Project N°: **FP6-015905**

Project Acronym: **MOBIUS**

Project Title: **Mobility, Ubiquity and Security**

Instrument: **Integrated Project**

Priority 2: **Information Society Technologies**

Future and Emerging Technologies

Deliverable D3.2

Intermediate report on embedding type-based analysis into program logics

Due date of deliverable: 2007-03-01 (T0+18)

Actual submission date: 2007-03-28

Start date of the project: **1 September 2005** Duration: **48 months**

Organisation name of lead contractor for this deliverable: **CTH**

Revision 3746M — Final

Project co-funded by the European Commission in the Sixth Framework Programme (2002-2006)		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Contributions

SITE	Contributed to Chapter
CTH	Chapter 1, 2, 5, 6
IoC	Chapter 4
WU	Chapter 1, 3

Executive Summary:
Intermediate report on embedding type-based analyses into program logics

This document summarises deliverable D3.2 of project FP6-015905 (MOBIUS), co-funded by the European Commission within the Sixth Framework Programme. Full information on this project, including the contents of this deliverable, is available online at `http://mobius.inria.fr`.

Secure information flow of programs guarantees that no classified information is leaked to an unauthorised user. Language-based security techniques analyse programs if they satisfy this property for any possible program run. The given guarantees are beyond what encryption based approaches can achieve, which capture only the aspect of access control rights, but do not consider the flow of information in a program, which might access higher classified information then its user is allowed to access.

Type-based systems are a successfully applied technique in this area, which type only those programs secure that do not leak classified information. The type-based systems are highly-efficient and automatic. The drawback is that these systems are also highly specialised and reject too many secure programs as insecure. The high specialisation makes it also necessary to use different type systems for different secure information flow policies, which may lead to an increase in the number of different certificates that need to be supported.

Program logics on the other side are far more general and can be used to encode a wide variety of policies. Further as program logics model programming languages faithfully, they allow to prove properties beyond secure information flow like functional correctness of programs or the adherence to resource policies. Such a program logic has been already developed as part of the MOBIUS project. The drawback of using program logics is that due to their degree of precision and expressiveness, they lack the efficiency of type-based systems and require often user interaction.

In this working task, we develop a logic framework that allows to embed the type based systems in the general logic framework in order to combine the strengthens of both worlds, i.e. the efficiency of type-based system and the expressiveness of program logics. Another advantage is that these embeddings allow to use a uniform certificate for all kinds of properties, which reduces the requirements posed on the certification checkers and therewith their code base.

Contents

Chapter 1

Introduction

The challenge for MOBIUS Task 3.5 "Combining Type-based and Logical Analyses" is to provide the foundations of a framework that allows to integrate logic and type-based analyses of programs. The main goals to be achieved during its run-time are the combination of type-based and logic-based approaches in order to benefit from the advantages of both formalisms as well as to circumvent their disadvantages and to make use of type-based analysis results in order to generate simpler and smaller certificates.

The approaches presented in this deliverable show how type-based analyses can be embedded into (resp. emulated within) a general logical framework. Using a common formalism allows further the creation of uniform certificates, thus reducing the complexity of the certificate checkers on the client's side.

Type-based analyses can be found in manifold variations used to check statically that programs satisfy specific and well-defined properties like

1. the adherence to certain secure information flow policies as identified in Task 1.1 and further worked out in Task 3.2.

2. the liveness of variables, which allows an optimising compiler to eliminate dead-code.

Two of the three approaches presented in this intermediate report aim on embedding type-based analyses for secure information flow. Secure information flow means that programs accessing and processing confidential data do not leak classified information to a potential observer (for example, the user), she is not entitled to receive. Approaches like cyphering are not strong enough as they are restricted to access control, but—as many scenarios require programs to access information classified for their users—it must be ensured that these programs do not leak any secret information. Analysing the flow of information in a program allows to guarantee that no confidential information is leaked while the program is processed. For example, in its simplest case by printing the password in clear text form on the screen.

But information leaks must not be thus obvious: Assume that a program computes a checksum of a password and stores it in a variable that can be publicly observed. A user might then be able to reduce the search space considerably. The possible search space reduction depends of course on the properties of the chosen checksum function.

Ensuring secure information flow becomes even harder as not all information leaks can be prevented in practise. For example, entering a password releases at least the information, if it has been right or wrong. Consequently, secure information flow cannot be assumed to be an all-or-nothing property, instead one must be able to specify more elaborated security properties, which allow to define the amount of information that might be released.

Type-based and logic-based systems can be used to ensure secure information flow inside a program. The integration of type-based systems into a program logic allows to combine the strengthen of both approaches, i.e. the high efficiency and automation of type-based systems and the expressiveness and generality of logic-based systems.

The third approach presented in this deliverable, focuses on the liveness of variables. Intuitively a variable is considered to be live at a certain program point, if its current assigned value is used and contributes to the programs result or behaviour. The results of a liveness analysis can be easily used by an optimising compiler to eliminate dead-code in programs. In particular the results of the third approach indicate how to transform functional correctness proofs of the original (not optimised) program into corresponding functional correctness proofs of the optimised program.

In this deliverable we give an intermediate report about the results achieved so far in the task's first few months lifespan.

1.1 Type and logic-based systems

Different approaches—for example type-based program analysis, deductive program verification or model checking—have been developed to check that a program satisfies certain properties. These approaches differ in the kind of properties they allow to ensure, or more general, in their expressiveness (number of properties that can be expressed within the same formalism), in soundness (absence of false positives, i.e. no incorrect programs are classified as sound), completeness (no false negatives, i.e. no correct programs is rejected as unsound) and in the degree of automation. Usually an advantage in one of these categories comes with a drawback in one of the others.

In this working task, we deal with formal systems, which are required and have to be proven to be sound. These formal systems speak about properties of programs and therefore need a (formal) language to represent these programs and to state the properties of interest.

Formal languages are usually defined in terms of grammars, which describe precisely the allowed set of symbols and how these are composed to form valid expressions (words) and/or sentences. Typical representatives of formal languages are program languages like Java or logics.

These languages are assigned a well-defined meaning, or in other words, a mathematical semantics by their defining formal system. Such a semantic can be defined in different ways (or flavours), for example operational or descriptive.

Example 1 (Semantics) *Let* `x`, `y` *be program variables and* `x*y` *be a valid expression in the given program language (formal system) of interest. An operational semantics may assign a meaning to this expression by a detailed description of a multiplication algorithm. In contrast, a descriptive semantics would simply interpret* $*$ *as the mathematical multiplication function* $\times$ *and define its value without the necessity to give a concrete algorithm.*

In this deliverable type-based systems and program logics are the formal systems of interest. Bicolano defines an operational semantics for the subset of Java bytecode supported in MOBIUS. It models the precise, but informal natural language specification of the Java virtual machine within a strict mathematical framework. In the end any formalism developed in MOBIUS must be proven sound wrt. Bicolano.

The MOBIUS base logic is a Hoare style logic that provides a suitable abstraction layer designed to simplify the reasoning about properties of Java bytecode programs. The base logic has been proven sound relative to Bicolano. Both, Bicolano and the MOBIUS base logic, are explained in detail in [15]. The authors of [15] implemented both as theories of the theorem prover Coq [16].

Writing specifications in the MOBIUS base logic itself require a profound knowledge of logic, MOBIUS defines therefore the specification language called Bytecode Modelling Language (BML), which allows to annotate programs in a *design by contract* style using a Java like notation, that is easier to handle and more familiar to developers. BML specifications can then be translated into Hoare assertions.

Secure information flow is one of the main application scenarios of this working task and will serve as a running example in the following explanations. Nevertheless most of the drawn conclusions are also valid for other kinds of analysis like liveness of variables.

When establishing a property like secure information flow, we are mainly interested to prove that no classified information is leaked, while processing a program. Language-based information flow analysis is intended to ensure this kind of end-to-end secure information flow in a program. Type-based systems have proved to be a useful approach to automatically ensure secure information flow. Their use has been suggested first by Volpano et al. in [36]. Sect. 2 is centred around the information flow sensitive type-based system developed in [24], while Sect. 3 refers to the type system introduced in [8].

While type-based systems allow to check programs for the properties they have been designed for automatically. The necessary specialisation of today's type systems, which is on one side the reason for their high-efficiency, but contemporary also their greatest disadvantage.

Currently, many secure programs are rejected as insecure. One reason for this behaviour is that those type-based systems do not keep track of information like the possible range of values a location may have at a certain execution point. But this kind of value sensitivity would be necessary to exclude certain kinds of exceptions as for example, those raised by division-by-zero, whose absence can only be proven if knowledge about the value of the divisor is kept.

Although we are confident that current and future research in this area will increase the precision of the type-based systems, the use of the more general framework provided by program logics offers several additional advantages:

- *higher precision* as program logics usually axiomatise the semantics of programming language faithfully.
- *higher expressiveness* as program logics allow to specify and guarantee nearly arbitrary functional properties of programs.
- generation of *uniform certificates* for all kinds of properties keeping the code base of the certificate checkers small.

The first results that we present in this deliverable encourage us that the embedding of type-based systems into a program logic is a realistic and achievable goal. Further, these results strongly indicate that these embeddings combine the advantages of type-based system with those of logic frameworks.

1.1.1 Type-based systems

Many of today's programming languages are typed, i.e. they are equipped with a set of basic types like `int` or `boolean` and often allow the programmer to define their own types, which are composites of others, e.g. classes or records. These types are attached to locations (program variables, fields etc.) and methods. The compiler checks then statically, if the program to be compiled is typed correctly, such that for example no integer typed program variable is assigned a boolean value or vice versa. To perform this test the compiler gets equipped with a set of typing rules, which allow to derive the type of expressions (and/or statements) by looking at the types of their components. If the compiler is not able to derive that the program is typed correctly, it will reject the program and print an error message.

A program language is then called *type safe* if it respects its typing at runtime, or the other way round, a type system is sound if only programs can be typed within it, that respects the typing at runtime.

Besides additional complications like inheritance structures the above mentioned typing is of a very basic shape and not sufficient to establish further properties like definite assignment of local program variables, liveness of program variables or the secure information flow policies considered later in this deliverable. In order to ensure conformance to these properties additional types are required, which capture more information than pure value restrictions: for the definite assignment case one could model a local program variable type as a tuple $(valueType, valueAssigned)$, where $valueTupe$ is the normal user defined type of the value and $valueAssigned \in \{asgn, notAsgn\}$ with $asgn$ indicating that the variable has already been assigned a value. An assignment statement x=y∗z; shall than be only typable, if the $valueAssigned$ components of variable y and variable z is $asgn$.

The previous example is also useful to explain the difference between flow sensitive and insensitive type systems. Flow-sensitive type systems respect the execution order of statements, whereas insensitive systems do not. This means a type insensitive system will usually not be able to type a program as safe wrt. to definite assignment if the declaration of a local program variable does not coincide with its first assignment[1].

[1]In order to circumvent this restriction one has to transform the program into an equivalent one that declares a variable at its first use. But those transformations are not always trivial and need to be proven sound.

$$\textsf{asgn\_high}\ \frac{}{[pc] \vdash \text{h=exp;}} \qquad \textsf{asgn\_low}\ \frac{\vdash exp : low}{[low] \vdash \text{l=exp;}}$$

$$\textsf{composition}\ \frac{[pc] \vdash C_1; \qquad [pc] \vdash C_2;}{[pc] \vdash C_1; C_2;} \qquad \textsf{while}\ \frac{\vdash exp : pc \qquad [pc] \vdash \alpha}{[pc] \vdash \textbf{while}\ (!exp)\ \alpha}$$

$$\textsf{condition}\ \frac{\vdash exp : pc \quad [pc] \vdash C_1 \quad [pc] \vdash C_2}{[pc] \vdash \textbf{if}\ (!exp)\ C_1\ \textbf{else}\ C_2}$$

Figure 1.1: Flow insensitive secure type system equivalent to Volpano et. al. taken from [32] (excerpt)

$$\textsf{assignment}_\textsf{H}\ \frac{}{\{P[x/exp]\}x\text{=}exp;\{P\}} \qquad \textsf{composition}_\textsf{H}\ \frac{\{P\}C_1;\{R\} \quad \{R\}C_2;\{Q\}}{\{P\}C_1;C_2;\{Q\}}$$

$$\textsf{condition}_\textsf{H}\ \frac{\{P \wedge exp\}C_1\{Q\} \quad \{P \wedge !exp\}C_2\{Q\}}{\{P\}\textbf{if}\ (exp)\ C_1\ \textbf{else}\ C_2\{Q\}}$$

Figure 1.2: Hoare style calculus for a simple while-language (excerpt)

The first type-based system used to check secure information flow in a program has been developed by [36]. The type system is flow insensitive and fails therefore to type a program like "$l = h; l = 0;$" as secure. The program is in fact secure as it is equivalent to l=0;. But flow insensitive type systems like the one shown in Fig. 1.1 fail to derive this property as they do not respect the order of statements and will reject the previously mentioned program as insecure because of the first assignment. This becomes clear when looking at the composition rule of the type-based system in Fig. 1.1, which requires both statements to be typed independently as secure.

This shortcome can be circumvented by the introduction of new variables or by using a flow sensitive type system as [24]. Another problem where today's type systems fail are programs like

$$\textbf{if}\ (h! = 0)\ h = 1/h;$$

which assigns only to a high level program variable and may therefore be considered as secure. The problem is the division by h which raises an ArithmeticException in cases where $h \doteq 0$ and could therewith leak information about h to an observer. Of course this situation cannot occur as the if-condition excludes the malicious case, but as current type-based systems for secure information flow analysis are value insensitive, they will reject the above program as insecure.

1.1.2 Program logics

In order to ensure a certain property with a type system one has to encode it in the set of available types and corresponding type derivation rules. The chosen setup is then only specific wrt. to the modelled property, i.e. for different properties different type systems are required and, in addition, often incomplete but admittedly complete automatic and efficient. Complementary to this are program logics, which allow to specify nearly any kind of property in the same framework, but usually require user interaction.

Program logics are often Hoare style logics or dynamic logics, which can be seen as a closure of Hoare logics under quantifiers. Dynamic logics are defined in more detail in Sect. 2.1.2. Therefore we concentrate on Hoare style logics for the remaining section. In a logic (functional) properties of programs are expressed in terms of logic formulae. For example the formula $result \doteq x * y$ evaluates to true (or holds) in states where the value of variable $result$ is equal to the product of the value of the variables x and y. A program specification is then given as a so called Hoare triple $\{pre\}\ prg\ \{post\}$, where pre and $post$ are logic formulas. The program prg satisfies its specification if started in state where formula pre holds and if program prg terminates then the formula $post$ holds in the final state. For this kind of logic, on can define a complete calculus that models the semantics of the programming language faithfully and allows to derive all properties that can be expressed by logic formulas.

Example 2 *The Hoare triple*

$$\{n \doteq c \land c >= 0\} result = 0;\ \textbf{\textit{while}}\ (n-- > 0)\ result{+}{=}m; \{result \doteq m \times c\}$$

specifies that if the program is executed in a state where the value of n *is equal to* c *and greater than* 0 *then in its final state the computed value of* $result$ *is the product of* m *and* c*.*

A Hoare style calculus for a simple while language is shown in Fig. 1.2. In contrast to the typing system, the calculus models the semantics faithfully, in particular the assignment rule keeps track of the precise value assigned to a program variable.

In order to formalise secure information flow in a logic setting one can use the following semantic encoding of non-interference of program variables:

> The observable result of a program run must not depend on the content of classified data.

Given a program $prg(l, h)$ using only the variables l and h. Further, let $prg(l', h')$ denote a copy of the previous program, but where the program variables l and h have been replaced by new variables l' resp. h'. Assume a user can observe the values of the low security level variables l, l' before and after a program run, but not the value of the high security level variables h, h'. The above non-interference semantics requires now that a program run with equal low level input results in a state which coincides on the value of the low level input value, this means the final state is indistinguishable from the user's point of view. A syntactical encoding of this property as formalised in [19, 10] can be given as:

$$\{l \doteq l'\} prg(l, h); prg(l', h'); \{l \doteq l'\}$$

1.2 Terminology and notation

In the current stage the developed approaches are too different to be presented using identical notations and terminology. In order to achieve at least as far as possible a uniform presentation this section identifies and gives the definition of some shared concepts together with the notation defined by and used to represent these concepts.

In order to model assignments of variables that means to update a certain (set of) location(s) in the store or when for example the execution of a statement leads to a change of a variable's security level, i.e. its security type, the corresponding formalisation usually maps the function (representing the memory or typing at a certain program point) to another one identical with the former except for the changed value (here: variable). The notion of *function updates* proves convenient for this kind of mappings:

Definition 1.2.1 (Function Update) *Given a not necessary total function* $f : S \rightarrow T$*. The function* $f[y \mapsto v]$ *coincides with* f *on all* $x \in S$ *except for* y *where it has value* v*, i.e.*

$$f[y \mapsto v](x) := \begin{cases} v, & x = y \\ f(x), & \textit{otherwise.} \end{cases}$$

Type-based systems and deductive systems define usually a purely syntactic calculus or derivation system, that relates different expressions of the used formal language. This relation is called *syntactic consequence relation* or as symbol $\vdash^S$: $\mathcal{P}(S) \rightarrow S$ (for a given system S). We omit the superscript S, if the referred system is obvious in the current context or irrelevant for the discussion.

These syntactic consequence relations are (often) defined as rule-based systems. A rule is a schema that describes how to perform a single derivation step by purely syntactic operations. These rules are usually defined using the notation shown below:

$$\mathsf{rname}\ \frac{premise_1 \quad \ldots \quad premise_n}{conclusion}$$

where

- the identifier *rname* is a name uniquely identifying the rule,
- $premise_1, \ldots, premise_n$ are the assumptions of the rule that need to be established in order to derive
- the consequence *conclusion* of the rule.

Example 3 (Conjunction Rule) *One example of a valid rule is the conjunction rule as used in many logic calculi:*

$$\text{Conjunction}\ \frac{\Gamma \vdash A,\ \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}$$

If A is valid and B is valid, then their conjunction is also a valid formula.

The syntax is assigned a mathematical meaning (semantics) by its formal system S, the idea of formal systems is to derive mathematical valid proofs of properties by performing only simple syntactical operations.

Definition 1.2.2 (Semantic Consequence) *Given a formal system S and a property P. If the property P is valid and expressible in S, we say P is a semantic consequence of S or short $\models^S P$. This means there exists a mathematical proof for P in the context of S.*

Definition 1.2.3 (Soundness and Completeness) *Given a formal system S and a property P expressed in terms of the formal language defined by S. The syntactical consequence relation $\vdash^S$ is called* sound *if $\vdash^S P$ implies $\models^S P$.*

If the opposite direction holds, i.e. if $\models^S P$ then there exists a syntactic derivation $\vdash^S P$, then we call $\vdash^S$ complete*.*

The next few definitions introduce a few common notions for modelling secure information information flow in a typed based setting. Elementary is the notion of a security type lattice, which defines the available security types and their order.

Definition 1.2.4 ((Security) Types; Type Lattice) *The symbol $\mathcal{T}$ denotes the set of all security types. Further we require that $(\mathcal{T}, \preceq)$ forms a complete finite lattice.*

Note 1 *The following sections may add additional requirements on the type lattice if necessary, for example that $\preceq$ has to be a linear order.*

Note 2 *Given types t_1, t_2, we defined $t_1 \sqcup t_2$ to denote the greatest upper bound, resp. $t_1 \sqcap t_2$ as the least lower bound of the types t_1 and t_2.*

The typing function assigns any typable program element a security type. Typable program elements are usually all expressions, but depending on the concrete type framework also statements can have a security type.

Definition 1.2.5 (Typing) *The typing function $\nabla : T(S) \to \mathcal{T}$ assigns each type-able element $e \in T(S)$ of the formal system S its (security) type $\nabla(e)$ or equivalent $e : \nabla(e)$.*

Example 4 (Security type of a variable) *Given the security types $\mathcal{T} := \{low, high\}(low \preceq high)$ and program variables x and y, we may define $\nabla(x) := low$ and $\nabla(y) := high$ declaring variable x to be of low confidentiality resp. variable y of high confidentiality.*

Such a typing can be continued easily on statements. For example, the copy assignment statement that assigns a low level variable a constant is usually also typed with a low security level: $\nabla(\mathtt{x := 10;}) = low$.

Note 3 *The general typing function ∇ is sometimes to coarse and will be often decomposed into a class of functions like κ_l or κ_h assigning local variables resp. heap locations a security type.*

In order to scale up to real world programs, it is necessary to treat method invocations in a modular way. Therefore one needs to specify the effects a method invocation may have on observable locations. In order to specify a method according to secure information flow properties the following notion of a method type signature is used:

Definition 1.2.6 (Method Type Signature) *The security signature of a method is defined as*

$$\kappa_l \longrightarrow^{\kappa_h} \kappa_r$$

where

- *κ_r describes the security levels of the return normal and exceptional return values of a methods. Therefore it is a list $n : \kappa_n, e_1 : \kappa_{e_1}, \ldots, \kappa_{e_n}$, where κ_n is the security level of the normal return value and the security types κ_{e_i} denotes the security level of an environment in which an exception of type e_i might be propagated. The notation $\kappa_r[n]$ resp. $\kappa_r[e_i]$ is used for κ_n resp. κ_{e_i}.*
- *κ_h describes the security level of heap changes that may be performed by the method.*
- *κ_l describes the security levels appropriate for the local variables of the method.*

The above introduced definitions establish some common notions of the succeeding sections. The section itself will recall and refine them if necessary to the required extend.

1.3 Translation approaches in comparison

This section roughly compares the different translation approaches introduced in the succeeding chapters. As already mentioned in the previous sections, all considered type systems and translations are sound with respect to the specified secure information flow policy. This means that only for programs satisfying the policy the corresponding type derivation or proof can be found.

The translation approach presented in Sect. 2 embeds a secure information flow type analysis into a logical framework by “emulating” the typing rules with derivation rules of the program logic. Starting from a semantical formalisation of the non-interference property in terms of a logical formula, a sound, abstraction-based program logic calculus is designed to abstract away from information that is not necessarily required for proving non-interference properties. The resulting framework is proven to be as complete as the emulated type based system, but due to lazy abstraction and other optimisations it remains more expressive and precise, which allows to treat more programs than the type system.

The generated proof obligation grows in its basic version linear with the number of program variables for which non-interference has to be proven. The expression “basic version” refers to the pure non-interference proof obligation without modelling declassification aspects. The certificate growth should be in acceptable (linear) ranges, but this has not been proven yet.

The specification of the program is required to provide a security typing of locations, which allows to be expressed (and computed) as a dependency relation ship between locations, i.e. on which other locations may the value of a location at most depend after an arbitrary program run. In order to model declassification aspects the specification must also provide an extensional described partition of the domain of the input data. The user/attacker is than allowed to deduce the partition by performing several program runs, but not more.

The suggested approach requires only a minor extension to the already provided logical MOBIUS framework, namely a more general shape of the proof obligation allowing nested quantification in order to express the dependency between the different locations. This generalisation is easily possible as the formalisation of the MOBIUS base logic is provided as a Coq theory and Coq allows already the required nesting of quantifiers. Lifting the described embedding to the complete MOBIUS logic and therewith to a real object oriented setting is current ongoing work.

The translation described in Sect. 3 is of a different kind. The presented approach translates the non-interference formalisation used by the type system defined in [9] in terms of the specification language BML developed in the MOBIUS project. The soundness of the translation and its completeness relative to the type-based system is proven.

The required program specification consists of a global policy of security levels of fields, a type specification for methods and in particular a region structure of the bytecode, which allows to treat certain program parts as units in order to compensate the lack of structure in bytecode programs. In addition some further functions and values are required which can be computed mostly statically have to be provided. In addition for each method the maximal stack size for any program run must be known. Based on these necessary information is encoded in ghost variables representing tables which capture necessary typing and model information used finally in the generated method specification expressing the wanted non-interference property.

The translation makes intensive use of ghost fields, whose support requires the base logic to be extended. Currently a transparent extension framework for the base logic is discussed and developed in Task 3.2. This framework will allow the support of ghost fields, which are not only required by the presented translation, but also by other working groups and tasks, e.g. those concerned with the specification of resource policies.

The provided translation allows to plug-in more elaborated declassification properties in a simple way. It already incorporates direct support for method calls and exception handling. Propositions about the growth of proofs and certificates are future work.

Sect. 4 justifies the embedding of type-based analyses into a logic framework on a foundational level. It shows along a type-based analysis of program variable liveness that type-based systems can be seen as an applied version of a more foundational Hoare style logic. The Hoare style logic models the same semantics as used by the type-based system in a direct manner or even in terms of a more general trace semantics. The soundness of such foundational Hoare logics is easier to understand and prove.

It is then shown that a type derivation step can be translated into a Hoare assertion and thus it is possible to construct a proof in the Hoare logic from a given type derivation. While the construction of a proof in the Hoare logic is in general much harder to find, the construction of such a proof based on a given type analysis is not. The sections demonstrates also how to perform program optimisations by extending the type-based system with transformations that allow optimisations like the elimination of dead-code. It mentions further shortly that the provided link and translation between type-based analysis and Hoare logic allows to transform functional correctness proofs for the original program into proofs for the transformed (optimised) program automatically.

A simplified overview about the different translation approaches is shown in Fig. 1.3.

1.4 Structure of the deliverable

The deliverable is structured as follows: In Sect. 2 CTH presents an embedding of the type-based secure information flow analysis developed in [24] into a program logic framework such that a given type derivation can be simulated by a proof in the logic. The theorem establishing the simulation property is proven in a constructive manner such that it allows to conclude that it is not significantly harder to find the proof than the equivalent type derivation and it would also allow to translate a given type derivation into a proof in a logic enabling the combination of static typing systems by maintaining a general uniform certificate.

An alternative approach to perform secure information flow analysis within a logic framework is presented in Sect. 3 by WU. In this approach the authors use the type-based information flow system for Java bytecode described in [8, 9] that guarantees non-interference as starting point and give a translation of the type-based system into the MOBIUS program logic. As main theorem it is shown that the translated proof obligation is a valid formula if the program is typable in the type based information flow system.

