Mobius

Information Society Technologies

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 D3.8

Report on embedding type-based analyses into program logics

Due date of deliverable:2009-03-01 (T0+42)

Actual submission date:

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

Organisation name of lead contractor for this deliverable: CTH

Final version

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	1,2,8
IoC	6,7
LMU	3
UEDIN	5
WU	4

This document has been written by Lennart Beringer (LMU), Aleksy Schubert (WU), Ian Stark (UEDIN), Tarmo Uustalu (IoC) and Richard Bubel (CTH).

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

This document summarises deliverable D3.8 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`.

In this deliverable we report on the work done in Task 3.5 on embedding type-based analyses into program logics to reason about specific program properties.

Several techniques exist to ensure that a program behaves according to a given specification. Amongst the most popular ones are type-based and program logic based approaches.

Type-based approaches are typically specialised towards certain domains, e.g. information-flow or resource properties, on which they are highly efficient and fully automatic. But they classify too many actually secure resp. safe programs as insecure or resource restraints violating, producing false warnings/errors.

On the other side, program logic based approaches are complementary in the sense that they usually allow to express a wide range of properties including functional correctness of programs. The drawback is that they require severe human interaction and experts performing the correctness proofs.

The ultimate goal is to combine both approaches to benefit from their complementary strengths and minimizing the effect of their disadvantages. In a proof-carrying code environment the integration of both approaches in a common notation allows also to produce uniform certificates. The participating sites took different approaches to reach the given goal:

- CTH reports in Chapt. 2 on their work on embedding type-based (information-flow) analyses into a program logic using abstraction-based techniques. Being able to classify correctly more programs as secure and achieving the same automation level as type-based analyses.

- LMU reports in Chapt. 3 on a derivation system for proving non-interference for a subset of sequential bytecode. The derivation system provides several additional characteristics to previously existing type systems like applicability to unstructured bytecode or tracking of copies.

- WU reports in Chapt. 4 about design and implementation of their prototype tool which generates bytecode modelling language (BML) annotations necessary to encode the type-based security policy framework [4].

- UEDIN reports in Chapt. 5 on their work on integrating resource aware type-based systems into a program logic. The proposed program logic has been formalised and proved correct with help of a proof assistant.

- IoC reports in Chapt. 6 on a type-systematic approach to data-flow analyses and optimizations to cover bidirectional analyses and in Chapt. 7 on a semantics-based approach to sound automatic program repair and reasoning about such program transformations.

Contents

1 Introduction 5

2 Abstraction-Based Reasoning 7

3 Relational bytecode correlations 9

4 An implementation of the translation from a type system to BML 11
- 4.1 Specification of security policies 11
  - 4.1.1 Policies to specify 11
  - 4.1.2 Representation of‘ security policies 12
- 4.2 Implementation issues 14
- 4.3 The result of the translation 15

5 Strict Hoare Logic for Procedures 17
- 5.1 Hoare Logic and procedures 17
- 5.2 Types and procedures 18
- 5.3 Strict Hoare semantics 19
- 5.4 Isabelle formalization 20

6 Type-systematic description of bidirectional data-flow analyses 21

7 Program repair as sound optimization of broken programs 23

8 Conclusions 25

Chapter 1

Introduction

Analysing the reasons why and when program analyses are adopted (resp. why their adoption fails) in real-world software development reveals besides others the following two crucial points:

- the analyses must be automatic and efficient. In the ideal case they run in incremental mode directly in the integrated development environment.
- the analyses must be reasonably precise, i.e. emitting only a low number of false warnings.

Both aspects coincide peculiarly with the strength resp. weakness of type-based and program logic based approaches used in a number of analyses. Type-based approaches are typically used to ensure specialised properties of programs as, for example, secure information-flow or obedience to resource restrictions. On these domains they are highly efficient and fully automatic. On the downside they classify too many actually secure resp. safe programs as insecure or resource restraints violating, producing false warnings/errors.

Program logic based approaches are complementary in the sense that they allow usually to express a wide range of properties including functional correctness of programs and reach almost the theoretically possible level of completeness. The drawback is that they require severe human interaction and experts to perform the correctness proofs.

