f_{\pi}(\iota) \tag{4.6}$$

Next we calculate:

$$\begin{aligned}
\mathrm{MOP}(n) &= \bigsqcup_{\pi \,\in\, \Pi_n \cup \widetilde{\Pi}_n} f_\pi(\iota) && \text{by def. 4.3.5}\\
&= \bigsqcup_{\pi \,\in\, \Pi_n} f_\pi(\iota) \;\sqcup\; \bigsqcup_{\pi \,\in\, \widetilde{\Pi}_n} f_\pi(\iota) && \text{by assoc. } \sqcup\\
&= \bigsqcup_{\pi \,\in\, \Pi_n} f_\pi(\iota) && \text{by (4.6)}\\
&= \bigsqcup_{(m,n)\in E} \Big(\bigsqcup_{\pi \,\in\, \Pi_m} f_{(m,n)}(f_\pi(\iota)) \Big) && \text{by def. } f_\pi\\
&= \bigsqcup_{(m,n)\in E} f_{(m,n)}\Big(\bigsqcup_{\pi \,\in\, \Pi_m} f_\pi(\iota) \Big) && \text{by (4.5)}\\
&= \bigsqcup_{(m,n)\in E_\rho} f_{(m,n)}\Big(\bigsqcup_{\pi \,\in\, \Pi_m} f_\pi(\iota) \Big) && \text{by (4.4a)}\\
&= \bigsqcup_{(m,n)\in E_\rho} f_{(m,n)}\Big(\bigsqcup_{\pi \,\in\, \Pi_m} f_\pi(\iota) \;\sqcup\; \bigsqcup_{\pi \,\in\, \widetilde{\Pi}_m} f_\pi(\iota) \Big) && \text{by (4.6)}\\
&= \bigsqcup_{(m,n)\in E_\rho} f_{(m,n)}\Big(\bigsqcup_{\pi \,\in\, \Pi_m \cup \widetilde{\Pi}_m} f_\pi(\iota) \Big) && \text{by assoc. } \sqcup\\
&= \bigsqcup_{(m,n)\in E_\rho} f_{(m,n)}(\mathrm{MOP}(m)) && \text{by def. 4.3.5}
\end{aligned}$$