The IoC presents in Sect. 4 a foundational justification for the translation of analysis type systems

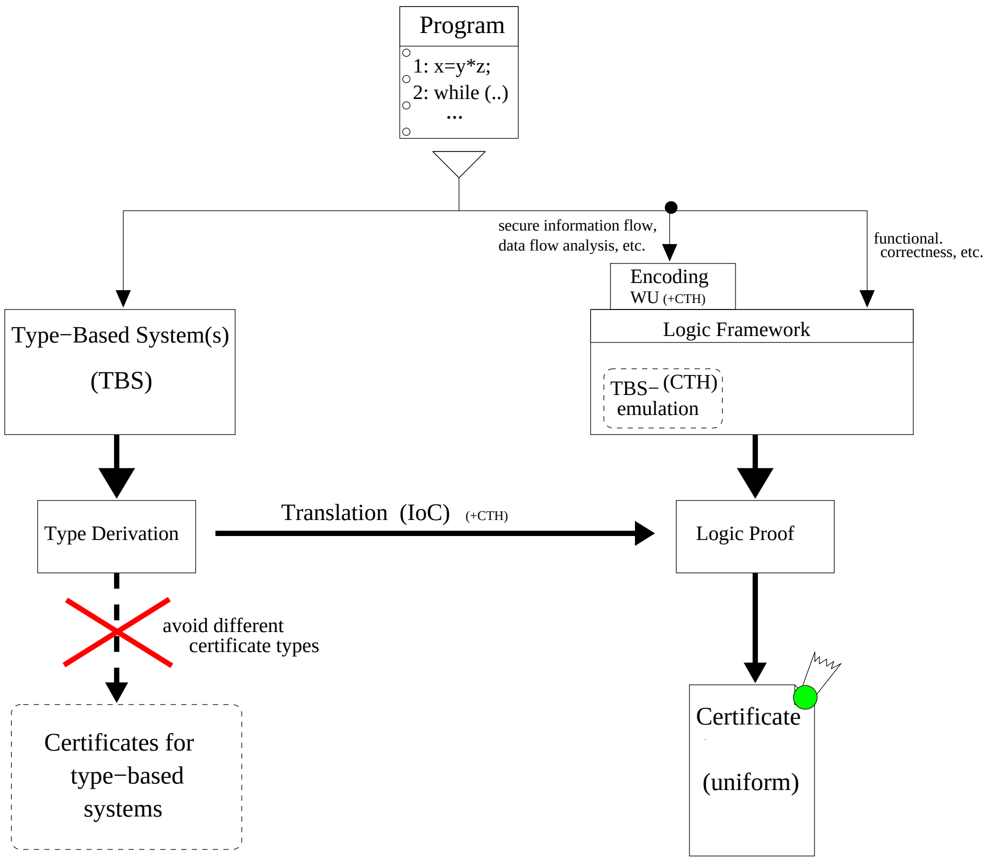


Figure 1.3: Overview about the different translation approaches

into a program logic. Therefore they prove that analysis type systems are an *applied* version of more *foundational* Hoare logics, where the latter one describes either the same property semantics as the applied version, but without approximation considerations or a more general transition trace semantics. These makes the foundational easier to trust and therewith, as the applied version can be proven sound wrt. to the foundational systems, allows to trust the more elaborated applied systems.

Sect. 5 gives an outline of the future work, before Sect. 6 summarises the achieved results and closes the report.

Chapter 2

Integration of a security type system into a program logic

In this chapter we present an embedding of a type-based analysis for secure information flow into a program logic. Therefore the type-based system is "emulated" by an abstraction based calculus for the program logics. The calculus is proven sound and to be as efficient as the type-based system in case of typeable programs, but provides higher expressiveness and precision.

In the following sections we will skip most proofs. A complete account can be found in [21].

2.1 Background

2.1.1 Non-interference analysis and declassification

Generally speaking, a program has secure information flow if no knowledge about some given secret data can be gained by executing this program. Whether or not a program has secure information flow can hence only be decided according to a given security policy discriminating secret from public data. In our considerations we adopt the common model where all input and output channels are taken to be program variables. The semantic concept underlying secure information flow then is that of non-interference: nothing can be learnt about a secret initially stored in variable `h`, by observing variable `l` after program execution, if the initial value of `h` *does not interfere with* the final value of `l`. Put differently, the final value of `l` must be *independent* of the initial value of `h`.

This non-interference property is commonly established via security type systems [32, 24, 36, 4], where a program is deemed secure if it is typable according to some given policy. Type systems are used to perform flow-sensitive as well as flow-insensitive analyses. Flow-insensitive approaches (e.g. [36]) require every subprogram to be well-typed according to the *same* policy. Recent flow-sensitive analyses [24, 4] allow the types of variables to change along the execution path, thereby providing more flexibility for the programmer. Like these type systems, the program logic developed in this chapter is termination insensitive, meaning that a security guarantee is only made about terminating runs of the program under consideration.

The type system of Hunt & Sands [24] is depicted in Fig. 2.1. The type p represents the security level of the program counter and serves to eliminate indirect information flow. The remaining components of typing judgements are a program α and two typing functions $\nabla, \nabla' : \text{PVar} \to \mathcal{L}$ mapping program variables to their respective pre- and post-types. The type system is parametric with respect to the choice of security types; it only requires them to form a (complete) lattice $\mathcal{L}$. In this chapter, we will only consider the most general[1] lattice $\mathcal{P}(\text{PVar})$. One may thus think of the type $\nabla(v)$ of a variable v as the set of all variables that v's value may depend on at a given point in the program. A judgement $p \vdash^{\text{HS}} \nabla \; \{ \; \alpha \; \} \; \nabla'$ states that in context p the program α transforms the typing (or dependency approximation) ∇ into ∇'. We note that rule $\mathsf{Assign}^{\text{HS}}$ gives the system its flow-sensitive character, stating that variable v's type is changed by an

[1]In the sense that any other type lattice is subsumed by it, see [24, Lem. 6.8].

$$\frac{}{p \vdash^{\mathrm{HS}} \nabla \ \{ \ \} \ \nabla} \ \mathsf{Skip}^{\mathrm{HS}}$$

$$\frac{\nabla \vdash E : t}{p \vdash^{\mathrm{HS}} \nabla \ \{ \ v = E \ \} \ \nabla[v \mapsto p \sqcup t]} \ \mathsf{Assign}^{\mathrm{HS}}$$

$$\frac{p \vdash^{\mathrm{HS}} \nabla \ \{ \ \alpha_1 \ \} \ \nabla' \qquad p \vdash^{\mathrm{HS}} \nabla' \ \{ \ \alpha_2 \ \} \ \nabla''}{p \vdash^{\mathrm{HS}} \nabla \ \{ \ \alpha_1 \ ; \ \alpha_2 \ \} \ \nabla''} \ \mathsf{Seq}^{\mathrm{HS}}$$

$$\frac{\nabla \vdash b : t \qquad p \sqcup t \vdash^{\mathrm{HS}} \nabla \ \{ \ \alpha_i \ \} \ \nabla' \ \ (i = 1, 2)}{p \vdash^{\mathrm{HS}} \nabla \ \{ \ \mathbf{if} \ b \ \alpha_1 \ \mathbf{else} \ \alpha_2 \ \} \ \nabla'} \ \mathsf{If}^{\mathrm{HS}}$$

$$\frac{\nabla \vdash b : t \qquad p \sqcup t \vdash^{\mathrm{HS}} \nabla \ \{ \ \alpha \ \} \ \nabla}{p \vdash^{\mathrm{HS}} \nabla \ \{ \ \mathbf{while} \ b \ \alpha \ \} \ \nabla} \ \mathsf{While}^{\mathrm{HS}}$$

$$\frac{p_1 \vdash^{\mathrm{HS}} \nabla_1 \ \{ \ \alpha \ \} \ \nabla'_1}{p_2 \vdash^{\mathrm{HS}} \nabla_2 \ \{ \ \alpha \ \} \ \nabla'_2} \ \mathsf{Sub}^{\mathrm{HS}} \qquad\qquad p_2 \sqsubseteq p_1, \nabla_2 \sqsubseteq \nabla_1, \ \nabla'_1 \sqsubseteq \nabla'_2$$

Figure 2.1: Hunt & Sands' flow-sensitive type system for information flow analysis

assignment $v = E$ to E's type as given by the pre-typing ∇ joined with the context type p. The type t of an expression E in a typing ∇ can simply be taken to be the join of the types $\nabla(v)$ of all free variables v occurring in E, which we denote by $\nabla \vdash E : t$. Joining with the context p is required to accommodate for leakage through the program context, as in the program "**if** (h) $\{l = 1\}$ **else** $\{l = 0)\}$", where the initial value of h is revealed in the final value of l. A modification of the context p can be observed, e.g., in rule $\mathsf{If}^{\mathrm{HS}}$, where the sub-derivation of the two branches of an if statement must be conducted in a context lifted by the type of the conditional.

2.1.2 A program logic with updates

Following [24], the program logic that we investigate is a simplified version of dynamic logic (DL) for Java Card [11]. The reason why we use a dynamic logic here in the first place and not the MOBIUS base logic itself is that at the time the work has been performed the base logic has not yet been available. As dynamic logic subsumes Hoare logics or in other words can be seen a first-order closure of Hoare logics, it is not to far away from the base logic and the transfer of the achieved results is currently ongoing and near future work (see also Sect. 2.3 and Sect. 5).

The most notable difference of DL to standard first-order dynamic logic for the simple while-language [23] is the presence of an explicit operator for simultaneous substitutions (called *updates* [30]). While updates become particularly useful when more complicated programming languages (with arrays or object-oriented features) are considered, in any case, they enable a more direct relation between program logic and type systems.

A *signature* of DL is a tuple $(\Sigma, \mathrm{PVar}, \mathrm{LVar})$ consisting of a set Σ of *function symbols* with fixed, non-negative arity, a set PVar of *program variables* and of a countably infinite set LVar of *logical variables.* Σ, PVar, LVar are pairwise disjoint. Because some of our rules need to introduce fresh function symbols, we assume that Σ contains infinitely many symbols for each arity n. Further, we require that a distinguished nullary symbol $\mathit{TRUE} \in \Sigma$ exists. *Rigid terms* t_{r}, *ground terms* t_{g}, *terms* t,[2] *programs* α, *updates* U and *formulae* ϕ are then defined by the following grammar, where $f \in \Sigma$ ranges over functions, $x \in \mathrm{LVar}$ over

[2]Both rigid terms and ground terms are terms.

logical variables and $v \in \mathrm{PVar}$ over program variables:

$$\begin{array}{rcl} t_{\mathrm{r}} & ::= & x \mid f(t_{\mathrm{r}}, \dots, t_{\mathrm{r}}) \\ t & ::= & t_{\mathrm{r}} \mid t_{\mathrm{g}} \mid f(t, \dots, t) \mid \{\, U \,\}\, t \\ \phi & ::= & \phi \wedge \phi \mid \forall x.\ \phi \mid \ \dots \ \mid t = t \mid [\alpha]\, \phi \mid \{\, U \,\}\, \phi \\ \alpha & ::= & \alpha \ ; \ \dots \ ; \ \alpha \mid v = t_{\mathrm{g}} \mid \mathbf{if}\ t_{\mathrm{g}}\ \alpha\ \mathbf{else}\ \alpha \mid \mathbf{while}\ t_{\mathrm{g}}\ \alpha \end{array} \qquad \begin{array}{rcl} t_{\mathrm{g}} & ::= & v \mid f(t_{\mathrm{g}}, \dots, t_{\mathrm{g}}) \\ U & ::= & \epsilon \mid v := t,\ U \end{array}$$

For the whole chapter, we assume a fixed signature $(\Sigma, \mathrm{PVar}, \mathrm{LVar})$ in which the set $\mathrm{PVar} = \{v_1, \dots, v_n\}$ is finite, containing exactly those variables occurring in the program under investigation.

A *structure* is a pair $S = (D, I)$ consisting of a non-empty *universe* D and an *interpretation* I of function symbols, where $I(f) : D^n \to D$ if $f \in \Sigma$ has arity n. *Program variable assignments* and *variable assignments* are mappings $\delta : \mathrm{PVar} \to D$ and $\beta : \mathrm{LVar} \to D$. The space of all program variable assignments over the universe D is denoted by $PA^D = \mathrm{PVar} \to D$, and the corresponding flat domain by $PA^D_\perp = PA^D \cup \{\perp\}$, where $\delta \sqsubseteq \delta'$ iff $\delta = \perp$ or $\delta = \delta'$.

While-programs α are evaluated in structures and operate on program variable assignments. We use a standard denotational semantics for such programs

$$[\![\alpha]\!]^S : PA^D \to PA^D_\perp$$

and define, for instance, the meaning of a loop "**while** $b\ \alpha$" through

$$[\![\mathbf{while}\ b\ \alpha]\!]^S \ =_{\mathrm{def}}\ \bigsqcup_i w_i, \qquad w_i : PA^D \to PA^D_\perp$$

$$w_0(\delta)\ =_{\mathrm{def}}\ \perp, \quad w_{i+1}(\delta)\ =_{\mathrm{def}} \begin{cases} (w_i)_\perp([\![\alpha]\!]^S(\delta)) & \text{for } val_{S,\delta}(b) = val_S(TRUE) \\ \delta & \text{otherwise} \end{cases}$$

where we make use of a 'bottom lifting': $(f)_\perp(x) = \mathit{if}\ (x = \perp)\ \mathit{then} \perp \mathit{else}\ f(x)$.

Likewise, updates are given a denotation as total operations on program variable assignments. The statements of an update are executed in parallel and statements that literally occur later can override the effects of earlier statements:

$$\begin{array}{l} [\![U]\!]^{S,\beta} : PA^D \to PA^D \\ [\![w_1 := t_1, \dots, w_k := t_k]\!]^{S,\beta}(\delta)\ =_{\mathrm{def}} \\ \quad (\cdots((\delta[w_1 \mapsto val_{S,\beta,\delta}(t_1)])[w_2 \mapsto val_{S,\beta,\delta}(t_2)]) \cdots)[w_k \mapsto val_{S,\beta,\delta}(t_k)] \end{array}$$

where $(\delta[w \mapsto a])(v) = \mathit{if}\ (v = w)\ \mathit{then}\ a\ \mathit{else}\ \delta(v)$ are ordinary function updates.

Evaluation $val_{S,\beta,\delta}$ of terms and formulae is mostly defined as it is common for first-order predicate logic. Formulas are mapped into a Boolean domain, where tt stands for semantic truth. The cases for programs and updates are

$$\begin{array}{rcl} val_{S,\beta,\delta}([\alpha]\, \phi) & =_{\mathrm{def}} & \begin{cases} val_{S,\beta,[\![\alpha]\!]^S(\delta)}(\phi) & \text{for } [\![\alpha]\!]^S(\delta) \neq \perp \\ \mathrm{tt} & \text{otherwise} \end{cases} \\ val_{S,\beta,\delta}(\{\, U \,\}\ \phi) & =_{\mathrm{def}} & val_{S,\beta,[\![U]\!]^{S,\beta}(\delta)}(\phi) \end{array}$$

We interpret free logical variables $x \in \mathrm{LVar}$ existentially: a formula ϕ is *valid* iff for each structure $S = (D, I)$ and each program variable assignment $\delta \in PA^D$ there is a variable assignment $\beta : \mathrm{LVar} \to D$ such that $val_{S,\beta,\delta}(\phi) = \mathrm{tt}$. Likewise, a sequent $\Gamma \vdash^{\mathrm{dl}} \Delta$ is called valid iff $\bigwedge \Gamma \to \bigvee \Delta$ is valid.

The set of unbound variables occurring in a term or a formula t is denoted by $\mathrm{vars}(t) \subseteq \mathrm{PVar} \cup \mathrm{LVar}$. For program variables $v \in \mathrm{PVar}$, this means $v \in \mathrm{vars}(t)$ iff v turns up anywhere in t. For logical variables $x \in \mathrm{LVar}$, we define $x \in \mathrm{vars}(t)$ iff x occurs in t and is not in the scope of $\forall x$ or $\exists x$.

We note that the semantic notion of non-interference can easily be expressed in the formalism of dynamic logic: One possibility [24] is to express the variable independence property introduced above as follows.

Assuming the set of program variables is $\mathrm{PVar} = \{v_1, \dots, v_n\}$, then v_j only depends on $v_1, \dots, v_i$ if variation of $v_{i+1}, \dots, v_n$ does not affect the final value of v_j:

$$\forall u_1, \dots, u_i.\ \exists r.\ \forall u_{i+1}, \dots, u_n.\ \{\, v_i := u_i \,\}_{1 \le i \le n}\, [\alpha]\, (v_j = r)\ . \tag{2.1}$$

The particular use of updates in this formula is a standard trick to quantify over program variables which is not allowed directly: in order to quantify over all values that a program variable v occurring in a formula ϕ can assume, we introduce a fresh logical variable u and quantify over the latter. In the following we use quantification over program variables as a shorthand, writing $\dot\forall v.\ \phi$ for $\forall u.\ \{\, v := u \,\}\ \phi$. One achieved result, which we are able to prove for our translation, is that simple, easily automated proofs of formulae such as (2.1) are viable in at least those cases where a corresponding derivation in the type system of Hunt and Sands exists.

2.2 Interpreting the type system in dynamic logic

We now present a calculus for dynamic logic in which the rules involving program statements employ abstraction instead of precise evaluation. The calculus facilitates automatic proofs of secure information flow. In particular, when proving loops the burden of finding invariants is reduced to the task of providing a dependency approximation between program variables. There is a close correspondence to the type system of [24] (Fig. 2.1). Intuitively, state updates in the DL calculus resemble security typings in the type system: updates arising during a proof will essentially take the form $\{\, v := f(\dots \mathit{vars} \dots) \,\}$, where the *vars* form the type of v in a corresponding derivation in the type system. To put our observation on a formal basis, we prove the soundness of the calculus and show that every derivation in the type system has a corresponding proof in our calculus.

2.2.1 Abstraction-based calculi

We introduce *extended type environments* as pairs (∇, I) consisting of a typing function $\nabla : \mathrm{PVar} \to \mathcal{P}(\mathrm{PVar})$ and an *invariance set* $I \subseteq \mathrm{PVar}$ used to indicate those variables whose value does not change after execution of the program. We write ∇_v for the syntactic sequence of variables $w_1, \dots, w_k$ with arbitrary ordering when $\nabla(v) = \{w_1, \dots, w_k\}$ and ∇_v^C for a sequence of all variables *not* in $\nabla(v)$. Ultimately, we want to prove non-interference properties of the form

$$\{\, \alpha \,\} \Downarrow (\nabla, I) \quad \equiv_{\mathrm{def}} \quad \bigwedge_{v \in \mathrm{PVar}} \begin{cases} \dot\forall v_1 \cdots v_n.\ \forall u.\ \{\, v := u \,\}[\alpha]\, v = u & , v \in I \\ \dot\forall \nabla_v.\ \exists r.\ \dot\forall \nabla_v^C.\ [\alpha]\, v = r & , v \notin I \end{cases} \tag{2.2}$$

where we assume $\mathrm{PVar} = \{v_1, \dots, v_n\}$. Validity of a judgment $\{\ \alpha\ \} \Downarrow (\nabla, I)$ ensures that all variables in the invariance set I remain unchanged after execution of the program α, and that any variable v of the rest only depends on variables in $\nabla(v)$. The invariance set I corresponds to the context p that turns up in judgments $p \vdash^{\mathrm{HS}} \nabla\ \{\ \alpha\ \}\ \nabla'$: while the type system ensures that p is a lower bound of the post-type $\nabla'(v)$ of variables v assigned in α, the set I can be used to ensure that variables with low post-type are not assigned (or, more precisely, not changed). The equivalence is formally stated in Lem. 2.2.2.

In the proof process we want to abstract program statements "**while** b α" and "**if** b α_1 **else** α_2" into updates modelling the effects of these statements. Thus we avoid having to split up the proof for the two branches of an if-statement, or having to find an invariant for a while-loop. Extended type environments capture the essence of these updates: the arguments for the abstraction functions and the unmodified variables. They are translated into updates as follows:

$$\begin{aligned} \mathrm{upd}(\nabla, I) \quad &=_{\mathrm{def}} \quad \{\, v := f_v(\nabla_v) \,\}_{v \in \mathrm{PVar} \setminus I} \\ \mathrm{ifUpd}(b, \nabla, I) \quad &=_{\mathrm{def}} \quad \{\, v := f_v(b, \nabla_v) \,\}_{v \in \mathrm{PVar} \setminus I} \end{aligned}$$

$$\frac{\Gamma \vdash^{\mathrm{dl}} \phi, \Delta \qquad \Gamma \vdash^{\mathrm{dl}} \psi, \Delta}{\Gamma \vdash^{\mathrm{dl}} \phi \wedge \psi, \Delta}\ \mathsf{and-right}^{\mathrm{dl}}$$

$$\frac{\Gamma \vdash^{\mathrm{dl}} \phi[x/f(X_1,\ldots,X_n)], \Delta}{\Gamma \vdash^{\mathrm{dl}} \forall x.\, \phi, \Delta}\ \mathsf{all-right}^{\mathrm{dl}} \qquad \begin{array}{l}\{X_1,\ldots,X_n\} = \mathrm{vars}(\phi) \cap \mathrm{LVar}\backslash\{x\},\\ f \text{ fresh}\end{array}$$

$$\frac{\Gamma \vdash^{\mathrm{dl}} \phi[x/X], \exists x.\, \phi, \Delta}{\Gamma \vdash^{\mathrm{dl}} \exists x.\, \phi, \Delta}\ \mathsf{ex-right}^{\mathrm{dl}} \qquad X \text{ fresh}$$

$$\frac{\begin{array}{c} * \\ [\, s \equiv t \,] \end{array}}{\Gamma \vdash^{\mathrm{dl}} s = t, \Delta}\ \mathsf{close-eq}^{\mathrm{dl}} \qquad s, t \text{ unifiable (with rigid unifier)}$$

$$\frac{(\Gamma \vdash^{\mathrm{dl}} \Delta)[x/f(\mathrm{vars}(t))]}{(\Gamma \vdash^{\mathrm{dl}} \Delta)[x/t]}\ \mathsf{abstract}^{\mathrm{dl}} \qquad f \text{ fresh}$$

$$\frac{\Gamma \vdash^{\mathrm{dl}} \{\, U \,\}\, \phi, \Delta}{\Gamma \vdash^{\mathrm{dl}} \{\, U \,\}\, [\,]\, \phi, \Delta}\ \mathsf{Skip}^{\mathrm{dl}} \qquad \frac{\Gamma \vdash^{\mathrm{dl}} \{\, U \,\}\, \{\, v := E \,\}\, [\ldots]\, \phi, \Delta}{\Gamma \vdash^{\mathrm{dl}} \{\, U \,\}\, [v = E\ ;\ \ldots]\, \phi, \Delta}\ \mathsf{Assign}^{\mathrm{dl}}$$

$$\frac{\begin{array}{c} \vdash^{\mathrm{dl}} \{\, \alpha_i \,\} \Downarrow (\nabla, I) \quad (i = 1, 2) \\ \Gamma \vdash^{\mathrm{dl}} \{\, U \,\}\, \{\, \mathrm{ifUpd}(b, \nabla, I) \,\}\, [\ldots]\, \phi, \Delta \end{array}}{\Gamma \vdash^{\mathrm{dl}} \{\, U \,\}\, [\mathbf{if}\ b\ \alpha_1\ \mathbf{else}\ \alpha_2\ ;\ \ldots]\, \phi, \Delta}\ \mathsf{If}^{\mathrm{dl}}$$

$$\frac{\begin{array}{c} \vdash^{\mathrm{dl}} \{\ \mathbf{if}\ b\ \alpha\ \mathbf{else}\ \{\}\ \} \Downarrow (\gamma^*_\nabla(\nabla), I) \\ \Gamma \vdash^{\mathrm{dl}} \{\, U \,\}\, \{\, \mathrm{upd}(\nabla, I) \,\}\, [\ldots]\, \phi, \Delta \end{array}}{\Gamma \vdash^{\mathrm{dl}} \{\, U \,\}\, [\mathbf{while}\ b\ \alpha\ ;\ \ldots]\, \phi, \Delta}\ \mathsf{While}^{\mathrm{dl}} \qquad v \in \nabla(v) \text{ for all } v \in \mathrm{PVar}$$

Figure 2.2: A dynamic logic calculus for information flow security. In the last four rules the update $\{\, U \,\}$ can also be empty and disappear.

The above updates assign to each v not in the invariance set I a *fresh* function symbol f_v whose arguments are exactly the variables given by the type $\nabla(v)$. In a program "**if** b α_1 **else** α_2" the final state may depend on the branch condition b, so the translation ifUpd 'injects' the condition into the update. This is the analogon of the context lifting present in $\mathsf{If}^{\mathrm{HS}}$. For the while-rule, we transform the loop body into a conditional, so that we must handle the context lifting only in the if-rule.

Figs. 2.2 and 2.3 contain the rules for a sequent calculus. We have only included those propositional and first-order rules (the first four rules of Fig. 2.2) that are necessary for proving the results in this section; more rules are required to make the calculus usable in practice. The calculus uses free logical variables $X \in \mathrm{LVar}$ ($\mathsf{ex-right}^{\mathrm{dl}}$) and unification ($\mathsf{close-eq}^{\mathrm{dl}}$) for handling existential quantification, where the latter rule works by applying the unifier of terms s and t to the whole proof tree. We have to demand that only rigid terms (not containing program variables) are substituted for free variables, because free variables can also occur in the scope of updates or the box modal operator. Skolemisation ($\mathsf{all-right}^{\mathrm{dl}}$) has to collect the free variables that occur in a quantified formula to ensure soundness. By definition of the non-interference properties (2.2) and by the design of the rules of the dynamic logic calculus it is sufficient to define update rules for terms, quantifier-free formulae, and other updates. Such rules can be used at any point in a proof to simplify expressions containing updates.