To meet the above mentioned necessary prerequisites for real-world adoption, we investigate how to embed type-based analyses into program logics to combine their complementary strengths by minimizing the effect of their respective disadvantages. In a proof-carrying code environment the integration of both approaches in a common notation allows also to produce uniform certificates.

The participating sites took different approaches to reach the given goal. As application scenarios served information-flow and resource analyses as well as general foundational aspects building also the theoretical basis for optimising, proof-preserving compilers and automated program repair.

The deliverable is organised as follows:

- In Chapt. 2 CTH reports on their efforts to encode different dimension of declassification into a program logic. In order to achieve full automation, they suggest an abstraction-based program logic. The proposed logic framework allows to perform lazy abstraction on the fly while symbolically executing the program under consideration. Lazy abstraction allows to maintain high precision as longs as possible. Abstraction is only performed when otherwise human interaction would be needed e.g. to provide a loop invariant. The abstraction is further restricted to parts of the state leaving untouched regions precise.
- In Chapt. 3 LMU reports on a derivation system for proving non-interference for a subset of sequential bytecode. The reported approach integrates ideas from relational program logics with separation, value abstraction and copy propagation. The application area covered by the derivation system concerns termination-insensitive non-interference in its end-to-end interpretation. The derivation system allows to track copies of values through objects and methods, is applicable for unstructured bytecode and

supports amongst others heap-local reasoning. The proposed approach has been formalised in a proof assistant.

- In Chapt. 4 WU reports on their encoding of the type-based security policy framework [4]. Starting from a specified policy security, the required program specification in terms of the bytecode modelling language (BML) is generated and annotated along the program.

  The report describes in particular the design of the prototype tool implementation producing the required BML code annotations for the encoding. The implemented tool is part of the MOBIUS program verification environment.

- In Chapt. 5 UEDIN reports on their work on integrating resource aware type-based systems into a program logic. A crucial insight of their work is that derivations in type-based systems prove a stronger property than classical Hoare triple, namely that the precondition is required to hold. Based on that observation, a stronger Hoare-like program logic is proposed, which allows to translate a derivation of the type-based system into a proof in the program logic in a more straight-forward manner. The proposed program logic has been formalised and proved correct with help of a proof assistant.

- In Chapt. 6 IoC reports on the extension of their prior work on a type-systematic approach to data-flow analyses and optimizations to cover bidirectional analyses. They evaluated their approach on two examples one for checking absence of type errors in a programming language where the type of a variable can change during the program run. The second example is a stack optimisation problem, namely load-pop pair eliminations, ensuring that the operand stack stays balanced.

- In Chapt. 7 IoC reports on a semantics-based approach to sound automatic program repair (including enforcing of coding conventions) and reasoning about such program transformations. This an applications of the techniques they originally developed for reasoning about program optimizations.

Chapter 2

Abstraction-Based Reasoning

Integrating type-based systems and program logics provides synergies of their respective strengthens levelling out some of their disadvantages. Type-based systems are highly efficient and completely automatic, but also incomplete and reject many programs which in fact satisfy the property to be proved. In an information-flow security based setting that means for instance, that they reject many secure programs as insecure. Program logics on the other side are usually complete and model a program languages semantics faithfully, but require severe interaction.

Aiming at the integration of type-based systems into a program logic, we concentrated in particular on information-flow analyses as application scenario. The focus was two-fold, on one-side we concentrated on how to express advanced information-flow properties in a program logic and on the other side on techniques allowing to achieve the automation of type-based systems by simultaneously maintaining a higher precision. The first two contributions concentrate on expressing a number of declassification dimensions in a program logic, while the last two papers describe necessary techniques to achieve full automation.

In the previous deliverable, we reported already about the development of a specialised program logic able to simulate the type-based system proposed by Hunt and Sands in [20]. The type-based system was able to check the non-interference property. We showed that type derivations of the type system could be translated into proofs of the program logic. The possibility of this translation indicated that it is principally not harder to find a non-interference proof in the program logic than in the type based system. On the other hand the program logic is able to prove more programs secure than the type-based system. In this deliverable we refer to an extended journal version of the paper [19] including a chapter about an extension to cover declassification of certain values.