Therefore, MOP is solution to the data flow equations. Since, by definition 4.3.1, MFP is the least solution to such equations, we have MFP $\sqsubseteq$ MOP. On the other hand, by theorem 4.3.6 it follows MOP $\sqsubseteq$ 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 $\mathsf{DFP} = \langle \mathsf{G}, \mathcal{L}_\mathsf{N}, f_\mathsf{E}, \iota \rangle$.
Output: the MFP solution for DFP.

```
WORKLIST-ITERATION(G, L_N, f_E, ι)
 1  W ← NIL
 2  Analysis[⊥_N] ← ι
 3  for each (m, n) in E_ρ do
 4     W ← CONS((m, n), W)
 5  for each n in N do
 6     Analysis[n] ← ⊥_{L_N}
 7  while W ≠ NIL do
 8     (m, n) ← HEAD(W)
 9     W ← TAIL(W)
10     if f_(m,n)(Analysis[m]) ⋢ Analysis[n]
11        then Analysis[n] ← Analysis[n] ⊔ f_(m,n)(Analysis[m])
12             for each (n, n') in E_ρ do
13                W ← 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 $\mathsf{n} \in \mathsf{N} \cup \{ \bot_\mathsf{N} \}$, we have:

$$\mathrm{MFP}(\mathsf{n}) \;=\; \mathsf{Analysis}[\mathsf{n}]$$

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 $|\mathsf{E}_\rho|$ and $|\mathsf{N}|$ times respectively. The worklist W contains $|\mathsf{E}_\rho|$ 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 $|\mathsf{E}_\rho|$ new elements (lines 12-13).

An edge $(\mathsf{n}, \mathsf{n}')$ can be added only if $f_{(\mathsf{m},\mathsf{n})}(\mathsf{Analysis}[\mathsf{m}]) \sqsupset \mathsf{Analysis}[\mathsf{n}]$ for some node m, or they are incomparable (line 10). In both cases, the new value assigned to $\mathsf{Analysis}[\mathsf{n}]$ at line 11 is strictly greater than its previous one. In fact, the inequality:

$$\mathsf{Analysis}[\mathsf{n}] \;\sqsubset\; \mathsf{Analysis}[\mathsf{n}] \sqcup f_{(\mathsf{m},\mathsf{n})}(\mathsf{Analysis}[\mathsf{m}])$$

is implied by the more general result:

$$\mathsf{l}' \not\sqsubseteq \mathsf{l} \;\Longrightarrow\; \mathsf{l} \sqsubset \mathsf{l} \sqcup \mathsf{l}'$$

The non-strict inequality $\mathsf{l} \sqsubseteq \mathsf{l} \sqcup \mathsf{l}'$ is always true, because, by definition of $\sqsubseteq$, this is equivalent to $\mathsf{l} \sqcup (\mathsf{l} \sqcup \mathsf{l}') = (\mathsf{l} \sqcup \mathsf{l}')$, which holds by associativity and idempotency of $\sqcup$.

Strictness is proved by contradiction. Assume $l = l \sqcup l'$. By definition of $\sqsubseteq$, this is the same as saying $l' \sqsubseteq l$, which clearly contradicts hypothesis $l' \not\sqsubseteq l$.

Now, any *strictly* increasing chain must be finite. To see why, suppose, by contradiction, that $l_0 \sqsubset l_1 \sqsubset \cdots \sqsubset l_k \sqsubset \cdots$ is an infinite strictly increasing chain. Then, since $l_i \sqsubset l_{i+1}$ clearly implies $l_i \sqsubseteq l_{i+1}$ for any $i \in \mathbb{N}$, by the ascending chain condition we find an index $k_0 \in \mathbb{N}$ such that $\forall k \geq k_0.\ l_k = l_{k_0}$, 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 $\mathsf{Analysis}[n] \sqsubseteq \mathrm{MFP}(n)$ is invariant for the **while** loop; then, we will conclude by showing that, on termination of the loop, also the converse inequality $\mathrm{MFP}(n) \sqsubseteq \mathsf{Analysis}[n]$ will hold.

For the first part, observe that, after the initialisation **for** loop at lines 5-6, we have $\mathsf{Analysis}[\bot_N] = \iota = \mathrm{MFP}(\bot_N)$, and $\mathsf{Analysis}[n] = \bot_{\mathcal{L}_N} \sqsubseteq \mathrm{MFP}(n)$ for each $n \neq \bot_N$. Therefore, the inequality 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 $\mathsf{Analysis}$ is left unchanged. Otherwise, only the value of $\mathsf{Analysis}[n]$ is updated, and we have:

$$
\begin{array}{lcll}
\mathsf{Analysis}[n] & \leftarrow & \mathsf{Analysis}[n] \sqcup f_{(m,n)}(\mathsf{Analysis}[m]) & \text{by the assignment at line 11} \\
& \sqsubseteq & \mathrm{MFP}(n) \sqcup f_{(m,n)}(\mathsf{Analysis}[m]) & \text{by hyp. } \mathsf{Analysis}[n] \sqsubseteq \mathrm{MFP}(n) \\
& \sqsubseteq & \mathrm{MFP}(n) \sqcup f_{(m,n)}(\mathrm{MFP}(m)) & \text{by monotonicity of } f_{(m,n)} \\
& \sqsubseteq & \mathrm{MFP}(n) \sqcup \bigsqcup_{(m,n) \in \mathsf{E}_\rho} f_{(m,n)}(\mathrm{MFP}(m)) & \text{by def. } \sqsubseteq \text{ and } \sqcup \\
& = & \mathrm{MFP}(n) \sqcup \mathrm{MFP}(n) & \text{by def. MFP} \\
& = & \mathrm{MFP}(n) & \text{by idempotency of } \sqcup
\end{array}
$$

For the second part, note that the inequality:

$$\forall (m,n) \in \mathsf{E}_\rho.\ \mathsf{Analysis}[n] \sqsupseteq f_{(m,n)}(\mathsf{Analysis}[m]) \tag{4.7}$$

is ensured at the exit of the **while** loop. In fact, whenever the condition of the **if** statement at line 10 is false for some $(m,n) \in \mathsf{E}_\rho$, the assignment at line 11 manages to estabilish the inequality $\mathsf{Analysis}[n] \sqsupseteq f_{(m,n)}(\mathsf{Analysis}[m])$, because:

$$\mathsf{Analysis}[n] \sqcup f_{(m,n)}(\mathsf{Analysis}[m]) \sqsupseteq f_{(m,n)}(\mathsf{Analysis}[m])$$

trivially follows by definition of $\sqcup$ and $\sqsupseteq$. Actually, after the value of $\mathsf{Analysis}[n]$ has been updated, it may happen that $\mathsf{Analysis}[n'] \not\sqsupseteq f_{(n,n')}(\mathsf{Analysis}[n])$ for some $(n,n') \in \mathsf{E}_\rho$. However, once the assignment at line 11 has been performed, the inner **for** loop at lines 12-13 takes care of inserting all edges $(n,n') \in \mathsf{E}_\rho$ in the worklist: then, the inequality $\mathsf{Analysis}[n'] \sqsupseteq f_{(n,n')}(\mathsf{Analysis}[n])$ will eventually be re-estabilished in a subsequent iteration of the **while**. Hence, the inequality (4.7) is satisfied at the exit of the loop. By definition of $\bigsqcup$, from (4.7) it follows that, for any $n \in \mathsf{N}$:

$$\mathsf{Analysis}[n] \sqsupseteq \bigsqcup_{(m,n) \in \mathsf{E}_\rho} f_{(m,n)}(\mathsf{Analysis}[m])$$

Then, $\mathsf{Analysis}$ turns out to be a *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:

$$\mathsf{Analysis}[n] \sqsupseteq \mathrm{MFP}(n) \qquad \square$$

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 analysis can be used to optimize the Java bytecode.

5.1 The Denied Permissions Analysis

We say that a permission P is *denied* to a state σ if $\sigma \nvdash \mathsf{JDK}(\mathsf{P})$. 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 *certainly* denied to any state reaching that node. We require the analysis to be *safe*, in the sense that it never claims that a permission P is denied to a node $\mathfrak{n}$ when an execution $[] \rhd \cdots \rhd \sigma : \mathfrak{n}$ exists such that $\sigma : \mathfrak{n} \vdash \mathsf{JDK}(\mathsf{P})$.

Definition 5.1.1. The sets of permissions denied, respectively, to a state σ and to a node $\mathfrak{n}$, are defined as:

$$\Delta(\sigma) \;=\; \begin{cases} \{\, \mathsf{P} \in \textbf{Permission} \mid \sigma \nvdash \mathsf{JDK}(\mathsf{P}) \,\} & \text{if } \mathsf{G} \vdash \sigma \\ \varnothing & \text{otherwise} \end{cases}$$

$$\Delta(\mathfrak{n}) \;=\; \bigcap_{\mathsf{G} \vdash \sigma : \mathfrak{n}} \Delta(\sigma : \mathfrak{n})$$

Given any control flow graph G, a DP analysis is specified by means of an equation system $\mathsf{DP}^{=}(\mathsf{G})$. We say that δ is a DP-solution if $\delta \models \mathsf{DP}^{=}(\mathsf{G})$.

Definition 5.1.2 (DP-soundness). A DP-solution δ is *sound* if:

$$\mathsf{P} \in \delta(\mathfrak{n}) \quad \Longrightarrow \quad \forall \sigma \in \Sigma.\ \mathsf{G} \vdash \sigma : \mathfrak{n} \implies \sigma : \mathfrak{n} \nvdash \mathsf{JDK}(\mathsf{P})$$

Lemma 5.1.3. Let δ be a sound DP-solution. Then, for each node $\mathfrak{n} \in \mathsf{N}$:

$$\delta(\mathfrak{n}) \;\subseteq\; \Delta(\mathfrak{n})$$

Proof. Let $\mathsf{P} \in \delta(\mathfrak{n})$, where δ is a sound DP-solution. If $\mathfrak{n}$ is not G-reachable, then $\Delta(\mathfrak{n})$ 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 $\Delta(\mathfrak{n})$.

Otherwise, if $\mathfrak{n}$ is G-reachable, by definition 5.1.2 we have that $\sigma : \mathfrak{n} \nvdash \mathsf{JDK}(\mathsf{P})$ for any G-reachable state $\sigma : \mathfrak{n}$. Actually, by definition 5.1.1, this means that $\mathsf{P} \in \Delta(\sigma : \mathfrak{n})$ for any state $\sigma : \mathfrak{n}$ such that $\mathsf{G} \vdash \sigma : \mathfrak{n}$, that is:

$$\mathsf{P} \;\in\; \bigcap_{\mathsf{G} \vdash \sigma : \mathfrak{n}} \Delta(\sigma : \mathfrak{n}) \qquad \square$$

Definition 5.1.4. A DP-solution δ is *non-trivial* if:

$$\mathsf{P} \notin \mathit{Perm}(\mathfrak{n}) \quad \Longrightarrow \quad \mathsf{P} \in \delta(\mathfrak{n})$$

Definition 5.1.5 (DP-completeness). A DP-solution δ is *complete* if:

$$\mathsf{P} \notin \delta(n) \quad \Longrightarrow \quad \exists \sigma \in \Sigma.\ \mathsf{G} \vdash \sigma : n \ \wedge\ \sigma : n \vdash \mathsf{JDK}(\mathsf{P})$$

Lemma 5.1.6. Let δ be a complete DP-solution. Then, for each $n \in \mathsf{N}$:

$$\delta(n) \ \supseteq\ \Delta(n)$$

Proof. Let δ be a complete DP-solution, and $\mathsf{P} \in \Delta(n)$. 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 $\mathsf{P} \notin \delta(n)$ 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 $\mathsf{P} \in \Delta(n)$, we have that $\sigma : n \nvdash \mathsf{JDK}(\mathsf{P})$ for any G-reachable state $\sigma : n$. By contradiction, assume $\mathsf{P} \notin \delta(n)$: then, by definition 5.1.5, at least one G-reachable state $\sigma : n$ exists such that $\sigma : n \vdash \mathsf{JDK}(\mathsf{P})$. This raises a contradiction with hypothesis $\sigma : n \nvdash \mathsf{JDK}(\mathsf{P})$. □

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*.

n	$\delta(n)$
n_0	$\varnothing$
n_1	*unreach*
$n_2 - n_4$	$\{\mathsf{P}_{loan}, \mathsf{P}_{read}, \mathsf{P}_{write}\}$
n_5	$\{\mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}, \mathsf{P}_{loan}, \mathsf{P}_{read}, \mathsf{P}_{write}\}$
n_6	*unreach*
$n_7 - n_8$	$\{\mathsf{P}_{loan}, \mathsf{P}_{read}, \mathsf{P}_{write}\}$
n_9	$\varnothing$
$n_{10} - n_{12}$	$\{\mathsf{P}_{loan}, \mathsf{P}_{read}, \mathsf{P}_{write}\}$
$n_{13} - n_{14}$	$\varnothing$
$n_{15} - n_{16}$	$\{\mathsf{P}_{loan}, \mathsf{P}_{read}, \mathsf{P}_{write}\}$
$n_{17} - n_{18}$	*unreach*
n_{19}	$\{\mathsf{P}_{loan}, \mathsf{P}_{read}, \mathsf{P}_{write}\}$
$n_{20} - n_{21}$	$\varnothing$
n_{22}	$\{\mathsf{P}_{loan}, \mathsf{P}_{read}, \mathsf{P}_{write}\}$
$n_{23} - n_{26}$	$\varnothing$

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 σ if $\sigma \vdash \mathsf{JDK}(\mathsf{P})$. 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 σ and to a node $\mathfrak{n}$, are defined as:

$$\Gamma(\sigma) \;=\; \begin{cases} \{\mathsf{P} \in \textbf{Permission} \mid \sigma \vdash \mathsf{JDK}(\mathsf{P})\} & \text{if } \mathsf{G} \vdash \sigma \\ \varnothing & \text{otherwise} \end{cases}$$

$$\Gamma(\mathfrak{n}) \;=\; \bigcap_{\mathsf{G} \vdash \sigma:\mathfrak{n}} \Gamma(\sigma:\mathfrak{n})$$

Given any control flow graph G, a GP analysis is specified by means of an equation system $\mathsf{GP}^{=}(\mathsf{G})$. We say that γ is a GP-solution if $\gamma \models \mathsf{GP}^{=}(\mathsf{G})$.

Definition 5.2.2 (GP-soundness). A GP-solution γ is *sound* if:

$$\mathsf{P} \in \gamma(\mathfrak{n}) \quad \Longrightarrow \quad \forall \sigma \in \Sigma.\ \mathsf{G} \vdash \sigma:\mathfrak{n} \implies \sigma:\mathfrak{n} \vdash \mathsf{JDK}(\mathsf{P})$$

Lemma 5.2.3. Let γ be a sound GP-solution. Then, for each node $\mathfrak{n} \in \mathsf{N}$:

$$\gamma(\mathfrak{n}) \;\subseteq\; \Gamma(\mathfrak{n})$$

Proof. Let $\mathsf{P} \in \gamma(\mathfrak{n})$, where γ is a sound GP-solution. If $\mathfrak{n}$ is not G-reachable, then $\Gamma(\mathfrak{n})$ 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 $\Gamma(\mathfrak{n})$.

Otherwise, if $\mathfrak{n}$ is G-reachable, by definition 5.2.2 we have that $\sigma:\mathfrak{n} \vdash \mathsf{JDK}(\mathsf{P})$ for any G-reachable state $\sigma:\mathfrak{n}$. Actually, by definition 5.2.1, this means that $\mathsf{P} \in \Gamma(\sigma:\mathfrak{n})$ for any state $\sigma:\mathfrak{n}$ such that $\mathsf{G} \vdash \sigma:\mathfrak{n}$, that is:

$$\mathsf{P} \;\in\; \bigcap_{\mathsf{G} \vdash \sigma:\mathfrak{n}} \Gamma(\sigma:\mathfrak{n})$$

□

Definition 5.2.4. A GP-solution γ is *non-trivial* if:

$$\mathsf{P} \in \mathit{Perm}(\mathfrak{n}) \;\wedge\; \mathit{Priv}(\mathfrak{n}) \quad \Longrightarrow \quad \mathsf{P} \in \gamma(\mathfrak{n})$$

Definition 5.2.5 (GP-completeness). A GP-solution γ is *complete* if:

$$\mathsf{P} \notin \gamma(\mathfrak{n}) \quad \Longrightarrow \quad \exists \sigma \in \Sigma.\ \mathsf{G} \vdash \sigma:\mathfrak{n} \;\wedge\; \sigma:\mathfrak{n} \nvdash \mathsf{JDK}(\mathsf{P})$$

Lemma 5.2.6. Let γ be a complete GP-solution. Then, for each $n \in N$:

$$\gamma(n) \supseteq \Gamma(n)$$

Proof. Let γ 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 $\sigma : n \nvdash JDK(P)$. This raises a contradiction with hypothesis $\sigma : n \vdash JDK(P)$. □

Theorem 5.2.7. Let δ a sound DP-solution and γ a sound GP-solution. Then, for any G-reachable node $n \in N$:

$$\gamma(n) \subseteq \overline{\delta}(n)$$

Proof. Assume $P \in \gamma(n)$, but also $P \in \delta(n)$. This is a contradiction, because, if $G \vdash \sigma : n$, then DP-soundness ensures $\sigma : n \nvdash JDK(P)$, while GP-soundness states $\sigma : n \vdash JDK(P)$. □

Example 5.2.8. The optimal GP-solution for example 3.1.2 is in Table 5.2.

n	$\gamma(n)$
n_0	$\{P_{canpay}, P_{credit}, P_{debit}, P_{loan}, P_{read}, P_{write}\}$
n_1	*unreach*
$n_2 - n_4$	$\{P_{canpay}, P_{credit}, P_{debit}\}$
n_5	$\varnothing$
n_6	*unreach*
$n_7 - n_8$	$\{P_{canpay}, P_{credit}, P_{debit}\}$
n_9	$\{P_{canpay}, P_{credit}, P_{debit}, P_{loan}, P_{read}, P_{write}\}$
$n_{10} - n_{12}$	$\{P_{canpay}, P_{credit}, P_{debit}\}$
$n_{13} - n_{14}$	$\{P_{canpay}, P_{credit}, P_{debit}, P_{loan}, P_{read}, P_{write}\}$
n_{15}	$\{P_{canpay}, P_{credit}, P_{debit}\}$
n_{16}	$\varnothing$
$n_{17} - n_{18}$	*unreach*
n_{19}	$\{P_{canpay}, P_{credit}, P_{debit}\}$
$n_{20} - n_{21}$	$\{P_{canpay}, P_{credit}, P_{debit}, P_{loan}, P_{read}, P_{write}\}$
n_{22}	$\{P_{canpay}, P_{credit}, P_{debit}\}$
$n_{23} - n_{26}$	$\{P_{canpay}, P_{credit}, P_{debit}, P_{loan}, P_{read}, P_{write}\}$

Table 5.2: The optimal GP-solution for the e-commerce application.

5.3 The DP^0 Analysis

$$\overline{\mathsf{DP}^0}_{in}(\mathtt{n}) = \bigcup_{(\mathtt{m},\mathtt{n})\in \mathsf{E}} \overline{\mathsf{DP}^0}_{out}(\mathtt{m},\mathtt{n})$$

$$\overline{\mathsf{DP}^0}_{out}(\mathtt{m},\mathtt{n}) = \begin{cases} \mathit{Perm}(\mathtt{n}) & \text{if } \bullet\!\!\longrightarrow \mathtt{n} \\ \overline{\mathsf{DP}^0}_{call}(\mathtt{m}) \cap \mathit{Perm}(\mathtt{n}) & \text{if } \mathtt{m} \longrightarrow \mathtt{n} \\ \overline{\mathsf{DP}^0}_{trans}(\mathtt{m}) & \text{if } \mathtt{m} \dashrightarrow \mathtt{n} \end{cases}$$

$$\overline{\mathsf{DP}^0}_{call}(\mathtt{n}) = \begin{cases} \mathit{Perm}(\mathtt{n}) & \text{if } \mathit{Priv}(\mathtt{n}) \\ \overline{\mathsf{DP}^0}_{in}(\mathtt{n}) & \text{otherwise} \end{cases}$$

$$\overline{\mathsf{DP}^0}_{trans}(\mathtt{n}) = \begin{cases} \varnothing & \text{if } \mathsf{kill}^0(\mathtt{n}) \\ \overline{\mathsf{DP}^0}_{in}(\mathtt{n}) \cup \{\mathsf{P}\} & \text{if } \ell(\mathtt{n}) = \mathtt{check}(\mathsf{P}) \text{ and } \neg\mathsf{kill}^0(\mathtt{n}) \\ \overline{\mathsf{DP}^0}_{in}(\mathtt{n}) & \text{otherwise} \end{cases}$$

$$\mathsf{kill}^0(\mathtt{n}) =_{def} \ell(\mathtt{n}) = \mathtt{check}(\mathsf{P}) \text{ and } \mathsf{P} \notin \mathit{Perm}(\mathtt{n})$$

Table 5.3: The DP^0 Analysis.

The DP^0 analysis is defined in Table 5.3 by means of the complemented analysis $\overline{\mathsf{DP}^0}$ 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 $\mathtt{n}$ enforcing a permission P kills all the permissions at its entry if $\mathsf{P} \notin \mathit{Perm}(\mathtt{n})$; 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 $\mathtt{n}$ always belong to $\mathit{Perm}(\mathtt{n})$.

Lemma 5.3.1. Let $\langle \delta_{in}, \delta_{out}, \delta_{call}, \delta_{trans} \rangle \models \mathsf{DP}^0(\mathsf{G})$. Then:

$$\mathsf{P} \in \delta_{in}(\mathfrak{n}) \ \cap \ \mathit{Perm}(\mathfrak{n}) \quad \Longrightarrow \quad \sigma \nvdash \mathsf{JDK}(\mathsf{P})$$

for any G-reachable state $\sigma : \mathfrak{n}$.

Proof. The proof is carried out by contradiction, assuming $\sigma \vdash \mathsf{JDK}(\mathsf{P})$. Then, we proceed by induction on the derivation used to estabilish $\mathsf{G} \vdash \sigma : \mathfrak{n}$. The base case corresponds to our single axiom:

$$\frac{\mathfrak{n} \in \mathsf{N}_{entry}}{[] \ \rhd \ [\mathfrak{n}]}$$

For $\mathfrak{n} \in \mathsf{N}_{entry}$, we have that $\mathsf{P} \in \delta_{in}(\mathfrak{n}) \implies \mathsf{P} \in \delta_{out}(\bot_{\mathsf{N}}, \mathfrak{n})$. Then $\mathsf{P} \notin \mathit{Perm}(\mathfrak{n})$, and it cannot be, at the same time, $\mathsf{P} \in \mathit{Perm}(\mathfrak{n})$. 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 $\sigma : \mathfrak{n}$, yielding:

- case [*call*]:

$$\frac{\ell(\mathfrak{n}') = \texttt{call} \quad \mathfrak{n}' \longrightarrow \mathfrak{n}}{\sigma' : \mathfrak{n}' \ \rhd \ \sigma' : \mathfrak{n}' : \mathfrak{n}} \qquad \text{where} \quad \sigma = \sigma' : \mathfrak{n}'$$

  By definition of DP^0_{in}, $\mathsf{P} \in \delta_{in}(\mathfrak{n}) \implies \mathsf{P} \in \delta_{out}(\mathfrak{n}', \mathfrak{n})$, that is $\mathsf{P} \in \delta_{call}(\mathfrak{n}')$ or $\mathsf{P} \notin \mathit{Perm}(\mathfrak{n})$. The latter option is prevented by our assumptions about P, then only the former one is considered. If $\neg \mathit{Priv}(\mathfrak{n}')$, then $\mathsf{P} \in \delta_{in}(\mathfrak{n}')$; assumption $\sigma' : \mathfrak{n}' \vdash \mathsf{JDK}(\mathsf{P})$ imposes $\mathsf{P} \in \mathit{Perm}(\mathfrak{n}')$, hence we can apply the inductive hypothesis to deduce $\sigma' \nvdash \mathsf{JDK}(\mathsf{P})$. This prevents the $\mathsf{JDK}_{\prec}$ rule to be applicable, and a contradiction arises with assumption $\sigma' : \mathfrak{n}' \vdash \mathsf{JDK}(\mathsf{P})$. Otherwise, if $\mathit{Priv}(\mathfrak{n}')$, then it should hold $\mathsf{P} \notin \mathit{Perm}(\mathfrak{n}')$, contradicting again assumption $\sigma' : \mathfrak{n}' \vdash \mathsf{JDK}(\mathsf{P})$.

- case [*check*]:

$$\frac{\ell(\mathfrak{n}') = \texttt{check}(\mathsf{P}') \quad \sigma : \mathfrak{n}' \vdash \mathsf{JDK}(\mathsf{P}') \quad \mathfrak{n}' \dashrightarrow \mathfrak{n}}{\sigma : \mathfrak{n}' \ \rhd \ \sigma : \mathfrak{n}}$$

  Here $\mathsf{P} \in \delta_{in}(\mathfrak{n}) \implies \mathsf{P} \in \delta_{trans}(\mathfrak{n}')$ and, by the premises of the $\rhd_{check}$ rule, we also have $\mathsf{P}' \in \mathit{Perm}(\mathfrak{n}')$. Then, it must be $\mathsf{P} \in \delta_{in}(\mathfrak{n}')$, and the inductive hypothesis can be applied to obtain a contradiction $\sigma \nvdash \mathsf{JDK}(\mathsf{P})$.

- case [*return*]:

$$\frac{\ell(\mathfrak{m}) = \texttt{return} \quad \mathfrak{n}' \dashrightarrow \mathfrak{n}}{\sigma : \mathfrak{n}' : \mathfrak{m} \ \rhd \ \sigma : \mathfrak{n}}$$

  Here, $\mathsf{P} \in \delta_{in}(\mathfrak{n}) \implies \mathsf{P} \in \delta_{trans}(\mathfrak{n}')$. By lemma 3.4.4, it must be $\ell(\mathfrak{n}') = \texttt{call}$, and again it turns out that $\mathsf{P} \in \delta_{in}(\mathfrak{n}')$. Since both $\mathfrak{n}$ and $\mathfrak{n}'$ carry the same permissions, we also have $\mathsf{P} \in \mathit{Perm}(\mathfrak{n}) \implies \mathsf{P} \in \mathit{Perm}(\mathfrak{n}')$. By lemma 3.4.1, any derivation of $\sigma : \mathfrak{n}' : \mathfrak{m}$ is of the form:

$$[] \ \rhd \ \cdots \ \rhd \ \sigma : \mathfrak{n}' \ \rhd \ \cdots \ \rhd \ \sigma : \mathfrak{n}' : \mathfrak{m}$$

  Therefore, we can apply the inductive hypothesis to $\sigma : \mathfrak{n}'$, and a contradiction arises by noticing that the inductive hypothesis yields $\sigma \nvdash \mathsf{JDK}(\mathsf{P})$.

□

Theorem 5.3.2. Let $\langle \delta_{in}, \delta_{out}, \delta_{call}, \delta_{trans} \rangle \models \mathsf{DP}^0(\mathsf{G})$. For any $\mathfrak{n} \in \mathsf{N}$, define:

$$\delta(\mathfrak{n}) \;=\; \delta_{call}(\mathfrak{n})$$

Then δ is a sound DP^0-solution.

Proof. Let $\sigma : \mathfrak{n}$ be a G-reachable state, and $\mathsf{P} \in \delta_{call}(\mathfrak{n})$. By contradiction, assume $\sigma : \mathfrak{n} \vdash \mathsf{JDK}(\mathsf{P})$. This clearly implies $\mathsf{P} \in \mathit{Perm}(\mathfrak{n})$: otherwise, neither the $\mathsf{JDK}_{\prec}$ nor the JDK_{Priv} rules are applicable. Then we have to distinguish between two cases:

- if $\neg \mathit{Priv}(\mathfrak{n})$, then $\mathsf{P} \in \delta_{in}(\mathfrak{n})$, and by lemma 5.3.1 it follows $\sigma \not\vdash \mathsf{JDK}(\mathsf{P})$. This prevents the $\mathsf{JDK}_{\prec}$ rule to be applicable, hence a contradiction arises with our assumption $\sigma : \mathfrak{n} \vdash \mathsf{JDK}(\mathsf{P})$.
- if $\mathit{Priv}(\mathfrak{n})$, then $\mathsf{P} \notin \mathit{Perm}(\mathfrak{n})$. Again, this contradicts $\sigma : \mathfrak{n} \vdash \mathsf{JDK}(\mathsf{P})$.

□

Example 5.3.3. Consider the e-commerce application of Fig. 3.1. By direct computation of the analysis at $\mathfrak{n}_2$, we obtain:

$$\begin{aligned}
\overline{\delta}_{in}(\mathfrak{n}_2) &= \overline{\delta}_{out}(\mathfrak{n}_0, \mathfrak{n}_2) \cup \overline{\delta}_{out}(\mathfrak{n}_3, \mathfrak{n}_2) \cup \overline{\delta}_{out}(\mathfrak{n}_4, \mathfrak{n}_2) \\
&= \big(\overline{\delta}_{call}(\mathfrak{n}_0) \cap \mathit{Perm}(\mathfrak{n}_2)\big) \cup \overline{\delta}_{trans}(\mathfrak{n}_3) \cup \overline{\delta}_{trans}(\mathfrak{n}_4) \\
&= \big(\overline{\delta}_{in}(\mathfrak{n}_0) \cap \mathit{Perm}(\mathfrak{n}_2)\big) \cup \overline{\delta}_{in}(\mathfrak{n}_3) \cup \overline{\delta}_{in}(\mathfrak{n}_4) \\
&= \big(\mathit{Perm}(\mathfrak{n}_0) \cap \mathit{Perm}(\mathfrak{n}_2)\big) \cup \overline{\delta}_{out}(\mathfrak{n}_2, \mathfrak{n}_3) \cup \overline{\delta}_{out}(\mathfrak{n}_2, \mathfrak{n}_4) \\
&= \{\mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}\} \cup \overline{\delta}_{trans}(\mathfrak{n}_2) \\
&= \{\mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}\} \cup \overline{\delta}_{in}(\mathfrak{n}_2)
\end{aligned}$$

This recursive equation is satisfied by any superset of $\{\mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}\}$: since the property space for $\overline{\mathsf{DP}^0}$ is partially ordered by $\subseteq$, this is just the MFP solution at $\mathfrak{n}_2$. Actually, this is the most accurate DP-sound solution: in fact, no permissions can be granted to an execution stack having $\mathfrak{n}_2$ as top element, outside those associated to the protection domain of $\mathfrak{n}_2$ (Client).

$$\begin{aligned}
\overline{\delta}_{in}(\mathfrak{n}_5) &= \overline{\delta}_{out}(\mathfrak{n}_0, \mathfrak{n}_5) \cup \overline{\delta}_{out}(\mathfrak{n}_6, \mathfrak{n}_5) = \big(\overline{\delta}_{call}(\mathfrak{n}_0) \cap \varnothing\big) \cup \overline{\delta}_{trans}(\mathfrak{n}_6) \\
&= \overline{\delta}_{in}(\mathfrak{n}_6) = \overline{\delta}_{out}(\mathfrak{n}_5, \mathfrak{n}_6) = \overline{\delta}_{trans}(\mathfrak{n}_5) = \overline{\delta}_{in}(\mathfrak{n}_5)
\end{aligned}$$

This recursive equation is satisfied by any set, so the MFP solution at $\mathfrak{n}_5$ is $\varnothing$. Again, this is the most accurate DP-sound solution: in fact, no permissions

can be granted to an execution stack having $\mathfrak{n}_5$ as top element, because the protection domain of $\mathfrak{n}_5$ (Unknown) does not carry any permission.

$$\begin{aligned}\overline{\delta}_{in}(\mathfrak{n}_{16}) &= \overline{\delta}_{out}(\mathfrak{n}_4,\mathfrak{n}_{16}) \cup \overline{\delta}_{out}(\mathfrak{n}_5,\mathfrak{n}_{16}) = \overline{\delta}_{call}(\mathfrak{n}_4) \cup \overline{\delta}_{call}(\mathfrak{n}_5)\\ &= \overline{\delta}_{in}(\mathfrak{n}_4) \cup \overline{\delta}_{in}(\mathfrak{n}_5) = \overline{\delta}_{out}(\mathfrak{n}_2,\mathfrak{n}_4) \cup \overline{\delta}_{in}(\mathfrak{n}_5)\\ &= \overline{\delta}_{trans}(\mathfrak{n}_2) \cup \overline{\delta}_{in}(\mathfrak{n}_5) = \overline{\delta}_{in}(\mathfrak{n}_2) \cup \overline{\delta}_{in}(\mathfrak{n}_5)\end{aligned}$$

By the previous computations, we have that this equation is satisfied by any superset of $\{\mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}\}$. Therefore, taking the complement of the MFP solution, it follows that all the permissions *except* $\{\mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}\}$ belong to $\delta(\mathfrak{n}_{16})$. Actually, $\mathsf{P}_{loan} \in \delta(\mathfrak{n}_{16})$, too: then, it is not necessary to perform stack inspection in order to enforce the security check at $\mathfrak{n}_{16}$, 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 $\overline{\mathrm{DP}^0}$, since its monotonicity clearly implies monotonicity of DP^0. The global property space for the complemented analysis is the pair $\mathcal{L} = \langle \mathcal{L}, \sqsubseteq \rangle$, 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 $\mathcal{P}(\textbf{Permission})$, while $\mathcal{L}_{out}$ is a total function space from E to $\mathcal{P}(\textbf{Permission})$. In symbols:

$$\begin{aligned}\mathcal{L}_{in}, \mathcal{L}_{call}, \mathcal{L}_{trans} &= \mathsf{N} \to \mathcal{P}(\textbf{Permission})\\ \mathcal{L}_{out} &= \mathsf{E} \to \mathcal{P}(\textbf{Permission})\end{aligned}$$

Since, for any control flow graph, each of the sets N, E and **Permission** is finite, then also $\mathcal{L}_{in}$, $\mathcal{L}_{call}$, $\mathcal{L}_{trans}$ and $\mathcal{L}_{out}$ are finite, with cardinality:

$$\begin{aligned}|\mathcal{L}_{in}| = |\mathcal{L}_{call}| = |\mathcal{L}_{trans}| &= 2^{|\mathsf{N}| \cdot |\textbf{Permission}|}\\ |\mathcal{L}_{out}| &= 2^{|\mathsf{E}| \cdot |\textbf{Permission}|}\end{aligned}$$

Assuming that $\mathcal{P}(\textbf{Permission})$ is partially ordered by the subset relation $\subseteq$, a standard construction equips each of these spaces with a pointwise order. More precisely, we have:

$$\begin{array}{lcll}\mathsf{l}_{in} & \sqsubseteq & \mathsf{l}'_{in} & \equiv \forall \mathfrak{n} \in \mathsf{N}.\ \mathsf{l}_{in}(\mathfrak{n}) \subseteq \mathsf{l}'_{in}(\mathfrak{n})\\ \mathsf{l}_{out} & \sqsubseteq & \mathsf{l}'_{out} & \equiv \forall (\mathfrak{m},\mathfrak{n}) \in \mathsf{E}.\ \mathsf{l}_{out}(\mathfrak{m},\mathfrak{n}) \subseteq \mathsf{l}'_{out}(\mathfrak{m},\mathfrak{n})\\ \mathsf{l}_{call} & \sqsubseteq & \mathsf{l}'_{call} & \equiv \forall \mathfrak{n} \in \mathsf{N}.\ \mathsf{l}_{call}(\mathfrak{n}) \subseteq \mathsf{l}'_{call}(\mathfrak{n})\\ \mathsf{l}_{trans} & \sqsubseteq & \mathsf{l}'_{trans} & \equiv \forall \mathfrak{n} \in \mathsf{N}.\ \mathsf{l}_{trans}(\mathfrak{n}) \subseteq \mathsf{l}'_{trans}(\mathfrak{n})\end{array}$$

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:

$$|\mathcal{L}| = 2^{(\,3|\mathsf{N}|+|\mathsf{E}|\,) \cdot |\textbf{Permission}|}$$

The bottom element of $\mathcal{L}$ is $\bot_{\mathcal{L}} = \langle \bot_{\mathcal{L}_{in}}, \bot_{\mathcal{L}_{out}}, \bot_{\mathcal{L}_{call}}, \bot_{\mathcal{L}_{trans}} \rangle$, where:

$$\begin{array}{rcl} \bot_{\mathcal{L}_{in}}, \bot_{\mathcal{L}_{call}}, \bot_{\mathcal{L}_{trans}} & = & \bot_{\mathsf{N} \to \mathcal{P}(\textbf{Permission})} = \lambda \mathtt{n} : \mathsf{N}.\ \varnothing \\ \bot_{\mathcal{L}_{out}} & = & \bot_{\mathsf{E} \to \mathcal{P}(\textbf{Permission})} = \lambda (\mathtt{m}, \mathtt{n}) : \mathsf{E}.\ \varnothing \end{array}$$

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 $f = \mu(\mathsf{G})$ is defined by the equation system in Table 5.3 as a quadruple:

$$f = \langle f_{in},\ f_{out},\ f_{call},\ f_{trans} \rangle$$

By definition of monotonicity, we have for prove that, for any $\mathtt{l}, \mathtt{l}' \in \mathcal{L}$:

$$\mathtt{l} \sqsubseteq \mathtt{l}' \quad \Longrightarrow \quad f(\mathtt{l}) \sqsubseteq f(\mathtt{l}')$$

So, let's assume $\mathtt{l} \sqsubseteq \mathtt{l}'$. By definition of the $\sqsubseteq$ relation, we have to deal with four cases:

- $f_{in}(\mathtt{l}) \sqsubseteq f_{in}(\mathtt{l}')$. Let $\mathtt{n} \in \mathsf{N}$. We have:

$$\begin{array}{rcl} f_{in}(\mathtt{l})(\mathtt{n}) & = & \bigcup_{(\mathtt{m},\mathtt{n}) \in \mathsf{E}} \mathtt{l}_{out}(\mathtt{m}, \mathtt{n}) \\ f_{in}(\mathtt{l}')(\mathtt{n}) & = & \bigcup_{(\mathtt{m},\mathtt{n}) \in \mathsf{E}} \mathtt{l}'_{out}(\mathtt{m}, \mathtt{n}) \end{array}$$

  Here hypothesis $\mathtt{l} \sqsubseteq \mathtt{l}'$ ensures that, for any $(\mathtt{m}, \mathtt{n}) \in \mathsf{E}$, it is $\mathtt{l}_{out}(\mathtt{m}, \mathtt{n}) \subseteq \mathtt{l}'_{out}(\mathtt{m}, \mathtt{n})$, and the inequality is preserved by the union operator on sets.

- $f_{out}(\mathtt{l}) \sqsubseteq f_{out}(\mathtt{l}')$. Here we have to distinguish between the three kinds of edges. Let $(\mathtt{m}, \mathtt{n}) \in \mathsf{E}$. If $\bullet \!\!\rightarrow \mathtt{n}$, then $\mathtt{m} = \bot_{\mathsf{N}}$, and the inequality trivially holds, because:

$$f_{out}(\mathtt{l})(\mathtt{m}, \mathtt{n}) \ = \ \mathit{Perm}(\mathtt{n}) \ = \ f_{out}(\mathtt{l}')(\mathtt{m}, \mathtt{n}),$$

  Otherwise, if $\mathtt{m} \longrightarrow \mathtt{n}$, then:

$$\begin{array}{rcl} f_{out}(\mathtt{l})(\mathtt{m}, \mathtt{n}) & = & \mathtt{l}_{call}(\mathtt{m}) \cap \mathit{Perm}(\mathtt{n}) \\ f_{out}(\mathtt{l}')(\mathtt{m}, \mathtt{n}) & = & \mathtt{l}'_{call}(\mathtt{m}) \cap \mathit{Perm}(\mathtt{n}) \end{array}$$

  Here the inequality holds, as $\mathtt{l} \sqsubseteq \mathtt{l}'$ ensures $\mathtt{l}_{call}(\mathtt{m}) \subseteq \mathtt{l}'_{call}(\mathtt{m})$ for any node $\mathtt{m} \in \mathsf{N}$. Otherwise, if $\mathtt{m} \dashrightarrow \mathtt{n}$:

$$\begin{array}{rcl} f_{out}(\mathtt{l})(\mathtt{m}, \mathtt{n}) & = & \mathtt{l}_{trans}(\mathtt{m}) \\ f_{out}(\mathtt{l}')(\mathtt{m}, \mathtt{n}) & = & \mathtt{l}'_{trans}(\mathtt{m}) \end{array}$$

  Again, f preserves the inequality, because $\mathtt{l} \sqsubseteq \mathtt{l}'$ ensures $\mathtt{l}_{trans}(\mathtt{m}) \subseteq \mathtt{l}'_{trans}(\mathtt{m})$.

- $f_{call}(\mathtt{l}) \sqsubseteq f_{call}(\mathtt{l}')$. There two cases, according to $\mathtt{n}$ being privileged or not. In the first case, we have:

$$f_{call}(\mathtt{l})(\mathtt{n}) \ = \ \mathit{Perm}(\mathtt{n}) \ = \ f_{call}(\mathtt{l}')(\mathtt{n})$$

  and the inequality trivially holds. In the other case, we have:

$$\begin{array}{rcl} f_{call}(\mathtt{l})(\mathtt{n}) & = & \mathtt{l}_{in}(\mathtt{n}) \\ f_{call}(\mathtt{l}')(\mathtt{n}) & = & \mathtt{l}'_{in}(\mathtt{n}) \end{array}$$

  Hypothesis $\mathtt{l} \sqsubseteq \mathtt{l}'$ implies $\mathtt{l}_{in}(\mathtt{n}) \subseteq \mathtt{l}'_{in}(\mathtt{n})$, hence the inequality is preserved.

- $f_{trans}(\mathsf{l}) \sqsubseteq f_{trans}(\mathsf{l}')$. There are three cases. If $\mathtt{kill}^0(\mathsf{n})$, then $\ell(\mathsf{n}) = \mathtt{check}(\mathsf{P})$ but $\mathsf{P} \notin \mathit{Perm}(\mathsf{n})$. Thus:

$$f_{trans}(\mathsf{l})(\mathsf{n}) \;=\; \varnothing \;=\; f_{trans}(\mathsf{l}')(\mathsf{n})$$

  Otherwise, if $\ell(\mathsf{n}) = \mathtt{check}(\mathsf{P})$ and $\mathsf{P} \in \mathit{Perm}(\mathsf{n})$, then:

$$\begin{aligned} f_{trans}(\mathsf{l})(\mathsf{n}) &= \mathsf{l}_{in}(\mathsf{n}) \cup \{\mathsf{P}\} \\ f_{trans}(\mathsf{l}')(\mathsf{n}) &= \mathsf{l}'_{in}(\mathsf{n}) \cup \{\mathsf{P}\} \end{aligned}$$

  Here $\mathsf{l} \sqsubseteq \mathsf{l}' \implies \mathsf{l}_{in}(\mathsf{n}) \subseteq \mathsf{l}'_{in}(\mathsf{n})$, and this suffices for the $\sqsubseteq$ relation to be preserved. The last case (n is a call) is just as above, after having discarded the singleton $\{\mathsf{P}\}$.

□

Although we have proved that the DP^0 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 $\mathsf{DP}^0 = \langle \mathcal{L}, \mathcal{F}, \mu \rangle$ constructed above can be turned into a distributive framework $\langle \mathcal{L}_\mathsf{N}, \mathcal{F}_\mathsf{E}, \mu_\mathsf{E}, \iota \rangle$. This enables us to use a standard worklist-iteration algorithm to efficiently solve any instance of the DP^0 analysis.

Theorem 5.3.5. DP^0 is a distributive data flow framework.

Proof. Again, for simplicity, we consider the complemented analysis $\overline{\mathsf{DP}^0}$. We have to build a quadruple $\langle \mathcal{L}_\mathsf{N}, \mathcal{F}_\mathsf{E}, \mu_\mathsf{E}, \iota \rangle$ satisfying the requirements in definitions 4.2.1 and 4.2.5.