Rule $\mathsf{Abstract}^{\mathrm{dl}}$ can be used to normalise terms occurring in updates to the form $f(\ldots \mathit{vars} \ldots)$. In rules $\mathsf{If}^{\mathrm{dl}}$ and $\mathsf{While}^{\mathrm{dl}}$ the second premise represents the actual abstraction of the program statement for a suitably chosen typing ∇ and invariance set I. This abstraction is justified through the first premise in terms of another non-interference proof obligation. The concretisation operator γ^* (cf. [24]) of rule $\mathsf{While}^{\mathrm{dl}}$

$$\begin{array}{rcll}
\{\, w_1 := t_1, \ldots, w_k := t_k \,\}\; w_i & \to^{\mathrm{dl}} & t_i & \text{if } w_j \neq w_i \text{ for } i < j \leq k \\
\{\, w_1 := t_1, \ldots, w_k := t_k \,\}\; t & \to^{\mathrm{dl}} & t & \text{if } w_1, \ldots, w_k \notin \mathrm{vars}(t) \\
\{\, U \,\}\; f(t_1, \ldots, t_n) & \to^{\mathrm{dl}} & f(\{\, U \,\}\; t_1, \ldots, \{\, U \,\}\; t_n) & \\
\{\, U \,\}\; (t_1 = t_2) & \to^{\mathrm{dl}} & \{\, U \,\}\; t_1 = \{\, U \,\}\; t_2 & \\
\{\, U \,\}\; \neg\phi & \to^{\mathrm{dl}} & \neg\{\, U \,\}\; \phi & \\
\{\, U \,\}\; (\phi_1 * \phi_2) & \to^{\mathrm{dl}} & \{\, U \,\}\; \phi_1 * \{\, U \,\}\; \phi_2 & \text{for } * \in \{\vee, \wedge\}
\end{array}$$

$$\{\, U \,\}\; \{\, w_1 := t_1, \ldots, w_k := t_k \,\}\; \phi \;\to^{\mathrm{dl}}\; \{\, U,\; w_1 := \{\, U \,\}\; t_1, \ldots, w_k := \{\, U \,\}\; t_k \,\}\; \phi$$

Figure 2.3: Application rules for updates in dynamic logic, as far as they are required for Lem. 2.2.6. Further application and simplification rules are necessary in general.

is generally defined as

$$\gamma^*_{\nabla_1}(\nabla_2)(x) \;=_{\mathrm{def}}\; \{y \in \mathrm{PVar} \mid \nabla_1(y) \subseteq \nabla_2(x)\} \qquad (x \in \mathrm{PVar})\ . \tag{2.3}$$

Together with the side condition that for all v we require $v \in \nabla(v)$, a closure property on dependencies is ensured: $w \in \gamma^*_{\nabla}(\nabla)(v)$ implies $\gamma^*_{\nabla}(\nabla)(w) \subseteq \gamma^*_{\nabla}(\nabla)(v)$: if a variable depends on another, the latter's dependencies are included in the former's. This accounts for the fact that the loop body can be executed more than once, which, in general, causes transitive dependencies.

Function Arguments Ensure Soundness. A recurring proof obligation in a non-interference proof is a statement of the form $\dot\forall \nabla_v.\ \exists r.\ \dot\forall \nabla^C_v.\ [\alpha]\, v = r$. To prove this statement without abstraction essentially is to find a function of the variables ∇_v that yields the value of v under α for every given pre-state: one must find the strongest post-condition w.r.t. v's value. Logically, one must create this function as a term for the existentially quantified variable r in which the ∇^C_v do not occur. In a unification-based calculus the occurs check will let all those proofs fail where an actual information flow takes places from ∇^C_v to v. The purpose of function arguments for f_v is exactly to retain this crucial property in the abstract version of the calculus. We must make sure that a function f_v – abstracting the effect of α on v – gets at least those variables as arguments that are parts of the term representing the final value of v after α.

Theorem 2.2.1 (Soundness) *The rules of the DL calculus given in Figs. 2.2 and 2.3 are sound: the root of a closed proof tree is a valid sequent.*

2.2.2 Simulating type derivations in dynamic logic

In order to show subsumption of the type system in the logic, we first put the connection between invariance sets and context on solid ground. It suffices to approximate the invariance of variables v with the requirement that v must not occur as left-hand side of assignments ($Lhs(\alpha)$ is the set of all left-hand sides of assignments in α).

Lemma 2.2.2 *In the type system of [24], see Fig. 2.1, the following equivalence holds:*

$$p \vdash^{\mathrm{HS}} \nabla \,\{\, \alpha \,\}\, \nabla' \quad \textit{iff} \quad \bot \vdash^{\mathrm{HS}} \nabla \,\{\, \alpha \,\}\, \nabla' \ \ \textit{and}\ \ \textit{f.a. } v \in \textit{Lhs}(\alpha):\ p \sqsubseteq \nabla'(v)$$

Furthermore, we can normalise type derivations thanks to the Canonical Derivations Lemma of [24]. The crucial ingredient is the concretisation operator γ^* defined in (2.3).

Lemma 2.2.3 (Canonical Derivations)

$$\bot \vdash^{\mathrm{HS}} \nabla \,\{\, \alpha \,\}\, \nabla' \quad \textit{iff} \quad \bot \vdash^{\mathrm{HS}} \Delta_0 \,\{\, \alpha \,\}\, \gamma^*_{\nabla}(\nabla') \qquad \textit{where } \Delta_0 = \lambda x.\, \{x\}$$

For brevity, we must refer to Hunt and Sands' paper for details, but in the setting at hand one can intuitively take Lemma 2.2.3 as stating that any typing judgment can also be understood as a dependency judgment: the typing on the left-hand side is equivalent to the statement that the final value of x *may depend on* the initial value of y only if y appears in the post-type, or dependence set, $\gamma^*_\nabla(\nabla')(x)$.

The type system of Fig. 2.4 only mentions judgments with a pre-type Δ_0 as depicted on the right-hand side of the equivalence in Lemma 2.2.3. Further, the context p has been replaced by equivalent side conditions (Lemma 2.2.2), and rule $\mathsf{Seq}^{\mathrm{HS}}$ is built into the other rules, i.e., the rules always work on the initial statement of a program. Likewise, rule $\mathsf{Sub}^{\mathrm{HS}}$ has been integrated in $\mathsf{Skip}^{\mathrm{cf}}$ and $\mathsf{While}^{\mathrm{cf}}$. The type system is equivalent to Hunt and Sands' system (Fig. 2.1):

Lemma 2.2.4

$$\bot \vdash^{\mathrm{HS}} \Delta_0 \; \{ \; \alpha \; \} \; \nabla \qquad \textit{if and only if} \qquad \vdash^{\mathrm{cf}} \Delta_0 \; \{ \; \alpha \; \} \; \nabla$$

The proof proceeds in multiple steps by devising intermediate type systems, each of which adds a modification towards the system in Fig. 2.4 and which is equivalent to Hunt and Sands' system.

Obviously, due to the approximating character of $\mathsf{If}^{\mathrm{dl}}$ and $\mathsf{While}^{\mathrm{dl}}$ (and the lack of arithmetic), our DL calculus is not (relatively) complete in the sense of [23]. For the particular judgements $\{ \; \alpha \; \} \Downarrow (\nabla, I)$ the calculus is, however, not more incomplete than the type system of Fig. 2.1: every typable program can also be proven secure using the DL calculus.[3]

Theorem 2.2.5

$$\bot \vdash^{\mathrm{HS}} \Delta_0 \; \{ \; \alpha \; \} \; \nabla \qquad \textit{implies} \qquad \vdash^{\mathrm{dl}} \; \{ \; \alpha \; \} \Downarrow (\nabla, \emptyset)$$

The proof of the theorem is constructive: A method for translating type derivations into DL proofs is given. The existence of this translation mapping shows that proving in the DL calculus is in principle not more difficult than typing programs using the system of Fig. 2.1.

The first part of the translation is accomplished by Lem. 2.2.4, which covers structural differences between type derivations and DL proofs. Applications of the rules of Fig. 2.4 can then almost directly be replaced with the corresponding rules of the DL calculus, which is ensured by the following lemma:

Lemma 2.2.6

$$\vdash^{\mathrm{cf}} \Delta_0 \; \{ \; \alpha \; \} \; \nabla \qquad \textit{implies} \qquad \vdash^{\mathrm{dl}} \; \{ \; \alpha \; \} \Downarrow (\nabla, \emptyset)$$

Obviously the combination of Lem. 2.2.4 and Lem. 2.2.6 allows us to immediately prove the main theorem 2.2.5. We concentrate therefore on a proof sketch for Lem. 2.2.6 where most of the work is done and in particular the constructive flavour of the proof can be demonstrated.

For showing that derivations in cf can be translated to proofs in the DL calculus, we first need a bit of further notation. For an update U and a term s, we write $U\,[s]$ for the (unique) irreducible term s' that is obtained by repeatedly applying rules of Fig. 2.3:

$$\{ \; U \; \} \; s \xrightarrow{*}{}^{\mathrm{dl}} s'$$

Note that terms $U\,[s]$ do not contain updates.

Further, for an update U, a type $t \subseteq \mathrm{PVar}$ and a logical variable $R \in \mathrm{LVar}$, we write $rel(t, R, U)$ if the following identity holds:

$$\{v \in \mathrm{PVar} \mid R \in \mathrm{vars}(U\,[v])\} = \mathrm{PVar}\backslash t \tag{2.4}$$

Intuitively, this means that all variables $w \in \mathrm{PVar}\backslash t$ whose interference is prohibited are "poisoned" by U with a free variable R. Removing the quantifiers in a non-interference statement like

$$\forall u_1\, u_2 \; \exists r. \; \forall u_3\, u_4. \; \{ \; v_i := u_i \; \}_{1 \le i \le 4} \; [p] \, (v_1 = r)$$

[3]The converse of Theorem 2.2.5 does not hold. In the basic version of the calculus of Fig. 2.2, untypable programs like "**if** (h) $\{l = 1\}$ **else** $\{l = 0\}$" can be proven secure. Sect. 2.2.3 discusses how the precision of the DL calculus can be further augmented.

using rules $\mathsf{all}-\mathsf{right}^{\mathrm{dl}}$ and $\mathsf{ex}-\mathsf{right}^{\mathrm{dl}}$ creates this situation (in the example for $t = \nabla(v_1) = \{v_1, v_2\}$).

We denote the update obtained by *sequentially composing* two updates U_1 and $U_2 = v_1 := t_1, \ldots, v_k := t_k$ by

$$U_1; U_2 \quad := \quad U_1,\ v_1 := \{\, U_1 \,\}\, t_1, \ldots, v_k := \{\, U_1 \,\}\, t_k$$

(note the similarity to the last rule of Fig. 2.3).

In order to prove the completeness of our translation relative to the type system of Hunt and Sands, we need to further technical lemmas that are given here without proofs.

Lemma 2.2.7 *Suppose that for an update U' and a typing $\nabla' : \mathrm{PVar} \to \mathcal{P}(\mathrm{PVar})$ the following property holds:*

$$\textit{f.a. } v \in \mathrm{PVar}. \qquad \nabla'(v) = \mathrm{vars}(U'\,[v]) \cap \mathrm{PVar} \quad \textit{and} \quad R \notin \mathrm{vars}(U'\,[v])$$

Then

$$\mathit{rel}(t, R, U) \qquad \textit{implies} \qquad \mathit{rel}(\gamma_{\nabla'}(t), R, (U; U'))$$

One further technical lemma that required for proving the main theorem is:

Lemma 2.2.8

$$p \vdash^{\mathrm{HS}} \nabla \ \{\ \alpha\ \}\ \nabla' \quad \textit{and} \quad v \notin \mathit{Lhs}(\alpha) \quad \textit{implies} \quad \nabla(v) \sqsubseteq \nabla'(v)$$

Proof of Lem. 2.2.6:

We show the stronger implication

$$\vdash^{\mathrm{cf}} \Delta_0\ \{\ \alpha\ \}\ \nabla \quad \Longrightarrow \quad I \cap \mathit{Lhs}(\alpha) = \emptyset \quad \Longrightarrow \quad \vdash^{\mathrm{dl}}\ \{\ \alpha\ \} \Downarrow (\nabla, I)$$

by induction on the program α. It appears easiest to use noetherian induction and the sub-program-order: For showing the implication for a program α, we will assume that it holds for all programs $\alpha' \neq \alpha$ that literally occur as part of α (in particular for the empty program).

In the whole proof, given a type environment $\nabla : \mathrm{PVar} \to \mathcal{P}(\mathrm{PVar})$ we write $\nabla_{\downarrow A}$ for the environment defined by

$$\nabla_{\downarrow A}(v) \ := \ \begin{cases} \{v\} & \text{for } v \in A \\ \nabla(v) & \text{otherwise} \end{cases}$$

We then first decompose α into a list $\alpha = \alpha_1 \ ;\ \ldots\ ;\ \alpha_m$ of statements ($m = 0$ is possible) and assume $\vdash^{\mathrm{cf}} \Delta_0\ \{\ \alpha\ \}\ \nabla$ and $I \cap \mathit{Lhs}(\alpha) = \emptyset$. $\{\ \alpha\ \} \Downarrow (\nabla, I)$ consists of two kinds of proof obligations:

Non-interference obligations: We pick one of the obligations,

$$PO = \dot{\forall} \nabla_v .\ \exists r.\ \dot{\forall} \nabla_v^C .\ [\alpha]\, r = v$$

(for $v \notin I$), and by induction on a $k \in \mathbb{N}$, $k \leq m$ we show that the following properties hold:

- There is a dl proof tree with PO as root that has exactly one open branch:

$$\vdash^{\mathrm{dl}}\ \{\, U \,\}\ [\alpha_{k+1}\ ;\ \ldots\ ;\ \alpha_m]\, R = v$$

  where U is an update

- There is a type derivation that corresponds to the open goal:

$$\vdash^{\mathrm{cf}} \Delta_0\ \{\ \alpha_{k+1}\ ;\ \ldots\ ;\ \alpha_m\ \}\ \nabla'$$

  for some typing ∇' with $\mathit{rel}(\nabla'(v), R, U)$.

The induction is conducted as follows:
Induction base case ($k = 0$): (Just eliminate the quantifiers of PO)
Induction step (the properties hold for a $0 \leq k < m$): There are different cases depending on the next statement α_{k+1}:

- α_{k+1} is an assignment $w = E$: There is a derivation ending with

$$\frac{\Delta_0 \vdash E : t \qquad \vdash^{\mathrm{cf}} \Delta_0 \; \{ \; \alpha_{k+2} \; ; \; \ldots \; ; \; \alpha_m \; \} \; \gamma^*_{\Delta_0[w \mapsto t]}(\nabla')}{\vdash^{\mathrm{cf}} \Delta_0 \; \{ \; w = E \; ; \; \alpha_{k+2} \; ; \; \ldots \; ; \; \alpha_m \; \} \; \nabla'} \; \mathsf{Assign}^{\mathrm{cf}}$$

  In the dl proof, we apply rule $\mathsf{Assign}^{\mathrm{dl}}$ to the open branch:

$$\frac{\vdash^{\mathrm{dl}} \; \{U; \; w := E\} \; [\alpha_{k+2} \; ; \; \ldots \; ; \; \alpha_m] \, R = v}{\vdash^{\mathrm{dl}} \; \{U\} \; [w = E \; ; \; \alpha_{k+2} \; ; \; \ldots \; ; \; \alpha_m] \, R = v} \; \mathsf{Assign}^{\mathrm{dl}}, \rightarrow^{\mathrm{dl}}$$

  and by Lem. 2.2.7 we have

$$\mathit{rel}(\gamma^*_{\Delta_0[w \mapsto t]}(\nabla')(v), R, (U; \; w := E))$$

- α_{k+1} is a conditional statement **if** b β_1 **else** β_2: Let $A := \mathit{Lhs}(\beta_1) \cup \mathit{Lhs}(\beta_2)$. There is a type derivation ending with

$$\frac{\begin{array}{c}\Delta_0 \vdash b : t \qquad \vdash^{\mathrm{cf}} \Delta_0 \; \{ \; \alpha_{k+2} \; ; \; \ldots \; ; \; \alpha_m \; \} \; \gamma^*_{\nabla''}(\nabla') \\ \vdash^{\mathrm{cf}} \Delta_0 \; \{ \; \beta_i \; \} \; \nabla'' \quad (i = 1, 2)\end{array}}{\vdash^{\mathrm{cf}} \Delta_0 \; \{ \; \mathbf{if} \; b \; \beta_1 \; \mathbf{else} \; \beta_2 \; ; \; \alpha_{k+2} \; ; \; \ldots \; ; \; \alpha_m \; \} \; \nabla'} \; \mathsf{If}^{\mathrm{cf}}$$

  and the condition f.a. $v \in A$. $t \sqsubseteq \nabla''(v)$ holds. In order to continue the dl proof we apply $\mathsf{If}^{\mathrm{dl}}$ for the extended type environment $(\nabla''_{\downarrow A^C}, A^C)$:

$$\frac{\begin{array}{c}\vdash^{\mathrm{dl}} \; \{ \; \beta_i \; \} \Downarrow (\nabla''_{\downarrow A^C}, A^C) \quad (i = 1, 2) \\ \vdash^{\mathrm{dl}} \; \{U; \; \mathrm{ifUpd}(b, \nabla''_{\downarrow A^C}, A^C)\} \; [\alpha_{k+2} \; ; \; \ldots \; ; \; \alpha_m] \, R = v\end{array}}{\vdash^{\mathrm{dl}} \; \{U\} \; [\mathbf{if} \; b \; \beta_1 \; \mathbf{else} \; \beta_2 \; ; \; \alpha_{k+2} \; ; \; \ldots \; ; \; \alpha_m] \, R = v} \; \mathsf{If}^{\mathrm{dl}}, \rightarrow^{\mathrm{dl}}$$

  For proving the first two premisses, the type judgements

$$\vdash^{\mathrm{cf}} \Delta_0 \; \{ \; \beta_i \; \} \; \nabla'' \quad (i = 1, 2) \tag{2.5}$$

  and the induction hypothesis entail that there are dl proofs of

$$\vdash^{\mathrm{dl}} \; \{ \; \beta_1 \; \} \Downarrow (\nabla'', A^C), \qquad \vdash^{\mathrm{dl}} \; \{ \; \beta_2 \; \} \Downarrow (\nabla'', A^C) \tag{2.6}$$

  Because of the definition of non-interference proof obligations, these proofs are also proofs of the first two premisses

$$\vdash^{\mathrm{dl}} \; \{ \; \beta_1 \; \} \Downarrow (\nabla''_{\downarrow A^C}, A^C), \qquad \vdash^{\mathrm{dl}} \; \{ \; \beta_2 \; \} \Downarrow (\nabla''_{\downarrow A^C}, A^C)$$

  Finally, from Lem. 2.2.8 and (2.5) we obtain $\nabla''_{\downarrow A^C} \sqsubseteq \nabla''$, i.e., $\gamma^*_{\nabla''}(\nabla') \sqsubseteq \gamma^*_{\nabla''_{\downarrow A^C}}(\nabla')$, and thus the typing

$$\vdash^{\mathrm{cf}} \Delta_0 \; \{ \; \alpha_{k+2} \; ; \; \ldots \; ; \; \alpha_m \; \} \; \gamma^*_{\nabla''_{\downarrow A^C}}(\nabla')$$

  which is related to the open goal of the dl proof: By Lem. 2.2.7 and the condition

$$\text{f.a. } v \in A. \; \mathrm{vars}(b) = t \sqsubseteq \nabla''(v)$$

  we have

$$\mathit{rel}(\gamma^*_{\nabla''_{\downarrow A^C}}(\nabla')(v), R, (U; \; \mathrm{ifUpd}(b, \nabla''_{\downarrow A^C}, A^C)))$$

- α_{k+1} is a loop **while** $b\ \beta$: Let $A := Lhs(\beta)$. There is a type derivation ending with

$$\frac{\Delta_0 \vdash b : t \qquad \vdash^{\mathrm{cf}} \Delta_0 \{\ \alpha_{k+2}\ ;\ \dots\ ;\ \alpha_m\ \}\ \gamma^*_{\nabla''}(\nabla') \qquad \vdash^{\mathrm{cf}} \Delta_0 \{\ \beta\ \}\ \gamma^*_{\nabla''}(\nabla'')}{\vdash^{\mathrm{cf}} \Delta_0 \{\ \textbf{while}\ b\ \beta\ ;\ \alpha_{k+2}\ ;\ \dots\ ;\ \alpha_m\ \}\ \nabla'}\ \mathsf{While}^{\mathrm{cf}}$$

and the conditions $\Delta_0 \sqsubseteq \nabla''$ and f.a. $v \in A.\ t \sqsubseteq \gamma^*_{\nabla''}(\nabla'')(v)$ hold. In order to continue the dl proof, we apply rule $\mathsf{While}^{\mathrm{dl}}$ using the extended type environment (∇'', A^C):

$$\frac{\vdash^{\mathrm{dl}}\ \{\ \textbf{if}\ b\ \beta\ \textbf{else}\ \{\}\ \} \Downarrow (\gamma^*_{\nabla''}(\nabla''), A^C) \qquad \vdash^{\mathrm{dl}}\ \{U;\ \mathrm{upd}(\nabla'', A^C)\ \}\ [\alpha_{k+2}\ ;\ \dots\ ;\ \alpha_m]\ R = v}{\vdash^{\mathrm{dl}}\ \{U\}\ [\textbf{while}\ b\ \beta\ ;\ \alpha_{k+2}\ ;\ \dots\ ;\ \alpha_m]\ R = v}\ \mathsf{While}^{\mathrm{dl}}, \rightarrow^{\mathrm{dl}}$$

The first premiss again leads to two different kinds of proof obligations, non-interference obligations and invariance obligations, one for each existing program variable.

  – Non-interference obligations: We pick one of the obligations (for a $w \in A$), eliminate the quantifiers and apply $\mathsf{If}^{\mathrm{dl}}$ using the type environment $(\gamma^*_{\nabla''}(\nabla''), A^C)$:

$$\frac{\vdash^{\mathrm{dl}}\ \{\ \beta\ \} \Downarrow (\gamma^*_{\nabla''}(\nabla''), A^C) \qquad \vdash^{\mathrm{dl}}\ \{\ \ \} \Downarrow (\gamma^*_{\nabla''}(\nabla''), A^C) \qquad \vdash^{\mathrm{dl}}\ \{U';\ \mathrm{ifUpd}(b, \gamma^*_{\nabla''}(\nabla''), A^C)\ \}\ [\,]\ R' = w}{\vdash^{\mathrm{dl}}\ \{U'\}\ [\textbf{if}\ b\ \beta\ \textbf{else}\ \{\}]\ R' = w}\ \mathsf{If}^{\mathrm{dl}}, \rightarrow^{\mathrm{dl}}$$

$$\vdots$$

$$\vdash^{\mathrm{dl}}\ \dot\forall \nabla_w.\ \exists r.\ \dot\forall \nabla^C_w.\ [\textbf{if}\ b\ \beta\ \textbf{else}\ \{\}]\ r = w$$

  Because of (the second judgement follows because of $\Delta_0 \sqsubseteq \gamma^*_{\nabla''}(\nabla'')$)

$$\vdash^{\mathrm{cf}} \Delta_0\ \{\ \beta\ \}\ \gamma^*_{\nabla''}(\nabla''), \qquad \vdash^{\mathrm{cf}} \Delta_0\ \{\ \ \}\ \gamma^*_{\nabla''}(\nabla'')$$

  and the induction hypothesis there are dl proofs of the first two premises.
  For the last premise, because of f.a. $v \in A$. $\mathrm{vars}(b) = t \sqsubseteq \gamma^*_{\nabla''}(\nabla'')(v)$ and by Lem. 2.2.7 we have (for $\nabla''' := \gamma^*_{\nabla''}(\nabla'')_{\downarrow A^C}$)

$$rel(\gamma^*_{\nabla'''}(\gamma^*_{\nabla''}(\nabla''))(w), R', (U';\ \mathrm{ifUpd}(b, \gamma^*_{\nabla''}(\nabla''), A^C)))$$

  Because of $w \in \gamma^*_{\nabla'''}(\gamma^*_{\nabla''}(\nabla''))(w)$, that is

$$R' \notin \mathrm{vars}((U';\ \mathrm{ifUpd}(b, \gamma^*_{\nabla''}(\nabla''), A^C))\ [w])$$

  by (2.4), the premise can be proven by

$$\cfrac{\cfrac{\cfrac{*}{[\,R' \equiv (U';\ \mathrm{ifUpd}(b, \gamma^*_{\nabla''}(\nabla''), A^C))\ [w]\,]}}{\vdash^{\mathrm{dl}}\ R' = (U';\ \mathrm{ifUpd}(b, \gamma^*_{\nabla''}(\nabla''), A^C))\ [w]}\ \mathsf{close-eq}^{\mathrm{dl}}}{\vdash^{\mathrm{dl}}\ \{U'\}\ \{\ \mathrm{ifUpd}(b, \gamma^*_{\nabla''}(\nabla''), A^C)\ \}\ [\,]\ R' = w}\ \mathsf{Skip}^{\mathrm{dl}}, \xrightarrow{*}^{\mathrm{dl}}$$

  – Invariance obligations: Again we pick one of the obligations (for $w \notin A$), eliminate the quantifiers and apply $\mathsf{If}^{\mathrm{dl}}$ using the type environment $(\gamma^*_{\nabla''}(\nabla''), A^C)$:

$$\frac{\vdash^{\mathrm{dl}}\ \{\ \beta\ \} \Downarrow (\gamma^*_{\nabla''}(\nabla''), A^C) \qquad \vdash^{\mathrm{dl}}\ \{\ \ \} \Downarrow (\gamma^*_{\nabla''}(\nabla''), A^C) \qquad \vdash^{\mathrm{dl}}\ \{U';\ \mathrm{ifUpd}(b, \gamma^*_{\nabla''}(\nabla''), A^C)\ \}\ [\,]\ u_c = w}{\vdash^{\mathrm{dl}}\ \{U'\}\ [\textbf{if}\ b\ \beta\ \textbf{else}\ \{\}]\ u_c = w}\ \mathsf{If}^{\mathrm{dl}}, \rightarrow^{\mathrm{dl}}$$

$$\vdots$$

$$\vdash^{\mathrm{dl}}\ \dot\forall v_1 \cdots v_n.\ \forall u.\ \{\ w := u\ \}[\textbf{if}\ b\ \beta\ \textbf{else}\ \{\}]\ u = w$$

The first two premises can be handled as in the first case. For the last premise, because of $w \notin A$ we obtain

$$(U'; \mathrm{ifUpd}(b, \gamma^*_{\nabla''}(\nabla''), A^C))\,[w] = u_c$$

and the branch can be proven by

$$\dfrac{\dfrac{\dfrac{*}{\vdash^{\mathrm{dl}}\ \ u_c = u_c}\ \mathsf{close-eq}^{\mathrm{dl}}}{\vdash^{\mathrm{dl}}\ \ u_c = (U'; \mathrm{ifUpd}(b, \gamma^*_{\nabla''}(\nabla''), A^C))\,[w]}}{\vdash^{\mathrm{dl}}\ \ \{\,U';\ \mathrm{ifUpd}(b, \gamma^*_{\nabla''}(\nabla''), A^C)\,\}\,[\,]\,u_c = w}\ \mathsf{Skip}^{\mathrm{dl}},\ \xrightarrow{*}^{\mathrm{dl}}$$

As in the if-case, we finally have $\Delta_0 \sqsubseteq \nabla''$, that is $\nabla''_{\downarrow A^C} \sqsubseteq \nabla''$, that is $\gamma^*_{\nabla''}(\nabla') \sqsubseteq \gamma^*_{\nabla''_{\downarrow A^C}}(\nabla')$, and we obtain

$$\vdash^{\mathrm{cf}} \Delta_0 \ \{\ \alpha_{k+2}\ ;\ \ldots\ ;\ \alpha_m\ \}\ \gamma^*_{\nabla''_{\downarrow A^C}}(\nabla')$$

This type judgement is related with the open branch of the dl proof because of Lem. 2.2.7:

$$rel(\gamma^*_{\nabla''_{\downarrow A^C}}(\nabla')(v), R, (U;\ \mathrm{upd}(\nabla'', A^C)))$$

Harvesting: Having finished the induction on k, we know that

- There is a dl proof tree with PO as root that has exactly one open branch:

  $$\vdash^{\mathrm{dl}}\ \ \{\,U\,\}\,[\,]\,R = v$$

  where U is an update
- There is a type derivation that corresponds to the open goal:

  $$\vdash^{\mathrm{cf}} \Delta_0\ \{\ \ \}\ \nabla'$$

  for some typing ∇' with $rel(\nabla'(v), R, U)$.