In the survey paper [34] the authors Sabelfeld and Sands categorise a number of declassification approaches along several dimensions. In [11] we report about how to encode several of these declassification properties into a program logic covering aspects of the

- what dimension, i.e. which kind of information is allowed to be released to the public like for example the average of incomes of employers, if an entered password was wrong or right etc.
- when dimension, i.e. how to express conditions under which a certain release is allowed, for instance, a movie maybe download only when the payment has been received.
- where dimension, i.e. denoting explicitly the position in the program from which on a certain information may be released.

Our main contribution is presented in the paper [12]. Type-based systems provide a fixed level of abstraction encoded in the definition of their types, i.e. the relevant information is tracked in the types and all other information discarded. The impressive degree of automation obtained using type-based system is a direct consequence thereof.

We report about a logic framework for abstract interpretation of symbolic execution. The principle idea is to symbolically execute a program precisely and to resort to abstraction only when necessary. For instance,

to find sufficient loop invariants automatically without querying the user. Fig. 2.1 illustrates the principle process. The state change induced by a program during symbolic execution is computed incrementally and captured as updates to the initial state. The next symbolic execution step is then executed wrt. to the updated state. Instead of performing an immediate abstraction of the resulting state resp. state update (left side of the figure) we postpone the abstraction as long as possible. The approach is complementary

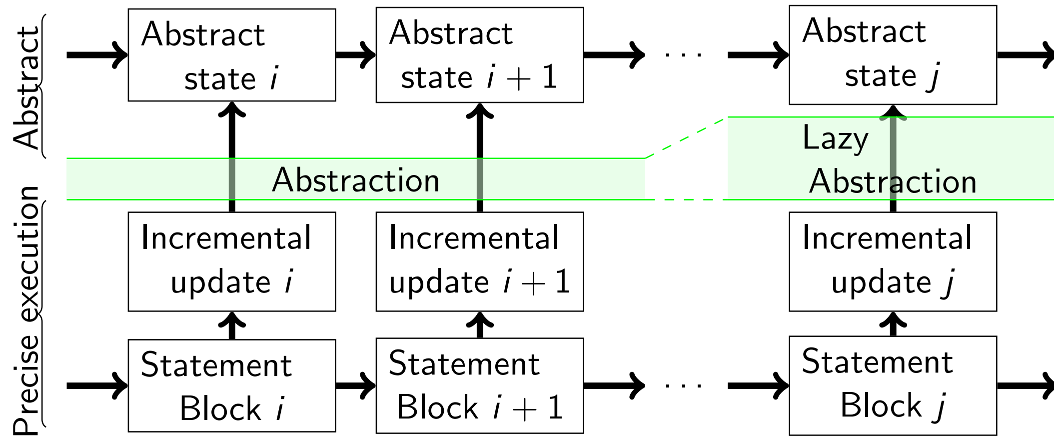

Figure 2.1: Lazy abstraction in symbolic execution

to other abstract interpretation approaches which usually start from the most abstract domain and refine stepwise if the abstraction was to coarse. The advantage of our approach is that we start from a faithful and complete modelling of the program language semantic keeping precise record of the actual symbolic state and lose precision only at designated points in a lazy manner. Lazy abstraction means that we delay the computation of an abstraction as long as possible and, then, only abstract a limited part of the symbolic state.

Finally, the paper describes how to extend the program logic to keep book about variable dependencies of locations. The book-keeping facility is then used to encode secure information-flow properties. We show further how the abstraction framework extends to the variable dependency aware program logic and allows to automatically prove more programs secure than typical type-based approaches.

Our last contribution is a technical paper [10] on how to represent dependencies on locations of state dependent predicates and functions in an efficient way. The presented approach allows to encode such dependencies directly in the syntax of logic symbols. It provides a concise syntax to express dependencies of single locations and arbitrary sets of heap locations, i.e. object or array fields. These symbols provide a useful tool and reduce the need of interaction. Furthermore, they are necessary to formalise correctness conditions of rules in [12] in presence of heap locations.