The local property space is a triple $\langle \mathcal{L}_\mathsf{N}, \sqcup, \bot_{\mathcal{L}_\mathsf{N}} \rangle$, where $\mathcal{L}_\mathsf{N} = \mathcal{P}(\textbf{Permission})$, the join operator $\sqcup$ is the binary union on sets, and $\bot_{\mathcal{L}_\mathsf{N}}$ is the empty set (which is an unit for $\sqcup$, since $\varnothing \cup \mathsf{l} = \mathsf{l} = \mathsf{l} \cup \varnothing$). Since $\cup$ is idempotent, commutative and associative, this definition actually makes $\mathcal{L}_\mathsf{N}$ a join semi-lattice. By the fact that **Permission** is finite for any control flow graph, it clearly follows that $\mathcal{L}_\mathsf{N}$ satisfies the ascending chain condition.

Moreover, a standard construction equips $\mathcal{L}_\mathsf{N}$ with a *complete lattice* structure:

$$\mathcal{L}_\mathsf{N} \;=\; \langle \mathcal{L}_\mathsf{N},\, \sqsubseteq,\, \bigsqcup,\, \bigsqcap,\, \bot_{\mathcal{L}_\mathsf{N}},\, \top_{\mathcal{L}_\mathsf{N}} \rangle$$

where the partial order $\sqsubseteq$ is the inclusion relation $\subseteq$ on sets, and each subset X of $\mathcal{L}_\mathsf{N}$ has least upper bound $\bigsqcup \mathsf{X} = \bigcup \mathsf{X}$ and greatest lower bound $\bigsqcap \mathsf{X} = \bigcap \mathsf{X}$. Furthermore, the bottom element is $\bot_{\mathcal{L}_\mathsf{N}} = \bigsqcup \varnothing = \varnothing$, and the top element is $\top_{\mathcal{L}_\mathsf{N}} = \bigsqcap \mathcal{L}_\mathsf{N} = \textbf{Permission}$.

Given a control flow graph G, the set of local transfer functions $f_\mathsf{E} = \mu_\mathsf{E}(\mathsf{G})$ is defined in Table 5.4. Again, if each $f \in f_\mathsf{E}$ is distributive regardless of G, we can take $\mathcal{F}_\mathsf{E}$ to be the space of distributive functions over $\mathcal{L}_\mathsf{N}$. By definition of distributivity, we have for prove that, for any $\mathsf{l}, \mathsf{l}' \in \mathcal{L}_\mathsf{N}$ and $(\mathsf{m}, \mathsf{n}) \in \mathsf{E}_\rho$:

$$f_{(\mathsf{m},\mathsf{n})}(\mathsf{l} \sqcup \mathsf{l}') \;=\; f_{(\mathsf{m},\mathsf{n})}(\mathsf{l}) \;\sqcup\; f_{(\mathsf{m},\mathsf{n})}(\mathsf{l}')$$

We have to deal with four cases:

- case [*entry*]: if $\bullet \longrightarrow \mathsf{n}$, then $f_{\bullet \longrightarrow n}(\mathsf{l}) = \mathit{Perm}(\mathsf{n})$ for any $\mathsf{l} \in \mathcal{L}_\mathsf{N}$. Thus:

$$f_{\bullet \longrightarrow n}(\mathsf{l} \sqcup \mathsf{l}') \;=\; \mathit{Perm}(\mathsf{n}) \;=\; f_{\bullet \longrightarrow n}(\mathsf{l}) \;\sqcup\; f_{\bullet \longrightarrow n}(\mathsf{l}')$$

  follows by idempotency of $\sqcup$.

$$
\begin{array}{lcl}
f_{\bullet\longrightarrow n}(\mathtt{l}) & = & \mathit{Perm}(\mathtt{n}) \\[2ex]
f_{m\longrightarrow n}(\mathtt{l}) & = & \mathit{Perm}(\mathtt{n}) \cap \begin{cases} \mathit{Perm}(\mathtt{m}) & \text{if } \mathit{Priv}(\mathtt{m}) \\ \mathtt{l} & \text{otherwise} \end{cases} \\[2ex]
f_{m\dashrightarrow n}(\mathtt{l}) & = & \begin{cases} \varnothing & \text{if } \mathtt{kill}^0(\mathtt{l},\mathtt{m}) \\ \mathtt{l} \cup \{\mathtt{P}\} & \text{if } \ell(\mathtt{m}) = \mathtt{check}(\mathtt{P}) \text{ and } \neg\mathtt{kill}^0(\mathtt{l},\mathtt{m}) \\ \mathtt{l} & \text{otherwise} \end{cases} \\[2ex]
f_{m\hookrightarrow n}(\mathtt{l}) & = & \varnothing \\[2ex]
\mathtt{kill}^0(\mathtt{l},\mathtt{n}) & =_{def} & \ell(\mathtt{n}) = \mathtt{check}(\mathtt{P}) \text{ and } \mathtt{P} \notin \mathit{Perm}(\mathtt{n})
\end{array}
$$

Table 5.4: Local transfer functions for the DP^0 Analysis.

- case [*call*]: if $\mathtt{m} \longrightarrow \mathtt{n}$, we have two sub-cases, depending on $\mathtt{m}$ being privileged or not. In the first case, $f_{m\longrightarrow n}(\mathtt{l}) = \mathit{Perm}(\mathtt{n}) \cap \mathit{Perm}(\mathtt{m})$ for any $\mathtt{l} \in \mathcal{L}_{\mathsf{N}}$. Then:

$$f_{m\longrightarrow n}(\mathtt{l} \sqcup \mathtt{l}') \;=\; \mathit{Perm}(\mathtt{n}) \cap \mathit{Perm}(\mathtt{m}) \;=\; f_{m\longrightarrow n}(\mathtt{l}) \sqcup f_{m\longrightarrow n}(\mathtt{l}')$$

  follows by idempotency of $\sqcup$. Otherwise, if $\mathtt{m}$ is not privileged:

$$
\begin{array}{rcl}
f_{m\longrightarrow n}(\mathtt{l} \sqcup \mathtt{l}') & = & \mathit{Perm}(\mathtt{n}) \cap (\mathtt{l} \cup \mathtt{l}') \\
& = & (\mathit{Perm}(\mathtt{n}) \cap \mathtt{l}) \cup (\mathit{Perm}(\mathtt{n}) \cap \mathtt{l}') \\
& = & f_{m\longrightarrow n}(\mathtt{l}) \sqcup f_{m\longrightarrow n}(\mathtt{l}')
\end{array}
$$

  follows by distributivity of $\cap$ over $\cup$.

- case [*transfer*]: if $\mathtt{m} \dashrightarrow \mathtt{n}$, we have three sub-cases, because the predicate $\mathtt{kill}^0$ only depends on $\mathtt{m}$. If $\mathtt{kill}^0(\mathtt{l},\mathtt{m})$ then $\ell(\mathtt{m}) = \mathtt{check}(\mathtt{P})$ but $\mathtt{P} \notin \mathit{Perm}(\mathtt{m})$. Therefore:

$$f_{m\dashrightarrow n}(\mathtt{l} \sqcup \mathtt{l}') \;=\; \varnothing \;=\; f_{m\dashrightarrow n}(\mathtt{l}) \sqcup f_{m\dashrightarrow n}(\mathtt{l}')$$

  follows by idempotency of $\sqcup$. Otherwise, if $\ell(\mathtt{m}) = \mathtt{check}(\mathtt{P})$ and $\mathtt{P} \in \mathit{Perm}(\mathtt{m})$:

$$
\begin{array}{rcl}
f_{m\dashrightarrow n}(\mathtt{l} \sqcup \mathtt{l}') & = & (\mathtt{l} \cup \mathtt{l}') \cup \{\mathtt{P}\} \\
& = & (\mathtt{l} \cup \{\mathtt{P}\}) \cup (\mathtt{l}' \cup \{\mathtt{P}\}) \\
& = & f_{m\dashrightarrow n}(\mathtt{l}) \sqcup f_{m\dashrightarrow n}(\mathtt{l}')
\end{array}
$$

  follows by associativity and idempotency of $\cup$. The last case ($\mathtt{m}$ is not a check node) is just as above, after having discarded the singleton $\{\mathtt{P}\}$.

- case [*return*]: if $\mathtt{m} \hookrightarrow \mathtt{n}$, then $f_{m\hookrightarrow n}(\mathtt{l}) = \varnothing$ for any $\mathtt{l} \in \mathcal{L}_{\mathsf{N}}$. Therefore:

$$f_{m\hookrightarrow n}(\mathtt{l} \sqcup \mathtt{l}') \;=\; \varnothing \;=\; f_{m\hookrightarrow n}(\mathtt{l}) \sqcup f_{m\hookrightarrow n}(\mathtt{l}')$$

  follows by idempotency of $\sqcup$.

□

Lemma 5.3.6. Let $\langle \delta_{in}, \delta_{out}, \delta_{call}, \delta_{trans} \rangle$ be the MFP solution to DP^0. Then:

$$\mathsf{P} \in \overline{\delta}_{in}(\mathsf{n}) \quad \Longrightarrow \quad \mathsf{P} \in \mathit{Perm}(\mathsf{n})$$

for any $\mathsf{n} \in \mathsf{N}$.

Proof. Let f be the global transfer function for the complemented analysis $\overline{\mathsf{DP}^0}$. Then we know that the MFP solution to $\overline{\mathsf{DP}^0}$ is the least upper bound of the (finite) chain:

$$\bot_{\mathcal{L}} \sqsubseteq f(\bot_{\mathcal{L}}) \sqsubseteq f^2(\bot_{\mathcal{L}}) \sqsubseteq \cdots \sqsubseteq f^{\mathsf{k}}(\bot_{\mathcal{L}}) = f^{\mathsf{k}+1}(\bot_{\mathcal{L}})$$

Now let $\mathsf{l} = f^{\mathsf{i}}(\bot_{\mathcal{L}})$ for some index $\mathsf{i} \in \mathbb{N}$. If we define:

$$\phi(\mathsf{l}) \quad \equiv \quad \forall \mathsf{n} \in \mathsf{N}.\ \mathsf{P} \in \mathsf{l}_{in}(\mathsf{n}) \implies \mathsf{P} \in \mathit{Perm}(\mathsf{n})$$

then we can use a straightforward mathematical induction to prove ϕ is true on the least $\overline{\mathsf{DP}^0}$-solution. The base case asks for $\phi(\bot_{\mathcal{L}})$: this is trivially true, since $\bot_{\mathcal{L}_{in}}(\mathsf{n}) = \varnothing$ for any node $\mathsf{n} \in \mathsf{N}$. For the inductive case, assume $\phi(\mathsf{l})$ is true: we prove that this actually implies $\phi(f(\mathsf{l}))$. Take $\mathsf{P} \in f_{in}(\mathsf{l})(\mathsf{n})$. Then, we have:

$$\mathsf{P} \in \bigcup_{(\mathsf{m},\mathsf{n}) \in \mathsf{E}} \mathsf{l}_{out}(\mathsf{m},\mathsf{n})$$

Now, in order to prove that $\forall (\mathsf{m},\mathsf{n}) \in \mathsf{E}.\ \mathsf{P} \in \mathsf{l}_{out}(\mathsf{m},\mathsf{n}) \implies \mathsf{P} \in \mathit{Perm}(\mathsf{n})$, we have to distinguish between the three kinds of (abstract) edges:

- if $\bullet\!\!\longrightarrow \mathsf{n}$, then $\mathsf{P} \in \mathit{Perm}(\mathsf{n})$, and the statement clearly holds.
- if $\mathsf{m} \longrightarrow \mathsf{n}$, by monotonicity of f and definition of f_{out} we have:

  $$\mathsf{l}_{out}(\mathsf{m},\mathsf{n}) \subseteq f_{out}(\mathsf{l})(\mathsf{m},\mathsf{n}) = \mathsf{l}_{call}(\mathsf{m}) \cap \mathit{Perm}(\mathsf{n})$$

  hence $\mathsf{P} \in \mathit{Perm}(\mathsf{n})$.
- if $\mathsf{m} \dashrightarrow \mathsf{n}$, then:

  $$\mathsf{l}_{out}(\mathsf{m},\mathsf{n}) \subseteq f_{out}(\mathsf{l})(\mathsf{m},\mathsf{n}) = \mathsf{l}_{trans}(\mathsf{m}) \subseteq f_{trans}(\mathsf{l})(\mathsf{m})$$

  In this case we have just two possibilities: $\mathsf{P} \in \mathsf{l}_{in}(\mathsf{m})$, or $\mathsf{P} = \mathsf{P}'$ if $\ell(\mathsf{m}) = \mathtt{check}(\mathsf{P}')$. In the first case, the hypothesis $\phi(\mathsf{l})$ implies $\mathsf{P} \in \mathit{Perm}(\mathsf{m})$, and we can deduce $\mathsf{P} \in \mathit{Perm}(\mathsf{n})$ by the fact that both m and n lie in the same protection domain. The second case can only happen if $\mathsf{P}' \in \mathit{Perm}(\mathsf{m})$, and the same argument used above still works.

□

Lemma 5.3.7. The MFP solution to DP^0 is non-trivial.

Proof. Let δ be the greatest DP^0-solution for G and n a node. Since, by duality, $\overline{\delta}$ is the least $\overline{\mathsf{DP}^0}$-solution, lemma 5.3.6 tells $\mathsf{P} \in \overline{\delta}_{in}(\mathsf{n}) \implies \mathsf{P} \in \mathit{Perm}(\mathsf{n})$. This indeed implies non-triviality of δ, as $\mathsf{P} \in \overline{\delta}_{call}(\mathsf{n}) \implies \mathsf{P} \in \overline{\delta}_{in}(\mathsf{n}) \vee \mathsf{P} \in \mathit{Perm}(\mathsf{n})$. □

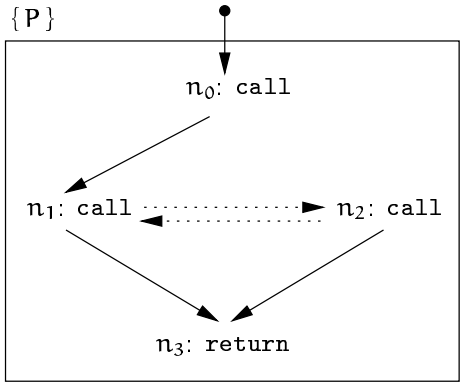

Figure 5.1: Control flow graph for counterexample 5.3.8

Counterexample 5.3.8. Not all DP^0-solutions are non-trivial.

Proof. Consider the control flow graph in Fig. 5.1. By direct computation, we have:

$$\begin{aligned}\overline{\delta}_{in}(n_1) &= \overline{\delta}_{out}(n_0,n_1) \cup \overline{\delta}_{out}(n_2,n_1) = \left(\overline{\delta}_{call}(n_0) \cap \{P\}\right) \cup \overline{\delta}_{trans}(n_2)\\ &= \left(\overline{\delta}_{in}(n_0) \cap \{P\}\right) \cup \overline{\delta}_{in}(n_2) = \left(\{P\} \cap \{P\}\right) \cup \overline{\delta}_{out}(n_1,n_2)\\ &= \{P\} \cup \overline{\delta}_{trans}(n_1) = \{P\} \cup \overline{\delta}_{in}(n_1)\end{aligned}$$

Note that the equation $\overline{\delta}_{in}(n_1) = \{P\} \cup \overline{\delta}_{in}(n_1)$ is solved by any superset of $\{P\}$. Define:

$$\begin{aligned}\overline{\delta}(n_0) &= \{P\}\\ \overline{\delta}(n_i) &= \{P, P'\} \quad \text{for } i = 1..3\end{aligned}$$

Then, δ is a *trivial* fixed point solution to DP^0: in fact, $P' \notin \mathit{Perm}(n_1)$ but $P' \notin \delta(n_1)$. □

5.4 The GP^0 Analysis

$$
\begin{aligned}
\mathsf{GP}^0_{in}(\mathtt{n}) &= \bigcap_{(\mathtt{m},\mathtt{n})\in \mathsf{E}} \mathsf{GP}^0_{out}(\mathtt{m},\mathtt{n}) \\
\mathsf{GP}^0_{out}(\mathtt{m},\mathtt{n}) &= \begin{cases} \mathit{Perm}(\mathtt{n}) & \text{if } \bullet\!\longrightarrow \mathtt{n} \\ \mathsf{GP}^0_{call}(\mathtt{m}) \cap \mathit{Perm}(\mathtt{n}) & \text{if } \mathtt{m} \longrightarrow \mathtt{n} \\ \mathsf{GP}^0_{trans}(\mathtt{m}) & \text{if } \mathtt{m} \dashrightarrow \mathtt{n} \end{cases} \\
\mathsf{GP}^0_{call}(\mathtt{n}) &= \begin{cases} \mathit{Perm}(\mathtt{n}) & \text{if } \mathit{Priv}(\mathtt{n}) \\ \mathsf{GP}^0_{in}(\mathtt{n}) & \text{otherwise} \end{cases} \\
\mathsf{GP}^0_{trans}(\mathtt{n}) &= \begin{cases} \varnothing & \text{if } \mathtt{kill}^0(\mathtt{n}) \\ \mathsf{GP}^0_{in}(\mathtt{n}) \cup \{\mathsf{P}\} & \text{if } \ell(\mathtt{n}) = \mathtt{check}(\mathsf{P}) \text{ and } \neg\mathtt{kill}^0(\mathtt{n}) \\ \mathsf{GP}^0_{in}(\mathtt{n}) & \text{otherwise} \end{cases}
\end{aligned}
$$

Table 5.5: The GP^0 Analysis.

The 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 $\mathtt{n}$ enforcing a permission P kills all the permissions at its entry if $\mathsf{P} \notin \mathit{Perm}(\mathtt{n})$; 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 $\mathtt{n}$ always belong to $\mathit{Perm}(\mathtt{n})$.

Lemma 5.4.1. Let $\langle \gamma_{in}, \gamma_{out}, \gamma_{call}, \gamma_{trans} \rangle \models \mathsf{GP}^0(\mathsf{G})$. Then:

$$\mathsf{P} \in \gamma_{in}(\mathtt{n}) \quad \Longrightarrow \quad \sigma \vdash \mathsf{JDK}(\mathsf{P}) \ \wedge \ \mathsf{P} \in \mathit{Perm}(\mathtt{n})$$

for any G-reachable state $\sigma : \mathtt{n}$.

Proof. We proceed by induction on the derivation used to estabilish $\mathsf{G} \vdash \sigma : \mathtt{n}$. The base case corresponds to our single axiom:

$$\frac{\mathtt{n} \in \mathsf{N}_{entry}}{[] \rhd [\mathtt{n}]}$$

Here rule $\mathsf{JDK}_{\varnothing}$ ensures that $[] \vdash \mathsf{JDK}(\mathsf{P})$ for any permission P, and $\mathsf{P} \in \mathit{Perm}(\mathtt{n})$ is due to the fact that $\mathsf{P} \in \gamma_{in}(\mathtt{n}) \implies \mathsf{P} \in \gamma_{out}(\bot_{\mathsf{N}}, \mathtt{n}) = \mathit{Perm}(\mathtt{n})$ when $\mathtt{n} \in \mathsf{N}_{entry}$. For the inductive case, we proceed by case analysis on the last rule used to derive $\sigma : \mathtt{n}$, yielding:

- case [*call*]:
$$\frac{\ell(\mathtt{n}') = \mathtt{call} \quad \mathtt{n}' \longrightarrow \mathtt{n}}{\sigma' : \mathtt{n}' \rhd \sigma' : \mathtt{n}' : \mathtt{n}} \qquad \text{where } \sigma = \sigma' : \mathtt{n}'$$
By definition of GP^0_{in}, $\mathsf{P} \in \gamma_{in}(\mathtt{n}) \implies \mathsf{P} \in \gamma_{out}(\mathtt{n}', \mathtt{n})$, that is $\mathsf{P} \in \gamma_{call}(\mathtt{n}')$ and $\mathsf{P} \in \mathit{Perm}(\mathtt{n})$. If $\neg \mathit{Priv}(\mathtt{n}')$, then $\mathsf{P} \in \gamma_{in}(\mathtt{n}')$, so we can apply the inductive hypothesis to deduce $\sigma' \vdash \mathsf{JDK}(\mathsf{P}) \wedge \mathsf{P} \in \mathit{Perm}(\mathtt{n}')$: then rule $\mathsf{JDK}_{\prec}$ ensures $\sigma' : \mathtt{n}' \vdash \mathsf{JDK}(\mathsf{P})$. Otherwise, if $\mathit{Priv}(\mathtt{n}')$, then $\mathsf{P} \in \mathit{Perm}(\mathtt{n}')$: in this case, $\sigma' : \mathtt{n}' \vdash \mathsf{JDK}(\mathsf{P})$ is ensured by rule JDK_{Priv}.
- case [*check*]:
$$\frac{\ell(\mathtt{n}') = \mathtt{check}(\mathsf{P}') \quad \sigma : \mathtt{n}' \vdash \mathsf{JDK}(\mathsf{P}') \quad \mathtt{n}' \dashrightarrow \mathtt{n}}{\sigma : \mathtt{n}' \rhd \sigma : \mathtt{n}}$$
Here $\mathsf{P} \in \gamma_{in}(\mathtt{n}) \implies \mathsf{P} \in \gamma_{trans}(\mathtt{n}')$ and, by the premises of the $\rhd_{check}$ rule, we also have $\mathsf{P}' \in \mathit{Perm}(\mathtt{n}')$. There are two cases: $\mathsf{P} \in \gamma_{in}(\mathtt{n}')$ or $\mathsf{P} = \mathsf{P}'$. In the first case, we can apply the inductive hypothesis to deduce $\sigma \vdash \mathsf{JDK}(\mathsf{P})$ and $\mathsf{P} \in \mathit{Perm}(\mathtt{n}')$. Observe that, by the protection domain constraint, $\mathtt{n}' \dashrightarrow \mathtt{n}$ implies $\mathit{Perm}(\mathtt{n}') = \mathit{Perm}(\mathtt{n})$, thus we can deduce $\mathsf{P} \in \mathit{Perm}(\mathtt{n})$. In the second case, $\sigma : \mathtt{n}' \vdash \mathsf{JDK}(\mathsf{P})$ is assumed in the premises of the $\rhd_{check}$ rule. As check nodes are not privileged, this can be derived only by rule $\mathsf{JDK}_{\prec}$, whose premises are just $\sigma \vdash \mathsf{JDK}(\mathsf{P})$ and $\mathsf{P} \in \mathit{Perm}(\mathtt{n}')$.
- case [*return*]:
$$\frac{\ell(\mathtt{m}) = \mathtt{return} \quad \mathtt{n}' \dashrightarrow \mathtt{n}}{\sigma : \mathtt{n}' : \mathtt{m} \rhd \sigma : \mathtt{n}}$$
Here $\mathsf{P} \in \gamma_{in}(\mathtt{n}) \implies \mathsf{P} \in \gamma_{trans}(\mathtt{n}')$. As, by lemma 3.4.4, it must be $\ell(\mathtt{n}') = \mathtt{call}$, then $\mathsf{P} \in \gamma_{in}(\mathtt{n}')$. By lemma 3.4.1, any derivation of $\sigma : \mathtt{n}' : \mathtt{m}$ is of the form:
$$[] \rhd \cdots \rhd \sigma : \mathtt{n}' \rhd \cdots \rhd \sigma : \mathtt{n}' : \mathtt{m}$$
Therefore, we can apply the inductive hypothesis to $\sigma : \mathtt{n}'$, obtaining $\sigma \vdash \mathsf{JDK}(\mathsf{P})$ and $\mathsf{P} \in \mathit{Perm}(\mathtt{n}')$. Again, this implies $\mathsf{P} \in \mathit{Perm}(\mathtt{n})$, since both $\mathtt{n}$ and $\mathtt{n}'$ lie in the same protection domain.

□

Theorem 5.4.2. Let $\langle \gamma_{in}, \gamma_{out}, \gamma_{call}, \gamma_{trans} \rangle \models GP^0(G)$. For any $n \in N$, define:

$$\gamma(n) \;=\; \gamma_{call}(n)$$

Then γ is a sound 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 $\neg Priv(n)$, then $P \in \gamma_{in}(n)$. Lemma 5.4.1 tells $\sigma \vdash JDK(P)$ and $P \in Perm(n)$: but these are exactly the premises of rule $JDK_{\prec}$, therefore $\sigma : n \vdash JDK(P)$.
- if $Priv(n)$, then $P \in Perm(n)$. Here, $\sigma : n \vdash JDK(P)$ is ensured by rule JDK_{Priv}.

□

Example 5.4.3. Consider the e-commerce application of Fig. 3.1. By direct computation of the analysis at n_{23}, we obtain:

$$\begin{aligned}
\gamma_{in}(n_{23}) &= \gamma_{out}(n_9, n_{23}) \cap \gamma_{out}(n_{13}, n_{23}) \cap \gamma_{out}(n_{20}, n_{23}) \\
&= \big(\gamma_{call}(n_9) \cap \gamma_{call}(n_{13}) \cap \gamma_{call}(n_{20})\big) \cap Perm(n_{23}) \\
&= Perm(n_9) \cap Perm(n_{13}) \cap Perm(n_{20}) \\
&= \{P_{canpay}, P_{credit}, P_{debit}, P_{loan}, P_{read}, P_{write}\}
\end{aligned}$$

This equation admits an unique solution, i.e. the set of permissions associated with the protection domain Bank. Since $P_{read} \in \gamma(n_{23})$, the soundness result for the 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}:

$$\begin{aligned}
\gamma_{in}(n_{25}) &= \gamma_{out}(n_{14}, n_{25}) \cap \gamma_{out}(n_{21}, n_{25}) \\
&= \big(\gamma_{call}(n_{14}) \cap \gamma_{call}(n_{21})\big) \cap Perm(n_{25}) \\
&= Perm(n_{14}) \cap Perm(n_{21}) \\
&= \{P_{canpay}, P_{credit}, P_{debit}, P_{loan}, P_{read}, P_{write}\}
\end{aligned}$$

The full MFP solution to GP^0 for the e-commerce example is in Table A.1.

Theorem 5.4.4. 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 GP^0 analysis are just those defined in Table 5.4. The local property space is dual to the space for the GP^0 analysis: now $\sqcup$ is the binary intersection on sets, while $\bot_{\mathcal{L}_N}$ is the universal set **Permission**. □

Lemma 5.4.5. Let $\langle \gamma_{in}, \gamma_{out}, \gamma_{call}, \gamma_{trans} \rangle \models \mathsf{GP}^0(\mathsf{G})$. Then, for any $\mathsf{n} \in \mathsf{N}$:

$$\mathsf{P} \in \gamma_{in}(\mathsf{n}) \quad \Longrightarrow \quad \mathsf{P} \in \mathit{Perm}(\mathsf{n})$$

Proof. Let $\delta = \langle \delta_{in}, \delta_{out}, \delta_{call}, \delta_{trans} \rangle$ be the MFP solution to DP^0, and assume $\mathsf{P} \in \gamma_{in}(\mathsf{n})$. As $\gamma_{in}(\mathsf{n}) \subseteq \gamma_{call}(\mathsf{n})$ regardless n being privileged or not, theorem 5.2.7 ensures $\mathsf{P} \in \overline{\delta}_{call}(\mathsf{n})$. Then, since δ is non-trivial by lemma 5.3.7, we deduce $\mathsf{P} \in \mathit{Perm}(\mathsf{n})$. □

Lemma 5.4.6. Any GP^0-solution is non-trivial.

Proof. Let $\langle \gamma_{in}, \gamma_{out}, \gamma_{call}, \gamma_{trans} \rangle \models \mathsf{GP}^0(\mathsf{G})$ Then, for any privileged node n, we have exactly $\gamma_{call}(\mathsf{n}) = \mathit{Perm}(\mathsf{n})$. □

5.5 The DP^1 Analysis

$$\overline{DP^1}_{in}(n) = \bigcup_{(m,n)\in E} \overline{DP^1}_{out}(m,n)$$

$$\overline{DP^1}_{out}(m,n) = \begin{cases} Perm(n) & \text{if } \bullet\!\!\longrightarrow n \\ \overline{DP^1}_{call}(m) \cap Perm(n) & \text{if } m \longrightarrow n \\ \overline{DP^1}_{trans}(m) & \text{if } m \dashrightarrow n \end{cases}$$

$$\overline{DP^1}_{call}(n) = \begin{cases} Perm(n) & \text{if } Priv(n) \\ \overline{DP^1}_{in}(n) & \text{otherwise} \end{cases}$$

$$\overline{DP^1}_{trans}(n) = \begin{cases} \varnothing & \text{if } \mathtt{kill}^1(n) \\ \bigcup_{\substack{(m,n)\in E \\ P\in\overline{DP^1}_{out}(m,n)}} \overline{DP^1}_{out}(m,n) & \text{if } \ell(n) = \mathtt{check}(P) \text{ and } \neg\mathtt{kill}^1(n) \\ \overline{DP^1}_{in}(n) & \text{otherwise} \end{cases}$$

$$\mathtt{kill}^1(n) =_{def} \ell(n) = \mathtt{check}(P) \text{ and } P \notin \overline{DP^1}_{in}(n)$$

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 *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.

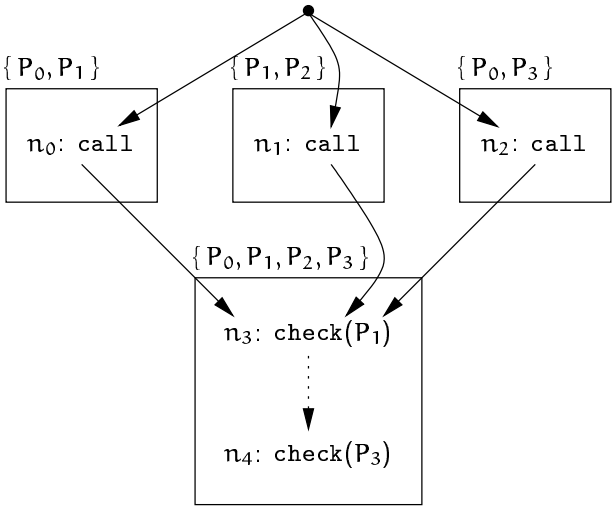

Figure 5.2: Control flow graph for example 5.5.1

Example 5.5.1. Consider the control flow graph in Fig. 5.2. By direct computation of the DP^0 analysis at n_4, we obtain:

$$\begin{aligned}
\overline{\delta^0}_{in}(n_4) &= \overline{\delta^0}_{trans}(n_3) = \overline{\delta^0}_{in}(n_3) \\
&= \overline{\delta^0}_{out}(n_0, n_3) \cup \overline{\delta^0}_{out}(n_1, n_3) \cup \overline{\delta^0}_{out}(n_2, n_3) \\
&= \overline{\delta^0}_{in}(n_0) \cup \overline{\delta^0}_{in}(n_1) \cup \overline{\delta^0}_{in}(n_2) \\
&= \{P_0, P_1\} \cup \{P_1, P_2\} \cup \{P_0, P_3\} = \{P_0, P_1, P_2, P_3\}
\end{aligned}$$

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:

$$\begin{aligned}
\overline{\delta^1}_{out}(n_0, n_3) &= \overline{\delta^1}_{in}(n_0) = \{P_0, P_1\} \\
\overline{\delta^1}_{out}(n_1, n_3) &= \overline{\delta^1}_{in}(n_1) = \{P_1, P_2\} \\
\overline{\delta^1}_{out}(n_2, n_3) &= \overline{\delta^1}_{in}(n_2) = \{P_0, P_3\}
\end{aligned}$$

Then, according to the DP^1 analysis, the permissions non-denied to n_4 are:

$$\begin{aligned}
\overline{\delta^1}_{in}(n_4) &= \overline{\delta^1}_{trans}(n_3) = \bigcup_{(m,n_3)\in E} \{\overline{\delta^1}_{out}(m, n_3) \mid P_1 \in \overline{\delta^1}_{out}(m, n_3)\} \\
&= \overline{\delta^1}_{out}(n_0, n_3) \cup \overline{\delta^1}_{out}(n_1, n_3) = \{P_0, P_1, P_2\}
\end{aligned}$$

This shows that the DP^1 analysis is stricly more accurate than the DP^0. Theorem 5.5.3 will ensure that DP^1 enjoys the soundness property, too.

Lemma 5.5.2. Let $\langle \delta_{in}, \delta_{out}, \delta_{call}, \delta_{trans} \rangle \models \mathsf{DP}^1(\mathsf{G})$, and:

$$[] = \sigma_0 \rhd \sigma_1 \rhd \cdots \rhd \sigma_k = \sigma : n$$

be a derivation. Then:

$$\mathsf{P} \in \delta_{out}(\eta(\sigma_{k-1}, \sigma_k)) \cap \mathit{Perm}(n) \implies \sigma \nvdash \mathsf{JDK}(\mathsf{P})$$

Proof. The proof is carried out by contradiction, assuming $\sigma \vdash \mathsf{JDK}(\mathsf{P})$. Then, we proceed by induction on the length of derivation $[] \rhd \cdots \rhd \sigma_{k-1} \rhd \sigma : n$. The base case corresponds to our single axiom:

$$\frac{n \in \mathsf{N}_{entry}}{[] \rhd [n]}$$

For $n \in \mathsf{N}_{entry}$, we have defined $\eta([], [n]) = (\bot_{\mathsf{N}}, n)$. As $\mathsf{P} \in \delta_{out}(\bot_{\mathsf{N}}, n) \implies \mathsf{P} \notin \mathit{Perm}(n)$, P is prevented from being in $\mathit{Perm}(n)$. 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 rule used to derive $\sigma_{k-1} \rhd \sigma : n$, yielding:

- case [*call*]:

$$\frac{\ell(n') = \texttt{call} \quad n' \longrightarrow n}{\sigma' : n' \rhd \sigma' : n' : n} \qquad \text{where } \sigma_{k-1} = \sigma = \sigma' : n'$$

  Here $\eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma' : n', \sigma' : n' : n) = (n', n) \in \mathsf{E}_{call}$, hence $\mathsf{P} \in \delta_{out}(n', n) \implies \mathsf{P} \in \delta_{call}(n')$ or $\mathsf{P} \notin \mathit{Perm}(n)$. The latter option is prevented by our assumptions about P, then only the former one is considered. If $\neg \mathit{Priv}(n')$, then $\mathsf{P} \in \delta_{in}(n')$, and, using lemma 3.8.2, we obtain $\mathsf{P} \in \delta_{out}(\eta(\sigma_{k-2}, \sigma' : n'))$. Now, assumption $\sigma' : n' \vdash \mathsf{JDK}(\mathsf{P})$ implies $\mathsf{P} \in \mathit{Perm}(n')$, so we can apply the inductive hypothesis to deduce $\sigma' \nvdash \mathsf{JDK}(\mathsf{P})$. This prevents the $\mathsf{JDK}_{\prec}$ rule to be applicable, and a contradiction arises with assumption $\sigma' : n' \vdash \mathsf{JDK}(\mathsf{P})$. Otherwise, if $\mathit{Priv}(n')$, then it should hold $\mathsf{P} \notin \mathit{Perm}(n')$, contradicting again assumption $\sigma' : n' \vdash \mathsf{JDK}(\mathsf{P})$.

- case [*check*]:

$$\frac{\ell(n') = \texttt{check}(\mathsf{P}') \quad \sigma : n' \vdash \mathsf{JDK}(\mathsf{P}') \quad n' \dashrightarrow n}{\sigma : n' \rhd \sigma : n}$$

  Here $\eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma : n', \sigma : n) = (n', n) \in \mathsf{E}_{trans}$, then $\mathsf{P} \in \delta_{out}(n', n) \implies \mathsf{P} \in \delta_{trans}(n')$. As both n and n' lie in the same protection domain, assumption $\mathsf{P} \in \mathit{Perm}(n)$ also ensures $\mathsf{P} \in \mathit{Perm}(n')$. Now, it can be shown that:

$$\mathsf{P} \in \bigcap_{\substack{(m,n') \in \mathsf{E} \\ \mathsf{P}' \notin \delta_{out}(m,n')}} \delta_{out}(m, n') \tag{5.1}$$

  As check nodes are not privileged, by rule $\mathsf{JDK}_{\prec}$ we know that $\sigma : n' \vdash \mathsf{JDK}(\mathsf{P}')$ can only be true if $\sigma \vdash \mathsf{JDK}(\mathsf{P}')$. Then, by applying contrapositively the inductive hypothesis, we deduce that $\mathsf{P}' \notin \delta_{out}(\eta(\sigma_{k-2}, \sigma_{k-1})) \cap \mathit{Perm}(n')$. Now, premise $\sigma : n' \vdash \mathsf{JDK}(\mathsf{P}')$ of the $\rhd_{check}$ rule requires $\mathsf{P}' \in \mathit{Perm}(n')$. Therefore, it is indeed $\mathsf{P}' \notin \delta_{out}(\eta(\sigma_{k-2}, \sigma_{k-1}))$: by equation (5.1), this implies $\mathsf{P} \in \delta_{out}(\eta(\sigma_{k-2}, \sigma_{k-1}))$. Applying again the inductive hypothesis, we finally obtain a contradiction $\sigma \nvdash \mathsf{JDK}(\mathsf{P})$.

- case [*return*]:

$$\frac{\ell(m) = \texttt{return} \quad n' \dashrightarrow n}{\sigma : n' : m \rhd \sigma : n}$$

Here, $\eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma : n' : m, \sigma : n) = (n', n) \in E_{trans}$: thus, $P \in \delta_{out}(n', n) \implies P \in \delta_{trans}(n')$. By lemma 3.4.4, it must be $\ell(n') = \texttt{call}$, then it turns out that $P \in \delta_{in}(n')$. 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 $P \in \delta_{out}(\eta(\sigma_{i-1}, \sigma : n'))$, and, again, assumption $P \in \mathit{Perm}(n)$ implies $P \in \mathit{Perm}(n')$ as both n and n' carry the same permissions. Hence, the inductive hypothesis can be applied to yield $\sigma \not\vdash \mathsf{JDK}(P)$.

□

Theorem 5.5.3. Let $\langle \delta_{in}, \delta_{out}, \delta_{call}, \delta_{trans} \rangle \models \mathsf{DP}^1(G)$. For any $n \in N$, define:

$$\delta(n) \;=\; \delta_{call}(n)$$

Then δ is a sound DP^1-solution.

Proof. Let $\sigma : n$ be a G-reachable state, and $P \in \delta_{call}(n)$. By contradiction, assume $\sigma : n \vdash \mathsf{JDK}(P)$. This clearly implies $P \in \mathit{Perm}(n)$: otherwise, neither the $\mathsf{JDK}_{\prec}$ nor the JDK_{Priv} rules are applicable. Then we have to distinguish between two cases:

- if $\neg \mathit{Priv}(n)$, then $P \in \delta_{in}(n)$. Consider a derivation $[] \rhd \cdots \rhd \sigma_{k-1} \rhd \sigma : n$. By definition of DP^1_{in}, we then have $P \in \delta_{out}(\eta(\sigma_{k-1}, \sigma : n))$, and by lemma 5.5.2 it follows $\sigma \not\vdash \mathsf{JDK}(P)$. Since the $\mathsf{JDK}_{\prec}$ rule is not applicable in this case, a contradiction arises with our assumption $\sigma : n \vdash \mathsf{JDK}(P)$.
- if $\mathit{Priv}(n)$, then $P \notin \mathit{Perm}(n)$. Again, this contradicts $\sigma : n \vdash \mathsf{JDK}(P)$.

□

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:

$$\begin{aligned} \overline{\delta}_{in}(n_2) &= \{P_{canpay}, P_{credit}, P_{debit}\} \\ \overline{\delta}_{in}(n_5) &= \varnothing \end{aligned}$$

Now, by direct computation of the analysis at edges (n_3, n_{11}) and (n_6, n_{11}):

$$\begin{aligned} \overline{\delta}_{out}(n_3, n_{11}) &= \overline{\delta}_{call}(n_3) \cap \mathit{Perm}(n_{11}) = \overline{\delta}_{in}(n_3) \cap \mathit{Perm}(n_{11}) \\ &= \overline{\delta}_{out}(n_2, n_3) \cap \mathit{Perm}(n_{11}) = \overline{\delta}_{trans}(n_2) \cap \mathit{Perm}(n_{11}) \\ &= \overline{\delta}_{in}(n_2) \cap \mathit{Perm}(n_{11}) = \{P_{canpay}, P_{credit}, P_{debit}\} \end{aligned}$$

$$\begin{aligned} \overline{\delta}_{out}(n_6, n_{11}) &= \overline{\delta}_{call}(n_6) \cap \mathit{Perm}(n_{11}) = \overline{\delta}_{in}(n_6) \cap \mathit{Perm}(n_{11}) \\ &= \overline{\delta}_{out}(n_5, n_6) \cap \mathit{Perm}(n_{11}) = \overline{\delta}_{trans}(n_5) \cap \mathit{Perm}(n_{11}) \\ &= \overline{\delta}_{in}(n_5) \cap \mathit{Perm}(n_{11}) = \varnothing \end{aligned}$$

This shows that the security check at n_{11} may be passed by executions reaching n_{11} through (n_3, n_{11}), whereas the check will always fail for any execution flowing through (n_6, n_{11}). This is ensured by by lemma 5.5.2, because:

$$\mathsf{P}_{debit} \in \delta_{out}(n_6, n_{11}) \cap \mathit{Perm}(n_{11}) \quad \Longrightarrow \quad \sigma : n_6 \not\vdash \mathsf{JDK}(\mathsf{P}_{debit})$$

for any derivation $[] \rhd \cdots \rhd \sigma : n_6 \rhd \sigma : n_6 : n_{11}$, and this clearly implies that $\sigma : n_6 : n_{11} \not\vdash \mathsf{JDK}(\mathsf{P}_{debit})$.

The full MFP solution to DP^1 for the e-commerce example is in Table A.1.

Theorem 5.5.5. DP^1 is a monotone data flow analysis.

Proof. The property space for $\overline{\mathsf{DP}^1}$ is exactly the same as that for the $\overline{\mathsf{DP}^0}$ analysis. Let f be the $\overline{\mathsf{DP}^1}$ transfer function, and $l \sqsubseteq l'$. The proof of monotonicity for f differs from the relative proof for the $\overline{\mathsf{DP}^0}$ analysis in just one case, i.e. $f_{trans}(l) \sqsubseteq f_{trans}(l')$ when $\ell(n) = \mathtt{check}(\mathsf{P})$ and $\neg \mathsf{kill}^1(n)$:

$$\begin{aligned}
f_{trans}(l)(n) &= \bigcup_{\substack{(m,n)\in E \\ \mathsf{P}\in l_{out}(m,n)}} l_{out}(m,n) && \text{by def. } f_{trans} \\
&\subseteq \bigcup_{\substack{(m,n)\in E \\ \mathsf{P}\in l'_{out}(m,n)}} l_{out}(m,n) && \text{by hyp. } l_{out} \subseteq l'_{out} \\
&\subseteq \bigcup_{\substack{(m,n)\in E \\ \mathsf{P}\in l'_{out}(m,n)}} l'_{out}(m,n) && \text{by hyp. } l_{out} \subseteq l'_{out} \\
&= f_{trans}(l')(n) && \text{by def. } f_{trans}
\end{aligned}$$

□

Theorem 5.5.6. DP^1 is a monotone data flow framework.

Proof. Again, it is worth considering the complemented analysis $\overline{\mathsf{DP}^1}$. The local property space is a triple $\langle \mathcal{L}_N, \sqcup, \bot_{\mathcal{L}_N} \rangle$, where:

$$\mathcal{L}_N = \mathcal{P}(\textbf{Permission}) \times \mathcal{P}(\textbf{Permission})$$

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 coordinatewise fashion:

$$\langle l_0, l_1 \rangle \sqcup \langle l'_0, l'_1 \rangle = \langle l_0 \cup l'_0,\ l_1 \cup l'_1 \rangle$$

The bottom element is $\bot_{\mathcal{L}_N} = \langle \varnothing, \varnothing \rangle$. Since $\cup$ 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:

$$\mathcal{L}_N = \langle \mathcal{L}_N,\ \sqsubseteq,\ \bigsqcup,\ \bigsqcap,\ \bot_{\mathcal{L}_N},\ \top_{\mathcal{L}_N} \rangle$$

$$
\begin{array}{rcl}
f_{(m,n)}(\langle l_0, l_1\rangle) & = & \langle f^0_{(m,n)},\, f^1_{(m,n)}\rangle(\langle l_0, l_1\rangle) \\
f^0_{\bullet\longrightarrow n}(\langle l_0, l_1\rangle) & = & \mathit{Perm}(n) \\
f^0_{m\longrightarrow n}(\langle l_0, l_1\rangle) & = & \mathit{Perm}(n) \cap \begin{cases} \mathit{Perm}(m) & \text{if } \mathit{Priv}(m) \\ l_1 & \text{otherwise} \end{cases} \\
f^0_{m\dashrightarrow n}(\langle l_0, l_1\rangle) & = & l_1 \\
f^0_{m\hookrightarrow n}(\langle l_0, l_1\rangle) & = & \varnothing \\
f^1_{(m,n)}(\langle l_0, l_1\rangle) & = & \begin{cases} \varnothing & \text{if } \mathtt{kill}^1(l'_0, n) \\ l'_0 & \text{otherwise} \end{cases} \quad \text{where } l'_0 = f^0_{(m,n)}(\langle l_0, l_1\rangle) \\
\mathtt{kill}^1(l, n) & =_{def} & \ell(n) = \mathtt{check}(P) \text{ and } P \notin l
\end{array}
$$

Table 5.7: Local transfer functions for the DP^1 Analysis.

The partial order $\sqsubseteq$ is the determined coordinatewise:

$$\langle l_0, l_1\rangle \sqsubseteq \langle l'_0, l'_1\rangle \qquad \text{iff } l_0 \subseteq l'_0 \text{ and } l_1 \subseteq l'_1$$

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 $f \in f_E$ is monotone regardless of G, we can take $\mathcal{F}_E$ to be the space of monotone functions over $\mathcal{L}_N$. So, let $\langle l_0, l_1\rangle \sqsubseteq \langle l'_0, l'_1\rangle$ be two arbitrary elements of $\mathcal{L}_N$. By definition of monotonicity, we have for prove that, for any $(m, n) \in E_\rho$:

$$f_{(m,n)}(\langle l_0, l_1\rangle) \;\sqsubseteq\; f_{(m,n)}(\langle l'_0, l'_1\rangle)$$

For the $f^0_{(m,n)}$ component, we have to deal with four cases:

- case [*entry*]: if $\bullet\longrightarrow n$, then $f^0_{\bullet\longrightarrow n}(l) = \mathit{Perm}(n)$ for any $l \in \mathcal{L}_N$. Thus:

$$f^0_{\bullet\longrightarrow n}(\langle l_0, l_1\rangle) \;=\; \mathit{Perm}(n) \;=\; f^0_{\bullet\longrightarrow n}(\langle l'_0, l'_1\rangle)$$

- case [*call*]: if $m \longrightarrow n$, we have two cases, depending on m being privileged or not. In the first case, $f^0_{m\longrightarrow n}(l) = \mathit{Perm}(n) \cap \mathit{Perm}(m)$ for any $l \in \mathcal{L}_N$. Then:

$$f^0_{m\longrightarrow n}(\langle l_0, l_1\rangle) \;=\; \mathit{Perm}(n) \cap \mathit{Perm}(m) \;=\; f^0_{m\longrightarrow n}(\langle l'_0, l'_1\rangle)$$

  Otherwise, if m is not privileged:

$$f^0_{m\longrightarrow n}(\langle l_0, l_1\rangle) \;=\; \mathit{Perm}(n) \cap l_1 \;\sqsubseteq\; \mathit{Perm}(n) \cap l'_1 \;=\; f^0_{m\longrightarrow n}(\langle l'_0, l'_1\rangle)$$

  follows by hypothesis $l_1 \sqsubseteq l'_1$.

- case [*transfer*]: if $m \dashrightarrow n$, by hypothesis $l_1 \sqsubseteq l'_1$, we have:

$$f^0_{m \dashrightarrow n}(\langle l_0, l_1\rangle) \;=\; l_1 \;\sqsubseteq\; l'_1 \;=\; f^0_{m \dashrightarrow n}(\langle l'_0, l'_1\rangle)$$

- case [*return*]: if $m \hookrightarrow n$, then $f^0_{m \hookrightarrow n}(l) = \varnothing$ for any $l \in \mathcal{L}_N$. Therefore:

$$f^0_{m \hookrightarrow n}(\langle l_0, l_1\rangle) \;=\; \varnothing \;=\; f^0_{m \hookrightarrow n}(\langle l'_0, l'_1\rangle)$$

For the $f^1_{(m,n)}$ component, let $\widehat{l_0} = f^0_{(m,n)}(\langle l_0, l_1\rangle$ and $\widehat{l'_0} = f^0_{(m,n)}(\langle l'_0, l'_1\rangle$. Observe that, by definition of kill^1, we have, for any $l, l' \in \mathcal{P}$(**Permission**):

$$l \subseteq l' \;\wedge\; \mathsf{kill}^1(l', n) \;\Longrightarrow\; \mathsf{kill}^1(l, n)$$

By monotonicity of f^0, we know $\widehat{l_0} \sqsubseteq \widehat{l'_0}$: therefore, $\mathsf{kill}^1(\widehat{l'_0}, n) \implies \mathsf{kill}^1(\widehat{l_0}, n)$, and we have to deal with the following cases:

- if $\mathsf{kill}^1(\widehat{l'_0}, n)$, then it must be $\mathsf{kill}^1(\widehat{l_0}, n)$, too. Thus:

$$f^1_{(m,n)}(\langle l_0, l_1\rangle) \;=\; \varnothing \;=\; f^1_{(m,n)}(\langle l'_0, l'_1\rangle)$$

- if $\mathsf{kill}^1(\widehat{l_0}, n)$, but $\neg\mathsf{kill}^1(\widehat{l'_0}, n)$, then:

$$f^1_{(m,n)}(\langle l_0, l_1\rangle) \;=\; \varnothing \;\sqsubseteq\; \widehat{l'_0} \;=\; f^1_{(m,n)}(\langle l'_0, l'_1\rangle)$$

- otherwise, if $\neg\mathsf{kill}^1(\widehat{l_0}, n)$, then:

$$f^1_{(m,n)}(\langle l_0, l_1\rangle) \;=\; \widehat{l_0} \;\sqsubseteq\; \widehat{l'_0} \;=\; f^1_{(m,n)}(\langle l'_0, l'_1\rangle)$$

□

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 $\overline{\mathsf{DP}^1}$ 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:

$$\begin{aligned}
\overline{\delta}_{out}(n_2, n_3) &= \overline{\delta}_{call}(n_2) \;=\; \overline{\delta}_{in}(n_2) \\
&= \overline{\delta}_{out}(n_0, n_2) \cup \overline{\delta}_{out}(n_1, n_2) \\
&= \left(\overline{\delta}_{call}(n_0) \cap \{P_0, P_1\}\right) \cup \left(\overline{\delta}_{call}(n_1) \cap \{P_0, P_1\}\right) \\
&= \left(\overline{\delta}_{in}(n_0) \cup \overline{\delta}_{in}(n_1)\right) \cap \{P_0, P_1\} \\
&= \left(\{P_0\} \cup \{P_1\}\right) \cap \{P_0, P_1\} \;=\; \{P_0, P_1\}
\end{aligned}$$

Since $P_0 \in \overline{\delta}_{out}(n_2, n_3)$, then $\neg\mathsf{kill}^1(n_3)$, and:

$$\mathrm{MFP}(n_4) \;=\; \overline{\delta}_{in}(n_4) \;=\; \overline{\delta}_{out}(n_3, n_4) \;=\; \overline{\delta}_{trans}(n_3) \;=\; \{P_0, P_1\}$$

For the MOP solution, we have two (abstract) paths leading to n_4, i.e. $\pi_0 = \langle n_0, n_2, n_3, n_4\rangle$ and $\pi_1 = \langle n_1, n_2, n_3, n_4\rangle$. Given a path $\langle n_0, \ldots, n_k\rangle$, we write $\iota \mapsto_{n_0} l_0 \mapsto_{n_1} \cdots \mapsto_{n_k} l_k$ for the sequence $\langle \iota, f_{\langle n_0\rangle}(\iota), \ldots f_{\langle n_0,\ldots,n_k\rangle}(\iota)\rangle$. Then the values of $f_{\pi_0}(\iota)$ and $f_{\pi_1}(\iota)$ are accumulated along the paths π_0 and π_1 as follows:

$$\begin{array}{lllllll}
\iota & \longmapsto_{n_0} & \{P_0\} & \longmapsto_{n_2} & \{P_0\} & \longmapsto_{n_3} & \{P_0\} \\
\iota & \longmapsto_{n_1} & \{P_1\} & \longmapsto_{n_2} & \{P_1\} & \longmapsto_{n_3} & \varnothing
\end{array}$$

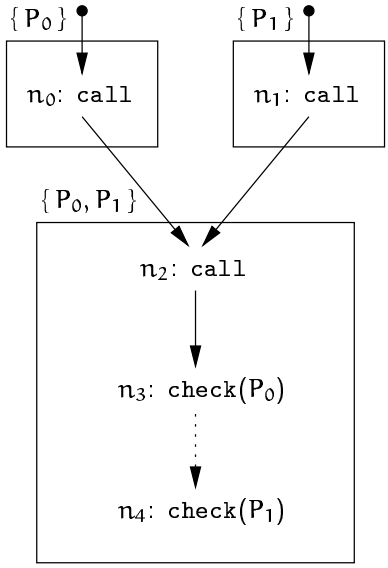

Figure 5.3: Control flow graph for counterexample 5.5.7

Thus, the MOP solution at node n_4 is:

$$\mathrm{MOP}(n_4) \;=\; f_{\pi_0}(\iota) \sqcup f_{\pi_1}(\iota) \;=\; \{P_0\} \sqcup \varnothing \;=\; \{P_0\}$$

Since $\mathrm{MFP}(n_4) \neq \mathrm{MOP}(n_4)$, the DP^1 analysis cannot be distributive. □

Theorem 5.5.8. Let δ^0 and δ^1 be the MFP solutions for the DP^0 and DP^1 analyses, respectively. Then δ^1 is *more accurate* than δ^0, i.e.:

$$\delta^1 \;\sqsubseteq\; \delta^0$$

Proof. Let f and g be the global transfer functions for the complemented analyses $\overline{\mathsf{DP}^0}$ and $\overline{\mathsf{DP}^1}$, respectively . We know that the MFP solutions for $\overline{\mathsf{DP}^0}$ and $\overline{\mathsf{DP}^1}$ are the least upper bounds of the (finite) chains:

$$\begin{array}{ccccccccccc}
\bot_{\mathcal{L}} & \sqsubseteq & f(\bot_{\mathcal{L}}) & \sqsubseteq & f^2(\bot_{\mathcal{L}}) & \sqsubseteq & \cdots & \sqsubseteq & f^k(\bot_{\mathcal{L}}) & = & f^{k+1}(\bot_{\mathcal{L}}) \\
\bot_{\mathcal{L}} & \sqsubseteq & g(\bot_{\mathcal{L}}) & \sqsubseteq & g^2(\bot_{\mathcal{L}}) & \sqsubseteq & \cdots & \sqsubseteq & g^h(\bot_{\mathcal{L}}) & = & g^{h+1}(\bot_{\mathcal{L}})
\end{array}$$

Now define:

$$\phi(l^0, l^1) \;\equiv\; l^0 \sqsupseteq l^1$$

The base case $\phi(\bot_{\mathcal{L}}, \bot_{\mathcal{L}})$ is true, as $\bot_{\mathcal{L}_{in}}(n) = \bot_{\mathcal{L}_{out}}(m,n) = \bot_{\mathcal{L}_{call}}(n) = \bot_{\mathcal{L}_{trans}}(n) = \varnothing$ for any $n, m \in \mathsf{N}$, and clearly $\varnothing \sqsupseteq \varnothing$. For the inductive case, we will prove:

$$l^0 = f^i(\bot_{\mathcal{L}}) \;\wedge\; l^1 = g^j(\bot_{\mathcal{L}}) \;\wedge\; \phi(l^0, l^1) \qquad \Longrightarrow \qquad \phi(f^2(l^0), g(l^1))$$

This clearly implies that $g^h(l^1) \sqsubseteq f^{2k}(l^0) = f^k(l^0)$. Note also that we are allowed to prove just $\phi(f(l^0), g(l^1))$ whenever possible. In fact, by monotonicity of f, we have:

$$f^2(l^0) \;=\; f(f(l^0)) \;\sqsupseteq\; f(l^0)$$

provided $l^0 = f^i(\bot_{\mathcal{L}})$. 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:

- $g_{in}(\mathsf{l}^1) \sqsubseteq f_{in}(\mathsf{l}^0)$. Here we have:

$$\begin{aligned} f_{in}(\mathsf{l}^0)(\mathtt{n}) &= \bigcup_{(\mathtt{m},\mathtt{n})\in\mathsf{E}} \mathsf{l}^0_{out}(\mathtt{m},\mathtt{n}) \\ g_{in}(\mathsf{l}^1)(\mathtt{n}) &= \bigcup_{(\mathtt{m},\mathtt{n})\in\mathsf{E}} \mathsf{l}^1_{out}(\mathtt{m},\mathtt{n}) \end{aligned}$$

  Here hypothesis $\phi(\mathsf{l}^0,\mathsf{l}^1)$ ensures $\mathsf{l}^0_{out}(\mathtt{m},\mathtt{n}) \supseteq \mathsf{l}^1_{out}(\mathtt{m},\mathtt{n})$ for any $(\mathtt{m},\mathtt{n}) \in \mathsf{E}$, and the inequality is preserved by the union operator on sets.

- $g_{out}(\mathsf{l}^1) \sqsubseteq f_{out}(\mathsf{l}^0)$. Here we have to distinguish between the three kinds of edges. If $\bullet\longrightarrow \mathtt{n}$, then:

$$f_{out}(\mathsf{l}^0)(\mathtt{m},\mathtt{n}) = \mathit{Perm}(\mathtt{n}) = f_{out}(\mathsf{l}^1)(\mathtt{m},\mathtt{n})$$

  If $\mathtt{m} \longrightarrow \mathtt{n}$, then:

$$\begin{aligned} f_{out}(\mathsf{l}^0)(\mathtt{m},\mathtt{n}) &= \mathsf{l}^0_{call}(\mathtt{m}) \cap \mathit{Perm}(\mathtt{n}) \\ g_{out}(\mathsf{l}^1)(\mathtt{m},\mathtt{n}) &= \mathsf{l}^1_{call}(\mathtt{m}) \cap \mathit{Perm}(\mathtt{n}) \end{aligned}$$

  and the inequality holds, because $\mathsf{l}^0 \sqsupseteq \mathsf{l}^1$ ensures $\mathsf{l}^0_{call}(\mathtt{m}) \supseteq \mathsf{l}^1_{call}(\mathtt{m})$ for any node $\mathtt{m} \in \mathsf{N}$. Otherwise, if $\mathtt{m} \dashrightarrow \mathtt{n}$:

$$\begin{aligned} f_{out}(\mathsf{l}^0)(\mathtt{m},\mathtt{n}) &= \mathsf{l}^0_{trans}(\mathtt{m}) \\ g_{out}(\mathsf{l}^1)(\mathtt{m},\mathtt{n}) &= \mathsf{l}^1_{trans}(\mathtt{m}) \end{aligned}$$

  Again the inequality is preserved, because $\mathsf{l}^0 \sqsupseteq \mathsf{l}^1$ ensures $\mathsf{l}^0_{trans}(\mathtt{m}) \supseteq \mathsf{l}^1_{trans}(\mathtt{m})$.

- $g_{call}(\mathsf{l}^1) \sqsubseteq f_{call}(\mathsf{l}^0)$. Here there are two cases, according to $\mathtt{n}$ being privileged or not. In the first case, we have:

$$f_{call}(\mathsf{l}^0)(\mathtt{n}) = \mathit{Perm}(\mathtt{n}) = g_{call}(\mathsf{l}^1)(\mathtt{n})$$

  and the inequality trivially holds. In the other case, we have:

$$\begin{aligned} f_{call}(\mathsf{l}^0)(\mathtt{n}) &= \mathsf{l}^0_{in}(\mathtt{n}) \\ g_{call}(\mathsf{l}^1)(\mathtt{n}) &= \mathsf{l}^1_{in}(\mathtt{n}) \end{aligned}$$

  Assumption $\mathsf{l}^0 \sqsupseteq \mathsf{l}^1$ implies $\mathsf{l}^0_{in}(\mathtt{n}) \supseteq \mathsf{l}^1_{in}(\mathtt{n})$, hence the inequality is preserved.

- $g_{trans}(\mathsf{l}^1) \sqsubseteq f^2_{trans}(\mathsf{l}^0)$. The most interesting case is $\ell(\mathtt{n}) = \mathtt{check}(\mathsf{P})$ and $\neg\mathsf{kill}^1(\mathtt{n})$. In such case, lemma 5.5.9 ensures $\mathsf{P} \in \mathit{Perm}(\mathtt{n})$, hence we have also $\neg\mathsf{kill}^0(\mathtt{n})$. Thus:

$$\begin{aligned} g_{trans}(\mathsf{l}^1)(\mathtt{n}) &= \bigcup_{\substack{(\mathtt{m},\mathtt{n})\in\mathsf{E} \\ \mathsf{P}\in\mathsf{l}^1_{out}(\mathtt{m},\mathtt{n})}} \mathsf{l}^1_{out}(\mathtt{m},\mathtt{n}) && \text{by def. } g_{trans} \\ &= \bigcup_{\substack{(\mathtt{m},\mathtt{n})\in\mathsf{E} \\ \mathsf{P}\in\mathsf{l}^1_{out}(\mathtt{m},\mathtt{n})}} \mathsf{l}^1_{out}(\mathtt{m},\mathtt{n}) \cup \{\mathsf{P}\} && \text{by a property of } \bigcup \\ &\subseteq \bigcup_{(\mathtt{m},\mathtt{n})\in\mathsf{E}} \mathsf{l}^1_{out}(\mathtt{m},\mathtt{n}) \cup \{\mathsf{P}\} && \text{by a property of } \bigcup \\ &\subseteq \bigcup_{(\mathtt{m},\mathtt{n})\in\mathsf{E}} \mathsf{l}^0_{out}(\mathtt{m},\mathtt{n}) \cup \{\mathsf{P}\} && \text{by hypothesis } \mathsf{l}^0_{out} \sqsupseteq \mathsf{l}^1_{out} \\ &= f_{in}(\mathsf{l}^0)(\mathtt{n}) \cup \{\mathsf{P}\} && \text{by def. } f_{in} \\ &= f_{trans}(f(\mathsf{l}^0))(\mathtt{n}) && \text{by def. } f_{trans} \end{aligned}$$

Otherwise, if $\ell(n) = \texttt{check}(P)$ and $\texttt{kill}^1(n)$, we have $P \notin l^1_{in}(n)$. Here we have to distinguish between two cases, i.e. $P \in \mathit{Perm}(n)$ or not. In the first case, we have:

$$f_{trans}(l^0)(n) \;=\; l^0_{in}(n) \cup \{P\} \quad \supseteq \quad \varnothing \;=\; g_{trans}(l^1)(n)$$

If $P \notin \mathit{Perm}(n)$, then $\texttt{kill}^0(n)$ holds too. Thus:

$$f_{trans}(l^0)(n) \;=\; \varnothing \;=\; g_{trans}(l^1)(n)$$

The last case $\ell(m) = \texttt{call}$ involves $f_{trans}(l^0)(n) = l^0_{in}(n)$ and $g_{trans}(l^1)(n) = l^1_{in}(n)$, hence assumption $\phi(l^0, l^1)$ does the job.

□

Lemma 5.5.9. Let $\langle \delta_{in}, \delta_{out}, \delta_{call}, \delta_{trans} \rangle$ be the MFP solution to DP^1. Then:

$$P \in \overline{\delta}_{in}(n) \quad \Longrightarrow \quad P \in \mathit{Perm}(n)$$

for any $n \in \mathsf{N}$.

Proof. The proof is similar to the proof of theorem 5.3.6. □

Lemma 5.5.10. The MFP solution to DP^1 is non-trivial.

Proof. The result easily follows from 5.5.9, as it is shown in 5.3.7. □

5.6 The GP^1 Analysis

$$GP^1_{in}(n) = \bigcap_{(m,n)\in E} GP^1_{out}(m,n)$$

$$GP^1_{out}(m,n) = \begin{cases} Perm(n) & \text{if } \bullet\longrightarrow n \\ GP^1_{call}(m) \cap Perm(n) & \text{if } m \longrightarrow n \\ GP^1_{trans}(m) & \text{if } m \dashrightarrow n \end{cases}$$

$$GP^1_{call}(n) = \begin{cases} Perm(n) & \text{if } Priv(n) \\ GP^1_{in}(n) & \text{otherwise} \end{cases}$$

$$GP^1_{trans}(n) = \begin{cases} \varnothing & \text{if } \mathtt{kill}^1(n) \\ \bigcap_{\substack{(m,n)\in E \\ P\in \overline{DP^1}_{out}(m,n)}} GP^1_{out}(m,n) \cup \{P\} & \text{if } \ell(n) = \mathtt{check}(P) \text{ and } \neg\mathtt{kill}^1(n) \\ GP^1_{in}(n) & \text{otherwise} \end{cases}$$

Table 5.8: The GP^1 Analysis.

The GP^1 analysis is defined in Table 5.8. In only differs from the GP^0 analysis in the the set of granted permissions propagated through check nodes.

In the GP^0 analysis, a check node n propagates all the permissions at its entry, unless the permission it enforces does not belong to *Perm(n)*. This is a main source of degradation, as example 5.6.1 shows.

The 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 GP^0 analysis.

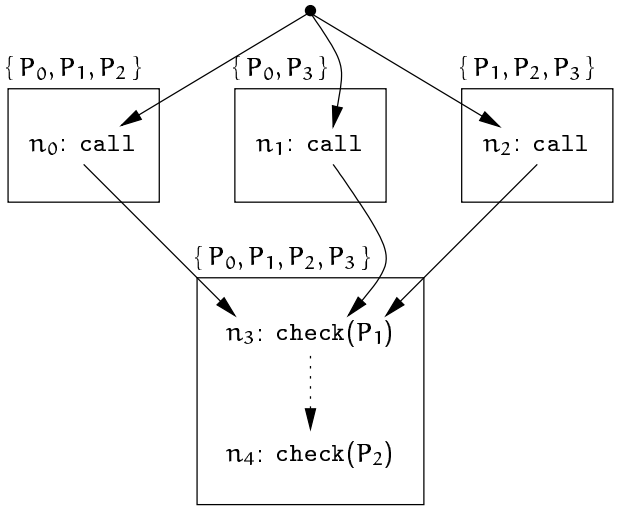

Figure 5.4: Control flow graph for example 5.6.1

Example 5.6.1. Consider the control flow graph in Fig. 5.4. By direct computation of the GP^0 analysis at n_4, we obtain:

$$\begin{aligned}
\gamma^0_{in}(n_4) &= \gamma^0_{trans}(n_3) = \gamma^0_{in}(n_3) \cup \{P_1\} \\
&= \left(\gamma^0_{out}(n_0, n_3) \cap \gamma^0_{out}(n_1, n_3) \cap \gamma^0_{out}(n_2, n_3)\right) \cup \{P_1\} \\
&= \left(\gamma^0_{in}(n_0) \cap \gamma^0_{in}(n_1) \cap \gamma^0_{in}(n_2)\right) \cup \{P_1\} \\
&= \left(\{P_0, P_1, P_2\} \cap \{P_0, P_3\} \cap \{P_1, P_2, P_3\}\right) \cup \{P_1\} = \{P_1\}
\end{aligned}$$

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:

$$\begin{aligned}
\overline{\delta^1}_{out}(n_0, n_3) &= \overline{\delta^1}_{in}(n_0) = \{P_0, P_1\} \\
\overline{\delta^1}_{out}(n_1, n_3) &= \overline{\delta^1}_{in}(n_1) = \{P_1, P_2\} \\
\overline{\delta^1}_{out}(n_2, n_3) &= \overline{\delta^1}_{in}(n_2) = \{P_0, P_3\}
\end{aligned}$$

Then, according to the GP^1 analysis, the permissions granted to n_4 are:

$$\begin{aligned}
\gamma^1_{in}(n_4) &= \gamma^1_{trans}(n_3) = \bigcap_{(m,n_3)\in E} \{\gamma^1_{out}(m, n_3) \mid P_1 \in \overline{\delta^1}_{out}(m, n_3)\} \cup \{P_1\} \\
&= \left(\gamma^1_{out}(n_0, n_3) \cap \gamma^1_{out}(n_2, n_3)\right) \cup \{P_1\} = \{P_1, P_2\}
\end{aligned}$$

This shows that the GP^1 analysis is stricly more accurate than the GP^0. Theorem 5.6.3 will ensure that GP^1 enjoys the soundness property, too.

Lemma 5.6.2. Let $\langle \gamma_{in}, \gamma_{out}, \gamma_{call}, \gamma_{trans} \rangle \models \mathsf{GP}^1(\mathsf{G})$, and:

$$[] = \sigma_0 \rhd \sigma_1 \rhd \cdots \rhd \sigma_k = \sigma : n$$

be a derivation. Then:

$$\mathsf{P} \in \gamma_{out}(\eta(\sigma_{k-1}, \sigma_k)) \implies \sigma \vdash \mathsf{JDK}(\mathsf{P}) \ \wedge \ \mathsf{P} \in \mathit{Perm}(n)$$

Proof. We proceed by induction on the length of derivation $[] \rhd \cdots \rhd \sigma_{k-1} \rhd \sigma : n$. The base case corresponds to our single axiom:

$$\frac{n \in \mathsf{N}_{entry}}{[] \rhd [n]}$$

For $n \in \mathsf{N}_{entry}$, we have defined $\eta([], [n]) = (\bot_{\mathsf{N}}, n)$, so $\mathsf{P} \in \gamma_{out}(\bot_{\mathsf{N}}, n) \implies \mathsf{P} \in \mathit{Perm}(n)$, and $\sigma = [] \vdash \mathsf{JDK}(\mathsf{P})$ by rule $\mathsf{JDK}_{\varnothing}$. For the inductive case, we proceed by case analysis on the rule used to derive $\sigma_{k-1} \rhd \sigma : n$, yielding:

- case [*call*]:

$$\frac{\ell(n') = \mathtt{call} \quad n' \longrightarrow n}{\sigma' : n' \rhd \sigma' : n' : n} \qquad \text{where } \sigma_{k-1} = \sigma = \sigma' : n'$$

  Here $\eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma' : n', \sigma' : n' : n) = (n', n) \in \mathsf{E}_{call}$, hence $\mathsf{P} \in \gamma_{out}(n', n) \implies \mathsf{P} \in \gamma_{call}(n')$ and $\mathsf{P} \in \mathit{Perm}(n)$. If $\neg \mathit{Priv}(n')$, then $\mathsf{P} \in \gamma_{in}(n')$, and, using lemma 3.8.2, we obtain $\mathsf{P} \in \gamma_{out}(\eta(\sigma_{k-2}, \sigma_{k-1}))$. Then we can apply the inductive hypothesis to deduce $\sigma' \vdash \mathsf{JDK}(\mathsf{P})$ and $\mathsf{P} \in \mathit{Perm}(n')$, thus obtaining $\sigma' : n' \vdash \mathsf{JDK}(\mathsf{P})$ by rule $\mathsf{JDK}_{\prec}$. Otherwise, if $\mathit{Priv}(n')$, then $\mathsf{P} \in \mathit{Perm}(n')$: in this case, $\sigma' : n' \vdash \mathsf{JDK}(\mathsf{P})$ is ensured by rule JDK_{Priv}.

- case [*check*]:

$$\frac{\ell(n') = \mathtt{check}(\mathsf{P}') \quad \sigma : n' \vdash \mathsf{JDK}(\mathsf{P}') \quad n' \dashrightarrow n}{\sigma : n' \rhd \sigma : n}$$

  Here $\eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma : n', \sigma : n) = (n', n) \in \mathsf{E}_{trans}$, then $\mathsf{P} \in \gamma_{out}(n', n) \implies \mathsf{P} \in \gamma_{trans}(n')$. Now, it can be shown that:

$$\mathsf{P} \in \bigcap_{\substack{(m,n') \in \mathsf{E} \\ \mathsf{P}' \notin \delta_{out}(m,n')}} \gamma_{out}(m, n') \cup \{\mathsf{P}'\} \tag{5.2}$$

  If $\mathsf{P} = \mathsf{P}'$, then $\sigma : n' \vdash \mathsf{JDK}(\mathsf{P})$ follows by the premises of the $\rhd_{check}$ rule, and this obviously implies $\mathsf{P} \in \mathit{Perm}(n')$. As check nodes are not privileged, $\sigma \vdash \mathsf{JDK}(\mathsf{P})$ follows from rule $\mathsf{JDK}_{\prec}$, and $\mathsf{P} \in \mathit{Perm}(n)$ from the fact that n and n' carry the same permissions. Otherwise, if $\mathsf{P} \neq \mathsf{P}'$, premise $\sigma : n' \vdash \mathsf{JDK}(\mathsf{P}')$ implies $\mathsf{P} \in \mathit{Perm}(n')$, and rule $\mathsf{JDK}_{\prec}$ also states $\sigma \vdash \mathsf{JDK}(\mathsf{P}')$. Then, by applying contrapositively lemma 5.5.2, it turns out that $\mathsf{P}' \notin \delta_{out}(\eta(\sigma_{k-2}, \sigma_{k-1}))$. Therefore, by equation (5.2), it is indeed $\mathsf{P} \in \gamma_{out}(\eta(\sigma_{k-2}, \sigma_{k-1}))$, and, applying the inductive hypothesis, we finally obtain $\sigma \vdash \mathsf{JDK}(\mathsf{P})$.

- case [*return*]:

$$\frac{\ell(m) = \texttt{return} \quad n' \dashrightarrow n}{\sigma : n' : m \rhd \sigma : n}$$

Here, $\eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma : n' : m, \sigma : n) = (n', n) \in \mathsf{E}_{trans}$: thus, $\mathsf{P} \in \gamma_{out}(n', n) \implies \mathsf{P} \in \gamma_{trans}(n')$. By lemma 3.4.4, it must be $\ell(n') = \texttt{call}$, then it turns out that $\mathsf{P} \in \gamma_{in}(n')$. 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 $\mathsf{P} \in \gamma_{out}(\eta(\sigma_{i-1}, \sigma : n'))$. Therefore we can apply the inductive hypothesis, obtaining $\sigma \vdash \mathsf{JDK}(\mathsf{P})$ and $\mathsf{P} \in \mathit{Perm}(n')$: again, $\mathsf{P} \in \mathit{Perm}(n') \implies \mathsf{P} \in \mathit{Perm}(n)$, as both n and n' lie in the same protection domain.

□

Theorem 5.6.3. Let $\langle \gamma_{in}, \gamma_{out}, \gamma_{call}, \gamma_{trans} \rangle \models \mathsf{GP}^1(\mathsf{G})$. For any $n \in \mathsf{N}$, define:

$$\gamma(n) \quad = \quad \gamma_{call}(n)$$

Then γ is a sound GP^1-solution.

Proof. Let $\gamma \models \mathsf{GP}^1(\mathsf{G})$, $\sigma : n$ a G-reachable state, and $\mathsf{P} \in \gamma_{call}(n)$. We have to distinguish between two cases:

- if $\neg \mathit{Priv}(n)$, then $\mathsf{P} \in \gamma_{in}(n)$. Consider a derivation $[] \rhd \cdots \rhd \sigma_{k-1} \rhd \sigma : n$. By definition of GP^1_{in}, we then have $\mathsf{P} \in \gamma_{out}(\eta(\sigma_{k-1}, \sigma : n))$, and by lemma 5.6.2 it follows $\sigma \vdash \mathsf{JDK}(\mathsf{P})$ and $\mathsf{P} \in \mathit{Perm}(n)$. But these are exactly the premises of rule $\mathsf{JDK}_{\prec}$, therefore $\sigma : n \vdash \mathsf{JDK}(\mathsf{P})$.
- if $\mathit{Priv}(n)$, then $\mathsf{P} \in \gamma_{in}(n)$ or $\mathsf{P} \in \mathit{Perm}(n)$. In the former case, we showed above that $\sigma : n \vdash \mathsf{JDK}(\mathsf{P})$; in the latter, this is ensured by rule JDK_{Priv}.

□

Example 5.6.4. Consider the e-commerce application of Fig. 3.1. By direct computation of the analysis at n_2, we obtain:

$$\begin{aligned}
\gamma_{in}(n_2) &= \gamma_{out}(n_0, n_2) \cap \gamma_{out}(n_3, n_2) \cap \gamma_{out}(n_4, n_2) \\
&= \big(\gamma_{call}(n_0) \cap \mathit{Perm}(n_2)\big) \cap \gamma_{trans}(n_3) \cap \gamma_{trans}(n_4) \\
&= \gamma_{in}(n_0) \cap \mathit{Perm}(n_2) \cap \gamma_{in}(n_3) \cap \gamma_{in}(n_4) \\
&= \mathit{Perm}(n_0) \cap \mathit{Perm}(n_2) \cap \gamma_{out}(n_2, n_3) \cap \gamma_{out}(n_2, n_4) \\
&= \{\mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}\} \cap \gamma_{trans}(n_2) \\
&= \{\mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}\} \cap \gamma_{in}(n_2)
\end{aligned}$$

This recursive equation is satisfied by any subset of $\{\mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}\}$: since, regardless of some techical details, the property space for 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:

$$\gamma_{trans}(n_{11}) \quad = \bigcap_{\substack{(m,n_{11})\in E \\ P_{debit}\notin\delta_{out}(m,n_{11})}} \gamma_{out}(m,n_{11}) \cup \{P_{debit}\}$$

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:

$$\begin{aligned}
\gamma_{trans}(n_{11}) &= \gamma_{out}(n_3,n_{11}) \cup \{P_{debit}\} = \gamma_{call}(n_3) \cup \{P_{debit}\} \\
&= \gamma_{in}(n_3) \cup \{P_{debit}\} = \gamma_{out}(n_2,n_3) \cup \{P_{debit}\} \\
&= \gamma_{trans}(n_2) \cup \{P_{debit}\} = \gamma_{in}(n_2) \cup \{P_{debit}\} \\
&= \{P_{canpay}, P_{credit}, P_{debit}\}
\end{aligned}$$

Therefore, the set of permissions granted at the entry of node n_8 is:

$$\begin{aligned}
\gamma_{in}(n_8) &= \gamma_{out}(n_2,n_8) \cap \gamma_{out}(n_{12},n_8) = \gamma_{call}(n_2) \cap \gamma_{call}(n_{12}) \\
&= \gamma_{in}(n_2) \cap \gamma_{in}(n_{12}) = \gamma_{in}(n_2) \cap \gamma_{trans}(n_{11}) \\
&= \{P_{canpay}, P_{credit}, P_{debit}\}
\end{aligned}$$

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 δ 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 γ^0 and γ^1 be the MFP solutions for the GP^0 and GP^1 analyses, respectively. Then γ^1 is *more accurate* than γ^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.

$$
\begin{array}{lcl}
f_{(m,n)}(\langle \mathsf{l}_0, \mathsf{l}_1 \rangle) & = & \langle f^0_{(m,n)},\, f^1_{(m,n)} \rangle(\langle \mathsf{l}_0, \mathsf{l}_1 \rangle) \\
f^0_{\bullet \longrightarrow n}(\langle \mathsf{l}_0, \mathsf{l}_1 \rangle) & = & \mathit{Perm}(\mathsf{n}) \\
f^0_{m \longrightarrow n}(\langle \mathsf{l}_0, \mathsf{l}_1 \rangle) & = & \mathit{Perm}(\mathsf{n}) \cap \begin{cases} \mathit{Perm}(\mathsf{m}) & \text{if } \mathit{Priv}(\mathsf{m}) \\ \mathsf{l}_1 & \text{otherwise} \end{cases} \\
f^0_{m \dashrightarrow n}(\langle \mathsf{l}_0, \mathsf{l}_1 \rangle) & = & \mathsf{l}_1 \\
f^0_{m \hookrightarrow n}(\langle \mathsf{l}_0, \mathsf{l}_1 \rangle) & = & \varnothing \\
f^1_{(m,n)}(\langle \mathsf{l}_0, \mathsf{l}_1 \rangle) & = & \begin{cases} \varnothing & \text{if } \mathsf{kill}^1(\overline{\delta}(\mathsf{n}), \mathsf{n}) \\ f^0_{(m,n)}(\langle \mathsf{l}_0, \mathsf{l}_1 \rangle) & \text{otherwise} \end{cases} \\
\\
\mathsf{kill}^1(\mathsf{l}, \mathsf{n}) & =_{def} & \ell(\mathsf{n}) = \mathtt{check}(\mathsf{P}) \text{ and } \mathsf{P} \notin \mathsf{l}
\end{array}
$$

Table 5.9: Local transfer functions for the GP^1 Analysis.

5.7 The DP^2 Analysis

$$\overline{\mathsf{DP}^2}_{in}(\mathtt{n}) = \begin{cases} \mathit{unreach} & \text{if } \forall(\mathtt{m},\mathtt{n}) \in \mathsf{E}.\ \overline{\mathsf{DP}^2}_{out}(\mathtt{m},\mathtt{n}) = \mathit{unreach} \\ \bigcup\limits_{\substack{(\mathtt{m},\mathtt{n})\in\mathsf{E} \\ \overline{\mathsf{DP}^2}_{out}(\mathtt{m},\mathtt{n})\neq\mathit{unreach}}} \overline{\mathsf{DP}^2}_{out}(\mathtt{m},\mathtt{n}) & \text{otherwise} \end{cases}$$

$$\overline{\mathsf{DP}^2}_{out}(\mathtt{m},\mathtt{n}) = \begin{cases} \mathit{unreach} & \text{if } \overline{\mathsf{DP}^2}_{in}(\mathtt{m}) = \mathit{unreach} \\ \mathit{Perm}(\mathtt{n}) & \text{if } \bullet\!\!\longrightarrow \mathtt{n} \\ \overline{\mathsf{DP}^2}_{call}(\mathtt{m}) \cap \mathit{Perm}(\mathtt{n}) & \text{if } \mathtt{m} \longrightarrow \mathtt{n} \\ \overline{\mathsf{DP}^2}_{trans}(\mathtt{m}) & \text{if } \mathtt{m} \dashrightarrow \mathtt{n} \end{cases}$$

$$\overline{\mathsf{DP}^2}_{call}(\mathtt{n}) = \begin{cases} \mathit{unreach} & \text{if } \overline{\mathsf{DP}^2}_{in}(\mathtt{n}) = \mathit{unreach} \\ \mathit{Perm}(\mathtt{n}) & \text{if } \mathit{Priv}(\mathtt{n}) \\ \overline{\mathsf{DP}^2}_{in}(\mathtt{n}) & \text{otherwise} \end{cases}$$

$$\overline{\mathsf{DP}^2}_{trans}(\mathtt{n}) = \begin{cases} \mathit{unreach} & \text{if } \overline{\mathsf{DP}^2}_{in}(\mathtt{n}) = \mathit{unreach} \text{ or } \mathsf{kill}^2(\mathtt{n}) \\ \bigcup\limits_{\substack{(\mathtt{m},\mathtt{n})\in\mathsf{E} \\ \mathsf{P}\in\overline{\mathsf{DP}^2}_{out}(\mathtt{m},\mathtt{n})}} \overline{\mathsf{DP}^2}_{out}(\mathtt{m},\mathtt{n}) & \text{if } \ell(\mathtt{n}) = \mathtt{check}(\mathsf{P}) \text{ and } \neg\mathsf{kill}^2(\mathtt{n}) \\ \overline{\mathsf{DP}^2}_{in}(\mathtt{n}) & \text{otherwise} \end{cases}$$

$$\mathsf{kill}^2(\mathtt{n}) =_{def} \begin{array}{l} \ell(\mathtt{n}) = \mathtt{check}(\mathsf{P}) \text{ and } \mathsf{P} \notin \overline{\mathsf{DP}^2}_{in}(\mathtt{n}) \text{ , or} \\ \ell(\mathtt{n}) = \mathtt{call} \text{ and } \forall \mathtt{m} \in \rho(\mathtt{n}).\ \overline{\mathsf{DP}^2}_{in}(\mathtt{m}) = \mathit{unreach} \end{array}$$

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.

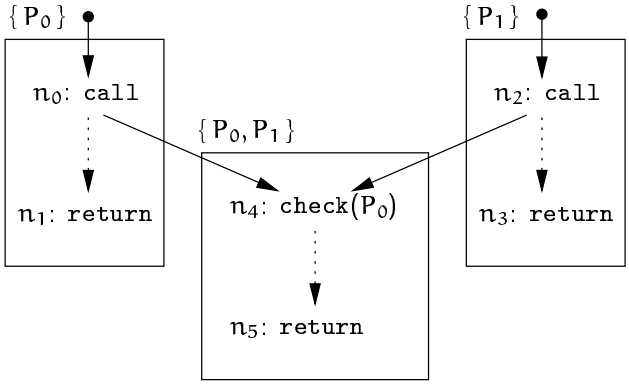

Figure 5.5: Control flow graph for example 5.7.1

Three kinds of unreachable nodes are detected by the DP^2 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, $m \dashrightarrow n$, 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 $[] \rhd \cdots \rhd \sigma : n$ 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 $\rho(n_0) = \{n_5\} = \rho(n_2)$. 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 $[n_2, n_4] \not\vdash \mathsf{JDK}(P_0)$.

Lemma 5.7.2. Let $\langle \delta_{in}, \delta_{out}, \delta_{call}, \delta_{trans} \rangle \models \mathsf{DP}^2(\mathsf{G})$, and:

$$[] = \sigma_0 \rhd \sigma_1 \rhd \cdots \rhd \sigma_k = \sigma : n$$

be a derivation. Then both the following statements hold:

$$\delta_{out}(\eta(\sigma_{k-1}, \sigma_k)) \neq \mathit{unreach} \tag{5.3a}$$

$$\mathsf{P} \in \delta_{out}(\eta(\sigma_{k-1}, \sigma_k)) \cap \mathit{Perm}(n) \implies \sigma \nvdash \mathsf{JDK}(\mathsf{P}) \tag{5.3b}$$

Proof. The proof is carried out by contradiction: we assume that $\sigma \vdash \mathsf{JDK}(\mathsf{P})$ whenever $\mathsf{P} \in \delta_{out}(\eta(\sigma_{k-1}, \sigma_k)) \cap \mathit{Perm}(n)$. Then, we proceed by induction on the length of derivation $[] \rhd \cdots \rhd \sigma_{k-1} \rhd \sigma : n$. The base case corresponds to our single axiom:

$$\frac{n \in \mathsf{N}_{entry}}{[] \rhd [n]}$$

For $n \in \mathsf{N}_{entry}$, we have defined $\eta([], [n]) = (\bot_{\mathsf{N}}, n)$, hence $\overline{\delta}_{out}(\bot_{\mathsf{N}}, n) = \mathit{Perm}(n)$. Then (5.3a) is true because $\mathit{Perm}(n) \neq \mathit{unreach}$, and (5.3b) trivially holds, as $\mathsf{P} \in \delta_{out}(\bot_{\mathsf{N}}, n) \implies \mathsf{P} \notin \mathit{Perm}(n)$, and the premises of the implication are never satisfied. For the inductive case, we proceed by case analysis on the rule used to derive $\sigma_{k-1} \rhd \sigma : n$, yielding:

- case [*call*]:

$$\frac{\ell(n') = \texttt{call} \quad n' \longrightarrow n}{\sigma' : n' \rhd \sigma' : n' : n} \qquad \text{where } \sigma_{k-1} = \sigma = \sigma' : n'$$

  Here $\eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma' : n', \sigma' : n' : n) = (n', n) \in \mathsf{E}_{call}$. By the inductive hypothesis, it must be $\delta_{out}(\eta(\sigma_{k-2}, \sigma' : n')) \neq \mathit{unreach}$, hence $\delta_{in}(n') \neq \mathit{unreach}$: this actually implies $\delta_{out}(n', n) \neq \mathit{unreach}$, so the (5.3a) is estabilished.

  For the (5.3b), $\mathsf{P} \in \delta_{out}(n', n) \implies \mathsf{P} \in \delta_{call}(n')$ or $\mathsf{P} \notin \mathit{Perm}(n)$. The latter option is prevented by our assumptions about P, then only the former one is considered. If $\neg \mathit{Priv}(n')$, then $\mathsf{P} \in \delta_{in}(n')$, and, using lemma 3.8.2, we obtain $\mathsf{P} \in \delta_{out}(\eta(\sigma_{k-2}, \sigma_{k-1}))$. Note that assumption $\sigma' : n' \vdash \mathsf{JDK}(\mathsf{P})$ imposes $\mathsf{P} \in \mathit{Perm}(n')$, so we can apply the inductive hypothesis to deduce $\sigma' \nvdash \mathsf{JDK}(\mathsf{P})$. This prevents the $\mathsf{JDK}_{\prec}$ rule to be applicable, and a contradiction arises with assumption $\sigma' : n' \vdash \mathsf{JDK}(\mathsf{P})$. Otherwise, if $\mathit{Priv}(n')$, then it should hold $\mathsf{P} \notin \mathit{Perm}(n')$, contradicting again assumption $\sigma' : n' \vdash \mathsf{JDK}(\mathsf{P})$.

- case [*check*]:

$$\frac{\ell(n') = \texttt{check}(\mathsf{P}') \quad \sigma : n' \vdash \mathsf{JDK}(\mathsf{P}') \quad n' \dashrightarrow n}{\sigma : n' \rhd \sigma : n}$$

  Here $\eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma : n', \sigma : n) = (n', n) \in \mathsf{E}_{trans}$. By the inductive hypothesis, it must be $\delta_{out}(\eta(\sigma_{k-2}, \sigma' : n')) \neq \mathit{unreach}$, hence $\delta_{in}(n') \neq \mathit{unreach}$. Therefore, $\delta_{trans}(n')$ can be *unreach* only if $\mathsf{P}' \in \delta_{in}(n')$, i.e. $\forall (m, n') \in \mathsf{E}.\ \mathsf{P}' \in \delta_{out}(m, n')$. If this happens, by lemma 3.8.2 we also have $\mathsf{P}' \in \delta_{out}(\eta(\sigma_{k-2}, \sigma : n'))$, and premise $\sigma : n' \vdash \mathsf{JDK}(\mathsf{P}')$ implies $\mathsf{P}' \in \mathit{Perm}(n')$. Now the inductive hypothesis can be applied to obtain $\sigma \nvdash \mathsf{JDK}(\mathsf{P}')$: since n' is not privileged, this is a contradiction. Then it must be $\delta_{trans}(n') \neq \mathit{unreach}$, and the (5.3a) is estabilished, because this clearly implies $\delta_{out}(n', n) \neq \mathit{unreach}$.

Now, for the (5.3b), we have that $\mathsf{P} \in \delta_{out}(\mathtt{n}', \mathtt{n}) \implies \mathsf{P} \in \delta_{trans}(\mathtt{n}')$. As both $\mathtt{n}$ and $\mathtt{n}'$ lie in the same protection domain, assumption $\mathsf{P} \in \mathit{Perm}(\mathtt{n})$ also ensures $\mathsf{P} \in \mathit{Perm}(\mathtt{n}')$. Now, it can be shown that:

$$\mathsf{P} \in \bigcap_{\substack{(\mathtt{m},\mathtt{n}') \in \mathsf{E} \\ \mathsf{P}' \notin \delta_{out}(\mathtt{m},\mathtt{n}')}} \delta_{out}(\mathtt{m}, \mathtt{n}') \tag{5.4}$$

As check nodes are not privileged, by rule $\mathsf{JDK}_{\prec}$ we know that $\sigma : \mathtt{n}' \vdash \mathsf{JDK}(\mathsf{P}')$ can only be true if $\sigma \vdash \mathsf{JDK}(\mathsf{P}')$. Then, by applying contrapositively the inductive hypothesis, we deduce that $\mathsf{P}' \notin \delta_{out}(\eta(\sigma_{k-2}, \sigma_{k-1})) \cap \mathit{Perm}(\mathtt{n}')$. Now, premise $\sigma : \mathtt{n}' \vdash \mathsf{JDK}(\mathsf{P}')$ requires $\mathsf{P}' \in \mathit{Perm}(\mathtt{n}')$: therefore, it is indeed $\mathsf{P}' \notin \delta_{out}(\eta(\sigma_{k-2}, \sigma_{k-1}))$. By equation (5.4), this implies $\mathsf{P} \in \delta_{out}(\eta(\sigma_{k-2}, \sigma_{k-1}))$, and, applying again the inductive hypothesis, we finally obtain a contradiction $\sigma \not\vdash \mathsf{JDK}(\mathsf{P})$.

- case [*return*]:

$$\frac{\ell(\mathtt{m}) = \texttt{return} \quad \mathtt{n}' \dashrightarrow \mathtt{n}}{\sigma : \mathtt{n}' : \mathtt{m} \rhd \sigma : \mathtt{n}}$$

Here, $\eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma : \mathtt{n}' : \mathtt{m}, \sigma : \mathtt{n}) = (\mathtt{n}', \mathtt{n}) \in \mathsf{E}_{trans}$. By the inductive hypothesis, it must be $\delta_{out}(\eta(\sigma_{k-2}, \sigma' : \mathtt{n}' : \mathtt{m})) \neq \mathit{unreach}$, hence $\delta_{in}(\mathtt{m}) \neq \mathit{unreach}$. Now, lemma 3.4.1 states that there is an index $i < k$ such that $\sigma_i = \sigma : \mathtt{n}'$: hence, the inductive hypothesis is also applicable to the derivation $\sigma_0 \rhd \cdots \rhd \sigma_i$, yielding $\delta_{out}(\eta(\sigma_{i-1}, \sigma' : \mathtt{n}')) \neq \mathit{unreach}$ and $\delta_{in}(\mathtt{n}') \neq \mathit{unreach}$. By lemma 3.4.4, it must be $\ell(\mathtt{n}') = \texttt{call}$, then the only case leading to $\delta_{trans}(\mathtt{n}') = \mathit{unreach}$ is when $\forall \mathtt{m}' \in \rho(\mathtt{n}')$. $\delta_{in}(\mathtt{m}') = \mathit{unreach}$: indeed, this cannot actually happen, as lemma 3.5.16 implies $\mathtt{m} \in \rho(\mathtt{n}')$ and we already know $\delta_{in}(\mathtt{m}) \neq \mathit{unreach}$. This proves the (5.3a).

For the (5.3b), we have that $\mathsf{P} \in \delta_{out}(\mathtt{n}', \mathtt{n}) \implies \mathsf{P} \in \delta_{trans}(\mathtt{n}')$. By lemma 3.4.4, it must be $\ell(\mathtt{n}') = \texttt{call}$, then it turns out that $\mathsf{P} \in \delta_{in}(\mathtt{n}')$. Now, lemma 3.4.1 states that $\exists i \in 1..k-2.\ \sigma_i = \sigma : \mathtt{n}'$: by lemma 3.8.2, we then have that $\mathsf{P} \in \delta_{out}(\eta(\sigma_{i-1}, \sigma : \mathtt{n}'))$, and, again, assumption $\mathsf{P} \in \mathit{Perm}(\mathtt{n})$ implies $\mathsf{P} \in \mathit{Perm}(\mathtt{n}')$ as both $\mathtt{n}$ and $\mathtt{n}'$ carry the same permissions. Hence, the inductive hypothesis can be applied to yield $\sigma \not\vdash \mathsf{JDK}(\mathsf{P})$.