The second item entails $\Delta_0 \sqsubseteq \nabla'$ (because the only applicable rule is $\mathsf{Skip}^{\mathrm{cf}}$), that means $v \in \nabla'(v)$, and we can finish the dl proof with

$$\dfrac{\dfrac{\overset{*}{[\,R \equiv U\,[v]\,]}}{\vdash^{\mathrm{dl}}\ \ R = U\,[v]}\ \mathsf{close-eq}^{\mathrm{dl}}}{\vdash^{\mathrm{dl}}\ \ \{\,U\,\}\,[\,]\,R = v}\ \mathsf{Skip}^{\mathrm{dl}},\ \xrightarrow{*}^{\mathrm{dl}}$$

Invariance obligations: As for non-interference obligations, we construct a dl proof by induction. There are in fact very simple proofs of the invariance obligations, because the type environments ∇ that are chosen when applying $\mathsf{If}^{\mathrm{dl}}$ and $\mathsf{While}^{\mathrm{dl}}$ are irrelevant. Nevertheless, it is meaningful to select certain type environments, because this demonstrates that the same choices for ∇ can be made as for the non-interference obligations, and actually *the same proof* can be used for all proof obligations.

We pick one of the obligations,

$$PO = \dot{\forall} v_1 \cdots v_n.\ \forall u.\ \{\,v := u\,\}[\alpha]\,u = v$$

(for $v \in I$), and by induction on a $k \in \mathbb{N}$, $k \leq m$ we show that the following properties hold:

- There is a dl proof tree with PO as root that has exactly one open branch:

  $$\vdash^{\mathrm{dl}}\ \ \{\,U\,\}\ [\alpha_{k+1}\ ;\ \ldots\ ;\ \alpha_m]\,u_c = v$$

  where U is an update with $U\,[v] = u_c$

- There is a type derivation that corresponds to the open goal:

$$\vdash^{\mathrm{cf}} \Delta_0 \ \{\ \alpha_{k+1} \ ;\ \ldots\ ;\ \alpha_m \ \}\ \nabla'$$

The induction is conducted as follows:
Induction base case ($k = 0$): (Just eliminate the quantifiers of PO)
Induction step (the properties hold for a $0 \le k < m$): There are different cases depending on the next statement α_{k+1}:

- α_{k+1} is an assignment $w = E$: There is a derivation ending with

$$\frac{\Delta_0 \vdash E : t \qquad \vdash^{\mathrm{cf}} \Delta_0 \ \{\ \alpha_{k+2} \ ;\ \ldots\ ;\ \alpha_m \ \}\ \gamma^*_{\Delta_0[w\mapsto t]}(\nabla')}{\vdash^{\mathrm{cf}} \Delta_0 \ \{\ w = E \ ;\ \alpha_{k+2} \ ;\ \ldots\ ;\ \alpha_m \ \}\ \nabla'}\ \mathsf{Assign}^{\mathrm{cf}}$$

  In the dl proof, we apply rule $\mathsf{Assign}^{\mathrm{dl}}$ to the open branch:

$$\frac{\vdash^{\mathrm{dl}}\ \{U;\ w := E\}\ [\alpha_{k+2} \ ;\ \ldots\ ;\ \alpha_m]\, u_c = v}{\vdash^{\mathrm{dl}}\ \{U\}\ [w = E \ ;\ \alpha_{k+2} \ ;\ \ldots\ ;\ \alpha_m]\, u_c = v}\ \mathsf{Assign}^{\mathrm{dl}}, \rightarrow^{\mathrm{dl}}$$

  Because of $v \notin Lhs(\alpha)$ we have $v \neq w$ and thus

$$(U; w := E)\,[v] = u_c$$

- α_{k+1} is a conditional statement **if** b β_1 **else** β_2: Let $A := Lhs(\beta_1) \cup Lhs(\beta_2)$. There is a type derivation ending with

$$\frac{\begin{array}{c}\Delta_0 \vdash b : t \qquad \vdash^{\mathrm{cf}} \Delta_0 \ \{\ \alpha_{k+2} \ ;\ \ldots\ ;\ \alpha_m \ \}\ \gamma^*_{\nabla''}(\nabla') \\ \vdash^{\mathrm{cf}} \Delta_0 \ \{\ \beta_i \ \}\ \nabla'' \quad (i = 1, 2)\end{array}}{\vdash^{\mathrm{cf}} \Delta_0 \ \{\ \mathbf{if}\ b\ \beta_1\ \mathbf{else}\ \beta_2 \ ;\ \alpha_{k+2} \ ;\ \ldots\ ;\ \alpha_m \ \}\ \nabla'}\ \mathsf{If}^{\mathrm{cf}}$$

  In order to continue the dl proof we apply $\mathsf{If}^{\mathrm{dl}}$:

$$\frac{\begin{array}{c}\vdash^{\mathrm{dl}}\ \{\ \beta_i\ \} \Downarrow (\nabla''_{\downarrow A^C}, A^C) \quad (i = 1, 2) \\ \vdash^{\mathrm{dl}}\ \{U;\ \mathrm{ifUpd}(b, \nabla''_{\downarrow A^C}, A^C)\}\ [\alpha_{k+2} \ ;\ \ldots\ ;\ \alpha_m]\, u_c = v\end{array}}{\vdash^{\mathrm{dl}}\ \{U\}\ [\mathbf{if}\ b\ \beta_1\ \mathbf{else}\ \beta_2 \ ;\ \alpha_{k+2} \ ;\ \ldots\ ;\ \alpha_m]\, u_c = v}\ \mathsf{If}^{\mathrm{dl}}, \rightarrow^{\mathrm{dl}}$$

  The first two premises can be proven as for non-interference obligations. Further, because of $v \in A^C$, we obtain

$$(U; \mathrm{ifUpd}(b, \nabla''_{\downarrow A^C}, A^C))\,[v] = u_c$$

- α_{k+1} is a loop **while** b β: Let $A := Lhs(\beta)$. There is a type derivation ending with

$$\frac{\begin{array}{c}\Delta_0 \vdash b : t \qquad \vdash^{\mathrm{cf}} \Delta_0 \ \{\ \alpha_{k+2} \ ;\ \ldots\ ;\ \alpha_m \ \}\ \gamma^*_{\nabla''}(\nabla') \\ \vdash^{\mathrm{cf}} \Delta_0 \ \{\ \beta\ \}\ \gamma^*_{\nabla''}(\nabla'')\end{array}}{\vdash^{\mathrm{cf}} \Delta_0 \ \{\ \mathbf{while}\ b\ \beta \ ;\ \alpha_{k+2} \ ;\ \ldots\ ;\ \alpha_m \ \}\ \nabla'}\ \mathsf{While}^{\mathrm{cf}}$$

  In order to continue the dl proof, we apply rule $\mathsf{While}^{\mathrm{dl}}$ using the extended type environment (∇'', A^C):

$$\frac{\begin{array}{c}\vdash^{\mathrm{dl}}\ \{\ \mathbf{if}\ b\ \beta\ \mathbf{else}\ \{\}\ \} \Downarrow (\gamma^*_{\nabla''}(\nabla''), A^C) \\ \vdash^{\mathrm{dl}}\ \{U;\ \mathrm{upd}(\nabla'', A^C)\}\ [\alpha_{k+2} \ ;\ \ldots\ ;\ \alpha_m]\, u_c = v\end{array}}{\vdash^{\mathrm{dl}}\ \{U\}\ [\mathbf{while}\ b\ \beta \ ;\ \alpha_{k+2} \ ;\ \ldots\ ;\ \alpha_m]\, u_c = v}\ \mathsf{While}^{\mathrm{dl}}, \rightarrow^{\mathrm{dl}}$$

  The first premise can again be handled as for non-interference obligations. Further, because of $v \in A^C$, we obtain

$$(U; \mathrm{upd}(\nabla'', A^C))\,[v] = u_c$$

Harvesting: Having finished the induction on k, we know that there is a dl proof tree with PO as root that has exactly one open branch:

$$\vdash^{\mathrm{dl}} \{U\}\,[\,]\,u_c = v$$

where U is an update with $U[v] = u_c$. Hence, the dl proof can directly be finished with

$$\dfrac{\dfrac{\dfrac{*}{\vdash^{\mathrm{dl}}\ u_c = u_c}\ \mathsf{close-eq}^{\mathrm{dl}}}{\vdash^{\mathrm{dl}}\ u_c = U[v]}}{\vdash^{\mathrm{dl}}\ \{U\}\,[\,]\,u_c = v}\ \mathsf{Skip}^{\mathrm{dl}},\ \xrightarrow{*}{}^{\mathrm{dl}}$$

2.2.3 Higher precision and delimited information release

The previous section shows that the suggested framework is at least so good and efficient as the type based approaches. In this section we describe how to improve the precision of the abstraction-based calculi in order to be able to treat a greater set of programs efficiently and to support more complex information flow policies.

Many realistic languages feature exceptions as a means to indicate failure. The occurrence of an exception can also lead to information leakage. Therefore, an information flow analysis for such a language must, at each point where an exception might possibly occur, either ensure that this will indeed not happen at runtime or verify that the induced information flow is benign. The Jif system [26] which implements a security type system for a large subset of the Java language employs a simple data flow analysis to retain a practically acceptable precision w.r.t. exceptions. The data flow analysis can verify the absence of null pointer exceptions and class cast exceptions in certain cases. However, to enhance the precision of this analysis to an acceptable level one is forced to apply a slightly cumbersome programming style.

The need for treatment of exceptions is an example showing that we actually gain something from the fact that our analysis is embedded in a more general program logic: there is no need to stack one analysis on top of the other to scale the approach up to larger languages, but we can coherently deal with added features, in this case exceptions, within one calculus. In the precise version of the calculus for Java Card as implemented in [1] and also in the MOBIUS base logic, exceptions are handled like conditional statements by branching on the condition under which an exception would occur. An uncaught exception is treated as non-termination. As an example, the division v_1/v_2 would have the condition that v_2 is zero ("$..\ ...$" denotes a context possibly containing exception handlers):

$$\frac{v_2 \neq 0\ \vdash^{\mathrm{dl}}\ \{w := v_1/v_2\}\,[..\ ...]\,\phi \qquad v_2 = 0\ \vdash^{\mathrm{dl}}\ [..\ \mathbf{throw}\ E\ ...]\,\phi}{\vdash^{\mathrm{dl}}\ [..\ w = v_1/v_2\ ...]\,\phi}\ .$$

If we knew $v_2 \neq 0$ at this point of the proof, implying that the division does in fact not raise an exception, the right branch could be closed immediately. Because our DL calculus stores the values of variables (instead of only the type) as long as no abstraction occurs, this information is often available: 1. rule $\mathsf{Assign}^{\mathrm{dl}}$ does not involve abstraction, which means that sequential programs can be executed without loss of information, and 2. invariance sets I in non-interference judgments allow to retain information about unchanged variables also across conditional statements and loops.

This can be seen for a program like "$v = 2\ ;\ \mathbf{while}\ b\ \alpha\ ;\ w = w/v$" in which α does not assign to v. By including v in the invariance set for "$\mathbf{while}\ b\ \alpha$" we can deduce $v = 2$ also after the loop, and thus be sure that the division will succeed. This is a typical example for a program containing an initialisation part that establishes invariants, and a use part that relies on the invariants. The pattern recurs in many flavours: examples are the initialisation and use of libraries and the well-definedness of references after object creation. We are optimistic to gather empirical evidence of our claim that the increased precision is useful in practice through future experiments.

$$\dfrac{}{\vdash^{\mathrm{cf}} \Delta_0 \{\ \} \nabla}\ \mathsf{Skip}^{\mathrm{cf}} \qquad v \in \nabla(v) \text{ for all } v \in \mathrm{PVar}$$

$$\dfrac{\Delta_0 \vdash E : t \qquad \vdash^{\mathrm{cf}} \Delta_0 \{ \dots \} \gamma^*_{\Delta_0[v\mapsto t]}(\nabla)}{\vdash^{\mathrm{cf}} \Delta_0 \{ v = E \ ; \ \dots \} \nabla}\ \mathsf{Assign}^{\mathrm{cf}}$$

$$\dfrac{\begin{array}{c}\Delta_0 \vdash b : t \qquad \vdash^{\mathrm{cf}} \Delta_0 \{ \dots \} \gamma^*_{\nabla}(\nabla') \\ \vdash^{\mathrm{cf}} \Delta_0 \{ \alpha_i \} \nabla \quad (i = 1, 2)\end{array}}{\vdash^{\mathrm{cf}} \Delta_0 \{ \textbf{if } b\ \alpha_1 \textbf{ else } \alpha_2 \ ; \ \dots \} \nabla'}\ \mathsf{If}^{\mathrm{cf}} \qquad \begin{array}{l}\text{f.a. } v \in \mathit{Lhs}(\alpha_1).\ t \sqsubseteq \nabla(v) \\ \text{f.a. } v \in \mathit{Lhs}(\alpha_2).\ t \sqsubseteq \nabla(v)\end{array}$$

$$\dfrac{\begin{array}{c}\Delta_0 \vdash b : t \qquad \vdash^{\mathrm{cf}} \Delta_0 \{ \dots \} \gamma^*_{\nabla}(\nabla') \\ \vdash^{\mathrm{cf}} \Delta_0 \{ \alpha \} \gamma^*_{\nabla}(\nabla)\end{array}}{\vdash^{\mathrm{cf}} \Delta_0 \{ \textbf{while } b\ \alpha \ ; \ \dots \} \nabla'}\ \mathsf{While}^{\mathrm{cf}} \qquad \begin{array}{l} v \in \nabla(v) \text{ for all } v \in \mathrm{PVar} \\ \text{f.a. } v \in \mathit{Lhs}(\alpha).\ t \sqsubseteq \gamma^*_{\nabla}(\nabla)(v)\end{array}$$

Figure 2.4: Intermediate flow-sensitive type system for information flow analysis

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{[\, f'_l(\mathit{TRUE}) \equiv R \,]}}{\mathit{odd}(f_h(R)) \ \vdash^{\mathrm{dl}}\ f'_l(\mathit{TRUE}) = R}\ \mathsf{close-eq}^{\mathrm{dl}}}{\mathit{odd}(f_h(R)) \ \vdash^{\mathrm{dl}}\ f'_l(\mathit{odd}(f_h(R))) = R}\ \mathsf{apply-eq}^{\mathrm{dl}}}{\mathit{odd}(f_h(R)) \ \vdash^{\mathrm{dl}}\ \{ l := f_l(R), h := f_h(R) \} \{ l := f'_l(\mathit{odd}(h)) \}\, l = R}\ \xrightarrow{*}{}^{\mathrm{dl}}}{\mathcal{D}}$$

$$\dfrac{\cdots \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{\vdash^{\mathrm{dl}}\ \{ l = 0 \} \Downarrow (\nabla, \{h\})} \quad \dfrac{*}{\vdash^{\mathrm{dl}}\ \{ l = 1 \} \Downarrow (\nabla, \{h\})} \qquad \mathcal{D}}{\mathit{odd}(f_h(R)) \ \vdash^{\mathrm{dl}}\ \{ l := f_l(R), h := f_h(R) \}\, [\alpha]\, l = R}\ \mathsf{If}^{\mathrm{dl}}}{\vdash^{\mathrm{dl}}\ \{ l := f_l(R), h := f_h(R) \}\, (\mathit{odd}(h) \to [\alpha]\, l = R)}\ \xrightarrow{*}{}^{\mathrm{dl}}, \mathsf{imp-right}^{\mathrm{dl}}}{\vdash^{\mathrm{dl}}\ \exists r.\, \dot\forall l.\, \dot\forall h.\ (\mathit{odd}(h) \to [\alpha]\, l = r)}\ \mathsf{ex-right}^{\mathrm{dl}}, \mathsf{all-right}^{\mathrm{dl}}}{\vdash^{\mathrm{dl}}\ \{ \alpha \} \Downarrow (\nabla, \{h\}, \mathit{odd}(h))}\ (\mathrm{Def}), \mathsf{and-right}^{\mathrm{dl}}$$

Figure 2.5: Non-interference proof with delimited information release: The precondition $\mathit{odd}(h)$ entails that only the parity of h is leaked into l. For sake of brevity, we use odd both as function and predicate, and only in one step ($\mathsf{apply-eq}^{\mathrm{dl}}$) make use of the fact that $\mathit{odd}(f_h(R))$ actually represents the equation $\mathit{odd}(f_h(R)) = \mathit{TRUE}$.

Increasing precision.

While our DL calculus is able to maintain state information *across* statements, the rules $\mathsf{If}^{\mathrm{dl}}$ and $\mathsf{While}^{\mathrm{dl}}$ lose this information in the first premisses, containing non-interference proofs for the statement *bodies.* This makes it impossible to deduce that no exceptions can occur in the program "$v = 2$; **while** b $\{w = w/v\}$". As another shortcoming, the branch predicate is not taken into account, so that absence of exceptions cannot be shown for a program like "**if** $(v \neq 0)$ $\{w = 1/v\}$".

One way to remedy these issues might be to relax the first premisses in $\mathsf{If}^{\mathrm{dl}}$ and $\mathsf{While}^{\mathrm{dl}}$. The idea is to generalise non-interference judgments and introduce *preconditions* ϕ under which the program must satisfy non-interference.

$$\{\ \alpha\ \} \Downarrow (\nabla, I, \phi) \quad \equiv_{\mathrm{def}} \quad \bigwedge_{v\in\mathrm{PVar}} \begin{cases} \dot\forall v_1 \cdots v_n.\ (\phi \to [\alpha]\, v = u) & , v \in I \\ \dot\forall \nabla_v.\ \exists r.\ \dot\forall \nabla_v^C.\ (\phi \to [\alpha]\, v = r) & , v \notin I \end{cases}$$

In an extended rule for if-statements, for instance, such a precondition can be used to 'carry through' side formulae and state information contained in the update U, as well as to integrate the branch predicates: we may assume arbitrary preconditions ϕ_1, ϕ_2 in the branches if we can show that they hold before the if-statement:

$$\frac{\begin{array}{cc} \vdash^{\mathrm{dl}}\ \{\ \alpha_1\ \} \Downarrow (\nabla, I, \phi_1) & \vdash^{\mathrm{dl}}\ \{\ \alpha_2\ \} \Downarrow (\nabla, I, \phi_2) \\ \Gamma, \{\,U\,\}\, b = \mathit{TRUE}\ \vdash^{\mathrm{dl}}\ \{\,U\,\}\,\phi_1, \Delta & \Gamma, \{\,U\,\}\, b \neq \mathit{TRUE}\ \vdash^{\mathrm{dl}}\ \{\,U\,\}\,\phi_2, \Delta \\ \multicolumn{2}{c}{\Gamma\ \vdash^{\mathrm{dl}}\ \{\,U\,\}\,\{\,\mathrm{ifUpd}(b, \nabla, I)\,\}\,[\ldots]\,\phi, \Delta} \end{array}}{\Gamma\ \vdash^{\mathrm{dl}}\ \{\,U\,\}\,[\mathbf{if}\ b\ \alpha_1\ \mathbf{else}\ \alpha_2\ ;\ \ldots]\,\phi, \Delta}$$

Probably more interestingly is that preconditions allow us to handle delimited information release in the style of [24], i.e. situations in which non-interference does not strictly hold and some well-defined information about secret values may be released.

Consider the following program piece α that tests the parity of a given variable and writes the result in a low level security variable. Further, assume that the declassified amount of information, i.e. the knowledge that a user is allowed to learn by observing variable l is exact the parity of h:

$$\mathbf{if}\ (\mathit{odd}(h))\ \{l = 0\}\ \mathbf{else}\ \{l = 1\}$$

The typing ∇ of the variables h and l assigns the variable l the value $\nabla(l) = \emptyset$ and variable h the type $\nabla(h) = \{h\}$, indicating that only declassified information flows into the low level security variable l. Parts of the non-interference proof for the above program with delimited information release is shown in Fig. 2.5. Performing a similar proof with the negated preconditions ensures that only the specified amount of declassified information can be learnt by an observer.

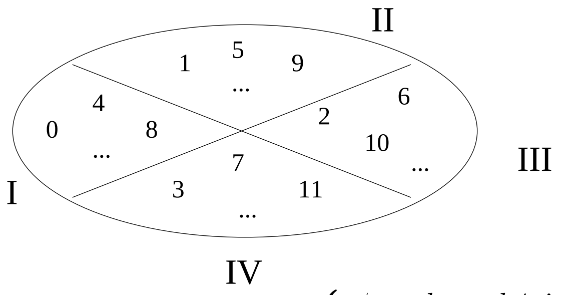
This approach allows us to treat all delimited information release properties, where the declassified information can be modelled as a partition on the values of high level variables. The observer is than allowed to deduce the partition in which the value of a high security level variable lies by observing the value of low variables, but not more. The precondition formulae $\phi_i,\ i \in \{1, \ldots, n\}$ must be then chosen as the characteristic functions of the partitions (see Fig. 2.6).

2.3 Ongoing and future work

2.3.1 Lifting to the MOBIUS program logic

As mentioned in the background section of this chapter, we used a dynamic logic as program logic as at the time most of the work has been done, the MOBIUS base logic has not yet been available. The transfer of the achieved results to the MOBIUS program logic is the currently ongoing work and will be done as a next step. In order to complete this transfer the following two steps have to be done

Program: $l = h\ \textbf{mod}\ 4;$



$$\text{Partitions (as preconditions):} \begin{cases} \phi_I \equiv h\ mod\ 4 \doteq 0 \\ \ldots \\ \phi_{IV} \equiv h\ mod\ 4 \doteq 3 \end{cases}$$

(a) Declassification of Partitions

$$\bigwedge_{i\in\{I,\ldots,IV\}} \big(\{\ l = h\ \textbf{mod}\ 4;\ \} \Downarrow (\nabla(l) := \emptyset, \nabla(h) := \{h, l\}, \{h\}, \phi_i)\big)$$

(b) Secure Information Flow Proof-Obligation with Delimited Information Release

Figure 2.6: Delimited information release: Partitioning of domains

1. the abstraction-based calculus and the non-interference formalisation have to be adopted to capture object-oriented features as heaps or method invocations.
2. the used formalism has to be translated into the Coq formalisation of the MOBIUS base logic.

We expect both issues to be completed within a relative short period of time. For the extension of the approach to a complete object-oriented programming language, we will make use of the ideas presented in [3, 9, 7] suggesting a type system that can treat heap structures and other object-oriented features. For the transfer of these ideas into a logic framework, the authors of this chapter can build up on their experience gained actively in the KeY-Project, which has successfully developed a deductive verification environment that can treat full Java Card, which is comparable to the Java fragment modelled in MOBIUS.

The translation of the approach into the MOBIUS formalism should be easily possible as we plan to use the Coq formalisation of the MOBIUS base logic. The MOBIUS base logic itself is mainly a Hoare style logics lacking the possibility to use nested quantifiers surrounding the complete Hoare triple. Using the Coq formalisation allows to make additional use of the Coq framework, in particular it allows the necessary nesting of quantifiers. With this on hand the translation is expected to be relative straight forward modulo some possibly occurring technical difficulties.

2.3.2 Improving support for declassification properties and further future work

The presented ideas for increasing the precision of the abstraction based calculus and the treatment of delimited information releases will be extended. A good classification of several declassification properties is given in [33], we plan to go through the classification systematically and to provide support for most of the listed properties. For those that will not be treated the reason will be given in detail.

Further future work that remains is a strict result for the growth of certificates, when using the suggested translation instead of the type based derivation. As the completeness proof of the translation is constructive, it gives already a good intuition that the translated proof is not significantly larger than the original proof. But still this has to be shown formal.

In order to evaluate the practical applicability of the suggested approach, we plan to perform some substantial case studies. These shall give conclusion about the performance of the translation and if it captures most of the security policies that are of practical importance.