Chapter 3

Relational bytecode correlations

We present a derivation system for proving non-interference (short: NI) for an idealized subset of sequential bytecode. We integrate ideas from relational program logics with separation, value abstraction and copy propagation. Our derivation system concerns termination-insensitive non-interference in its end-to-end interpretation. This means that two programs are already deemed equivalent if either of the two fails to terminate, and that we are only interested in initial and final, but not intermediate, states. Compared to previous type systems, it exhibits the following characteristics:

Descriptive, extensional formulation of NI Deviating from approaches inspired by Volpano et al.'s high-level type system [?], neither code regions nor variables or fields are a priori associated with fixed security levels. Instead, the correlated occurrences of values in states of competing executions is captured by a notion of relational state abstractions that generalizes the partial bijections on heap addresses used by Banerjee-Naumann [3] and Barthe et al. [4]. Instead of the distinction between high and low code regions, a dichotomy is introduced between correlated and uncorrelated events, where events include allocations, arithmetic operations, method invocations, and conditionals. Roughly speaking, the treatment of correlated events generalizes traditional low proof rules, and the treatment of uncorrelated events resembles high proof rules.

Applicability to unstructured bytecode No calculation of control dependence regions is required, and the soundness of derivations holds independently of additional control flow properties. Occurrences of field or variable assignments and method invocations inside branches are admitted in a substantially more flexible way than in previous systems.

Tracking of copies, through methods and objects The system includes the statically approximative tracking of copies of values through the components of states, including their flow through objects and methods. In accordance with the previous items, this includes the flow of - using the traditional terminology - private values though public objects, fields, or methods.

Heap-local reasoning The interpretation of state abstractions regarding objects is defined in a separating fashion, enabling the fine-grained description of sharing and non-sharing between heap regions. In accordance with this, our system includes frame rules similar to that of separation logic.

Abstract representation predicates Exploiting the separation structure, we introduce abstract representation predicates. These enable the specification of high-level data structures according to their heap layout and the formulation of complex non-interference properties. In particular, our formalism allows us to specify high-level recursive data types where the privacy of content data may vary at different positions. We illustrate this feature by specifying lists containing private (uncorrelated) and public (correlated) content elements in an alternating fashion.

Object-level confinement of heap effects Contrary to systems that delimit the effect of methods on objects along the axis of visibility, our technology allows a fine-grained control which ensures that objects

that are not reachable from the method arguments remain unchanged. This property is essential for heap-local reasoning as described above. In particular, this property enables the verification of recursive methods over heap representations of inductive high-level data structures: the methods can be given a specification which ensures that environmental objects whose connectivity is essential for the well-structuredness of data structures remain unchanged. We illustrate this feature by verifying a method for copying lists, and prove that the above-mentioned alternation pattern is preserved.

A main ingredient of our calculus is a notion of abstract states that expresses when values held in different storage locations of a single state are guaranteed to coincide, or are guaranteed to differ. Coincidence of values is tracked for integer as well as reference values, definite separation only for reference values only. Each value is identified by a corresponding color, such that abstract states have a similar form as concrete states but map storage locations to colors instead of concrete values.

Pairs of abstract states Σ and Σ' are combined with certain administrative components N to relational state descriptions (short: RSD's) $\phi = (\Sigma, N, \Sigma')$. These generalize the partial bijections mentioned above as the occurrence of identical colors in Σ and Σ' expresses the indistinguishability of the associated concrete values. RSD's subsequently play the role of types in our calculus: our main judgment form is of the form $G \vdash \ell \sim \ell' : \phi \to \psi$ where ℓ and ℓ' are program labels, ϕ and ψ are RSD's, and G is a polyvariant relational proof context with entries of the form $((\ell_i, \ell'_i), (\phi_i, \psi_i))$. An intuitive reading of such a judgment is as follows. Consider executions of the phrases at ℓ and ℓ' from initial states s and s' that jointly respect RSD ϕ. Here, "respect" in particular means that s abstracts to Σ, and s' to Σ'. Next, suppose that both executions terminate. Then, the terminal states t and t' are required to respect RSD ψ. Our derivation system thus exposes the two-program nature of non-interference explicitly at the level of judgments, extending ideas from the relational program logics introduced by Benton [5], Yang [36] and Amtoft et al. [1]. However, proof checking is computationally easier than in these general-purpose logics as the only side conditions of the proof rules concern freshness and equality conditions over the set of colors.