□

Theorem 5.7.3. Let $\langle \delta_{in}, \delta_{out}, \delta_{call}, \delta_{trans} \rangle \models \mathsf{DP}^2(\mathsf{G})$. For any $\mathtt{n} \in \mathsf{N}$, define:

$$\delta(\mathtt{n}) = \begin{cases} \varnothing & \text{if } \delta_{in}(\mathtt{n}) = \mathit{unreach} \\ \delta_{call}(\mathtt{n}) & \text{otherwise} \end{cases}$$

Then δ is a sound DP^2-solution.

Proof. The proof is exactly the same as the one of theorem 5.5.3. □

Lemma 5.7.4. Let $\langle \delta_{in}, \delta_{out}, \delta_{call}, \delta_{trans} \rangle \models \mathsf{DP}^2(\mathsf{G})$. Then, for any $n \in \mathsf{N}$:

$$\delta_{in}(n) = \mathit{unreach} \quad \Longrightarrow \quad \neg\exists \sigma \in \Sigma.\ \mathsf{G} \vdash \sigma : n$$

Proof. By contradiction, assume $\delta_{in}(n) = \mathit{unreach}$ and n is G-reachable, i.e. a derivation

$$[] = \sigma_0 \rhd \sigma_1 \rhd \cdots \rhd \sigma_k = \sigma : n$$

exists. Now, lemma 5.7.2 ensures that $\delta_{out}(\eta(\sigma_{k-1}, \sigma : n)) \neq \mathit{unreach}$. Then, we have a contradiction, as this would imply $\delta_{in}(n) \neq \mathit{unreach}$. □

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:

$$\begin{aligned} \overline{\delta}_{in}(n_2) &= \{\mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}\} \\ \overline{\delta}_{in}(n_5) &= \varnothing \end{aligned}$$

By direct computation of the analysis at n_{16}, we have:

$$\begin{aligned} \overline{\delta}_{in}(n_{16}) &= \overline{\delta}_{out}(n_4, n_{16}) \cup \overline{\delta}_{out}(n_5, n_{16}) \\ &= \big(\overline{\delta}_{call}(n_4) \cup \overline{\delta}_{call}(n_5)\big) \cap \mathit{Perm}(n_{16}) \\ &= \big(\overline{\delta}_{in}(n_4) \cup \overline{\delta}_{in}(n_5)\big) \cap \mathit{Perm}(n_{16}) \\ &= \big(\overline{\delta}_{out}(n_2, n_4) \cup \varnothing\big) \cap \mathit{Perm}(n_{16}) \\ &= \overline{\delta}_{trans}(n_2) \cap \mathit{Perm}(n_{16}) \\ &= \overline{\delta}_{in}(n_2) \cap \mathit{Perm}(n_{16}) \\ &= \{\mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}\} \end{aligned}$$

Since $\ell(n_{16}) = \mathtt{check}(\mathsf{P}_{loan})$ and $\mathsf{P}_{loan} \notin \overline{\delta}_{in}(n_{16})$, it turns out that $\mathtt{kill}^2(n_{16})$. Thus, $\overline{\delta}_{trans}(n_{16}) = \mathit{unreach}$, and $\overline{\delta}_{in}(n_{17}) = \mathit{unreach}$, too. By lemma 5.7.4, this shows that node n_{17} is unreachable.

Moreover, we have $\mathtt{kill}^2(n_5)$, because $\overline{\delta}_{in}(n_{18}) = \mathit{unreach}$ and $\rho(n_5) = \{n_{18}\}$. Then $\overline{\delta}_{in}(n_6) = \mathit{unreach}$, 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_6 and n_{17}.

The full MFP solution to DP^2 for the e-commerce example is in Table A.1.

Theorem 5.7.6. DP^2 is a monotone data flow framework.

Proof. Consider the complemented analysis $\overline{\mathsf{DP}^2}$. The local property space is a product:

$$\mathcal{L}_\mathsf{N} \;=\; \mathcal{P}(\textbf{Permission}) \times \mathcal{P}(\textbf{Permission}) \times \mathbf{O}$$

where $\mathbf{O}$ is the two-element cpo $\bot \sqsubseteq \top$. Similarly to the data flow framework constructed for the DP^1 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:

$$\langle \mathsf{l}_0, \mathsf{l}_1, \mathsf{l}_2 \rangle \sqcup \langle \mathsf{l}'_0, \mathsf{l}'_1, \mathsf{l}'_2 \rangle \;=\; \langle \mathsf{l}_0 \cup \mathsf{l}'_0,\ \mathsf{l}_1 \cup \mathsf{l}'_1, \mathsf{l}_2 \sqcup \mathsf{l}'_2 \rangle$$

where $\bot \sqcup \mathsf{l} = \mathsf{l}$ and $\top \sqcup \mathsf{l} = \top$ for any $\mathsf{l} \in \mathbf{O}$. The bottom element is $\bot_{\mathcal{L}_\mathsf{N}} = \langle \varnothing, \varnothing, \bot \rangle$. Since both the join operators of $\mathcal{P}(\textbf{Permission})$ and $\mathbf{O}$ are idempotent, commutative and associative, this definition actually makes $\mathcal{L}_\mathsf{N}$ a join semi-lattice. By the fact that **Permission** is finite for any control flow graph, it clearly follows that $\mathcal{L}_\mathsf{N}$ satisfies the ascending chain condition. Again, a standard construction equips $\mathcal{L}_\mathsf{N}$ with a *complete lattice* structure:

$$\mathcal{L}_\mathsf{N} \;=\; \langle \mathcal{L}_\mathsf{N},\ \sqsubseteq,\ \bigsqcup,\ \bigsqcap,\ \bot_{\mathcal{L}_\mathsf{N}},\ \top_{\mathcal{L}_\mathsf{N}} \rangle$$

The partial order $\sqsubseteq$ is the determined coordinatewise:

$$\langle \mathsf{l}_0, \mathsf{l}_1, \mathsf{l}_2 \rangle \sqsubseteq \langle \mathsf{l}'_0, \mathsf{l}'_1, \mathsf{l}'_2 \rangle \qquad \text{iff } \mathsf{l}_0 \subseteq \mathsf{l}'_0 \text{ and } \mathsf{l}_1 \subseteq \mathsf{l}'_1 \text{ and } \mathsf{l}_2 \sqsubseteq \mathsf{l}'_2$$

and the other components of the lattice are defined in the obvious manner. The solution for the isolated entry node is defined as:

$$\iota \;=\; \langle\, \varnothing, \varnothing, \top \,\rangle$$

Given a control flow graph G, the set of local transfer functions $f_\mathsf{E} = \mu_\mathsf{E}(\mathsf{G})$ is defined in Table 5.11. If each $f \in f_\mathsf{E}$ is monotone regardless of G, we can take $\mathcal{F}_\mathsf{E}$ to be the space of monotone functions over $\mathcal{L}_\mathsf{N}$. So, let $\langle \mathsf{l}_0, \mathsf{l}_1, \mathsf{l}_2 \rangle \sqsubseteq \langle \mathsf{l}'_0, \mathsf{l}'_1, \mathsf{l}'_2 \rangle$ be two arbitrary elements of $\mathcal{L}_\mathsf{N}$. By definition of monotonicity, we have for prove that, for any $(\mathsf{m}, \mathsf{n}) \in \mathsf{E}_\rho$:

$$f_{(\mathsf{m},\mathsf{n})}(\langle \mathsf{l}_0, \mathsf{l}_1, \mathsf{l}_2 \rangle) \;\sqsubseteq\; f_{(\mathsf{m},\mathsf{n})}(\langle \mathsf{l}'_0, \mathsf{l}'_1, \mathsf{l}'_2 \rangle)$$

Consider the case $\mathsf{l}_2 = \bot$, first. We have:

$$f^0_{(m,n)}(\langle \mathsf{l}_0, \mathsf{l}_1, \bot \rangle) \;=\; f^1_{(m,n)}(\langle \mathsf{l}_0, \mathsf{l}_1, \bot \rangle) \;=\; f^2_{(m,n)}(\langle \mathsf{l}_0, \mathsf{l}_1, \bot \rangle) \;=\; \varnothing$$

Then $f_{(m,n)}(\langle \mathsf{l}_0, \mathsf{l}_1, \bot \rangle) = \bot_{\mathcal{L}_\mathsf{N}} \sqsubseteq f_{(m,n)}(\langle \mathsf{l}'_0, \mathsf{l}'_1, \mathsf{l}'_2 \rangle)$, regardless of l'_0, l'_1 and l'_2.

Otherwise, let $\mathsf{l}_2 = \top$: since we have assumed $\mathsf{l}_2 \sqsubseteq \mathsf{l}'_2$, it must be $\mathsf{l}'_2 = \top$, too. Under these hypotheses, the proof of monotonicity for the components $f^0_{(m,n)}$ and $f^1_{(m,n)}$ is identical to the relative proof for the DP^1 analysis (theorem 5.5.6).

For the $f^2_{(\mathsf{m},\mathsf{n})}$ component, we have to deal with the following cases:

- if $\ell(\mathsf{m}) = \texttt{call}$ and $\mathsf{m} \dashrightarrow \mathsf{n}$, then:

$$f^2_{(\mathsf{m},\mathsf{n})}(\langle \mathsf{l}_0, \mathsf{l}_1, \top \rangle) \;=\; \bot \;=\; f^2_{(\mathsf{m},\mathsf{n})}(\langle \mathsf{l}'_0, \mathsf{l}'_1, \top \rangle)$$

- if $\ell(\mathsf{m}) = \texttt{check}(\mathsf{P})$ and $\mathsf{P} \notin \mathsf{l}'_0$, then $\mathsf{P} \notin \mathsf{l}_0$, because $\mathsf{l}_0 \subseteq \mathsf{l}'_0$ by hypothesis. Thus:

$$f^2_{(\mathsf{m},\mathsf{n})}(\langle \mathsf{l}_0, \mathsf{l}_1, \top \rangle) \;=\; \bot \;=\; f^2_{(\mathsf{m},\mathsf{n})}(\langle \mathsf{l}'_0, \mathsf{l}'_1, \top \rangle)$$

$$
\begin{array}{lcl}
f_{(m,n)}(\langle l_0, l_1, l_2\rangle) & = & \langle f^0_{(m,n)},\, f^1_{(m,n)},\, f^2_{(m,n)}\rangle(\langle l_0, l_1, l_2\rangle) \\[2ex]
f^0_{(m,n)}(\langle l_0, l_1, \bot\rangle) & = & \varnothing \\
f^0_{\bullet\longrightarrow n}(\langle l_0, l_1, \top\rangle) & = & \mathit{Perm}(n) \\
f^0_{m\longrightarrow n}(\langle l_0, l_1, \top\rangle) & = & \mathit{Perm}(n) \cap \begin{cases} \mathit{Perm}(m) & \text{if } \mathit{Priv}(m) \\ l_1 & \text{otherwise} \end{cases} \\
f^0_{m\dashrightarrow n}(\langle l_0, l_1, \top\rangle) & = & l_1 \\
f^0_{m\hookrightarrow n}(\langle l_0, l_1, \top\rangle) & = & \varnothing \\[2ex]
f^1_{(m,n)}(\langle l_0, l_1, \bot\rangle) & = & \varnothing \\
f^1_{(m,n)}(\langle l_0, l_1, \top\rangle) & = & \begin{cases} \varnothing & \text{if } \mathtt{kill}^2(l'_0, n) \\ l'_0 & \text{otherwise} \end{cases} \quad \text{where } l'_0 = f^0_{(m,n)}(\langle l_0, l_1, \top\rangle) \\[2ex]
f^2_{(m,n)}(\langle l_0, l_1, \bot\rangle) & = & \bot \\
f^2_{(m,n)}(\langle l_0, l_1, \top\rangle) & = & \begin{cases} \bot & \text{if } \ell(m) = \mathtt{call} \text{ and } m \dashrightarrow n, \\ & \text{or } \ell(m) = \mathtt{check}(\mathsf{P}) \text{ and } \mathsf{P} \notin l_0 \\ \top & \text{otherwise} \end{cases} \\[2ex]
\mathtt{kill}^2(l, n) & =_{def} & \ell(n) = \mathtt{check}(\mathsf{P}) \text{ and } \mathsf{P} \notin l
\end{array}
$$

Table 5.11: Local transfer functions for the DP^2 Analysis.

- if $\ell(m) = \mathtt{check}(\mathsf{P})$ and $\mathsf{P} \in l'_0$, but $\mathsf{P} \notin l_0$, then:

$$f^2_{(m,n)}(\langle l_0, l_1, \top\rangle) \;=\; \bot \;\sqsubseteq\; \top \;=\; f^2_{(m,n)}(\langle l'_0, l'_1, \top\rangle)$$

- in the remaining cases, we have:

$$f^2_{(m,n)}(\langle l_0, l_1, \top\rangle) \;=\; \top \;=\; f^2_{(m,n)}(\langle l'_0, l'_1, \top\rangle)$$

□

Theorem 5.7.7. Let δ^1 and δ^2 be the MFP solutions for the DP^1 and DP^2 analyses, respectively. Then δ^2 is *more accurate* than δ^1, i.e.:

$$\delta^2 \sqsubseteq \delta^1$$

Proof. Let f_{E} and g_{E} be the sets of local transfer functions for the complemented analyses $\overline{\mathsf{DP}^1}$ and $\overline{\mathsf{DP}^2}$ respectively, as defined in tables 5.7 and 5.11. Moreover, let $\mathcal{L}^1_{\mathsf{N}}$ and $\mathcal{L}^2_{\mathsf{N}}$ the local property spaces for $\overline{\mathsf{DP}^1}$ and $\overline{\mathsf{DP}^2}$.

By looking at the proofs of theorems 5.5.6 and 5.7.6, it follows that, for each element in $\mathcal{L}^1_{\mathsf{N}}$, there is a counterpart in $\mathcal{L}^2_{\mathsf{N}}$ carrying the same intuitive meaning. More precisely, if $\langle l_0, l_1\rangle \in \mathcal{L}^1_{\mathsf{N}}$, then the element $\langle l_0, l_1, \top\rangle \in \mathcal{L}^2_{\mathsf{N}}$ 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 $\mathcal{L}^1_{\mathsf{N}}$, because the DP^1 analysis does not distinguish between reachable and unreachable nodes.

Now, assume we can prove that:

$$\forall (m,n) \in \mathsf{E}_\rho.\ g_{(m,n)}(\langle l_0, l_1, \top\rangle) \sqsubseteq \langle f_{(m,n)}(\langle l'_0, l'_1\rangle), \top\rangle \tag{5.5}$$

for any $\langle l'_0, l'_1\rangle \in {\mathcal{L}_{\mathsf{N}}}^1$ and $\langle l_0, l_1, \top\rangle \in {\mathcal{L}_{\mathsf{N}}}^2$ such that $l_0 \sqsubseteq l'_0$ and $l_1 \sqsubseteq l'_1$. Since the inclusion relation is preserved by greatest lower bounds, the global transfer functions satisfy:

$$g(l) \sqsubseteq \langle f(l'), \top\rangle$$

for each $l = \lambda n.\ \langle l_0(n), l_1(n), \top\rangle$ and $l' = \lambda n.\ \langle l'_0(n), l'_1(n)\rangle$ such that:

$$\forall n \in \mathsf{N}.\ l_0(n) \sqsubseteq l'_0(n) \ \wedge\ l_1(n) \sqsubseteq l'_1(n)$$

Since the MFP solutions for $\overline{\mathsf{DP}^1}$ and $\overline{\mathsf{DP}^2}$ are the least upper bound of the (finite) chains:

$$\begin{array}{ccccccccccc}
\bot_{\mathcal{L}^1} & \sqsubseteq & f(\bot_{\mathcal{L}^1}) & \sqsubseteq & f^2(\bot_{\mathcal{L}^1}) & \sqsubseteq & \cdots & \sqsubseteq & f^k(\bot_{\mathcal{L}^1}) & = & f^{k+1}(\bot_{\mathcal{L}^1}) \\
\bot_{\mathcal{L}^2} & \sqsubseteq & g(\bot_{\mathcal{L}^2}) & \sqsubseteq & g^2(\bot_{\mathcal{L}^2}) & \sqsubseteq & \cdots & \sqsubseteq & g^h(\bot_{\mathcal{L}^2}) & = & g^{h+1}(\bot_{\mathcal{L}^2})
\end{array}$$

and, by (5.5), $g^i(\bot_{\mathcal{L}^2}) \sqsubseteq f^i(\bot_{\mathcal{L}^1})$ for any $i \in \mathbb{N}$, we eventually conclude that $\delta^2 \sqsubseteq \delta^1$.

Therefore, we proceed with the proof of (5.5). For each $(m,n) \in \mathsf{E}_\rho$, let $g^0_{(m,n)}$, $g^1_{(m,n)}$, $g^2_{(m,n)}$ be the components of $g_{(m,n)}$, and $f^0_{(m,n)}$, $f^1_{(m,n)}$ those of $f_{(m,n)}$.

Since $g^2_{(m,n)}(\langle l_0, l_1, \top\rangle) \sqsubseteq \top$ trivially follows by definition of the cpo $\mathbf{O}$, 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 $\bullet\!\longrightarrow n$, then:

$$g^0_{\bullet\longrightarrow n}(\langle l_0, l_1, \top\rangle) = \mathit{Perm}(n) = f^0_{\bullet\longrightarrow n}(\langle l'_0, l'_1\rangle)$$

- if $m \longrightarrow n$, there are two cases. If m is privileged then:

$$g^0_{m\longrightarrow n}(\langle l_0, l_1, \top\rangle) = \mathit{Perm}(m) = f^0_{m\longrightarrow n}(\langle l'_0, l'_1\rangle)$$

  and the inequality trivially holds. In the other case, we have:

$$g^0_{m\longrightarrow n}(\langle l_0, l_1, \top\rangle) = l_1 \sqsubseteq l'_1 = f^0_{m\longrightarrow n}(\langle l'_0, l'_1\rangle)$$

- if $m \dashrightarrow n$, then:

$$g^0_{m \dashrightarrow n}(\langle l_0, l_1, \top \rangle) \;=\; l_1 \;\sqsubseteq\; l'_1 \;=\; f^0_{m \dashrightarrow n}(\langle l'_0, l'_1 \rangle)$$

- if $m \hookrightarrow n$, then:

$$g^0_{m \hookrightarrow n}(\langle l_0, l_1, \top \rangle) \;=\; \varnothing \;=\; f^0_{m \hookrightarrow n}(\langle l'_0, l'_1 \rangle)$$

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 kill^1 and kill^2, we have, for any $l, l' \in \mathcal{P}(\textbf{Permission})$:

$$l \subseteq l' \;\wedge\; \mathsf{kill}^1(l', n) \quad\Longrightarrow\quad \mathsf{kill}^2(l, n)$$

By the part of the proof relative to $g^0_{(m,n)}$ and $f^0_{(m,n)}$, we know $\widehat{l_0} \sqsubseteq \widehat{l'_0}$: therefore, $\mathsf{kill}^1(\widehat{l'_0}, n) \Longrightarrow \mathsf{kill}^2(\widehat{l_0}, n)$, and we have to deal with the following cases:

- if $\mathsf{kill}^1(\widehat{l'_0}, n)$, then $\mathsf{kill}^2(\widehat{l_0}, n)$, and:

$$g^1_{(m,n)}(\langle l_0, l_1, \top \rangle) \;=\; \varnothing \;=\; f^1_{(m,n)}(\langle l'_0, l'_1 \rangle)$$

- if $\mathsf{kill}^2(\widehat{l_0}, n)$ but $\neg\mathsf{kill}^1(\widehat{l'_0}, n)$:

$$g^1_{(m,n)}(\langle l_0, l_1, \top \rangle) \;=\; \varnothing \;\sqsubseteq\; \widehat{l'_0} \;=\; f^1_{(m,n)}(\langle l'_0, l'_1 \rangle)$$

- in any remaining cases, we have:

$$g^1_{(m,n)}(\langle l_0, l_1, \top \rangle) \;=\; \widehat{l_0} \;\sqsubseteq\; \widehat{l'_0} \;=\; f^1_{(m,n)}(\langle l'_0, l'_1 \rangle)$$

□

Lemma 5.7.8. Let $\langle \delta_{in}, \delta_{out}, \delta_{call}, \delta_{trans} \rangle$ be the MFP solution to DP^2. Then:

$$P \in \overline{\delta}_{in}(n) \quad\Longrightarrow\quad P \in \mathit{Perm}(n)$$

for any $n \in \mathsf{N}$.

Proof. The proof is similar to the proof of theorem 5.3.6. □

Lemma 5.7.9. The MFP solution to DP^2 is non-trivial.

Proof. The result easily follows from 5.7.8, as it is shown in 5.3.7. □

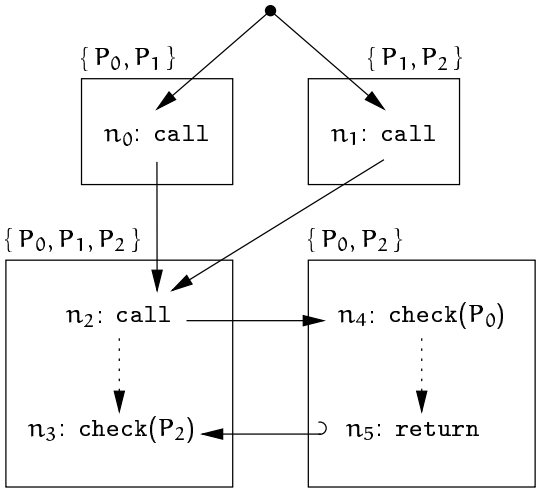

Figure 5.6: Control flow graph for counterexample 5.7.10

Counterexample 5.7.10. The DP^2 analysis is *not* complete.

Proof. Consider the control flow graph G in Fig. 5.6, and let δ the MFP solution to $DP^2(G)$. By direct computation of the analysis (with some shortcuts), we obtain:

$$\begin{aligned}
\overline{\delta}_{in}(n_2) &= \overline{\delta}_{out}(n_0, n_2) \cup \overline{\delta}_{out}(n_1, n_2) = \overline{\delta}_{call}(n_0) \cup \overline{\delta}_{call}(n_1) \\
&= \{P_0, P_1\} \cup \{P_1, P_2\} = \{P_0, P_1, P_2\} \\
\overline{\delta}_{in}(n_4) &= \overline{\delta}_{out}(n_2, n_4) = \overline{\delta}_{call}(n_2, n_4) \cap \{P_0, P_2\} \\
&= \{P_0, P_1, P_2\} \cap \{P_0, P_2\} = \{P_0, P_2\} \\
\overline{\delta}_{in}(n_5) &= \overline{\delta}_{out}(n_4, n_5) = \overline{\delta}_{trans}(n_4) = \overline{\delta}_{out}(n_2, n_4) = \{P_0, P_2\} \\
\overline{\delta}_{in}(n_3) &= \overline{\delta}_{out}(n_2, n_3) = \overline{\delta}_{trans}(n_2) = \overline{\delta}_{in}(n_2) = \{P_0, P_1, P_2\}
\end{aligned}$$

Observe that, in order to estimate the analysis at n_3, we have used the fact that $n_5 \in \rho(n_2)$ and $\overline{\delta}_{in}(n_5) \neq \mathit{unreach}$: 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 $\tilde{\pi}_0 = \langle n_0, n_2, n_4, n_5, n_3 \rangle$ and $\tilde{\pi}_1 = \langle n_1, n_2, n_4, n_5, n_3 \rangle$. Only the first of these paths is traversable, through the derivation:

$$[] \rhd [n_0] \rhd [n_0, n_2] \rhd [n_0, n_2, n_4] \rhd [n_0, n_2, n_5] \rhd [n_0, n_3]$$

On the contrary, the other path is not traversable. In fact, the derivation:

$$[] \rhd [n_1] \rhd [n_1, n_2] \rhd [n_1, n_2, n_4]$$

is blocked, because $[n_1, n_2, n_4] \not\vdash \mathsf{JDK}(P_0)$. Now, we have that P_2 is denied to *all* executions leading to n_3: in fact, $\tilde{\chi}(\tilde{\pi}_0) = [n_0, n_3] \not\vdash \mathsf{JDK}(P_2)$. Since, as we showed above, $P_2 \notin \delta(n_3)$, we deduce that the DP^2 analysis is not complete. □

5.8 The GP^2 Analysis

$$
\begin{array}{rcl}
GP^2_{in}(n) & = & \begin{cases} unreach & \text{if } \forall (m,n) \in E.\ \overline{DP^2}_{out}(m,n) = unreach \\ \bigcap_{\substack{(m,n)\in E \\ \overline{DP^2}_{out}(m,n) \neq unreach}} GP^2_{out}(m,n) & \text{otherwise} \end{cases} \\[2ex]
GP^2_{out}(m,n) & = & \begin{cases} unreach & \text{if } GP^2_{in}(m) = unreach \\ Perm(n) & \text{if } \bullet \longrightarrow n \\ GP^2_{call}(m) \cap Perm(n) & \text{if } m \longrightarrow n \\ GP^2_{trans}(m) & \text{if } m \dashrightarrow n \end{cases} \\[2ex]
GP^2_{call}(n) & = & \begin{cases} unreach & \text{if } GP^2_{in}(n) = unreach \\ Perm(n) & \text{if } Priv(n) \\ GP^2_{in}(n) & \text{otherwise} \end{cases} \\[2ex]
GP^2_{trans}(n) & = & \begin{cases} unreach & \text{if } GP^2_{in}(n) = unreach \text{ or } kill^2(n) \\ \bigcap_{\substack{(m,n)\in E \\ P \in \overline{DP^2}_{out}(m,n)}} GP^2_{out}(m,n) \cup \{P\} & \text{if } \ell(n) = \texttt{check}(P) \text{ and } \neg kill^2(n) \\ GP^2_{in}(n) & \text{otherwise} \end{cases}
\end{array}
$$

Table 5.12: The GP^2 Analysis.

The GP^2 analysis, defined in Table 5.12, refines the GP^1 analysis by taking into account the information about unreachable nodes computed by the DP^2 analysis. As we will see, this gives rise to more accurate (but still incomplete) solutions than those achievable by the GP^1 analysis.

Lemma 5.8.1. Let $\langle \gamma_{in}, \gamma_{out}, \gamma_{call}, \gamma_{trans} \rangle \models \mathsf{GP}^2(\mathsf{G})$, and:

$$[] = \sigma_0 \rhd \sigma_1 \rhd \cdots \rhd \sigma_k = \sigma : n$$

be a derivation. Then both the following statements hold:

$$\gamma_{out}(\eta(\sigma_{k-1}, \sigma_k)) \neq \mathit{unreach} \tag{5.6a}$$

$$\mathsf{P} \in \gamma_{out}(\eta(\sigma_{k-1}, \sigma_k)) \implies \sigma \vdash \mathsf{JDK}(\mathsf{P}) \ \wedge \ \mathsf{P} \in \mathit{Perm}(n) \tag{5.6b}$$

Proof. We proceed by induction on the length of derivation $[] \rhd \cdots \rhd \sigma_{k-1} \rhd \sigma : n$. The base case corresponds to our single axiom:

$$\frac{n \in \mathsf{N}_{entry}}{[] \rhd [n]}$$

For $n \in \mathsf{N}_{entry}$, we have defined $\eta([], [n]) = (\bot_{\mathsf{N}}, n)$, so $\mathsf{P} \in \gamma_{out}(\bot_{\mathsf{N}}, n) \implies \mathsf{P} \in \mathit{Perm}(n)$. Then (5.6a) is true because $\mathit{Perm}(n) \neq \mathit{unreach}$, and (5.6b) also holds because, by rule $\mathsf{JDK}_{\varnothing}$, $\sigma = [] \vdash \mathsf{JDK}(\mathsf{P})$. For the inductive case, we proceed by case analysis on the rule used to derive $\sigma_{k-1} \rhd \sigma : n$, yielding:

- case [*call*]:

$$\frac{\ell(n') = \mathtt{call} \quad n' \longrightarrow n}{\sigma' : n' \rhd \sigma' : n' : n} \qquad \text{where } \sigma_{k-1} = \sigma = \sigma' : n'$$

  Here $\eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma' : n', \sigma' : n' : n) = (n', n) \in \mathsf{E}_{call}$. By the inductive hypothesis, it must be $\gamma_{out}(\eta(\sigma_{k-2}, \sigma' : n')) \neq \mathit{unreach}$, hence $\gamma_{in}(n') \neq \mathit{unreach}$: this actually implies $\gamma_{out}(n', n) \neq \mathit{unreach}$, so the (5.6a) is estabilished.

  For the (5.6b), observe that $\mathsf{P} \in \gamma_{out}(n', n) \implies \mathsf{P} \in \gamma_{call}(n')$ and $\mathsf{P} \in \mathit{Perm}(n)$. If $\neg \mathit{Priv}(n')$, then $\mathsf{P} \in \gamma_{in}(n')$, and, using lemma 3.8.2, we obtain $\mathsf{P} \in \gamma_{out}(\eta(\sigma_{k-2}, \sigma_{k-1}))$. Then we can apply the inductive hypothesis to deduce $\sigma' \vdash \mathsf{JDK}(\mathsf{P})$ and $\mathsf{P} \in \mathit{Perm}(n')$, thus obtaining $\sigma' : n' \vdash \mathsf{JDK}(\mathsf{P})$ by rule $\mathsf{JDK}_{\prec}$. Otherwise, if $\mathit{Priv}(n')$, then $\mathsf{P} \in \mathit{Perm}(n')$: in this case, $\sigma' : n' \vdash \mathsf{JDK}(\mathsf{P})$ is ensured by rule JDK_{Priv}.

- case [*check*]:

$$\frac{\ell(n') = \mathtt{check}(\mathsf{P}') \quad \sigma : n' \vdash \mathsf{JDK}(\mathsf{P}') \quad n' \dashrightarrow n}{\sigma : n' \rhd \sigma : n}$$

  Here $\eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma : n', \sigma : n) = (n', n) \in \mathsf{E}_{trans}$. By the inductive hypothesis, it must be $\gamma_{out}(\eta(\sigma_{k-2}, \sigma' : n')) \neq \mathit{unreach}$, hence $\gamma_{in}(n') \neq \mathit{unreach}$. Therefore, $\gamma_{trans}(n')$ can be *unreach* only if $\mathsf{P}' \in \delta_{in}(n')$, i.e. $\forall (m, n') \in \mathsf{E}.\ \mathsf{P}' \in \delta_{out}(m, n')$. If this happens, by lemma 3.8.2 we also have $\mathsf{P}' \in \delta_{out}(\eta(\sigma_{k-2}, \sigma : n'))$. Since check nodes are not privileged, by the premise $\sigma : n' \vdash \mathsf{JDK}(\mathsf{P}')$ we deduce both $\sigma \vdash \mathsf{JDK}(\mathsf{P}')$ and $\mathsf{P}' \in \mathit{Perm}(n')$. Since, by (5.3b), it is also $\sigma \nvdash \mathsf{JDK}(\mathsf{P}')$, we have a contradiction. This proves the (5.6a).

  For the (5.6b), note that $\mathsf{P} \in \gamma_{out}(n', n) \implies \mathsf{P} \in \gamma_{trans}(n')$. It can be shown that:

$$\mathsf{P} \in \bigcap_{\substack{(m,n') \in \mathsf{E} \\ \mathsf{P}' \notin \delta_{out}(m,n')}} \gamma_{out}(m, n') \cup \{\mathsf{P}'\} \tag{5.7}$$

If $\mathsf{P} = \mathsf{P}'$, then $\sigma : \mathfrak{n}' \vdash \mathsf{JDK}(\mathsf{P})$ follows by the premises of the $\rhd_{check}$ rule, and this obviously implies $\mathsf{P} \in \mathit{Perm}(\mathfrak{n}')$. As check nodes are not privileged, $\sigma \vdash \mathsf{JDK}(\mathsf{P})$ follows from rule $\mathsf{JDK}_{\prec}$, and $\mathsf{P} \in \mathit{Perm}(\mathfrak{n})$ from the fact that $\mathfrak{n}$ and $\mathfrak{n}'$ carry the same permissions. Otherwise, if $\mathsf{P} \neq \mathsf{P}'$, premise $\sigma : \mathfrak{n}' \vdash \mathsf{JDK}(\mathsf{P}')$ implies $\mathsf{P} \in \mathit{Perm}(\mathfrak{n}')$, and rule $\mathsf{JDK}_{\prec}$ also states $\sigma \vdash \mathsf{JDK}(\mathsf{P}')$. Then, by applying contrapositively lemma 5.7.2, it turns out that $\mathsf{P}' \notin \delta_{out}(\eta(\sigma_{k-2}, \sigma_{k-1}))$. By equation (5.7), it follows $\mathsf{P} \in \gamma_{out}(\eta(\sigma_{k-2}, \sigma_{k-1}))$: then, by the inductive hypothesis, we finally obtain $\sigma \vdash \mathsf{JDK}(\mathsf{P})$.

- case [*return*]:

$$\frac{\ell(\mathfrak{m}) = \texttt{return} \quad \mathfrak{n}' \dashrightarrow \mathfrak{n}}{\sigma : \mathfrak{n}' : \mathfrak{m} \rhd \sigma : \mathfrak{n}}$$

Here, $\eta(\sigma_{k-1}, \sigma_k) = \eta(\sigma : \mathfrak{n}' : \mathfrak{m}, \sigma : \mathfrak{n}) = (\mathfrak{n}', \mathfrak{n}) \in \mathsf{E}_{trans}$. By the inductive hypothesis, it must be $\gamma_{out}(\eta(\sigma_{k-2}, \sigma' : \mathfrak{n}' : \mathfrak{m})) \neq \mathit{unreach}$, hence $\gamma_{in}(\mathfrak{m}) \neq \mathit{unreach}$. Now, lemma 3.4.1 states that there is an index $i < k$ such that $\sigma_i = \sigma : \mathfrak{n}'$: then, the inductive hypothesis is also applicable to the derivation $\sigma_0 \rhd \cdots \rhd \sigma_i$, yielding $\gamma_{out}(\eta(\sigma_{i-1}, \sigma' : \mathfrak{n}')) \neq \mathit{unreach}$ and $\gamma_{in}(\mathfrak{n}') \neq \mathit{unreach}$. By lemma 3.4.4, it must be $\ell(\mathfrak{n}') = \texttt{call}$, then $\gamma_{trans}(\mathfrak{n}') = \mathit{unreach}$ only occurs if $\gamma_{in}(\mathfrak{m}') = \mathit{unreach}$ for each $\mathfrak{m}' \in \rho(\mathfrak{n}')$. However, this cannot happen, because lemma 3.5.16 implies $\mathfrak{m} \in \rho(\mathfrak{n}')$, and we already know that $\gamma_{in}(\mathfrak{m}) \neq \mathit{unreach}$. This proves (5.6a).

For (5.6b), observe that $\mathsf{P} \in \gamma_{out}(\mathfrak{n}', \mathfrak{n}) \implies \mathsf{P} \in \gamma_{trans}(\mathfrak{n}')$. By lemma 3.4.4, it must be $\ell(\mathfrak{n}') = \texttt{call}$, then it turns out that $\mathsf{P} \in \gamma_{in}(\mathfrak{n}')$. Now, lemma 3.4.1 states that $\exists i \in 1..k-2.\ \sigma_i = \sigma : \mathfrak{n}'$, and by lemma 3.8.2, we then have that $\mathsf{P} \in \gamma_{out}(\eta(\sigma_{i-1}, \sigma : \mathfrak{n}'))$. Therefore we can apply the inductive hypothesis, obtaining $\sigma \vdash \mathsf{JDK}(\mathsf{P})$ and $\mathsf{P} \in \mathit{Perm}(\mathfrak{n}')$: again, $\mathsf{P} \in \mathit{Perm}(\mathfrak{n}') \implies \mathsf{P} \in \mathit{Perm}(\mathfrak{n})$, as both $\mathfrak{n}$ and $\mathfrak{n}'$ lie in the same protection domain.

□

Theorem 5.8.2. Let $\langle \gamma_{in}, \gamma_{out}, \gamma_{call}, \gamma_{trans} \rangle \models \mathsf{GP}^2(\mathsf{G})$. For any $\mathfrak{n} \in \mathsf{N}$, define:

$$\gamma(\mathfrak{n}) \;=\; \begin{cases} \varnothing & \text{if } \gamma_{in}(\mathfrak{n}) = \mathit{unreach} \\ \gamma_{call}(\mathfrak{n}) & \text{otherwise} \end{cases}$$

Then γ is a sound GP^2-solution.

Proof. The proof is exactly the same as the one of theorem 5.6.3. □

Lemma 5.8.3. Let $\langle \gamma_{in}, \gamma_{out}, \gamma_{call}, \gamma_{trans} \rangle \models \mathsf{GP}^2(\mathsf{G})$. Then, for any $\mathfrak{n} \in \mathsf{N}$:

$$\gamma_{in}(\mathfrak{n}) = \mathit{unreach} \quad \implies \quad \neg\exists \sigma \in \Sigma.\ \mathsf{G} \vdash \sigma : \mathfrak{n}$$

Proof. This proof is exactly the same as the one of lemma 5.7.4 □

Example 5.8.4. Consider the e-commerce application of Fig. 3.1. By direct computation of the analysis at $\mathfrak{n}_7$, we obtain:

$$\begin{aligned} \gamma_{in}(\mathfrak{n}_7) &= \gamma_{out}(\mathfrak{n}_0, \mathfrak{n}_7) \cap \gamma_{out}(\mathfrak{n}_7, \mathfrak{n}_7) \\ &= \big(\gamma_{call}(\mathfrak{n}_0) \cap \mathit{Perm}(\mathfrak{n}_7)\big) \cap \gamma_{trans}(\mathfrak{n}_7) \\ &= \{\mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}\} \cap \gamma_{in}(\mathfrak{n}_7) \end{aligned}$$

This recursive equation is satisfied by any subset of $\{\mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}\}$: since, regardless of some techical 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:

$$\gamma_{in}(n_{19}) \quad = \bigcap_{\substack{(m,n_{19})\in E \\ \overline{\delta}_{out}(m,n_{19})\neq unreach}} \gamma_{out}(m, n_{19})$$

Now, the only edges leading to n_{19} are (n_7, n_{19}) and (n_{17}, n_{19}). Since, by example 5.7.5, $\overline{\delta}_{in}(n_{17}) = unreach$, it follows that:

$$\begin{aligned}\gamma_{in}(n_{19}) &= \gamma_{out}(n_7, n_{19}) = \gamma_{call}(n_7) \cap Perm(n_{19}) \\ &= \gamma_{in}(n_7) \cap Perm(n_{19}) = \{\mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}\}\end{aligned}$$

Since $\mathsf{P}_{credit} \in \gamma(n_{19})$, 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:

$$\gamma_{in}(n_{11}) \quad = \bigcap_{\substack{(m,n_{11})\in E \\ \overline{\delta}_{out}(m,n_{11})\neq unreach}} \gamma_{out}(m, n_{11})$$

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 $\overline{\delta}_{in}(n_6) = unreach$: then, $\overline{\delta}_{out}(n_6, n_{11}) = unreach$, too. Proceeding as in example 5.6.4, we find that $\gamma_{in}(n_2) = \{\mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}\}$. Therefore:

$$\begin{aligned}\gamma_{in}(n_{11}) &= \gamma_{out}(n_3, n_{11}) = \gamma_{call}(n_3) \cap Perm(n_{11}) \\ &= \gamma_{in}(n_3) \cap Perm(n_{11}) = \gamma_{out}(n_2, n_3) \cap Perm(n_{11}) \\ &= \gamma_{trans}(n_2) \cap Perm(n_{11}) = \gamma_{in}(n_2) \cap Perm(n_{11}) \\ &= \{\mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}\}\end{aligned}$$

Since $\mathsf{P}_{debit} \in \gamma(n_{11})$, 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 GP^2 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:

$$\overline{\delta}_{trans}(n_{16}) = \bigcup_{\substack{(m,n_{16})\in E \\ \mathsf{P}_{loan}\in\overline{\delta}_{out}(m,n_{16})}} \overline{\delta}_{out}(m,n_{16})$$

Now, by direct computation of the DP^2 analysis at (n_4, n_{16}), we have:

$$\begin{aligned}\overline{\delta}_{out}(n_4,n_{16}) &= \overline{\delta}_{call}(n_4) \cap \mathit{Perm}(n_{16}) = \overline{\delta}_{in}(n_4) \cap \mathit{Perm}(n_{16}) \\ &= \overline{\delta}_{out}(n_2,n_4) \cap \mathit{Perm}(n_{16}) = \overline{\delta}_{trans}(n_2) \cap \mathit{Perm}(n_{16}) \\ &= \overline{\delta}_{in}(n_2) \cap \mathit{Perm}(n_{16}) = \{\mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}, \mathsf{P}_{loan}\}\end{aligned}$$

On the other hand, from example 5.7.5 we know that $\overline{\delta}_{out}(n_5, n_{16}) = \varnothing$. Moreover, since $\mathsf{P}_{loan} \in \overline{\delta}_{in}(n_{16})$, it follows that $\overline{\delta}_{in}(n_{17}) \neq \mathit{unreach}$. Thus:

$$\begin{aligned}\overline{\delta}_{in}(n_{18}) &= \overline{\delta}_{out}(n_{17},n_{18}) = \overline{\delta}_{trans}(n_{17}) = \overline{\delta}_{in}(n_{17}) \\ &= \overline{\delta}_{out}(n_{16},n_{17}) = \overline{\delta}_{trans}(n_{16}) = \overline{\delta}_{out}(n_4,n_{16}) \\ &= \{\mathsf{P}_{canpay}, \mathsf{P}_{credit}, \mathsf{P}_{debit}, \mathsf{P}_{loan}\}\end{aligned}$$

Since $n_{18} \in \rho(n_5)$ and $\overline{\delta}_{in}(n_{18}) \neq \mathit{unreach}$, we have $\neg\mathsf{kill}^2(n_5)$, although node n_6 is actually unreachable. Then $\gamma_{in}(n_6) = \gamma_{in}(n_5)$, and:

$$\begin{aligned}\gamma_{in}(n_{11}) &= \gamma_{out}(n_3,n_{11}) \cap \gamma_{out}(n_6,n_{11}) = \gamma_{call}(n_3) \cap \gamma_{call}(n_6) \cap \mathit{Perm}(n_{11}) \\ &= \gamma_{in}(n_3) \cap \gamma_{in}(n_6) \cap \mathit{Perm}(n_{11}) = \gamma_{in}(n_3) \cap \gamma_{in}(n_5) \cap \mathit{Perm}(n_{11}) \\ &= \gamma_{in}(n_3) \cap \varnothing \cap \mathit{Perm}(n_{11}) = \varnothing\end{aligned}$$

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 $n_0 : n_3 : n_{11} \vdash \mathsf{JDK}(\mathsf{P}_{debit})$. Thus, the GP^2 analysis is not complete. □

Theorem 5.8.6. GP^2 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 δ to the DP^2 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^2 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 γ^1 and γ^2 be the MFP solutions for the GP^1 and GP^2 analyses, respectively. Then γ^2 is *more accurate* than γ^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 GP^2-solution is non-trivial.

$$
\begin{array}{lcl}
f_{(m,n)}(\langle l_0, l_1, l_2\rangle) & = & \langle f^0_{(m,n)},\ f^1_{(m,n)},\ f^2_{(m,n)}\rangle(\langle l_0, l_1, l_2\rangle) \\
\\
f^0_{(m,n)}(\langle l_0, l_1, \bot\rangle) & = & \varnothing \\
f^0_{\bullet\longrightarrow n}(\langle l_0, l_1, \top\rangle) & = & \mathit{Perm}(n) \\
f^0_{m\longrightarrow n}(\langle l_0, l_1, \top\rangle) & = & \mathit{Perm}(n) \cap \begin{cases} \mathit{Perm}(m) & \text{if } \mathit{Priv}(m) \\ l_1 & \text{otherwise} \end{cases} \\
f^0_{m\dashrightarrow n}(\langle l_0, l_1, \top\rangle) & = & l_1 \\
f^0_{m\hookrightarrow n}(\langle l_0, l_1, \top\rangle) & = & \varnothing \\
\\
f^1_{(m,n)}(\langle l_0, l_1, \bot\rangle) & = & \varnothing \\
f^1_{(m,n)}(\langle l_0, l_1, \top\rangle) & = & \begin{cases} \varnothing & \text{if } \mathtt{kill}^2(\overline{\delta}(n), n) \\ f^0_{(m,n)}(\langle l_0, l_1, \top\rangle) & \text{otherwise} \end{cases} \\
\\
f^2_{(m,n)}(\langle l_0, l_1, \bot\rangle) & = & \bot \\
f^2_{(m,n)}(\langle l_0, l_1, \top\rangle) & = & \begin{cases} \bot & \text{if } \ell(m) = \mathtt{call} \text{ and } m \dashrightarrow n, \\ & \text{or } \ell(m) = \mathtt{check}(P) \text{ and } P \notin \overline{\delta}(n) \\ \top & \text{otherwise} \end{cases}
\end{array}
$$

Table 5.13: Local transfer functions for the GP^2 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 δ and γ 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 $\mathsf{P} \in \delta(n)$, then an `AccessControlException` is thrown. Otherwise, if $\mathsf{P} \in \gamma(n)$ 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.

Algorithm 3: Optimized stack inspection

CHECK-PERMISSION($\mathsf{P}, \sigma, \delta, \gamma$)