Chapter 3

Non-interference protection in a program logic

3.1 Introduction

The Java bytecode language is the target low-level language for the Java programming language. The application of formal specification methods at the level of the bytecode has several advantages. (1) This allows to provide descriptions and verify properties of programs written in the bytecode itself. (2) It allows to do a unified formalised development for languages other than Java, but compiling to the Java bytecode. In particular, it allows to conduct a unified formal development in projects with several source code languages. (3) Proof for bytecode programs may make possible several optimisations in a JIT compilers. (4) As Java bytecode is the language which is actually executed, it is possible to couple with programs their proof carrying-code (PCC) certificates. (5) Since Java programs are distributed in their bytecode version, it is possible for a software distributor to develop its own certificate to ensure a particular property its clients are interested in. As we can see, the formal methods at the bytecode level need not to be conjoined with the development in the Java source code. However, we will focus here mainly on this approach to make the presentation clearer.

Java security model is based on several dynamic and static techniques such as the absence of the arithmetic on references, access control for stacks, arrays and typed objects etc. which ensure that the sensitive internal information is separated from the external access. Still, it is difficult in Java to ensure more sophisticated security policies which ensure that security sensitive data flows only through secure paths in the code (e.g. every secret data does not leak to the network, a master password is not written to a file).

In this paper we consider the non-interference property of the data processed by bytecode programs. A program satisfies this property whenever the properties of the information from higher levels of confidentiality cannot be deduced from the data on lower ones. As a starting point we take an information-flow type system for Java bytecode that guarantees non-interference [8, 9]. The main contribution of this paper is a translation of the type system into a program logic developed within the MOBIUS project [15].

It is worth pointing out that the logical model for the non-interference property has several desirable features. First of all, it is possible to use a toolset based on logical methods rather than typed ones. The translation may be especially beneficial when the logical toolset is within the trusted code base. Second, the resulting model of non-interference is more flexible when the non-interference property must be enforced in a more sophisticated situation when the declassification is needed. It also allows in a flexible way to use different collections of the same type to contain values of different levels of confidentiality as well as associate with them in a safe way iterators and enumerators. Moreover, additional algorithms allow to easily discover in the source code the cases when the initial model is replaced by a more complicated solution.

The paper is structured as follows. In Section 3.2 we fix the notation and present the basic notions which are exploited in the paper. This is followed in Section 3.3 by an exposition of the translation from a type based system to the logic based one. This translation is supplemented by a proof that the resulting specifications guarantee the non-interference property in Section 3.4. The formal development is concluded by a proof that the non-interference property holds even when the bytecode program is extended with other specifications. This is presented in Section 3.4.1. At last we present the related work and conclusions in Section 3.5.

3.2 Preliminaries

Basic notation We use the expression $\mathrm{dom}(f)$ to denote the domain of the (partial) function f. Similarly, $\mathrm{rng}(f)$ denotes the image of the function f. To denote the fact that f is a partial function from the set A to B we write $f : A \rightharpoonup B$. The powerset of a set A is written $P(A)$. We write $\vec{k}$ to denote a vector of values. The notation $|\vec{k}|$ denotes the length of the vector and $\vec{k}(0), \ldots, \vec{k}(|\vec{k}| - 1)$ are the subsequent elements of the vector.

Java bytecode programs and specifications A Java bytecode program P is a set of classes with one distinctive method main_P. A class C is a set of fields and methods. Each field f has a name f_n and a type f_t. Similarly, a method m has a name N_m, a signature T_m and the body B_m. We assume that the method names are unique within a single program (possibly due to the standard Java prefixing with the object or class name). A method body is a sequence of bytecode instructions. The instructions are indexed by program points. For each method we distinguish the set of all program points in the method $\mathcal{PP}_m$ (we omit the subscript m when possible).

An annotated program $\hat{P}$ (following [15, Chapter 3]) has additionally for each class C

- a list Ghost_C of model and ghost fields,
- a list Inv_C of (static and object) class invariants,
- a list Constr_C of (static and object) history constraints,
- a method specification table M_C.

The method specification table M_C associates with each method m:

- a method specification $S_m = (R_m, T_m, \Phi_m)$ where R_m is the precondition of the method m, T_m is the postcondition of the method, and Φ_m is the method invariant;
- a local specification table G_m which assigns to each label in the method body B_m an additional assumption that may be used in the proof of the program verification clause associated with the label;
- a local annotation table Q_m which assigns to each label in the method body B_m an additional assertion,
- a local instruction table Ins_m which assigns to each label l in the method body B_m a sequence of bytecode instructions that operate on ghost variables which is supposed to be executed before the instruction at the label l.

Java Virtual Machine semantics We will use the Java Virtual Machine semantics proposed by Barthe et al. [9, Figure 10]. Here, we present only its brief overview. We use $\mathcal{V}$ to denote the set of possible values (i.e. $\mathbb{Z} \times \mathcal{L} \times \{null\}$), $\mathcal{L}$ to denote the set of possible locations in heaps. Heap is the set of all heaps (finite partial functions $\mathcal{L} \rightharpoonup \mathcal{O}$ where $\mathcal{O}$ is understood as the set of partial functions $\mathcal{C} \times \mathcal{F} \rightharpoonup \mathcal{V}$ with $\mathcal{C}$ being the set of all classes and $\mathcal{F}$ being the set of all possible fields in objects). The set $\mathcal{X}$ is the set of all possible variable names. The set Labels denotes the set of all the program instruction labels.

The semantics is described by a one step relation $\overset{(n)}{\leadsto}_{m,\tau}$, where n counts the number of method invocations, m is the current method, and τ is a tag which determines if the step is a normal step or an exceptional step and in the latter case the exception thrown. When the instruction is not a return from a method (normal or exceptional), the one step relation relates pairs:

$$\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_{m,\tau} \langle i', \rho', os', h'\rangle$$

where $i, i' \in \mathsf{Labels}$ are instruction lables in B_m, $\rho, \rho' : \mathcal{X} \rightharpoonup \mathcal{V} + \mathcal{L}$ define the values of the local variables, os, os' are local operand stacks of the method m, and $h, h' \in \mathsf{Heap}$ are heaps. We sometimes use the shorhand State to denote $\mathsf{Labels} \times (\mathcal{X} \rightharpoonup \mathcal{V} + \mathcal{L}) \times \mathcal{V}^* \times \mathsf{Heap}$, the set of values related by the normal step relation. In case a return action should be performed, the relation is slightly different:

$$\langle i, \rho, os, h\rangle \overset{(n)}{\rightsquigarrow}_{m,\tau} l, h'$$

where i, ρ, os, and h are as before and $l \in \mathcal{V} + \mathcal{L}$ defines the return value for the method, and $h' \in \mathsf{Heap}$ defines the resulting heap.

Additionally, the semantics uses a big-step $\Downarrow_m^{(n)}$ relation in case of the method invocation instructions (m and n have the same meaning as in the case of the small step semantics). This relation always relates pairs of the form:

$$\langle i, \rho, os, h\rangle \Downarrow_m^{(n)} l, h'$$

To make the notation clearer, we usually omit the superscript n.

This semantics uses auxiliary operations that statically calculate different properties:

1. A function classAnalysis which given a program point returns the set of exception classes of exceptions that may be thrown at the program point.
2. A function excAnalysis which given a method name N_m returns the set of exceptions that are possibly thrown by m.
3. A function nbLocals which given a method name N_m returns the number of its local variables.
4. A function nbArguments which given a method name N_m returns the number of its arguments.
5. A function $\mathsf{RuntimeExceptionHandling} : \mathsf{Heap} \times \mathcal{L} \times \mathcal{C} \times PP \times (\mathcal{X} \rightharpoonup \mathcal{V}) \to \mathsf{State} + (\mathcal{L} \times \mathsf{Heap})$ which chooses either to exit (in case the exception is not handled locally) exceptionally from a method or to jump to the local exception handling code (in case the exception can be handled locally).
6. Partial functions $\mathsf{Handler}_m : \mathcal{PP} \times \mathcal{C} \rightharpoonup \mathcal{PP}$ that for each method m take a program point i and an exception class τ and select the program point j at which the exception handler for τ when triggered by the instruction at i starts.

Security policy The security policy framework we employ in this paper is based on assumption that the attacker can observe the input/output of methods only (e.g. methods that perform network operations). This, however, is extended to the values of fields and heaps as otherwise it is difficult to guarantee statically the non-interference property. We also assume that the attacker is unable to observe the termination of the programs.

Formally, a security policy is described by a finite partial order $(\mathcal{S}, \leq)$. This order allows to express the capabilities of the attacker and the program to be analysed:

- A security level k_{obs} determines the observational capabilities of the attacker (she can observe field, local variables and return values whose level is less or equal to k_{obs}).
- A policy function ft assigns to each field its security level. This allows us to express the non-interference property we are interested in.

- A policy function Γ that associates to each method identifier N_m and security level $k \in S$ a security signature $\Gamma_m[k]$. This signature gives the security policy of the method m called on object of level k. The set of security signatures for a method m is defined as $\mathsf{Policies}_\Gamma(m) = \{\Gamma_m[k] \mid k \in \mathcal{S}\}$. The security signature has the following shape:

$$\vec{k_p} \xrightarrow{k_h} \vec{k_r}$$

  - $\vec{k_p}$ describes the security levels appropriate for the local variables of the method (in particular it assigns also the levels to the input parameters), $k_p[0]$ is the upper bound on the security level of an object that calls the method,
  - k_h describes the security level of the heap operations performed by the method
  - $\vec{k_r}$ describes the security levels for the method results (both normal and exceptional ones); it is a list of the form $\{n : k_n, e_1 : k_{e_1}, \ldots, e_n : k_{e_n}\}$, where k_n is the security level of the return value and e_i is an exception class that might be propagated by the method in a security environment of level k_i. We use the notation $k_r[n]$ and $k_r[e_i]$ for k_n and k_{e_i}.

For simplicity, the orders we actually use here are modelled as suborders of natural numbers. However, it is straightforward to extend the definitions to arbitrary finite orders. This can be achieved either by a more complicated definition of the inequality $\leq$ we use in the formulae below or by using objects for which the inequality can be encapsulated in one of their methods. This last solution allows also to consider certain infinite orders.

Non-interference In order to define the non-interference property we are interested in, we have to provide a few notions of indistinguishability.

Definition 1 `(value indistinguishability)`
Let β be a partial function $\mathcal{L} \rightharpoonup \mathcal{L}$ we say that $v_1, v_2 \in \mathcal{V}$ are indistinguishable (written $v_1 \sim_\beta v_2$) when

- $v_1 = v_2 = null$, or
- $v_1, v_2 \in \mathbb{Z}$ and $v_1 = v_2$, or
- $v_1, v_2 \in \mathcal{L}$ and $\beta(v_1) = v_2$.

Definition 2 `(local variables indistinguishability)`
Let $\rho, \rho' : \mathcal{X} \rightharpoonup \mathcal{V}$ be two assignments of values for local variables. Let β be a partial function $\mathcal{L} \rightharpoonup \mathcal{L}$. We write $\rho \sim_{\vec{k_p},\beta} \rho'$ if $\mathrm{dom}(\rho) = \mathrm{dom}(\rho')$ and $\rho(x) \sim_\beta \rho'(x)$ for all $x \in \mathrm{dom}(\rho)$ such that $\vec{k_p}(x) \leq k_{\mathrm{obs}}$.

Definition 3 `(object indistinguishability)`
Let β be a partial function $\mathcal{L} \rightharpoonup \mathcal{L}$ and $o_1, o_2 \in \mathcal{O}$. We write $o_1 \sim_\beta o_2$ if o_1 and o_2 are of the same class and for every field $f \in \mathrm{dom}(()o_1)$ such that $\mathsf{ft}(f) \leq k_{\mathrm{obs}}$ the value indistinguishability $o_1.f \sim_\beta o_2.f$ holds.

Definition 4 `(heap indistinguishability)`
Let β be a partial function $\mathcal{L} \rightharpoonup \mathcal{L}$. We write $h_1 \sim_\beta$ if and only if the following conditions hold:

1. β is a bijection between $\mathrm{dom}(\beta)$ and $\mathrm{rng}(\beta)$,
2. $\mathrm{dom}(\beta) \subseteq \mathrm{dom}(h_1)$ and $\mathrm{rng}(\beta) \subseteq \mathrm{dom}(h_2)$,
3. for each $l \in \mathrm{dom}(\beta)$ the object indistinguishability $h_1(l) \sim_\beta h_2(\beta(l))$ holds.

Definition 5 `(output indistinguishability)`
Let β be a partial function $\mathcal{L} \rightharpoonup \mathcal{L}$ and $\vec{k_r}$ be an output security level. We write $(v_1, h_1) \sim_{\vec{k_r},\beta} (v_2, h_2)$ when

1. $h_1 \sim_\beta h_2$ and if $\vec{k_r}[n] \leq k_{\text{obs}}$ then $v_1 \sim_\beta v_2$, or
2. $h_1 \sim_\beta h_2$, and $v_1, v_2 \in \mathcal{L}$ and $\mathsf{class}(h_1(l_1))$ has the level $k \in \vec{k_r}$, $k \leq k_{\text{obs}}$, and $v_1 \sim_\beta v_2$,
3. $h_1 \sim_\beta h_2$, and $v_1, v_2 \in \mathcal{L}$ and $\mathsf{class}(h_1(l_1))$ has the level $k \in \vec{k_r}$, $k \not\leq k_{\text{obs}}$,
4. $h_1 \sim_\beta h_2$, and $v_1, v_2 \in \mathcal{L}$ and $\mathsf{class}(h_2(l_2))$ has the level $k \in \vec{k_r}$, $k \not\leq k_{\text{obs}}$,
5. $h_1 \sim_\beta h_2$, and $v_1, v_2 \in \mathcal{L}$, and $\mathsf{class}(h_1(l_1))$ has the level $k_1 \in \vec{k_r}$, and $\mathsf{class}(h_2(l_2))$ has the level $k_2 \in \vec{k_r}$ $k_1 \not\leq k_{\text{obs}}$, and $k_2 \not\leq k_{\text{obs}}$.

Definition 6 (`non-interference of methods`)
A method m is non-intereferent with respect to a policy $\vec{k_p} \to \vec{k_r}$ if for every partial function $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ and every $\rho_1, \rho_2 : \mathcal{X} \rightharpoonup \mathcal{V}$, $h_1, h_2, h_1', h_2' \in \mathsf{Heap}$, $r_1, r_2 \in \mathcal{V} + \mathcal{L}$ such that $\rho_1, h_1 \Downarrow_m r_1, h_1'$ and $\rho_2, h_2 \Downarrow_m r_2, h_2'$ with $h_1 \sim_\beta h_2$ and $\rho_1 \sim_{\vec{k_p},\beta} \rho_2$ there exists a partial function $\beta' : \mathcal{L} \rightharpoonup \mathcal{L}$ such that $\beta \subseteq \beta'$ and $(r_1, h_1) \sim_{\vec{k_r},\beta'} (r_2, h_2)$.

Definition 7 (`side-effect preorder`)
Let $h, h' \in \mathsf{Heap}$. We write $h \preceq_k h'$ and say that the heaps are side effect preordered with respect to a security level $k \in \mathcal{S}$ when $\mathrm{dom}(h) \subseteq \mathrm{dom}(h')$ and for each location $l \in \mathrm{dom}(h)$ and each field $f \in \mathcal{F}$ such that $k \not\leq \mathsf{ft}(f)$ we have $h(l).f = h'(l).f$.

Definition 8 (`side-effect safe`)
A method m in a program P is side-effect-safe with respect to a security level k_h if for all local variables assignments $\rho : \mathcal{X} \rightharpoonup \mathcal{V}$, all heaps $h, h' \in \mathsf{Heap}$ and a value $v \in \mathcal{V}$, the reduction $\rho, h \Downarrow_m v, h'$ implies $h \preceq_{k_h} h'$.

Definition 9 (`safe methods`)
A method m in a program P is safe with respect to a security signature $\vec{k_p} \xrightarrow{k_h} \vec{k_r}$ when m is side-effect-safe with respect to k_h and non-intereferent with respect to the policy $\vec{k_p} \to \vec{k_r}$.

Definition 10 (`safe programs`)
A program P is safe with respect to a table of method signatures Γ if for each method m in P, m is safe with respect to all the security signatures in $\mathsf{Policies}_\Gamma(m)$.

Non-structured programs The bytecode programs do not have instructions typical for structured programming. Instead they provide control flow by means of jumps and conditional jumps. In order to reason on the information flow of such programs an additional structural information is needed. As we translate typings in an information flow system [9] we need the same descriptions of the bytecode program structure.

We use a binary successor relation $\mapsto^\tau \subseteq PP \to PP$ defined on the program points. This relation is parametrised by a tag τ since bytecode instructions may have several successors as it may execute normally (then the tag is $\emptyset$) or an exception may be triggered by the instruction (then the tag is the class of the exception). Intuitively, j is a successor of i ($i \mapsto j$) if performing one step execution from a state whose program point is i may lead to a state whose program point is j. We write $i \mapsto$ if i corresponds to a return instruction.

We assume that a bytecode program P comes equipped with a control dependence regions structure `cdr` which consists of a pair of partial functions (`region`, `jun`). The role of the functions is to arrange the program into compact parts for which the analysis of program invariants can be conducted separately. The `region` function describes the internal parts while `jun` the connections between them. The types of the functions are the following:

$$\mathtt{region}_m : \mathcal{PP} \times (\{\emptyset\} + \mathcal{C}) \to P(\mathcal{PP}) \qquad \mathtt{jun}_m : \mathcal{PP} \times (\{\emptyset\} + \mathcal{C}) \rightharpoonup \mathcal{PP}$$

The functions can be axiomatised by the following SOAP (Safe Over APproximation) properties:

SOAP1 for all program points i, j, k and tag τ such that $i \mapsto^{\emptyset} j$, $i \mapsto^{\tau} k$ and $j \neq k$ (i is a branching point), $k \in \mathtt{region}(i, \tau)$ or $k = \mathtt{jun}(i, \tau)$;

SOAP2 for all program points i, j, k and tag τ, if $j \in \mathtt{region}(i, \tau)$ and $j \mapsto^{\emptyset} k$ then either $k \in \mathtt{region}(i, \tau)$ or $k = \mathtt{jun}(i, \tau)$;

SOAP3 for all program points i, j and tag τ if $j \in \mathtt{region}(i, \tau)$ (or $i = j$) and j is a return point then $\mathtt{jun}(i, \tau)$ is undefined;

SOAP4 for each program point i and tags τ_1, τ_2, if $\mathtt{jun}(i, \tau_1)$ and $\mathtt{jun}(i, \tau_2)$ are defined and $\mathtt{jun}(i, \tau_1) \neq \mathtt{jun}(i, \tau_2)$ then $\mathtt{jun}(i, \tau_1) \in \mathtt{region}(i, \tau_2)$ or $\mathtt{jun}(i, \tau_2) \in \mathtt{region}(i, \tau_1)$;

SOAP5 for all program points i, j and tag τ if $j \in \mathtt{region}(i, \tau)$ (or $i = j$) and j is a return point then for each tag τ' such that $\mathtt{jun}(i, \tau')$ is defined, $\mathtt{jun}(i, \tau') \in \mathtt{region}(i, \tau)$.

Typable programs To check that a program is safe one may use a type system presented in [9]. In this type system every method is checked against its signatures separately. The type system is parametrised by:

- a table Γ of method signatures,
- a global policy ft that provides security levels of fields,
- a cdr structure $(\mathtt{region}_m, \mathtt{jun}_m)$ for every method m

Moreover, we suppose that functions classAnalysis, excAnalysis, nbLocals, nbArguments, Handler are given and that they are correct (computed in the trusted computing base).

Definition 11 `(typable programs)`
Program P is *typable* with the policy $(k_{\mathrm{obs}}, \mathsf{ft}, \Gamma)$ and cdr satisfying SOAP if every method m from P is typable with respect to ft, Γ, $\mathtt{region}_m$ and all signatures in $\mathsf{Policies}_\Gamma(m)$.

Definition 12 `(typable methods)`
Method m is *typable* with respect to ft, Γ, $\mathtt{region}_m$ and a signature sgn if there exists a security environment $\mathsf{se} : PP \to S$ and a function $\mathsf{st} : PP \to S^\star$ such that $st(1) = \epsilon$ and for all $i, j \in PP$, $e \in \{\emptyset\} \cup \mathcal{C}$:

1. if $i \mapsto^e j$ then there exists $s \in S^\star$ such that $\Gamma, \mathsf{ft}, \mathtt{region}_m, \mathsf{se}, \mathsf{sgn}, i \vdash^e \mathsf{st}(i) \implies s$ and $s \sqsubseteq \mathsf{st}(j)$,
2. if $i \mapsto^e$ then $\Gamma, \mathsf{ft}, \mathtt{region}_m, \mathsf{se}, \mathsf{sgn}, i \vdash^e \mathsf{st}(i) \implies$

where $\subseteq$ denote the point-wise partial order on type stack with respect to the partial order taken on security levels. Two exemplary rules for .. $\vdash^e$ .. $\implies$ .. relation are given in Figure 3.2. The set of all rules is given in [9].

Note that st and se are chosen for particular signature sgn. This signature, in turn, comes from $\mathsf{Policies}_\Gamma(m)$ and for each security level $s \in S$ we have a single signature. In this light we may consider se and st in collections indexed by elements of S only.

The main theorem of [9] states that typable programs are safe.

Theorem 3.2.1 *from typable to safe programs Let P be a Java program, $(k_{\mathrm{obs}}, \mathsf{ft}, \Gamma)$ a desired security policy, and cdr a control dependence regions structure satisfying SOAP. If P is typable with respect to $(k_{\mathrm{obs}}, \mathsf{ft}, \Gamma)$ and cdr then P is safe with respect to Γ.*

Proof:
See [9, Section 6]. □

$$\frac{P_m[i] = \texttt{ifeq}\ j \qquad \forall j' \in \texttt{region}(i,\emptyset),\ k \leq se(j')}{\Gamma, \mathsf{ft}, \texttt{region}, \mathsf{se}, \vec{k_p} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\emptyset} k :: st \implies \texttt{lift}_k(st)}$$

$$\frac{\begin{array}{c} P_m[i] = \texttt{invokevirtual}\ N_{m'} \qquad \Gamma_{N_{m'}}[k] = \vec{k'_p} \xrightarrow{k'_h} \vec{k'_r} \\ k \sqcup k_h \sqcup se(i) \leq k'_h \qquad \texttt{length}(st_1) = \mathsf{nbArguments}(N'_m) \\ k \leq k'_p[0] \qquad \forall i \in [0, \texttt{length}(st_1) - 1],\ st_1[i] \leq k'_p[i+1] \\ k_e = \sqcup\{k'_r[e] \mid e \in \mathsf{excAnalysis}(N_{m'})\} \\ \forall j \in \texttt{region}(i,\emptyset),\ k \sqcup k_e \leq se(j) \end{array}}{\Gamma, \mathsf{ft}, \texttt{region}, \mathsf{se}, \vec{k_p} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\emptyset} st_1 :: k :: st_2 \implies \texttt{lift}_{k \sqcup k_e}((k'_r[n] \sqcup se(i)) :: st_2)}$$

Figure 3.1: The rule for `ifeq` and a rule for `invokevirtual`

3.3 Translation from the information flow system

The translation of the information flow system into the MOBIUS base logic is done with the use of the BML syntax. Here is a brief summary of the syntax elements which are useful in the presentation of the translation.

We use the modifier **ghost** to indicate that a particular variable is not a program variable, but a specification variable. Other Java type modifiers such as **public**, **private**, **final**, **static** have the same meaning as in the case of Java declarations. We use JML syntax to denote the logical connectives i.e. && means the logical conjunction, || the logical alternative, and ! the logical negation. The logical implication is denoted as ==>. We also use the JML syntax to access and initialise elements of arrays and to denote types of variables. We also use here the general quantifier. The syntax of the quantifier expression is as follows:

(\forall variable declaration; bound on the quantification; actual formula)

where the *variable declaration* has the same form as a one line variable declaration in Java and introduces the variables over which the quantification takes place. The *bound on the quantification* is a formula the goal of which is to restrict potentially infinite domain of the quantification to be finite and it can be any boolean Java expression. At last, the *actual formula* is the formula we are interested in. A *bound on the quantification* B and an *actual formula* A are understood as the implication from B to A.

In order to express in MOBIUS base logic that there is no information flow we need to add some formula annotations to the program and to extend the method specification tables with the formulae encoding Definition 12. Both can be done separately for each method. The translation uses extensively ghost fields.

Data to translate The annotations we need are of two kinds. The first group consists of data used in the information-flow type system, namely:

- a table Γ of method signatures,
- a global policy ft that provides security levels of fields,
- a cdr structure $(\texttt{region}_m, \texttt{jun}_m)$,
- functions classAnalysis, excAnalysis, nbLocals, nbArguments, Handler,
- the successor relation $\mapsto^e$.

The second group comes from the Definition 12. For each policy signature of a given method these are:

- a security environment $\mathsf{se} : PP \to S$,
- a function $\mathsf{st} : PP \to S^\star$.

Security level of fields can be stored in ghost fields in the corresponding class. For each class field `f` (both static and instance) we define:

```
public final ghost S gft_f = ft(f);
```

this variable allows to consult the security level of the field f. The domain of the functions Γ, excAnalysis, nbLocals, nbArguments is the name space for methods names. It seems that the best place to store them are ghost fields of the class where the given method name appears. The other data are relative to the body of a concrete instance of the method and they should be stored as local ghost variables of the methods. We assume the following additional numbers are calculated statically:

1. lm the maximal label of the method m,
2. maxStack the maximal height of the stack during the execution of method m,
3. maxEx the number of all exception types in program P,
4. maxS the maximal security level; the level $0 \notin S$ will be used to mark the undefined value. Let $S0 = S \cup \{0\}$

These values are inlined in the specifications (i.e. we do not use any special ghost variable definitions to refer to them).