The proof system contains rules for correlated events and uncorrelated events. The former class includes pairs of allocations, pairs of conditionals, or pairs of method invocations, the latter single allocations, single conditionals, or single method invocations. Rules for correlated events affect both program components of a judgment. The rules for uncorrelated events only affect one of the two phrases, and are isolated as a stand-alone system of unary proof rules. In particular, this subsystem treats all those instruction forms that merely transfer existing values between different components of states, such as load/store, getfield/putfield, and simple operand stack instructions swap/dup/pop etc..

Tracking the flow of colors through methods is enabled by requiring that colors retain their interpretation throughout a judgment. Post-RSD's may contain additional colors, in case the program phrases allocate objects. The abstract states of final RSD's may also or lack some integer colors that existed in the abstract states of the pre-RSD, in case all occurrences of a color are overwritten. However, all colors that occur in the abstract states of both RSD's must be interpreted identically. Moreover, object colors must be preserved, as no garbage collection is modeled.

In order to enable the justification of the frame rules, we formally employ a Kripke-style extension of the above-mentioned intuitive interpretation of judgments: a judgment must in fact be obeyed with respect to all pre/post-RSD's that arise from the RSD's mentioned explicitly in the judgment by arbitrary separated heap extensions. A similar technique was used by Birkedal and Yang to formally justify higher-order separation logics, i.e. the extension of ordinary separation logics with recursive methods [9].

Our work is supported by a formalization in a proof assistant. The formalization contains soundness proofs for all proof rules with respect to an operational semantics, and formal proof derivations for selected example programs. The latter part includes the proof for certifying the non-interference of the list-copying routine, for lists of arbitrary length with elements of alternating visibility.

The paper [7] has been submitted to a journal.

Chapter 4

An implementation of the translation from a type system to BML

A translation from a type system which guarantees the non-interference has a few desirable features. First of all, once the translation is in place it is possible to use a toolset based on logical methods rather than typed ones. This allows to incorporate the guarantees of the type system into a foundational proof-carrying code (PCC, [2]) framework and use the non-interference property together with other properties originally formulated and expressed in the foundational fashion[1]. Moreover, the wide selection of JML based verification tools and methods [13] and a growing set of BML based ones [35, 18] is a solid basis to aim for a platform of foundational PCC certificates for Java bytecode.

Another desirable feature of this translation is the fact that the resulting model of non-interference is more flexible than the one based on typing. This is important whenever the non-interference property must be weakened, for example when declassification is needed (in particular when the code encrypts confidential data). The translation we provide is designed so that it is relatively straightforward to adjust it to various declassification needs.

We present a prototype implementation of a translation from the information-flow type system from [4] to BML. This prototype is based on an earlier work on the translation reported in [24]. This chapter summarises the main design issues associated with the development of the tool and presents an example which shows the result of the translation.

Throughout the text we use an example which illustrates the presented concepts. The example is based on a simple class in Figure 4.1.

4.1 Specification of security policies

The first main issue associated with the implementation is the way security policies should be specified. We present the policies system which is used for the work and then provide the way of describing it in bytecode files.

4.1.1 Policies to specify

We use the security policy framework from [4]. It is based on the assumption that the attacker can observe the input/output of methods only. This, however, is extended to the values of fields and heaps as otherwise it is difficult to guarantee statically the non-interference property. We assume also that the attacker is unable to observe the termination of the programs.