```
1  while σ ≠ NIL do
2     n ← POP(σ)
3     if P ∈ δ(n)
4        then throw  "access control exception"
5     if P ∈ γ(n)
6        then return
```

Figure 5.7: The optimized stack inspection algorithm.

$$\frac{}{[] \vdash \mathsf{JDK}^\star(\mathsf{P})} \quad [\mathsf{JDK}^\star_\varnothing]$$

$$\frac{\mathsf{P} \notin \delta(\mathfrak{n}) \quad \sigma \vdash \mathsf{JDK}^\star(\mathsf{P})}{\sigma : \mathfrak{n} \vdash \mathsf{JDK}^\star(\mathsf{P})} \quad [\mathsf{JDK}^\star_\delta]$$

$$\frac{\mathsf{P} \in \gamma(\mathfrak{n})}{\sigma : \mathfrak{n} \vdash \mathsf{JDK}^\star(\mathsf{P})} \quad [\mathsf{JDK}^\star_\gamma]$$

Table 5.14: Specification of the optimized access control policy.

Theorem 5.9.1 (Correctness of $\mathsf{JDK}^\star$). Let G be a control flow graph and σ a reachable state. Then, for any permission P:

$$\sigma \vdash \mathsf{JDK}(\mathsf{P}) \quad \Longleftrightarrow \quad \sigma \vdash \mathsf{JDK}^\star(\mathsf{P})$$

Proof. The proof for the forward implication is carried out by induction on the derivation $\sigma \vdash \mathsf{JDK}(\mathsf{P})$. Case analysis on the last applied rule yields:

- case $[\mathsf{JDK}_\varnothing]$: here $\sigma = []$, and $[] \vdash \mathsf{JDK}^\star(\mathsf{P})$ is deduced by rule $\mathsf{JDK}^\star_\varnothing$.
- case $[\mathsf{JDK}_\prec]$: here $\sigma = \sigma' : \mathfrak{n}$, and the premises of the rule ensure $\mathsf{P} \in \mathit{Perm}(\mathfrak{n})$ and $\sigma' \vdash \mathsf{JDK}(\mathsf{P})$. Applying the inductive hypothesis, we deduce $\sigma' \vdash \mathsf{JDK}^\star(\mathsf{P})$. Now assume, by contradiction, that $\mathsf{P} \in \delta(\mathfrak{n})$. Since δ is DP-sound, this would imply that $\sigma' : \mathfrak{n} \nvdash \mathsf{JDK}(\mathsf{P})$, contradicing our assumption $\sigma' : \mathfrak{n} \vdash \mathsf{JDK}(\mathsf{P})$. Therefore, it must be $\mathsf{P} \notin \delta(\mathfrak{n})$, and the rule $\mathsf{JDK}^\star_\delta$ can be applied to obtain $\sigma' : \mathfrak{n} \vdash \mathsf{JDK}^\star(\mathsf{P})$.
- case $[\mathsf{JDK}_{\mathit{Priv}}]$: here $\sigma = \sigma' : \mathfrak{n}$, and the premises of the rule state $\mathsf{P} \in \mathit{Perm}(\mathfrak{n})$ and $\mathit{Priv}(\mathfrak{n})$. Since γ is non-trivial, this implies that $\mathsf{P} \in \gamma(\mathfrak{n})$, and the $\mathsf{JDK}^\star_\gamma$ rule then yields $\sigma' : \mathfrak{n} \vdash \mathsf{JDK}^\star(\mathsf{P})$.

For the backward implication, assume $\sigma \vdash \mathsf{JDK}^\star(\mathsf{P})$. Again, we proceed by induction on the depth of the call stack. Case analysis on the last applied rule gives:

- case $[\mathsf{JDK}^\star_\varnothing]$: here $\sigma = []$, and $[] \vdash \mathsf{JDK}(\mathsf{P})$ is deduced by rule $\mathsf{JDK}_\varnothing$.
- case $[\mathsf{JDK}^\star_\delta]$: here $\sigma = \sigma' : \mathfrak{n}$, and the premises of the rule say that $\mathsf{P} \notin \delta(\mathfrak{n})$ and $\sigma' \vdash \mathsf{JDK}^\star(\mathsf{P})$. By applying the inductive hypothesis, we obtain $\sigma' \vdash \mathsf{JDK}(\mathsf{P})$. Moreover, as δ is non-trivial, we know that $\mathsf{P} \in \mathit{Perm}(\mathfrak{n})$: then the rule $\mathsf{JDK}_\prec$ can be used to deduce $\sigma' : \mathfrak{n} \vdash \mathsf{JDK}(\mathsf{P})$.
- case $[\mathsf{JDK}^\star_\gamma]$: here $\sigma = \sigma' : \mathfrak{n}$, and by premise we have $\mathsf{P} \in \gamma(\mathfrak{n})$. Since γ is GP-sound, this indeed implies $\sigma' : \mathfrak{n} \vdash \mathsf{JDK}(\mathsf{P})$.

□

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;
    }
}
```