In each class we add the following ghost field definitions initialised according to the static data mentioned above. In the definitions below we use a fixed correspondence between the exception types and the natural numbers $0, \ldots, \mathsf{maxEx}-1$. For each method (both static and instance) m with the identifier N_m we define a set of ghost variables. These variables will be used as constants; they will never be changed. The initial values of all the ghost variables we use here are defined to correspond directly to the values of real values/functions.

```
public static final ghost int gnbArguments_$N_m$ =
                                        nbArguments($N_m$);
public static final ghost int gnbLocals_$N_m$ =
                                        nbLocals($N_m$);
public static final ghost boolean [maxEx] gexcAnalysis_$N_m$ =
                        { $e_0, \ldots, e_{\mathsf{maxEx}-1}$ };
public static final ghost S0
        [maxS][gnbLocals_$N_m$+3+maxEx] gsgn_$N_m$ =
            {
              { $s_{0,0}, \ldots, s_{0,gnbLocals\_N_m+3+\mathsf{maxEx}-1}$ },
              ...
              { $s_{\mathsf{maxS}-1,0}, \ldots, s_{\mathsf{maxS}-1,gnbLocals\_N_m+3+\mathsf{maxEx}-1}$ }
            };
```

This definition makes use of some additional values which are defined and explained below. The information contained in **gexcAnalysis**_N_m is defined with the use of:

$$e_i = \begin{cases} \textbf{true} & \text{when excAnalysis}(N_m) \text{ says} \\ & \text{that the exception number } i \text{ is rised in } m \\ \textbf{false} & \text{otherwise} \end{cases}$$

The security signature $\Gamma_m[i] = \vec{k_p} \xrightarrow{k_h} \vec{k_r}$ allows us to give the values for $s_{i,j}$.

$$s_{i,j} = \begin{cases} \vec{k_p}(j) & j < |\vec{k_p}| \\ k_h & j = |\vec{k_p}| \\ \vec{k_r}(j - |\vec{k_p}| - 1) & j > |\vec{k_p}| \end{cases}$$

This definition allows us to explain the meaning of $\mathsf{gsgn}[i][j]$ in the following way. For a given security level i, $\mathsf{gsgn}[i][0]$ is the security level of the object that calls the method m, $\mathsf{gsgn}[k][1 \ldots gnbLocals\_N_m]$ are security levels of parameters and local variables (note that $\mathsf{nbLocals}(N_m) = |\vec{k_p}| - 1$), $\mathsf{gsgn}[k][gnbLocals\_N_m + 1]$ is the level of heap operations, while $\mathsf{gsgn}[k][gnbLocals\_N_m + 2]$ is the level of a normal return value and $\mathsf{gsgn}[k][gnbLocals\_N_m + 3 \ldots gnbLocals\_N_m + 3 + \mathsf{maxEx} - 1]$ are the security levels in which corresponding exceptions might be propagated (note that $\mathsf{maxEx} \geq |\vec{k_r}| - 1$).

We define also local ghost variables associated with the method m:

```
ghost boolean [lm][maxEx] gclassAnalysis =
            {
              { $c_{0,0}, \ldots, c_{0,\mathsf{maxEx}-1}$ },
              ...
              { $c_{\mathsf{lm}-1,0}, \ldots, c_{\mathsf{lm}-1,\mathsf{maxEx}-1}$ }
            };
```

where

$$c_{i,j} = \begin{cases} \textbf{true} & \text{when } \mathsf{classAnalysis}(i) \text{ says that the exception number } j \\ & \ \text{can be rised in } m \text{ at } i, \\ \textbf{false} & \text{otherwise.} \end{cases}$$

```
ghost boolean [lm][maxEx + 1][lm] gregion =
            {
              {
                { $r_{0,0,0}, \ldots, r_{0,0,\mathsf{lm}-1}$ },
                ...
                { $r_{0,\mathsf{maxEx}-1,0}, \ldots, r_{0,\mathsf{maxEx}-1,\mathsf{lm}-1}$ }
              },
              ...
              {
                { $r_{\mathsf{lm}-1,0,0}, \ldots, r_{\mathsf{lm}-1,0,\mathsf{lm}-1}$ },
                ...
                { $r_{\mathsf{lm}-1,\mathsf{maxEx},0}, \ldots, r_{\mathsf{lm}-1,\mathsf{maxEx},\mathsf{lm}-1}$ }
              }
            };
```

where

$$r_{i,j,k} = \begin{cases} \textbf{true} & \text{when } k \in \mathtt{region}_m(i,e) \text{ and} \\ & \ \text{the exception number corresponding to } e \text{ is } j \\ \textbf{true} & \text{when } k \in \mathtt{region}_m(i,\emptyset) \text{ and} \\ & j = \mathsf{maxEx} \\ \textbf{false} & \text{otherwise.} \end{cases}$$

Note that we use the index maxEx on the second coordinate to encode the region information for the normal execution.

```
ghost int [lm][maxEx] gHandler =
            {
              { $h_{0,0}, \ldots, h_{0,\mathsf{maxEx}-1}$ },
              ...
              { $h_{\mathsf{lm}-1,0}, \ldots, h_{\mathsf{lm}-1,\mathsf{maxEx}-1}$ }
            };
```

where $h_{i,j} = \mathsf{Handler}_m(i,e)$ with e corresponding to the exception number j. Handler function is encoded as a two-dimensional array of boolean values, since the value of a handler is not used in the type-system.

As noted below Definition 12, we may assume that se and st are indexed with security levels from S. We use the notation se_i and st_i for $i \in S$ to refer to the elements of the indexed families.

```
ghost S [maxS][lm] gse =
            {
              { $v_{0,0}, \ldots, v_{0,\mathsf{lm}-1}$ },
              ...
            { $v_{\mathsf{maxS}-1,0},\dots,v_{\mathsf{maxS}-1,\mathsf{lm}-1}$ }
        };
```

where $v_{i,j} = \mathsf{se}_i(j)$.

```
  ghost S0 [maxS][lm][maxStack] gst =
        {
          {
            { $t_{0,0,0},\dots,t_{0,0,\mathsf{maxStack}-1}$ },
            ...
            { $t_{0,\mathsf{lm}-1,0},\dots,t_{0,\mathsf{lm}-1,\mathsf{maxStack}-1}$ }
          },
          ...
          {
            { $t_{\mathsf{maxS}-1,0,0},\dots,t_{\mathsf{maxS}-1,0,\mathsf{maxStack}-1}$ },
            ...
            { $t_{\mathsf{maxS}-1,\mathsf{lm}-1,0},\dots,t_{\mathsf{maxS}-1,\mathsf{lm}-1,\mathsf{maxStack}-1}$ }
          }
        };
```

where $t_{i,j,k} = n$ whenever $k+1$-st element of the sequence $\mathsf{st}_i(j)$ is n. Note that the function st_i gives security levels for the stack positions so that the length of each $\mathsf{st}_i(j)$ is less that the maximal stack length $\mathsf{maxStack}$. We also assume that for each i, j the elements of $gst[i][0][j]$ are zero which corresponds to the fact that the operand stack at the beginning of a method is empty.

Translating non-interference property Once we have all the needed annotations, we may translate the property described in Definition 12. We do it for each method m separately and we decide to use local annotation table Q_m (as in [15, Chapter 3]).

Q_m is a finite partial map which for a program label i occurring in m gives an assertion $Q(c0, c)$. If a program point i in m is annotated with Q then $Q(c0, c)$ is supposed to hold in very c encountered at label i during any execution of m with the initial state $c0$ satisfying $R_m(c0)$ (i.e. the precondition of the method).

Let us describe how to extend a given specification Q so that it ensures the non-interference property. According to Definition 12 we need to state that for every security level s, every label i, every exception e, and every j, such that $i \mapsto^e j$ (or $i \mapsto^e$) some properties hold. For every i they will be expressed by a formula $N(i)(s)$. We define QNI, the local annotation table extended with non-interference checking, as

$$\begin{aligned}\mathsf{QNI}(i) = \lambda c0 \in \mathsf{State} \quad & \lambda c \in \mathsf{State}.\\ & Q(c0, c)\ \&\&\ N(i)(1)\ \&\&\ \dots\ \&\&\ N(i)(\mathsf{maxS}).\end{aligned} \tag{3.1}$$

Additionally, we add to $\mathsf{QNI}(1)$ the formula:

$$\backslash\mathsf{forall\ int}\ i, j; 0 \le i\ \&\&\ i < \mathsf{maxS}\ \&\&\ 0 \le j\ \&\&\ j < \mathsf{maxStack}; gst[i][0][j] == 0 \tag{3.2}$$

The formulae $N(i)(s)$ have similar form; it is

$$\begin{aligned}(\backslash\mathsf{forall\ int}\ e,\ j;\ 1 \le e\ \&\&\ e \le \mathsf{maxEx}\ \&\&\ 1 \le j\ \&\&\ j \le \mathsf{lm};\\ (i \mapsto^e j \texttt{ ==> } (\mathsf{Reg}_1^{\mathsf{inst(i)}}(\vec{p_1})\ ||\ \dots\ ||\ \mathsf{Reg}_k^{\mathsf{inst(i)}}(\vec{p_k})))\ \&\&\\ (i \mapsto^e \quad \texttt{==> } (\mathsf{Reg}_1^{\mathsf{inst(i)}}(\vec{p_1})\ ||\ \dots\ ||\ \mathsf{Reg}_{k'}^{\mathsf{inst(i)}}(\vec{p_{k'}}))))\end{aligned} \tag{3.3}$$

where $\mathsf{inst(i)}$ is the instruction at the label i in the method body m. Note also that $i \mapsto^e j$ (as well as $i \mapsto^e$) is a static information which can be easily defined directly as a subformula to be inserted at the place or with help of an additional ghost array.

Every $\mathsf{Reg}_i^{\mathsf{inst(i)}}(\vec{p_i})$ corresponds to one of the possibly applicable typing rules for instruction $\mathsf{inst(i)}$ in case $i \mapsto^e j$ (or $i \mapsto^e$) where $\vec{p_i}$ are parameters of inst. For example $\mathsf{Reg}_1^{\mathsf{ifeq}}(j)$ corresponds to $\mathsf{ifeq\ j}$ and equals to

$$\begin{array}{l} e == \emptyset \ \ \&\& \\ (\backslash\mathsf{forall}\ \mathrm{int}\ j';\ 1 \le j' \ \&\&\ j' \le \mathsf{lm}; \\ \qquad\qquad gregion[i][0][j'-1] ==> gst[s][i-1][\mathsf{cntr}] \le gse[s][j'-1]) \quad \&\& \\ (\backslash\mathsf{forall}\ \mathrm{int}\ p;\ 0 \le p \ \&\&\ p \le \mathsf{cntr}-1; \\ \qquad\qquad gst[s][i-1][p] \sqcup gst[s][i-1][\mathsf{cntr}] \le gst[s][j-1][p]) \quad \&\& \\ (\backslash\mathsf{forall}\ \mathrm{int}\ p;\ \mathsf{cntr} \le p; \\ \qquad\qquad gst[s][j-1][p] = 0) \end{array} \tag{3.4}$$

The two last lines state that $\mathsf{lift}_k(st) \subseteq st(j)$; the last one explicitely checks that $st(j)$ is one element shorter than $st(i)$. $\mathsf{Reg}_1^{\mathsf{invokevirtual}}(N_{m'})$, for some called method m', corresponds to

$$\begin{array}{l} e == \emptyset \ \ \&\& \\ (\backslash\mathsf{forall}\ \mathrm{int}\ k, n, k_e;\ 1 \le k \ \&\&\ k_e \le \mathsf{maxS}; \\ \qquad (n == gnbArguments\_N_{m'} \ \&\& \\ \qquad \mathsf{cntr} \ge n+1 \ \&\& \\ \qquad k == gst[s][i-1][\mathsf{cntr}-n] \ \&\& \\ \qquad k_e == \sqcup\{gsgn\_N_{m'}[k][n+2+i] \mid gexcAnalysis\_N_{m'}[i]\}) \\ \qquad\quad ==> \\ \qquad k \sqcup gsgn\_N_m[s][gnbLocals\_N_m + 1] \sqcup gse[s][i] \le \\ \qquad\qquad gsgn\_N_{m'}[k][gnbLocals\_N_{m'} + 1] \ \&\& \\ \qquad k \le gsgn\_N_{m'}[k][0] \ \&\& \\ \qquad (\backslash\mathsf{forall}\ \mathrm{int}\ j;\ 0 \le j \ \&\&\ j \le n-1; \\ \qquad\qquad gst[s][i-1][\mathsf{cntr}-j] \le \\ \qquad\qquad\qquad gsgn\_N_{m'}[k][j+1]) \ \&\& \\ \qquad (\backslash\mathsf{forall}\ \mathrm{int}\ j';\ 1 \le j' \ \&\&\ j' \le \mathsf{lm}; \\ \qquad\qquad gregion[i][0][j'] \\ \qquad\qquad\quad ==> \\ \qquad\qquad k \sqcup k_e \le gse[s][j']) \ \&\& \\ \qquad (\backslash\mathsf{forall}\ \mathrm{int}\ p;\ 0 \le p \ \&\&\ p \le \mathsf{cntr}-n; \\ \qquad\qquad gst[s][i-1][p] \sqcup k \sqcup k_e \le gst[s][j-1][p]) \ \&\& \\ \qquad ((k \sqcup k_e \sqcup gsgn\_N_{m'}[k][n+2] \sqcup gse[s][i]) \le \\ \qquad\qquad gst[s][j-1][\mathsf{cntr}-n+1]) \ \&\& \\ \qquad (\backslash\mathsf{forall}\ \mathrm{int}\ p;\ \mathsf{cntr}-n+2 \le p; \\ \qquad\qquad gst[s][j-1][p] == 0) \end{array}$$

We recall that $N_{m'}$ stands for the method identifier of the called method m'. In order to shorten the formula we introduce shorthands n, k for the number of the parameters for the method to be called and the security level of the object that contains the called method, respectively. We also introduce the shorthand k_e ,with the informal use of $\sqcup$, to denote the least upper bound of security levels corresponding to exceptions allowed by the excAnalysis.

$$k_e == \sqcup\{gsgn\_N_{m'}[k][n+2+i] \mid gexcAnalysis\_N_{m'}[i]\})$$

Again, it is easy to define the appropriate formula which expresses this property as we can enumerate all the values that are involved here.

It is worth pointing out that the subformulae starting from the first occurrence of \forall int p state that

$$\mathsf{lift}_{k \sqcup k_e}((k'_r[n] \sqcup se(i)) :: st_2) \subseteq st(j).$$

3.4 Proof of non-interference

In our approach the non-interference property is obtained in two steps. The first one provides the link between the type system of Barthe et al. [9] and this is exploited in the second step which results in the non-interference by Theorem 3.2.1. We additionally provide a result which allows to mix the specifications that result from our translation with specifications that come from other sources (e.g. are written by hand).

Please recall that like in [9] we assume that functions $\mathsf{classAnalysis}$, $\mathsf{excAnalysis}$, $\mathsf{nbLocals}$, $\mathsf{nbArguments}$, $\mathsf{Handler}$ are correct (computed in the trusted computing base).

The goal of the translation we provided is to obtain the following theorem.

Theorem 3.4.1 *typechecking and verifyability Let P be a Java program, $(k_{\text{obs}}, \mathsf{ft}, \Gamma)$ a desired security policy, and cdr a control dependence regions structure satisfying SOAP. Let TR be the translation defined in Section 3.3 that adds base logic annotations to a bytecode program. For each security environment family $\{\mathsf{se}_i : PP \to \mathcal{P}PP\}_{i \in S}$ and a family of functions $\{\mathsf{st}_i : PP \to \mathcal{S}^\star\}_{i \in S}$,*

- $\Rightarrow$ *if the annotated program $\mathsf{TR}(P, k_{\text{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$ verifies correctly then P with the policy $(k_{\text{obs}}, \mathsf{ft}, \Gamma)$ and cdr is typable,*
- $\Leftarrow$ *if the program P with the policy $(k_{\text{obs}}, \mathsf{ft}, \Gamma)$ and cdr is typable with se, st, and all $Q(c0, c)$ in QNI in (3.1) on page 39) are empty then the annotated program $\mathsf{TR}(P, k_{\text{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$ verifies correctly.*

Proof:
We present here a sketch of the proof only.

($\Rightarrow$) Suppose that the annotated program $\mathsf{TR}(P, k_{\text{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$ verifies correctly. We want to verify that the condition in Definition 12 is fulfilled. We take se and st as above. $\mathsf{st}(1) = \epsilon$ is guaranteed by the presence of the subformula (3.2) in $\mathsf{QNI}(1)$. The conditions (1)–(2) from Definition 12 are guaranteed by the fact that all the typing rules are faithfully modelled in the logic. We provide here a motivation that the typing rule is faithfully modelled in case the instruction at the label i is $\mathsf{ifeq\ j}$. In other cases the proof is similar. Note that in this case we must ensure the point (1) only if i has a successor in the method. We have to ensure that the typing condition is fulfilled. This is guaranteed by the first \forall subformula of (3.4). The condition $s \sqsubseteq \mathsf{st}(j)$ is guaranteed in turn by the second \forall subformula of (3.4).

($\Leftarrow$) Suppose that the program P with the policy $(k_{\text{obs}}, \mathsf{ft}, \Gamma)$ and cdr is typable. We have to ensure that each $\mathsf{QNI}(i)$, for i being a label in the method m, holds. As $Q(c0, c)$ is empty, it is enough to check that each $N(i)(j)$ holds for $j \in \mathcal{S}$. Each of the $N(i)(s)$ has similar structure presented in (3.3). It is enough to show that one of the corresponding $\mathsf{Reg}_l^{\mathsf{inst(i)}}(\vec{p_l})$ holds in case $i \mapsto^e j$ (or in case $i \mapsto^e$). As the method is typable, we know that $\Gamma, \mathsf{ft}, \mathtt{region}_m, \mathsf{se}, \mathsf{sgn}, i \vdash^e \mathsf{st}(i) \implies s$ can be inferred. This is done with one of the rules, say $\mathsf{Reg}_l^{\mathsf{inst(i)}}(\vec{p_l})$. Now, we have to make sure that the corresponding translation formula holds. We show this in case $\mathsf{inst(i)}$ is $\mathsf{ifeq\ j}$.

- the subformula $e == \emptyset$ in the pattern (3.4) holds as the only rule for $\mathsf{ifeq\ j}$ works only when we deal with the normal execution,
- the second subformula of (3.4) holds as the typing rule guarantees that $\forall j' \in \mathtt{region}(i, \emptyset), k \leq \mathsf{se}(j')$,
- the third subformula of (3.4) holds as the typability requires that $\mathsf{lift}_k(\mathsf{st}(i)) \sqsubseteq \mathsf{st}(j)$,
- the fourth subformula of (3.4) holds as the $\mathsf{st}(j)$ is not determined for indices greater than the top of the operand stack.

This finishes the proof in this case. The cases of other instructions are similar. □
This theorem says in principle that whenever a program with annotations proposed in Section 3.3 successfully verifies it also successfully typechecks. This property implies the following result:

Theorem 3.4.2 *non-interference If $\mathsf{TR}(P, k_{\text{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$ verifies correctly then the program P is safe.*

Proof:
Suppose that $\mathsf{TR}(P, k_{\text{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$ verifies correctly. By Theorem 3.4.1 it means that P is typable with the policy $(k_{\text{obs}}, \mathsf{ft}, \Gamma)$ and cdr. As Theorem 3.2.1 states, this means that P is safe. □

3.4.1 Proof of stability

We want also obtain a result which allows safely to extend the specifications so that the non-interference property is preserved. This can be spelled out by the following Theorem 3.4.3.

Definition 13 `(specifications in conflict)`
We say that specifications are in conflict with the translation TR whenever any element of the ghost arrays or variables defined in Section 3.3 is set.

Theorem 3.4.3 *stability Let P' be a specificational extension of* $\mathsf{TR}(P, k_{\mathrm{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$ *that does not conflict with* $\mathsf{TR}(P, k_{\mathrm{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$*. If P' verifies correctly then P satisfies the non-interference property.*

Proof:
We present here a sketch of the proof only. When the specification extension does not conflict with the translation $\mathsf{TR}(P, k_{\mathrm{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$ then the values of all the variables used in the translation are the same. In this light, the logical values of the formulae are the same as in case there are no additional specifications. By Theorem 3.4.2 we obtain non-interference for P. □

3.5 Related work and conclusions

We presented here a translation of the information-flow type system defined in [8] and further explored in [9] to the MOBIUS base logic presented in [15]. The translation is in fact through the BML specification language syntax which should facilitate quicker adoption of the translation by tools. This translation gives possibility to combine the logical verification with the properties obtained using a type system. It seems also that our formalisation can enable an easy control over the declassification.

Conclusions The original order used in the information flow type system has not been restricted to be finite. Our translation relies crucially on the fact that the order is finite. In practise, however, it is very difficult to check the non-interference in case of essentially infinite policies—in particular such policies should be effectively enumerable and thus the checking that a policy is fulfilled becomes an algorithm verification task.

Chapter 4

Foundational certification of data-flow analysis

4.1 Introduction

In proof-carrying code [27], it is the responsibility of the code producer to produce evidence that the code shipped is safe and/or functionally correct. When code generation involves optimizations, an important useful intermediate mechanism is certification of the underlying program analyses, preferably based on a formalism rather than an informal mathematical theory.

It has been recognised, see, e.g., [29, 25], that classical data-flow analyses, such as live variables analysis, available expressions analysis etc., are usefully specifiable in a declarative way as *type systems* that may operate on source programs in a compositional (syntax-directed) manner, rather than on intermediate representations (such as flat control-flow graphs). These type systems make good formal vehicles for certification of analyses and can thus turn it very similar to customary certification of safety and functional correctness properties in program-logic like formalisms. A certificate is a typing derivation (or a typing judgement with sufficient additional information in the form of annotations to recover one), certificate checking is type(-derivation) checking and certificate generation amounts to principal type inference. The declarative character of type systems endows their use with additional value. For instance, there is no good reason for certificates to depend on an algorithmic definition of an analysis when only the certifier needs to produce analyses: the certificate checker should be able to check purported analyses based on a declarative definition (which, moreover, is probably more basic and thus easier to trust than any algorithmic one).

The analysis type systems are sound and, in the case of distributive analysis frameworks, complete wrt. appropriate *natural semantics* on abstract properties—a reformulation of the usual semantical justification of analyses.

In this chapter, we shed further light on the type-systematic method by showing that analysis type systems are in fact *applied* versions of more *foundational* Hoare logics. These describe either the same property semantics as the type system (but without recourse to any ideas about approximations) or a more basic semantics on transition traces of a suitable kind and are therefore easier to trust. The applied formalisms are justifiable as sound wrt. the more foundational formalisms (and also their underlying semantics). This is analogous to foundational proof-carrying code [6], motivated by exactly the same idea of reducing the burden of trusting an applied formalism of certification by switching to a more foundational one. Moreover, we learn that not only can a textbook definition of an analysis be cast as a program-logic like formalism, but the same is possible for the more basic considerations that justify this definition.

These contributions are all based on the classical theory of monotone analysis frameworks and abstract interpretation [17, 18], but they demonstrate that the applied vs. foundational spectrum in proof-carrying code for safety or functional correctness carries over to certified optimisation analyses. And they also emphasise that program analyses for optimisations are just as amenable to certification in program-logic like declarative formalisms as are functional correctness and safety.

$$\frac{}{\delta[x \mapsto \mathrm{dd}][y \mapsto \delta(y) \sqcup \delta(x) \mid y \in \mathrm{FV}(a)] \succ x := a \to \delta}\ :=_{\mathrm{lvns}}$$

$$\frac{}{\delta \succ \mathsf{skip} \to \delta}\ \mathrm{skip}_{\mathrm{lvns}} \qquad \frac{\delta \succ s_0 \to \delta' \quad \delta' \succ s_1 \to \delta''}{\delta \succ s_0; s_1 \to \delta''}\ \mathrm{comp}_{\mathrm{lvns}}$$

$$\frac{\delta \succ s_t \to \delta'}{\delta[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \to \delta'}\ \mathrm{if}^{\mathrm{tt}}_{\mathrm{lvns}} \qquad \frac{\delta \succ s_f \to \delta'}{\delta[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \to \delta'}\ \mathrm{if}^{\mathrm{ff}}_{\mathrm{lvns}}$$

$$\frac{\delta \succ s_t \to \delta' \quad \delta' \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \to \delta''}{\delta[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \to \delta''}\ \mathrm{while}^{\mathrm{tt}}_{\mathrm{lvns}} \qquad \frac{}{\delta[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \to \delta}\ \mathrm{while}^{\mathrm{ff}}_{\mathrm{lvns}}$$

Figure 4.1: Natural semantics for live variables

Further, in addition to standard data-flow analyses, we also look at conditional analyses, based on a combined "standard state and abstract property" semantics, as a possible variation. We also comment briefly on type-systematic definition and justification of program optimisations and type-systematic "optimisation" of functional correctness proofs alongside programs.

For the sake of brevity and intuitiveness of exposition, we limit our discussion to live variables analysis and dead code elimination for the WHILE language, but the approach is general and applies to a variety data-flow analyses and optimisations.

In Section 4.2, we introduce the method of defining data-flow analyses as type systems on the example of live variables analysis. We justify the type system by proving it sound and complete wrt. an appropriate natural semantics on liveness states and show that analysing a program amounts to principal type inference. In Section 4.3, we show that the type system is an applied version of a Hoare logic for the same natural semantics. In Section 4.4, we define liveness of a variable as a predicate on def/use transition traces and justify the natural semantics and Hoare logic on liveness states in terms of a natural semantics and Hoare logic on def/use transition traces. In Section 4.6, we treat conditional liveness, to then proceed to a discussion of type-systematic program optimisation on the example of dead code elimination in Section 4.6. Section 4.7 reviews some related work and Section 4.8 concludes.

4.2 A natural semantics and type system for live variables

We begin by an overview of the type-systematic approach to data-flow analyses [29, 25]. We do this on the example of live variables analysis (in Sec. 4.5, we also consider a variant, conditional liveness).

Informally, a variable is said to be live on a computation path, if it has a future useful use not preceded by a definition. A useful use is a use in an expression assigned to a live variable or a use in a guard expression. (This is the strong version of liveness, in contrast to the weaker one where any use triggers liveness.)

The textbook definition of live variables analysis and its justification, however, do not proceed directly from this definition but from derived considerations. The analysis (for source programs, not for the corresponding control-flow graphs) is justified by the following non-standard semantics, which we state as a natural (i.e., big-step) semantics.