[1] The translation considered in this work does not reduce the trusted logical base to the one of the foundational PCC. To achieve that one has to link the resulting formulae with the non-interference property expressed in the foundational logic e.g. the property expressed in [6] for while programs.

```
public class Levels {

        private int h;

        public int m() {
                return h;
        }
}
```

Figure 4.1: Levels class

Let us first present briefly the security policies. Formally, a security policy is expressed in terms of a finite partial order $(\mathcal{S}, \leq)$. This order allows to describe 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 fields, local variables and return values whose level is less or equal than 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 $\Gamma_{N(m)}[k]$. This signature gives the security policy of the method m called on an object of the level k. The set of security signatures for a method m is defined as $\mathsf{Policies}_\Gamma(m) = \{\Gamma_{N(m)}[k] \mid k \in \mathcal{S}\}$. The security signature has the shape $\vec{k_p} \xrightarrow{k_h} \vec{k_r}$:
  - The vector $\vec{k_p}$ 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.
  - The value k_h describes the lower bound in the security levels of the heap operations performed by the method.
  - The vector $\vec{k_r}$ 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 the class of an exception that might be thrown in the method and k_{e_i} is the upper bound on the security level of the exception. We use the notation $k_r[n]$ and $k_r[e_i]$ for k_n and k_{e_i}.

4.1.2 Representation of‘ security policies

Security policies must be represented in relation to bytecode programs. Therefore, a method of specifying them must be provided. As the starting point for that we take the textual notation of the bytecode files which is presented in [14]. This format is extended to provide the definition of security policies. We do not provide the representation of the policies in the binary format as the textual format is enough for the purposes of the translation.[2]

Security policies are specified in special comments of the form `/*! ... !*/`. The comments contain the elements of the security policy described above. These annotations will be added to bytecode files in the textual representation. A bytecode textual representation for the class in Figure 4.1 is presented in Figure 4.2.

[2] In fact, the result of the translation to BML can be regarded as a class file binary representation of the security policies and the typings in the information-flow type system.

```
package [default]

Constant pool:
  const #1 = Class #2;
  ... other constants ...

class Levels

private int h;

public void <init>()
0:     aload_0
1:     invokespecial      java.lang.Object.<init> ()V (10)
4:     return

public int m()
0:     aload_0
1:     getfield           Levels.h I (18)
4:     ireturn
```

Figure 4.2: The textual representation of the bytecode for the Levels class

First of all, we have to decide on how to define the partial order of the security levels, $\mathcal{S}$. In our translation, we assume that $\mathcal{S}$ is represented as positive numbers available in the type `int`. Thus, the lowest security level is 1 and the value 0 is reserved to mark undefined positions in various structures used in the translation. This design choice allows us to refrain from specifying the syntax to declare $\mathcal{S}$.

The next thing to be specified is the level k_{obs} visible to the attacker. We can specify it using the keyword `visibleLevel` in the following way:

```
/*! visibleLevel = 1; !*/
```

This lets k_{obs} to be equal 1. This annotation should be introduced right after the class header, `class Levels` in the case of our example.

The policy ft which assigns to each field its security level is specified as an additional field modifier containing the number that determines the level of the field. In case of the Levels class, this looks as follows:

```
private /*! 2 !*/ int h;
```

This specifies that the field h is on the level 2 of confidentiality, in case the visible level is equal to 1 this means that the field is not observable for the attacker.

The security signatures of the methods have the form $\vec{k_p}(l) \stackrel{k_h(l)}{\to} \vec{k_r}(l)$, where l is the level of the reference on which the target object resides, and are specified in a slightly more complicated way. First of all, we follow the general flavour of signatures in Java and write information pertinent to the method result right before the type of the method result and the information pertinent to the parameters right before the parameter types. In general terms this looks as follows:

```
modifiers /*! lr !*/ methodName( /*! l1 !*/ type name,....) /*! lh !*/
     throws /*! le1 !*/ exeptionType,...
```

where `lr` is the specification of the result levels, `l`n for $n > 0$ are the specifications of the levels for the parameters, `lh` is the lower bound in the security levels of the heap operations performed by the method, and `le`n is the specification of the levels of the particular exceptions.