n	$\overline{\delta^0}(n)$	$\gamma^0(n)$	$\overline{\delta^1}(n)$	$\gamma^1(n)$	$\overline{\delta^2}(n)$	$\gamma^2(n)$
n_0	**Permission**		**Permission**		**Permission**	
n_1					*unreach*	
n_2	$\{P_{canpay}, P_{credit}, P_{debit}\}$		$\{P_{canpay}, P_{credit}, P_{debit}\}$		$\{P_{canpay}, P_{credit}, P_{debit}\}$	
n_3						
n_4						
n_5	$\varnothing$		$\varnothing$		$\varnothing$	
n_6					*unreach*	
n_7	$\{P_{canpay}, P_{credit}, P_{debit}\}$		$\{P_{canpay}, P_{credit}, P_{debit}\}$		$\{P_{canpay}, P_{credit}, P_{debit}\}$	
n_8	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\varnothing$	$\{P_{canpay}, P_{credit}, P_{debit}\}$		$\{P_{canpay}, P_{credit}, P_{debit}\}$	
n_9	**Permission**	**Permission**	**Permission**		**Permission**	
n_{10}	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\varnothing$	$\{P_{canpay}, P_{credit}, P_{debit}\}$		$\{P_{canpay}, P_{credit}, P_{debit}\}$	
n_{11}	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\varnothing$	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\varnothing$	$\{P_{canpay}, P_{credit}, P_{debit}\}$	
n_{12}	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\varnothing$	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\{P_{canpay}, P_{credit}, P_{debit}\}$	
n_{13}	**Permission**	**Permission**	**Permission**	**Permission**	**Permission**	
n_{14}	**Permission**	**Permission**	**Permission**	**Permission**	**Permission**	
n_{15}	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\varnothing$	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\{P_{canpay}, P_{credit}, P_{debit}\}$	
n_{16}	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\varnothing$	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\varnothing$	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\varnothing$
n_{17}	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\varnothing$	$\varnothing$	$\varnothing$	*unreach*	*unreach*
n_{18}	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\varnothing$	$\varnothing$	$\varnothing$	*unreach*	*unreach*
n_{19}	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\varnothing$	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\varnothing$	$\{P_{canpay}, P_{credit}, P_{debit}\}$	
n_{20}	**Permission**	**Permission**	**Permission**	**Permission**	**Permission**	
n_{21}	**Permission**	**Permission**	**Permission**	**Permission**	**Permission**	
n_{22}	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\varnothing$	$\{P_{canpay}, P_{credit}, P_{debit}\}$	$\varnothing$	$\{P_{canpay}, P_{credit}, P_{debit}\}$	
n_{23}	**Permission**		**Permission**		**Permission**	
n_{24}						
n_{25}	**Permission**		**Permission**		**Permission**	
n_{26}						