States δ are assignments of values $\{\mathrm{dd}, \mathrm{ll}\}$, $\mathrm{dd} \sqsubseteq \mathrm{ll}$, to variables, understood as "liveness states". We define $\delta \sqsubseteq \delta'$ to mean that $\delta(y) \sqsubseteq \delta'(y)$ for any $y \in \mathbf{Var}$.

The evaluations of a statement are pairs of states (a prestate and a poststate) given by the rules in Figure 4.1, the notation $\delta \succ s \to \delta'$ meaning that δ and δ' are a possible pre- and poststate for s. The notation $\delta[x \mapsto v]$ stands for updating a state δ at a variable x with a value v and we also use a similar notation for simultaneous updates.

Intuitively, this semantics runs programs backwards. For any final liveness state, we get the initial liveness states corresponding to the computation paths the program can take. For example, the assignment rule expresses that, if the lhs variable x is live in a poststate, then it is dead in the midstate (where the rhs has been evaluated but not assigned yet), because the assignment defines it, and all variables y of the

$$\frac{}{x := a : d[x \mapsto \mathrm{dd}][y \mapsto d(y) \sqcup d(x) \mid y \in \mathrm{FV}(a)] \longrightarrow d}\ {:=}_{\mathrm{lvts}}$$

$$\frac{}{\mathsf{skip} : d \longrightarrow d}\ \mathrm{skip}_{\mathrm{lvts}} \qquad \frac{s_0 : d \longrightarrow d' \quad s_1 : d' \longrightarrow d''}{s_0; s_1 : d \longrightarrow d''}\ \mathrm{comp}_{\mathrm{lvts}}$$

$$\frac{s_t : d \longrightarrow d' \quad s_f : d \longrightarrow d'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : d[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \longrightarrow d'}\ \mathrm{if}_{\mathrm{lvts}} \qquad \frac{s_t : d \longrightarrow d[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]}{\mathsf{while}\ b\ \mathsf{do}\ s_t : d[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \longrightarrow d}\ \mathrm{while}_{\mathrm{lvts}}$$

$$\frac{d \leq d_0 \quad s : d_0 \longrightarrow d_0' \quad d_0' \leq d'}{s : d \longrightarrow d'}\ \mathrm{conseq}_{\mathrm{lvts}}$$

Figure 4.2: Type system for live variables

rhs (including perhaps x as well) are live in the prestate (because they are usefully used). If x is dead in a poststate, the prestate is the same. The semantics is non-deterministic, as if- and while-statements can take multiple computation paths: liveness states fix no values for the variables.

A version that is deterministic (still in the backwards direction: for any poststate, there is exactly one prestate), the collecting semantics, is defined by

$$[\![s]\!](\delta') =_{\mathrm{df}} \bigsqcup \{\delta \mid \delta \succ s \rightarrow \delta'\}$$

The collecting semantics calculates the MOP ("meet over all paths") upper bound on the liveness prestates for a given liveness poststate.

The analysis (working with approximations) can be formulated as a type system. Types d are assignments of values from $\{\mathrm{dd}, \mathrm{ll}\}$, $\mathrm{dd} \sqsubseteq \mathrm{ll}$ to variables, just as states, but their pragmatics is different: they function as non upper bounds (over-approximations) of liveness states. The negation here is a formality that results from the analysis being backward, but validity of subtyping and typing being forward (for conformity with the standard definitions of validity; this design decision will be useful for us especially in Sec. 4.5). Ignoring this negation, the values dd and ll in types can be understood to mean "certainly dead" resp. "possibly live": a state is of a type, if all variables dead in the type are dead in the state (a variable live in the type can be both dead and live in the state).

The type system has one subtyping rule, reading

$$\frac{d' \sqsubseteq d}{d \leq d'}$$

The types of a statement are pairs of types (a pretype and a posttype): we write to $s : d \rightarrow d'$ to denote that d and d' are a possible pre- and posttype of s. The typing rules are in Figure 4.2. Note that while the assignment rule of the type system is similar to that in the semantics, the rules for if- and while-statements are different: a typing of a statement pertains to all computation paths of a statement, not just one. The while rule is similar to the while rule from standard Hoare logic by involving an invariant type. Likewise, the subsumption rule is an analogue of the consequence rule. The type system accepts all valid analyses of a program, not only the strongest one, so for the statement $s =_{\mathrm{df}}$ $\mathsf{if}\ w = 3\ \mathsf{then}\ x := y\ \mathsf{else}\ x := z$ and posttype $[w \mapsto \mathrm{dd}, x \mapsto \mathrm{ll}, y, z \mapsto \mathrm{dd}]$, both $[w \mapsto \mathrm{ll}, x \mapsto \mathrm{dd}, y, z \mapsto \mathrm{ll}]$ and $[w, x, y, z \mapsto \mathrm{ll}]$ are derivable as pretypes, but the former pretype corresponds to the strongest analysis.

To state and prove the type system adequate wrt. the semantics, we define $\delta \models d$ to mean $\delta \not\sqsubseteq d$ in agreement with the explanations above. Adequacy of subtyping ($d \leq d'$ iff d' being an upper bound of a state implies that d is also an upper bound) is trivial.

Theorem 4.2.1 (Soundness and completeness of subtyping) *$d \leq d'$ iff for any δ, $\delta \models d$ implies $\delta \models d'$ (i.e., $\delta \sqsubseteq d'$ implies $\delta \sqsubseteq d$).*

Soundness and completeness of typing ($s : d \longrightarrow d'$ iff d' being an upper bound on a poststate implies that d is an upper bound on the prestates) are proven separately.

Theorem 4.2.2 (Soundness of typing) *If $s : d \longrightarrow d'$, then, for any δ, δ' such that $\delta \succ\!s\!\rightarrow \delta'$, $\delta \models d$ implies $\delta' \models d'$ (i.e., $\delta' \sqsubseteq d'$ implies $\delta \sqsubseteq d$).*

Proof. By induction on $s : d \longrightarrow d'$ and subordinate induction on $\delta \succ\!s\!\rightarrow \delta'$ in the case $s = \mathsf{while}\ b\ \mathsf{do}\ s_t$. □

To prove completeness, we define a syntactic weakest pretype operator wpt:

$$\begin{aligned}
&\mathrm{wpt}(x := a, d') \\
&\quad =_{\mathrm{df}}\ \ d'[x \mapsto \mathrm{dd}][y \mapsto d'(y) \sqcup d'(x) \mid y \in \mathrm{FV}(a)] \\
&\mathrm{wpt}(\mathsf{skip}, d')\ =_{\mathrm{df}}\ d' \\
&\mathrm{wpt}(s_0; s_1, d')\ =_{\mathrm{df}}\ \mathrm{wpt}(s_0, \mathrm{wpt}(s_1, d')) \\
&\mathrm{wpt}(\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f, d') \\
&\quad =_{\mathrm{df}}\ \ (\mathrm{wpt}(s_t, d') \wedge \mathrm{wpt}(s_f, d'))[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \\
&\mathrm{wpt}(\mathsf{while}\ b\ \mathsf{do}\ s_t, d') \\
&\quad =_{\mathrm{df}}\ \ \nu(F)\ \text{where} \\
&\qquad\qquad F(d) =_{\mathrm{df}} (\mathrm{wpt}(s_t, d) \wedge d')[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]
\end{aligned}$$

Here ν is the greatest (wrt. our subtyping $\leq$) fixpoint operation on monotone type transformers.

The wpt operator is the type-systematic formulation of an algorithm for computing the strongest analysis, i.e., the MFP ("maximal fixpoint") upper bound on the liveness prestates for a liveness poststate.

The following lemmata show that the wpt of a given type d' is a pretype of d', in the sense of typing, and greater than any semantic pretype of d'.

Lemma 1 $s : \mathrm{wpt}(s, d') \longrightarrow d'$.

Proof. By induction on s. □

Lemma 2 *If, for any δ, δ' such that $\delta \succ\!s\!\rightarrow \delta'$, $\delta \models d$ implies $\delta' \models d'$ (i.e., $\delta' \sqsubseteq d'$ implies $\delta \sqsubseteq d$), then $d \leq \mathrm{wpt}(s, d')$ (i.e., $\mathrm{wpt}(s, d') \sqsubseteq d$).*

Proof. Also by induction on s. □

Theorem 4.2.3 (Completeness of typing) *If, for any δ, δ' such that $\delta \succ\!s\!\rightarrow \delta'$, $\delta \models d$ implies $\delta' \models d'$ (i.e., $\delta' \sqsubseteq d'$ implies $\delta \sqsubseteq d$), then $s : d \longrightarrow d'$.*

Proof. Immediate from the two lemmata by the $\mathrm{conseq}_{\mathrm{llvts}}$ rule. □

¿From soundness and completeness we get that the collecting semantics and the weakest pretype agree perfectly, i.e., MOP=MFP.

Corollary 1 $[\![s]\!](\delta') \sqsubseteq \mathrm{wpt}(s, \delta')$.

Proof. By Lemma 1 and Thm. 4.2.2, $\delta \succ\!s\!\rightarrow \delta'$ yields $\delta \sqsubseteq \mathrm{wpt}(s, \delta')$ for any liveness state δ. □

Corollary 2 $\mathrm{wpt}(s, \delta') \sqsubseteq [\![s]\!](\delta')$

Proof. By Lemma 2. □

Lemma 2 and its consequences, including completeness of typing (Thm. 4.2.3) and MFP $\sqsubseteq$ MOP (Cor. 2), depend on the fact that the transfer functions of live variables analysis are distributive (preserve meets). They do not, for instance, hold for constant propagation, which fails to be distributive.

The following weaker semantics-independent property of the type system alone does not rest on distributivity (and holds thus also for non-distributive backward analyses): the wpt of a program wrt a posttype d' is greater than any typing-sense pretype of d'. We already know that it also is a typing-sense pretype of d' (Lemma 1). In summary, the wpt is the principal pretype of d', in type systems jargon. And computing the strongest analysis is principal type inference.

$$\frac{}{\{P\}\, x := a\, \{(ls(x) = \mathrm{ll} \supset P[ls(y) \Mapsto \mathrm{ll} \mid y \in \mathrm{FV}(a)][ls(x) \Mapsto \mathrm{dd}]) \wedge (ls(x) = \mathrm{dd} \supset P)\}}\ {:=}_{\mathrm{lvhoa}}$$

$$\frac{}{\{P\}\, \mathsf{skip}\, \{P\}}\ \mathrm{skip}_{\mathrm{lvhoa}} \qquad \frac{\{P\}\, s_0\, \{Q\} \quad \{Q\}\, s_1\, \{R\}}{\{P\}\, s_0; s_1\, \{R\}}\ \mathrm{comp}_{\mathrm{lvhoa}}$$

$$\frac{\{P[ls(y) \Mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]\}\, s_t\, \{Q\} \quad \{P[ls(y) \Mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]\}\, s_f\, \{Q\}}{\{P\}\, \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f\, \{Q\}}\ \mathrm{if}_{\mathrm{lvhoa}}$$

$$\frac{\{P[ls(y) \Mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]\}\, s_t\, \{P\}}{\{P\}\, \mathsf{while}\ b\ \mathsf{do}\ s_t\, \{P[ls(y) \Mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]\}}\ \mathrm{while}_{\mathrm{lvhoa}}$$

$$\frac{P \models P_0 \quad \{P_0\}\, s\, \{Q_0\} \quad Q_0 \models Q}{\{P\}\, s\, \{Q\}}\ \mathrm{conseq}_{\mathrm{lvhoa}}$$

Figure 4.3: Hoare logic for live variables

Lemma 3 *If* $d'_0 \leq d'$, *then* $\mathrm{wpt}(s, d'_0) \leq \mathrm{wpt}(s, d')$.

Theorem 4.2.4 *If* $s : d \longrightarrow d'$, *then* $d \leq \mathrm{wpt}(s, d')$.

Proof. By induction on $s : d \longrightarrow d'$, using Lemma 3 in some cases. (For live variables, one can also go via the semantics and get the theorem as an immediate corollary of Thm. 4.2.2 and Lemma 2.) □

4.3 A Hoare logic for live variables

While the type system is a description of the semantics, it is not a very direct one: the type system really relies on both the semantics and properties of upper bounds. Nor is the type system completely expressive; the only expressible assertions are negations of upper bound conditions. A more foundational formalism, which is also expressively complete, can be obtained by recasting the liveness semantics as a Hoare logic, in complete analogy with the Hoare logic characterisation of the standard semantics.

The assertions of the Hoare logic are (generally open) formulae of the (first-order) theory of $(\{\mathrm{dd}, \mathrm{ll}\}, \sqsubseteq)$ over the signature with an extralogical constant $ls(x)$ (for the liveness value of x in the understood liveness state) for any program variable $x \in \mathbf{Var}$. The proof rules are in Figure 4.3. The notation $P[x \Mapsto a]$ denotes substituting the occurrences of x in P by a. Again the design is for the standard notion of validity, i.e., a forward implication: the precondition holding for a prestate implies that the postcondition holds for all possible poststates. Reversing the implication is possible by contraposition, i.e., by negating the two conditions.

The Hoare logic is adequate (both sound and complete) with respect to the intended interpretation, which has $[\![ls(y)]\!](\delta) =_{\mathrm{df}} \delta(y)$ (i.e., $ls(y)$ means the current liveness value of y).

Theorem 4.3.1 (Soundness) *If* $\{P\}\, s\, \{Q\}$, *then, for any* δ, δ' *such that* $\delta \succ s \rightarrow \delta'$, $\delta \models_\alpha P$ *implies* $\delta' \models_\alpha Q$ *for any valuation* α.

Lemma 4 $\{P\}\, s\, \{\mathrm{slp}(P, s)\}$ *(for a correctly defined strongest postcondition operator).*

Theorem 4.3.2 (Completeness) *If, for any* δ, δ' *such that* $\delta \succ s \rightarrow \delta'$, $\delta \models_\alpha P$ *implies* $\delta' \models_\alpha Q$ *for any valuation* α, *then* $\{P\}\, s\, \{Q\}$.

Note that since the domain of the intended interpretation of the language of assertions is a doubleton, this language is, in fact, essentially propositional. Hence the strongest postcondition operation is trivially definable. And completeness would still hold absolutely (as opposed to relatively to the level of completeness of an axiomatization of arithmetic), if we replaced the two entailment side conditions in the consequence rule with side conditions of deducibility in an appropriate proof system.

Clearly the Hoare logic is more foundational than the type system, as it formalizes the semantics directly. But this has the price that, generally, Hoare triple derivations are harder to construct than type derivations. For certification of analyses, however, this is unproblematic. Constructing a Hoare-logic derivation for a typing judgement (a purported analysis result) is no harder than constructing a type-system derivation: types admit a translation into Hoare-logic assertions and a Hoare-logic derivation of a translated typing judgement is mechanically obtainable from its type-system derivation, i.e., the translation of types extends to type derivations.

In agreement with the semantic meaning of types, a type d can be translated into the Hoare-logic assertion $[d] =_{\mathrm{df}} \mathit{ls} \not\sqsubseteq d$ (i.e., $\neg \bigwedge\{\mathit{ls}(y) \sqsubseteq d(y) \mid y \in \mathbf{Var}\}$). This translation preserves subtyping and typing.

Theorem 4.3.3 (Preservation of subtyping) *If $d \leq d'$ in the type system, then $[d] \models [d']$.*

Theorem 4.3.4 (Preservation of typing) *If $s : d \longrightarrow d'$ in the type system, then $\{[d]\}\, s\, \{[d']\}$ in the Hoare logic.*

Proof. A non-constructive indirect proof is immediate from soundness of the type system and completeness of the Hoare logic. An alternative constructive direct proof (by extending translation to type derivations) is by induction on $s : d \longrightarrow d'$. □

Of course the weakest preconditions in the Hoare logic are stronger than those in the type system. For the statement $s =_{\mathrm{df}}$ if $w = 3$ then $x := y$ else $x := z$, for instance, we have that $\mathrm{wpt}(s, [w \mapsto \mathrm{dd}, x \mapsto \mathrm{ll}, y, z \mapsto \mathrm{dd}]) = [w \mapsto \mathrm{ll}, x \mapsto \mathrm{dd}, y, z \mapsto \mathrm{ll}]$ while

$$\begin{aligned}
&\mathrm{wlp}(s, \neg(\mathit{ls}(w) = \mathrm{dd} \wedge \mathit{ls}(y) = \mathrm{dd} \wedge \mathit{ls}(z) = \mathrm{dd})) \\
&\quad = \quad \neg(\mathit{ls}(w) = \mathrm{ll} \wedge \mathit{ls}(x) = \mathrm{dd} \\
&\qquad\qquad \wedge ((\mathit{ls}(y) = \mathrm{ll} \wedge \mathit{ls}(z) = \mathrm{dd}) \vee (\mathit{ls}(z) = \mathrm{ll} \wedge \mathit{ls}(y) = \mathrm{dd})))
\end{aligned}$$

(for a poststate where w, y, z are dead, the type system can only detect that x is dead in the prestate (w, y, z can be either dead or live), while the Hoare logic knows also that w is live and that exactly one of y and z is live).

4.4 A natural semantics and Hoare logic for future defs, uses

Our discussion of live variables analysis thus far has been quite detached from the (informal) definition of liveness we recalled in Sec. 4.2. Instead we built on a semantics on liveness states, which looks very different. In fact, the definition of liveness is part of a foundation for this semantics, but we did not show this. Now we will close the gap.

The only observation needed is that liveness states are an abstraction over another, more concrete (and thus more basic) non-standard notion of states. There is nothing like liveness states or a semantics for them "in the nature". Instead, the liveness definition speaks about "computation paths", more specifically, about future definitions and uses of variables on such paths. As no more information about paths is relevant for liveness, we can take abstract paths to future transition traces, where the transitions are defs and uses.

Thus we introduce a natural semantics where states are lists of tokens D_x with $x \in \mathbf{Var}$ and U^x_V with $V \subseteq \mathbf{Var}$ and $x \in \mathbf{Var} + \{\mathit{pc}\}$. A token D_x means a definition of x. A token U^x_V means a use of the variables V for defining x. The pseudovariable pc (for "program counter") is for the case where the variables V are used to evaluate a guard. Lists of such tokens should be understood as defs and uses to take place in the future.

The evaluation rules of the semantics are in Figure 4.4. ($\cdot$ stands for both "cons" and "append". Again the causality is backward, but compared to the rules of the liveness semantics, they are completely basic. The assignment rule, for instance, tells us that the unique preagenda for a postagenda τ is $\mathrm{U}^x_{\mathrm{FV}(a)} \cdot \mathrm{D}_x \cdot \tau$, i.e., to use all variables of a, to define x and then to do τ.

$$\frac{}{\mathrm{U}^{x}_{\mathrm{FV}(a)} \cdot \mathrm{D}_x \cdot \tau \succ x := a \rightarrow \tau}\ :=_{\mathrm{lvns}} \qquad \frac{}{\tau \succ \mathsf{skip} \rightarrow \tau}\ \mathrm{skip}_{\mathrm{lvns}} \qquad \frac{\tau \succ s_0 \rightarrow \tau' \quad \tau' \succ s_1 \rightarrow \tau''}{\tau \succ s_0; s_1 \rightarrow \tau''}\ \mathrm{comp}_{\mathrm{lvns}}$$

$$\frac{\tau \succ s_t \rightarrow \tau'}{\mathrm{U}^{pc}_{\mathrm{FV}(b)} \cdot \tau \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \rightarrow \tau'}\ \mathrm{if}^{\mathrm{tt}}_{\mathrm{lvns}} \qquad \frac{\tau \succ s_f \rightarrow \tau'}{\mathrm{U}^{pc}_{\mathrm{FV}(b)} \cdot \tau \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \rightarrow \tau'}\ \mathrm{if}^{\mathrm{ff}}_{\mathrm{lvns}}$$

$$\frac{\tau \succ s_t \rightarrow \tau' \quad \tau' \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow \tau''}{\mathrm{U}^{pc}_{\mathrm{FV}(b)} \cdot \tau \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow \tau''}\ \mathrm{while}^{\mathrm{tt}}_{\mathrm{lvns}} \qquad \frac{}{\mathrm{U}^{pc}_{\mathrm{FV}(b)} \cdot \tau \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow \tau}\ \mathrm{while}^{\mathrm{ff}}_{\mathrm{lvns}}$$

Figure 4.4: Natural semantics for future def/use traces

$$\frac{}{\{P\}\, x := a\, \{P[tr \mapsto \mathrm{U}^{x}_{\mathrm{FV}(a)} \cdot \mathrm{D}_x \cdot tr]\}}\ :=_{\mathrm{lvhoa}} \qquad \frac{}{\{P\}\, \mathsf{skip}\, \{P\}}\ \mathrm{skip}_{\mathrm{lvhoa}} \qquad \frac{\{P\}\, s_0\, \{Q\} \quad \{Q\}\, s_1\, \{R\}}{\{P\}\, s_0; s_1\, \{R\}}\ \mathrm{comp}_{\mathrm{lvhoa}}$$

$$\frac{\{P[tr \mapsto \mathrm{U}^{pc}_{\mathrm{FV}(b)} \cdot tr]\}\, s_t\, \{Q\} \quad \{P[tr \mapsto \mathrm{U}^{pc}_{\mathrm{FV}(b)} \cdot tr]\}\, s_f\, \{Q\}}{\{P\}\, \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f\, \{Q\}}\ \mathrm{if}_{\mathrm{lvhoa}} \qquad \frac{\{P[tr \mapsto \mathrm{U}^{pc}_{\mathrm{FV}(b)} \cdot tr]\}\, s_t\, \{P\}}{\{P\}\, \mathsf{while}\ b\ \mathsf{do}\ s_t\, \{P[tr \mapsto \mathrm{U}^{pc}_{\mathrm{FV}(b)} \cdot tr]\}}\ \mathrm{while}_{\mathrm{lvhoa}}$$

$$\frac{P \models P_0 \quad \{P_0\}\, s\, \{Q_0\} \quad Q_0 \models Q}{\{P\}\, s\, \{Q\}}\ \mathrm{conseq}_{\mathrm{lvhoa}}$$

Figure 4.5: Hoare logic for future def/use traces

Using traces as states, it is straightforward to formalize the intuitive definition of liveness. For a trace τ, the corresponding liveness state $\mathsf{LS}(\tau)$ is defined as follows.

$$\begin{array}{ll} & \mathsf{LS}(\tau)(z) = \mathrm{ll} \\ \text{iff}_{\ \mathrm{df}} & \exists \upsilon, \tau', x, V.\, \tau = \upsilon \cdot \mathrm{U}^{x}_{V} \cdot \tau' \wedge z \in V \\ & \wedge((\exists \tau''.\, \tau' = \mathrm{D}_x \cdot \tau'' \wedge \mathsf{LS}(\tau'')(x) = \mathrm{ll}) \vee x = pc) \end{array}$$

This definition is wellformed, because it can be reorganized into the following structurally recursive one.

$$\begin{array}{rcl} \mathsf{LS}(\varepsilon)(z) & =_{\mathrm{df}} & \mathrm{dd} \\ \mathsf{LS}(\mathrm{D}_x \cdot \tau)(z) & =_{\mathrm{df}} & \begin{cases} \mathrm{dd} & \text{if } z = x \\ \mathsf{LS}(\tau)(z) & \text{otherwise} \end{cases} \\ \mathsf{LS}(\mathrm{U}^{x}_{V} \cdot \mathrm{D}_x \cdot \tau)(z) & =_{\mathrm{df}} & \begin{cases} \mathsf{LS}(\tau)(z) \sqcup \mathsf{LS}(\tau)(x) & \text{if } z \in V \\ \mathsf{LS}(\mathrm{D}_x \cdot \tau)(z) & \text{otherwise} \end{cases} \\ \mathsf{LS}(\mathrm{U}^{pc}_{V} \cdot \tau)(z) & =_{\mathrm{df}} & \begin{cases} \mathrm{ll} & \text{if } z \in V \\ \mathsf{LS}(\tau)(z) & \text{otherwise} \end{cases} \end{array}$$

The abstraction function puts the semantics of liveness states into perfect agreement with the trace semantics. Hence it is right to say that the trace semantics and the abstraction function are a foundation for the liveness state semantics.

Theorem 4.4.1 *If* $\tau \succ s \rightarrow \tau'$*, then* $\mathsf{LS}(\tau) \succ s \rightarrow \mathsf{LS}(\tau')$*.*

Theorem 4.4.2 *If* $\delta \succ s \rightarrow \mathsf{LS}(\tau')$*, then there is a trace* τ *such that* $\tau \succ s \rightarrow \tau'$ *and* $\mathsf{LS}(\tau) = \delta$*.*

Similarly to the liveness state semantics, the trace semantics is also characterisable by a Hoare logic, by a completely analogous design. The assertions are (open) formulae of the (first-order) theory of lists of use, def tokens over the signature with an extralogical constant tr for the current def-use future. The inference rules are in Figure 4.5.

Again the logic is sound and complete for the obvious intended interpretation of the assertions with $[\![tr]\!](\tau) =_{\mathrm{df}} \tau$ (i.e., tr denotes the current trace). But completeness is only relative to the completeness level of an axiomatization of the theory of lists, if the entailments in the side conditions of the consequence rule are replaced by deducibilities: the incompleteness of axiomatisations of arithmetic applies.

Theorem 4.4.3 (Soundness) *If $\{P\}\, s\, \{Q\}$, then, for any τ, τ' such that $\tau \succ s \rightarrow \tau'$, $\tau \models_\alpha P$ implies $\tau' \models_\alpha Q$ for any valuation α.*

Theorem 4.4.4 (Completeness) *If, for any τ, τ' such that $\tau \succ s \rightarrow \tau'$, $\tau \models_\alpha P$ implies $\tau' \models_\alpha Q$ for any valuation α, then $\{P\}\, s\, \{Q\}$.*

Just as constructing proofs in the Hoare logic for liveness in general was harder than constructing type derivations in the type system for liveness, constructing proofs in the Hoare logic for traces is even harder, generally. But again, if we have a proof for a triple in the Hoare logic for liveness, the more foundational proof in the Hoare logic for traces is obtainable mechanically. The assertions of the Hoare logic for liveness can be translated into ones of the Hoare logic for traces and the translation extends to derivations. Define LS to be a syntactic version of LS (so $[\![LS(t)]\!](\tau) = \mathsf{LS}([\![t]\!](\tau))$. An assertion P about the current liveness state is naturally translated as the assertion $[P] =_{\mathrm{df}} P[ls \Mapsto LS(tr)]$ about the current trace. This translation preserves derivable triples.