As the method signatures of the form $\vec{k_p}(l) \stackrel{k_h(l)}{\to} \vec{k_r}(l)$ are parametrised by l, where l is the level of the reference on which the target object resides. Therefore, the specification of levels is a list of correspondences

of the form n_1 `->` n_2 where n_1 gives the level of the reference on which the target object resides (l) and n_2 the value of the security level of the particular part of the method signature as specified by the policy designated by n_1. The elements of the lists are separated with commas.

In case of the method m in class Levels this specification looks as follows:

```
public /*! 1->1, 2->2 !*/ int m() /*! 1->2, 2->2 !*/
```

and this represents the policy:

$$\text{level of target} = 1 : \quad \varepsilon \xrightarrow{2} 1,$$
$$\text{level of target} = 2 : \quad \varepsilon \xrightarrow{2} 2.$$

A Levels class with specifications expressed in the language above is presented in Figure 4.3. Note that we provide also the default security policy for the default constructor which is invisible in the source code in Figure 4.1. The levels of the target object are understood here as the levels of the object which is created by the constructor.

```
package [default]

Constant pool:
  const #1 = Class #2;
  ... other constants ...

class Levels

/*! visibleLevel = 1; !*/

private /*! 2 !*/ int h;

public void <init>() /*! 1->1, 2->2 !*/
0:      aload_0
1:      invokespecial      java.lang.Object.<init> ()V (10)
4:      return

public /*! 1->1, 2->2 !*/ int m() /*! 1->2, 2->2 !*/
0:      aload_0
1:      getfield           Levels.h I (18)
4:      ireturn
```

Figure 4.3: The textual representation of the bytecode for the Levels class

4.2 Implementation issues

Parsing of the specifications is done with help of the ANTLR parser generator and is integrated with BmlLib implementation [35].

One of the more complicated issues in the implementation is the generation of the additional structures necessary to make the translation. These structures include:

- maxEx the number of all exception types in P,
- maxNbArg the maximal number of arguments of all methods in P,
- lm the maximal label of the method m,
- maxStack the maximal height of the stack in the execution of method m,

- nbLocs which for a method name $N(m)$ returns the number of its local variables.

as well as:

- excAnalysis which for a method name $N(m)$ returns the set of exception classes that are possibly thrown by m,
- classAnalysis which for a program point returns the set of exception classes of exceptions that may be thrown at the program point,
- Handler which for a given point i in the method m and an exception e returns the point where the handler of the exception starts,
- a cdr structure ($\texttt{region}_m, \texttt{jun}_m$) the role of which is to arrange the program into compact parts for which the analysis of program invariants can be conducted separately,
- st the abstract representation of the stack,
- se the security environment.

The data mentioned in the first enumeration above can be easily extracted from standard Java verification software. In our implementation we rely on the verifier JustIce which is a part of BCEL library that is used in BmlLib. Note that the labels in actual methods are not successive natural numbers, but the translation needs the successive numbers. This issue must be taken into account in the implementation. The current implementation implicitly transforms from the actual labels to instruction numbers used in ghost arrays employed for the translation.

The generation of the other data requires an adaptation of the verification library. The values of excAnalysis and classAnalysis require an inter-method analysis. The current implementation relies on the method signatures. This information is, however, imprecise and usually does not include the information on the runtime exceptions thrown by a particular method. Therefore, we plan to extend the analysis to take into account the JML specifications and to use intra-method analysis to generate the precise information.

The generation of the Handler, st, se and cdr is a typical intra-method dataflow analysis problem for which we adapt the class `Pass3bVerifier` from JustIce iterating through the list of the instructions until the data stabilises.

4.3 The result of the translation

The result of the translation for the example presented in Figure 4.3 is on Figure 4.4. The figure presents the result of the translation for the method m. For the sake of clarity, we omit on the figure the declarations of the ghost arrays employed in the formulae. The formulae which are presented on the figure seem to be complicated, but in fact the formulae fall within a rather simple class of $\forall$ fragment of the first-order logic and the theorem provers should have no problem in establishing their validity. Therefore, it should be no problem in discharge the result of the translation in a mechanical way.