Table A.1: Solutions to the analyses for the e-commerce application.

Index of symbols

Bibliography

[ABLP93] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 4(15):706–734, September 1993.

[ARS97] Anurag Acharya, M. Ranganathan, and Joel Saltz. Sumatra: a language for resource-aware mobile programs, 1997.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[BJLT01] F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model checking security properties of control flow graphs. *Journal of computer security*, 9:217–250, 2001.

[BL00] Gaetano Bigliardi and Cosimo Laneve. A type system for JVM threads. Technical Report UBLCS-2000-06, June 2000.

[CGQ98] Alessandro Coglio, Allen Goldberg, and Zhenyu Qian. Toward a provably-correct implementation of the JVM bytecode verifier. Technical report, Kestrel Institute, Palo Alto, July 1998.

[Dea97] Drew Dean. The security of static typing with dynamic linking. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, Zurich, Switzerland, 1997.

[Dea99] Drew Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, 1999.

[ES99] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: a retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop*, Caledon Hills, September 1999.

[FC98] Philip W. L. Fong and Robert D. Cameron. Proof linking: An architecture for modular verification of dynamically-linked mobile code. In *Proceedings of the Sixth ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'98)*, pages 222–230, Orlando, Florida, November 1998.

[FC99] Philip W. L. Fong and Robert D. Cameron. Proof linking: Modular verification of mobile programs in the presence of lazy, dynamic linking. In *ACM Transactions on Software Engineering and Methodology*, June 1999.

[FC00] Philip W. L. Fong and Robert D. Cameron. Java proof linking with multiple classloaders. Technical Report SFU CMPT TR 2000-04, Simon Fraser University, August 2000.

[FG01] Cédric Fournet and Andrew D. Gordon. Stack inspection: theory and variants. Draft, August 2001.

[FM98] Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98)*, volume 33(10) of *SIGPLAN Notices*, pages 310–327, Vancouver, BC, Canada, October 1998. ACM Press.

[FM99a] Stephen N. Freund and John C. Mitchell. A formal framework for the Java bytecode language and verifier. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99)*, volume 34(10) of *SIGPLAN Notices*, pages 147–166, Denver, CO, November 1999. ACM Press.

[FM99b] Stephen N. Freund and John C. Mitchell. Specification and verification of Java Bytecode subroutines and exceptions. Technical Note STAN-CS-TN-99-91, Department of Computer Science, Stanford University, August 1999.

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '97)*, pages 108–124, 1997.

[GHM00] Etienne Gagnon, Laurie Hendren, and Guillaume Marceau. Efficient inference of static types for Java Bytecode. In Jens Palsberg, editor, *Static Analysis, 7th International Symposium (SAS'2000)*, volume 1824 of *LNCS*, pages 199–219, Santa Barbara, CA, June/July 2000. Springer-Verlag.

[Gon99] Li Gong. *Inside Java 2 platform security: architecture, API design, and implementation.* Addison-Wesley, 1999.

[HKK00] Manfred Hauswirth, Clemens Kerer, and Roman Kurmawytsch. A flexible and extensible security framework for Java code. Technical Report TUV-1841-99-14, Technical University of Vienna, May 2000.

[HT98] Masami Hagiya and Akihiko Tozawa. On a new method for dataflow analysis of Java Virtual Machine subroutines. In Giorgio Levi, editor, *Static Analysis, 5th International Symposium (SAS'98)*, volume 1503 of *LNCS*, pages 17–32, Pisa, Italy, September 1998. Springer-Verlag.

[JLT98a] T. Jensen, D. Le Métayer, and T. Thorn. Security and dynamic class loading in Java: A formalisation. In *Proceedings of the 1998 IEEE International Conference on Computer Languages*, pages 4–15. IEEE Computer Society Press, May 1998.

[JLT98b] T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security policies. Technical Report 1210, IRISA, October 1998.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Syposium on Principles of Programming Languages*, pages 194–206, Boston, MA, October 1973.

[KLO97] Gűnter Karjoth, Danny B. Lange, and Mitsuru Oshima. A security model for Aglets. *IEEE Internet Computing*, 1(4), July/August 1997.

[KS92] Jens Knoop and Bernhard Steffen. The interprocedural coincidence theorem. In *Proceedings of the 4th International Conference on Compiler Construction (CC'92)*, volume 641 of *LNCS*, pages 125–140. Springer-Verlag, 1992.

[KU76] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, 1976.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[LGK+99] C. Lai, L. Gong, L. Koved, A. Nadalin, and R. Schemers. User authentication and authorization in the java platform. In *15th Annual Computer Security Application Reference*, pages 285–290. IEEE Computer Society Press, 1999.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[MF99] Gary McGraw and Edward W. Felten. *Securing Java: Getting Down to Business with Mobile Code.* John Wiley & Sons, New York, NY, 1999.

[MR90] Thomas J. Marlowe and Barbara G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28(2):121–163, 1990.

[MS98] Nimisha V. Mehta and Karen R. Sollins. Expanding and extending the security features of Java. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[Muc97] S. Muchnick. *Advanced compiler design & implementation.* Morgan Kaufmann Publishers, 1997.

[Nip01] Tobias Nipkow. Verified bytecode verifiers. In F. Honsell, editor, *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, LNCS. Springer-Verlag, 2001.

[NNH99] F. Nielson, H. Riis Nielson, and C. L. Hankin. *Principles of Program Analysis.* Springer, 1999.

[NTS01] Naoya Nitta, Yoshiaki Takata, and Hiroyuki Seki. Security verification of programs with stack inspection. In *Proceedings of 6th ACM Symposium on Access Control Models and Technologies (ACM SACMAT2001)*, pages 31–40, Chantilly, VA, May 2001.

[Oak01] Scott Oaks. *Java Security.* O'Reilly & Associates, Inc., second edition, 2001.

[PSS01] François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. In David Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP '01)*, volume 2028 of *LNCS*, pages 30–45. Springer-Verlag, April 2001.

[PV98] Joachim Posegga and Harald Vogt. Java bytecode verification using model checking, October 1998.

[QGC00] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A formal specification of Java™ class loading. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000)*, volume 35(10) of *SIGPLAN Notices*, pages 325–336, Minneapolis, Minnesota, October 2000. ACM Press.

[RG98] A. Rubin and D. Geer. Mobile code security. *IEEE Internet Computing*, 2(6), November/December 1998.

[RRL99] Atanas Rountev, Barbara G. Ryder, and William Landi. Data-flow analysis of program fragments. In *ESEC / SIGSOFT FSE*, pages 235–252, 1999.

[SA98] Ramye Stata and Martín Abadi. A type system for Java Bytecode Subroutines. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, pages 149–160, San Diego, CA, January 1998. ACM Press.

[Sar97] V. Saraswat. Java is not type-safe. Web page at: `www.research.att.com/~vj/main.html`, 1997.

[SBC00] Vugranam C. Sreedhar, Michael G. Burke, and Jong-Deok Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–207, 2000.

[Sch98] Fred B. Schneider. Enforceable security policies. Technical Report TR98-1664, Cornell University, January 1998.

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000)*, volume 35(10) of *SIGPLAN Notices*, pages 264–280, Minneapolis, MN, October 2000. ACM Press.

[SM98] Insik Shin and John C. Mitchell. Java bytecode modification and applet security, 1998.

[SP81] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[SS00] Christian Skalka and Scott Smith. Static enforcement of security with types. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 34–45, Montréal, Canada, September 2000.

[TP00] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '00)*, pages 281–293, 2000.

[WAF00] Dan S. Wallach, Andrew W. Appel, and Edward W. Felten. SAFKASI: a security mechanism for language-based systems. *j-TOSEM*, 9(4):341–378, October 2000.

[Wal99] Daniel S. Wallach. *A New Approach to Mobile Code Security.* PhD thesis, Princeton University, January 1999.

[Wal00] David Walker. A type system for expressive security policies. In *Conference record of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 254–267. ACM Press, 2000.

[WF98] Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1998.

[Wil00] C. Wille. *Presenting* C♯. SAMS Publishing, 2000.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, December 1993.