Theorem 4.4.5 (Preservation of derivable Hoare triples) *If $\{P\}\, s\, \{Q\}$ in the Hoare logic for live variables, then $\{[P]\}\, s\, \{[Q]\}$ in the Hoare logic for traces.*

Proof. A non-constructive indirect proof is immediate from soundness of the Hoare logic for live variables, Thm. 4.4.1 and completeness of the Hoare logic for traces. An alternative constructive direct proof is by induction on $\{P\}\, s\, \{Q\}$. □

For analyses other than live variables analysis, the notion of a trace considered need not be suitable. For available expressions, for instance, it suffices to keep track of past evaluations and modifications of non-trivial expressions on a computation path. A more universal notion would record all past and future "atomic actions" (which are assignments and evaluations of guards).

4.5 Conditional liveness

Having outlined the foundational spectrum of certification for live variables analysis, we now sketch a variant, much to illustrate the flexibility of our setup. Namely, we look at conditional liveness à la Strom and Yellin [34]. This dwells on the same definition of liveness on a computation path as before, but the states in the underlying concrete semantics are pairs of a store (standard state) and a computation path, so only these transitions between computation paths are considered that are physically possible. Accordingly, the analysis is finer.

The analogue to the semantics in Sec. 4.2 has as states pairs (σ, δ) of stores (assignments of integers to variables) and liveness states. The evaluation rules are in Figure 4.6.

The type system defining the analysis has as types d assignments to variables of assertions of the standard Hoare logic (i.e., arithmetic formulae over a signature with an extralogical constant x for any variable x). There is one subtyping rule

$$\frac{d' \models d}{d \leq d'}$$

The typing rules are in Figure 4.7. In these rules, we have written $d' \models d$ to mean that $d'(y) \models d(y)$ for all $y \in \mathbf{Var}$, $d[x \Mapsto a]$ to mean $[y \mapsto d(y)[x \Mapsto a] \mid y \in \mathbf{Var}]$, $b \supset d$ to mean $[y \mapsto b \supset d(y) \mid y \in \mathbf{Var}]$ and $d_t \wedge d_f$ to mean $[y \mapsto d_t(y) \wedge d_f(y) \mid y \in \mathbf{Var}]$.

Adequacy (soundness and completeness) of the type system wrt. the semantics holds wrt. the intended interpretation of types, which is: $\sigma, \delta \models d$ iff it is <u>not</u> the case that, for all variables $y \in \mathbf{Var}$, $\delta(y) = \mathrm{ll}$ implies $\sigma \models d(y)$ (i.e., $d(y)$ is a necessary condition for y being live). (Again the negation is formal: the analysis is backward, but the validity notion definition is forward, hence the need for contraposition.)

$$\frac{}{\sigma, \delta[x \mapsto \mathrm{dd}][y \mapsto \delta(y) \sqcup \delta(x) \mid y \in \mathrm{FV}(a)] \succ x := a \rightarrow \sigma[x \mapsto [\![a]\!]\sigma], \delta} :=_{\mathrm{lvns}}$$

$$\frac{}{\sigma, \delta \succ \mathsf{skip} \rightarrow \sigma, \delta}\ \mathrm{skip}_{\mathrm{lvns}} \qquad \frac{\sigma, \delta \succ s_0 \rightarrow \sigma', \delta' \quad \sigma', \delta' \succ s_1 \rightarrow \sigma', \delta''}{\sigma, \delta \succ s_0; s_1 \rightarrow \sigma'', \delta''}\ \mathrm{comp}_{\mathrm{lvns}}$$

$$\frac{\sigma \models b \quad \sigma, \delta \succ s_t \rightarrow \sigma', \delta'}{\sigma, \delta[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \rightarrow \sigma', \delta'}\ \mathrm{if}^{\mathrm{tt}}_{\mathrm{lvns}} \qquad \frac{\sigma \not\models b \quad \sigma, \delta \succ s_f \rightarrow \sigma', \delta'}{\sigma, \delta[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \rightarrow \sigma', \delta'}\ \mathrm{if}^{\mathrm{ff}}_{\mathrm{lvns}}$$

$$\frac{\sigma \models b \quad \sigma, \delta \succ s_t \rightarrow \sigma', \delta' \quad \sigma', \delta' \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow \sigma', \delta''}{\sigma, \delta[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow \sigma'', \delta''}\ \mathrm{while}^{\mathrm{tt}}_{\mathrm{lvns}} \qquad \frac{\sigma \not\models b}{\sigma, \delta[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow \sigma, \delta}\ \mathrm{while}^{\mathrm{ff}}_{\mathrm{lvns}}$$

Figure 4.6: Natural semantics for conditional liveness

$$\frac{}{x := a : (d[x \mapsto \bot][y \mapsto d(y) \vee d(x) \mid y \in \mathrm{FV}(a)])[x \Mapsto a] \longrightarrow d} :=_{\mathrm{lvts}}$$

$$\frac{}{\mathsf{skip} : d \longrightarrow d}\ \mathrm{skip}_{\mathrm{lvts}} \qquad \frac{s_0 : d \longrightarrow d' \quad s_1 : d' \longrightarrow d''}{s_0; s_1 : d \longrightarrow d''}\ \mathrm{comp}_{\mathrm{lvts}}$$

$$\frac{s_t : b \supset d \longrightarrow d' \quad s_f : \neg b \supset d \longrightarrow d'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : d[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \longrightarrow d'}\ \mathrm{if}_{\mathrm{lvts}} \qquad \frac{s_t : b \supset d \longrightarrow d[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]}{\mathsf{while}\ b\ \mathsf{do}\ s_t : d[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \longrightarrow \neg b \supset d}\ \mathrm{while}_{\mathrm{lvts}}$$

$$\frac{d \leq d_0 \quad s : d_0 \longrightarrow d_0' \quad d_0' \leq d'}{s : d \longrightarrow d'}\ \mathrm{conseq}_{\mathrm{lvts}}$$

Figure 4.7: Type system for conditional liveness

The strongest analysis algorithm is described by the weakest pretype (wpt) operator defined as follows:

$$\begin{array}{l}
\mathrm{wpt}(x := a, d') \\
\quad =_{\mathrm{df}} \quad (d'[x \mapsto \bot][y \mapsto d'(y) \vee d'(x) \mid y \in \mathrm{FV}(a)])[x \Mapsto a] \\
\mathrm{wpt}(\mathsf{skip}, d') =_{\mathrm{df}} d' \\
\mathrm{wpt}(s_0; s_1, d') =_{\mathrm{df}} \mathrm{wpt}(s_0, \mathrm{wpt}(s_1, d')) \\
\mathrm{wpt}(\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f, d') \\
\quad =_{\mathrm{df}} \quad ((b \supset \mathrm{wpt}(s_t, d')) \wedge (\neg b \supset \mathrm{wpt}(s_f, d'))) \\
\qquad\qquad\qquad\qquad\qquad [y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \\
\mathrm{wpt}(\mathsf{while}\ b\ \mathsf{do}\ s_t, d') \\
\quad =_{\mathrm{df}} \quad \nu(F)\ \text{where} \\
\qquad\qquad F(d) =_{\mathrm{df}} ((b \supset \mathrm{wpt}(s_t, d)) \wedge (\neg b \supset d')) \\
\qquad\qquad\qquad\qquad\qquad [y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]
\end{array}$$

Type derivations in this type system can be translated into proofs in a Hoare logic. The assertions P are formulae of the two-sorted theory of both the integers and the liveness domain $(\{\mathrm{dd}, \mathrm{ll}\}, \sqsubseteq)$ over the signature with an integer constant y and liveness constant $ls(y)$ for any variable y. The proof rules are in Figure 4.8. A type d is translated as dictated by the interpretation of types, namely by the assertion $\neg \bigwedge\{ls(y) = \mathrm{ll} \supset d(y) \mid y \in \mathbf{Var}\}$.

We refrain from spelling out the semantics and Hoare logic for store and transition trace pairs.

4.6 Dead code elimination

As an example of a data-flow analysis based optimisation, let us look at dead code elimination. This optimisation removes assignments to dead variables. Type-systematically, it is straightforwardly defined by

$$\frac{}{\{P[x \mapsto a]\}\, x := a\, \{(ls(x) = \mathrm{ll} \supset P[ls(y) \Mapsto \mathrm{ll} \mid y \in \mathrm{FV}(a)][ls(x) \Mapsto \mathrm{dd}]) \wedge (ls(x) = \mathrm{dd} \supset P)\}} :=_{\mathrm{lvhoa}}$$

$$\frac{}{\{P\}\, \mathsf{skip}\, \{P\}}\ \mathrm{skip}_{\mathrm{lvhoa}} \qquad \frac{\{P\}\, s_0\, \{Q\} \quad \{Q\}\, s_1\, \{R\}}{\{P\}\, s_0; s_1\, \{R\}}\ \mathrm{comp}_{\mathrm{lvhoa}}$$

$$\frac{\{b \wedge P[ls(y) \Mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]\}\, s_t\, \{Q\} \quad \{\neg b \wedge P[ls(y) \Mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]\}\, s_f\, \{Q\}}{\{P\}\, \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f\, \{Q\}}\ \mathrm{if}_{\mathrm{lvhoa}}$$

$$\frac{\{b \wedge P[ls(y) \Mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]\}\, s_t\, \{P\}}{\{P\}\, \mathsf{while}\ b\ \mathsf{do}\ s_t\, \{\neg b \wedge P[ls(y) \Mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]\}}\ \mathrm{while}_{\mathrm{lvhoa}}$$

$$\frac{P \models P_0 \quad \{P_0\}\, s\, \{Q_0\} \quad Q_0 \models Q}{\{P\}\, s\, \{Q\}}\ \mathrm{conseq}_{\mathrm{lvhoa}}$$

Figure 4.8: Hoare logic for conditional liveness

$$\frac{d(x) = \mathrm{ll}}{x := a : d[x \mapsto \mathrm{dd}][y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(a)] \longrightarrow d \hookrightarrow x := a} :=^1_{\mathrm{lvts}} \qquad \frac{d(x) = \mathrm{dd}}{x := a : d \longrightarrow d \hookrightarrow \mathsf{skip}} :=^2_{\mathrm{lvts}}$$

$$\frac{}{\mathsf{skip} : d \longrightarrow d \hookrightarrow \mathsf{skip}}\ \mathrm{skip}_{\mathrm{lvts}} \qquad \frac{s_0 : d \longrightarrow d' \hookrightarrow s_0' \quad s_1 : d' \longrightarrow d'' \hookrightarrow s_1'}{s_0; s_1 : d \longrightarrow d'' \hookrightarrow s_0'; s_1'}\ \mathrm{comp}_{\mathrm{lvts}}$$

$$\frac{s_t : d \longrightarrow d' \hookrightarrow s_t' \quad s_f : d \longrightarrow d' \hookrightarrow s_f'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : d[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \longrightarrow d' \hookrightarrow \mathsf{if}\ b\ \mathsf{then}\ s_t'\ \mathsf{else}\ s_f'}\ \mathrm{if}_{\mathrm{lvts}}$$

$$\frac{s_t : d \longrightarrow d[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \hookrightarrow s_t'}{\mathsf{while}\ b\ \mathsf{do}\ s_t : d[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \longrightarrow d \hookrightarrow \mathsf{while}\ b\ \mathsf{do}\ s_t'}\ \mathrm{while}_{\mathrm{lvts}}$$

$$\frac{d \le d_0 \quad s : d_0 \longrightarrow d_0' \hookrightarrow s' \quad d_0' \le d'}{s : d \longrightarrow d' \hookrightarrow s'}\ \mathrm{conseq}_{\mathrm{lvts}}$$

Figure 4.9: Type system for dead code elimination

a transformational add-on to the analysis type system of Fig. 4.2. The rules of the extended type system are in Fig. 4.9. For any type assigned to a program, this also assigns a transformed program.

This type-systematic definition of the optimisation allows for a simple proof of relational soundness of the analysis. Defining $\sigma \sim_d \sigma'$ to denote that $\sigma(x) = \sigma'(x)$ for any $x \in \mathbf{Var}$ such that $d(x) = \mathrm{ll}$ (i.e., that two states agree on all variables live in a type), the optimisation is sound in the following sense: If $s : d \to d' \hookrightarrow s_*$ and $\sigma \sim_d \sigma_*$, then (i) $\sigma \succ\!s\!\rightarrow \sigma'$ implies that there exists σ_*' such that $\sigma' \sim_{d'} \sigma_*'$ and $\sigma_* \succ\!s_*\!\rightarrow \sigma_*'$, (ii) $\sigma_* \succ\!s_*\!\rightarrow \sigma_*'$ implies that there exists σ' such that $\sigma' \sim_{d'} \sigma_*'$ and $\sigma \succ\!s\!\rightarrow \sigma'$. The proof is by induction on $s : d \to d' \hookrightarrow s_*$. Moreover, the type-derivation based program transformation can be extended to a transformation of functional correctness proofs. For details, see [12, 31].

Conditional liveness analysis facilitates more refined dead code elimination optimisation. For assignment, one could, for instance, choose to give these rules:

$$\frac{d(x) \not\models \bot}{\begin{array}{l} x := a : \\ (d[x \mapsto \bot][y \mapsto d(y) \vee d(x) \mid y \in \mathrm{FV}(a)])[x \Mapsto a] \longrightarrow d \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \hookrightarrow x := a \end{array}}$$

$$\frac{d(x) \models \bot}{x := a : d[x \Mapsto a] \longrightarrow d \hookrightarrow \mathsf{skip}}$$

A more aggressive optimisation could be built on conditional liveness information together with conditional constant propagation information.

Consider this example (b does not contain z, y and x; the pretypes for all constituent statements have been computed from the global posttype on line 8).

```
1  $[z \mapsto b, y \mapsto \bot, x \mapsto \bot]$     $x := 5;$
2  $[z \mapsto b, y \mapsto \bot, x \mapsto \bot]$     if $b$ then
3  $[z \mapsto b, y \mapsto \bot, x \mapsto \neg b]$       $y := x$
                                                         else
4  $[z \mapsto b, y \mapsto \neg b]$                     skip;
5  $[z \mapsto b, y \mapsto \neg b]$                   if $b$ then
6  $[z \mapsto \top]$                                      $z := z + 1$
                                                         else
7  $[z \mapsto \bot, y \mapsto \top]$                      $z := y$
8  $[z \mapsto \top]$
```

Because of path sensitivity, the assignment to x on line 1 is eliminated by the rules above, since x is necessarily dead after it. The assignment to y on live 3 is conditionally live and could be removed on the basis of additional forward analysis information that it is liveness condition $\neg b$ cannot obtain after it.

Alternative designs would optimise partially dead assignments by code motion, e.g., move an assignment preceding an if into one branch of the if-statement.

4.7 Related work

We can only mention some items of related work. Use of type systems to define program analyses is an old idea, especially in the form of enrichments of standard type systems ("annotated types", especially for functional languages). The type systems philosophy is also central in the "flow logic" work of Nielsen and Nielsen [29]. Type-systematic accounts of classical data-flow analyses for optimisations (incl. soundness of optimisations and optimisation of functional correctness proofs, cf. translation validation) for imperative languages appear in [12, 25, 31]. Volpano et al.'s well-known type system [36] for secure information flow is of the same kind, but relatively weak (based on invariant state types instead of pre- and poststate types).

The idea of characterising non-standard semantics with program logics is old as well. For imperative languages, it appears already, e.g., in Andrews and Reitman's Hoare logic for secure information flow [5] and H. R. Nielson's Hoare logic for computation time [28]. Denney and Fischer [20] have characterized safety policies with Hoare logics and applied these to certification of safety.

The conditional data-flow analyses à la Strom and Yellin [34] are an extension of Strom and Yemini's typestate checking paradigm [35], originally meant to address basic safety.

Proof-carrying code was invented by Necula and Lee [27], who certified safety of programs against a verification condition generator. Foundationalism in PCC was pioneered by Appel [6], who suggested using a universal logic formalisation of the underlying semantics instead. Hamid, Zhao et al. [22] proposed that a foundational certificate can consist in a type derivation and a formal soundness proof of the type system.

Formal certification of data-flow analyses, especially for Java bytecode, is the subject of a number of recent works. Albert et al.'s analysis-carrying code [2] highlights that analysis results can serve as analysis certificates allowing for lightweight checking (without fixpoint re-computation, only fixpoint checking). Besson, Jensen et al. [14] have taken a more foundational approach, certifying also soundness of analysis algorithms and address the issue of minimising certificate size. Also foundational is the work by Beringer, Hofmann et al. [13] on certified heap consumption analysis based on a type system specialising a program logic.

4.8 Conclusions

We have shown that classical data-flow analyses, such as live variables analysis, can be certified on a variety of levels, completely analogous to certification of program safety or functional correctness. To accept a typing derivation in an analysis type system as a certificate of a computed analysis, one must believe in the textbook definition of the analysis (this is what the type system formalises) or in a justification of this definition. To accept a derivation in the corresponding Hoare logic for abstract properties, it suffices to trust the definition of the abstract property semantics (which is also the justification for the type system, but only together with additional ideas about approximations). Finally, to accept a derivation in the Hoare logic for future def/use traces, even this is not necessary: one must only believe in the trivial definition of the trace semantics and in the definition of the abstraction of traces into liveness states. How much prerequisite trust is required depends on the application, but a formalism is available for each level.

In the case of more foundational formalisms, one can ask what makes better certificates: direct derivations in the foundational formalism (obtainable from derivations in an applied formalism by translation) or derivations in an applied formalism accompanied by a proof of soundness of the applied formalism wrt. the foundational one. It is also an option to avoid trusting any program logics by relying instead on the descriptions of their underlying program semantics and universal (meta)logic. The program logics and program semantics we have presented here support the full spectrum of foundationalism.

Chapter 5

Road-Map for integrating the results into the MOBIUS framework and future work

In this section we sketch shortly the next steps to be realised by this working task. In the immediate future we will integrate the results achieved so far in the MOBIUS framework and tool suite. In detail this includes:

1. extending the presented translations and embeddings to treat all core features of the Java language supported by MOBIUS. This includes object-oriented features like presence of an object heap, method invocations or exceptions.

2. reformulation of the presented embeddings and translations in terms of the formalism developed and used in MOBIUS. In particular in terms of Bicolano or the MOBIUS base logic instead of the partially used external formalisms.

3. formalisation and implementation of the theoretic results within the suitable tools, like the Coq formalisation of the supported logics. This step depends on the completion of a modular extension framework for Bicolano and the base logic. This extension framework is currently discussed and developed in Task 3.2. The framework will allow to add support for concepts like ghost variables in a transparent manner. Transparent means that adding/removing or availability/absence of an extensions affects *only* those parts of the system that depend on this particular extension.

4. defining a clear interface to verification condition/proof obligation generators and fixing the information required to be provided by the user in order to perform the required analysis. This may result in suggestions for extensions of the bytecode modelling language BML.

5. furthermore, possible impacts on the general format of certificates have to be examined in order to be able to generate certificates from the found proofs and to achieve the desired uniform representation.

The above listed tasks, but especially the first three ones are scheduled for the immediate future. But their is further work to be done in order to improve the practical use of the presented approaches.

An important milestone to be reached is to provide support for more elaborated security policies. For information flow analyses the focus lies on the comprehensive treatment of declassification. Therefore we plan to go systematically through the classification of these properties listed in [33]. We aim to develop and provide support for most of the different mentioned property classes and to point out in detail, when certain declassification properties cannot be treated in the suggested framework.

Last but not least, the developed approaches have to be analysed and evaluated with respect to scalability and stability issues. These kinds of analyses are expected to answer questions about the growth of required annotations, proofs and certificates—both in size and run-time—in relation to the program size as well as the impact caused e.g. by changing a few lines of code. An admirable property would be that local changes remain local and will not cause numerous proofs to be redone.

In order to gain empirical results how the approaches behave in practise case studies will be performed. The case studies shall help to weight the theoretically gained insights in scalability and stability. This is necessary as often theoretical problems might not be of practical relevance as the problematic cases either occur only in rare cases and/or can be circumvented easily, e.g. by following or preventing certain programming patterns. In addition, we expect from these case studies further feedback about strengthen and weaknesses of the approaches as well as to shed light on potential optimisations.

Chapter 6

Conclusions

In this deliverable we gave an intermediate report of the current results achieved in the first months. We presented two approaches that allow to embed type-based analysis for secure information flow into a logical framework in detail. The presented approaches are complete wrt. to the original type-based systems, but allow more programs to be proven secure than the original type systems. Further, the higher precision and expressiveness allows to express more elaborated security policies within the same formalism. Further, some first proofs show that the complexity for performing these analysis within the logical framework are not significantly harder as long as the program to be checked could be proven secure in the type-based system. The impact on growth of proofs, certificates and required program annotations differs between the suggested approaches and has been analysed and quantified in the near future. The results will be important for insights and improvements of the approaches concerning stability and scalability.

In addition a foundational certification for data-flow analysis has been presented, which provides a foundational ground for type-based analysis and their representation in a Hoare style logic. The link between type-based system and foundational logics allow further insights on how to ensure soundness of program optimisations and allows to automatically compile functional correctness proofs of the original program into proofs for the optimised version.

Bibliography

[1] W. Ahrendt, Th. Baar, B. Beckert, R. Bubel, M. Giese, R. Hï¿$\frac{1}{2}$nle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 2004. To appear.

[2] E. Albert, G. Puebla, and M. V. Hermenegildo. Abstraction-carrying code. In *Logic for Programming Artificial Intelligence and Reasoning*, volume 3452 of *Lecture Notes in Computer Science*, pages 380–397. Springer-Verlag, 2005.

[3] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *Principles of Programming Languages*, pages 91–102. Association of Computing Machinery Press, 2006.

[4] T. Amtoft and A. Banerjee. Information flow analysis in logical form. In R. Giacobazzi, editor, *Static Analysis Symposium*, volume 3148 of *Lecture Notes in Computer Science*, pages 100–115. Springer-Verlag, 2004.

[5] G.R. Andrews and R.P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, 1980.

[6] A. W. Appel. Foundational proof-carrying code. In J. Halpern, editor, *Logic in Computer Science*, page 247. IEEE Press, June 2001. Invited Talk.

[7] A. Banerjee and D. Naumann. Secure information flow and pointer confinement in a java-like language. In *Computer Security Foundations Workshop*, page 253. IEEE Computer Society, 2002.

[8] G. Barthe, D. Naumann, and T. Rezk. Deriving an information flow checker and certifying compiler for Java. In *Symposium on Security and Privacy*. IEEE Press, 2006.

[9] G. Barthe, D. Pichardie, and T. Rezk. Non-interference for low level languages. Technical report, INRIA, 2006.

[10] P. D'Argenio G. Barthe and T. Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Computer Security Foundations Workshop*, pages 100–114. IEEE Press, 2004.

[11] B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security*, volume 2041 of *Lecture Notes in Computer Science*, pages 6–24. Springer-Verlag, 2001.

[12] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Principles of Programming Languages*, pages 14–25. Association of Computing Machinery Press, 2004.

[13] L. Beringer, Martin Hofmann, Alberto Momigliano, and Olha Shkaravska. Automatic certification of heap consumption. In *Logic for Programming Artificial Intelligence and Reasoning*, volume 3452, pages 347–362. Springer-Verlag, 2005.

[14] F. Besson, T. Jensen, and D. Pichardie. Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. *Theoretical Computer Science*, 2006. Extended version of [**?**], to appear.

[15] Mobius Consortium. Deliverable 3.1: Bytecode specification language and program logic. Available online from `http://mobius.inria.fr`, 2006.

[16] Coq development team. The Coq proof assistant reference manual V8.0. Technical Report 255, INRIA, France, mars 2004. `http://coq.inria.fr/doc/main.html`.

[17] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.

[18] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages*, pages 269–282, 1979.

[19] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In R. Gorrieri, editor, *Workshop on Issues in the Theory of Security.* IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003.

[20] E. Denney and B. Fischer. Correctness of source-level safety policies. In *Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 894–913. Springer-Verlag, 2003.

[21] R. Hähnle, J. Pan, P. Rümmer, and D. Walter. Integration of a security type system into a program logic, 2007.

[22] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. *Journal of Automated Reasoning*, 31(3–4):191–229, 2003.

[23] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic.* Foundations of Computing. MIT Press, October 2000.

[24] S. Hunt and D. Sands. On flow-sensitive security types. In *Principles of Programming Languages*, Charleston, South Carolina, USA, January 2006. Association of Computing Machinery Press.

[25] P. Laud, T. Uustalu, and V. Vene. Type systems equivalent to data-flow analyses for imperative languages. *Theoretical Computer Science*, 364(3):292–310, 2006.

[26] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Principles of Programming Languages*, pages 228–241. Association of Computing Machinery Press, 1999. Ongoing development at `http://www.cs.cornell.edu/jif/`.

[27] G.C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. Association of Computing Machinery Press.

[28] H.R. Nielson. A hoare-like proof system for analysing the computation time of programs. *Science of Computer Programming*, 9:107–136, 1987.

[29] H.R. Nielson and F. Nielson. Flow logic: A multi-paradigmatic approach to static analysis. In *The Essence of Computation, Complexity, Analysis, Transformation*, volume 2566 of *Lecture Notes in Computer Science*, pages 223–244. Springer-Verlag, 2002.

[30] P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *Logic for Programming Artificial Intelligence and Reasoning*, volume 4246 of *Lecture Notes in Computer Science*, pages 422–436. Springer-Verlag, 2006.

[31] A. Saabas and T. Uustalu. Program and proof optimizations with type systems. Submitted, 2006.

[32] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21:5–19, January 2003.

[33] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *IEEE Computer Security Foundations Workshop*, pages 255–269. IEEE Press, 2005.

[34] R.E. Strom and D.M. Yellin. Extended typestate checking using conditional liveness analysis. *IEEE Transactions on Software Engineering*, 19(5):487–485, 1993.

[35] R.E. Strom and S. Yemini. Typestate: a programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[36] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.