Each typing rule of the original system is transformed to a formula checked right before the instruction instance it concerns. Note that it is possible in BML to change the state of the ghost variables by means of the local `set` instructions. This enables an easy method to declassify information by means of the assignment to a ghost variable. For example the declassification should occur when a value on a high security level on the stack at a program point i should be returned as a low level value. In our example, this restriction is checked in the formulae from the assert preceding the `4: ireturn` instruction. We can insert a `set` instruction before the assert which changes the value of `gsgn_m[1][gnbLocs_m + 2]` to a higher level and in this way suppress the check. This action, however, requires supplementing all the methods of the program by adding set instructions at the beginning which transform back the entry in the array to its original value. In this way we obtain a clear declassification management mechanism — declassification is present when the `set` instructions manipulate the mentioned above arrays.

```
public /*! 1->1, 2->2 !*/ int m() /*! 1->2, 2->2 !*/
/*@ assert  //N(0)(1)
  @   (forall int e, j; 0 <= e && e <= maxEx && 0 <= j && j < 3;
  @     //i |->^0 j
  @     ((e == 0 && j == 1) ==>
  @              // Reg^push,1(0)
  @             (forall int p; 0 <= p && p <= cntr; gst[1][0][p] == gst[1][1][p]) &&
  @             gst[1][1][cntr + 1] == gse[1][0] &&
  @             (forall int p; cntr + 1 < p && p <= maxStack; gst[s][1][p] == 0)
  @     )
  @ &&  //N(0)(2)
  @   (forall int e, j; 0 <= e && e <= 0 && 0 <= j && j < 3;
  @     ((e == 0 && j == 1) ==>
  @              // Reg^push,2(0)
  @             (forall int p; 0 <= p && p <= cntr; gst[2][0][p] == gst[2][1][p]) &&
  @             gst[2][1][cntr + 1] == gse[2][0] &&
  @             (forall int p; cntr + 1 < p && p <= maxStack; gst[2][1][p] == 0)
  @     )
  @*/
0:    aload_0
/*@ assert //N(1)(1)
  @   (forall int e, j; 0 <= e && e <= 0 && 0 <= j && j < 3;
  @     //i |->^0 j
  @     ((e == 0 && j == 2) ==>
  @              // Reg^getfield,1(h)
  @             (forall int p; 0 <= p && p < cntr; gst[1][1][p] == gst[1][2][p]) &&
  @             (gst[1][1][cntr] >= gse[1][1] && gst[1][1][cntr] >= gft_h ==>
  @              gst[1][2][cntr] == gst[1][1][cntr]) &&
  @             (gft_h >= gse[1][1] && gft_h >= gst[1][1][cntr]  ==>
  @              gst[1][2][cntr] == gft_h) &&
  @             (gse[1][1] >= gst[1][1][cntr] && gse[1][1] >= gft_h  ==>
  @              gst[1][2][cntr] == gse[1][1]) &&
  @             (forall int p; cntr + 1 < p && p <= maxStack; gst[s][2][p] == 0)
  @     )
  @ && //N(1)(2)
  @   (forall int e, j; 0 <= e && e <= 0 && 0 <= j && j < 3;
  @     ((e == 0 && j == 2) ==>
  @              // Reg^getfield,2(h)
  @             (forall int p; 0 <= p && p < cntr; gst[2][1][p] == gst[2][2][p]) &&
  @             (gst[2][1][cntr] >= gse[2][1] && gst[2][1][cntr] >= gft_h ==>
  @              gst[2][2][cntr] == gst[2][1][cntr]) &&
  @             (gft_h >= gse[2][1] && gft_h >= gst[2][1][cntr]  ==>
  @              gst[2][2][cntr] == gft_h) &&
  @             (gse[2][1] >= gst[2][1][cntr] && gse[2][1] >= gft_h  ==>
  @              gst[2][2][cntr] == gse[2][1]) &&
  @             (forall int p; cntr + 1 < p && p <= maxStack; gst[2][2][p] == 0)
  @     )
  @   )
  @*/
1:    getfield Levels.h I (18)
/*@ assert //N(2)(1)
  @   (forall int e, j; 0 <= e && e <= 0 && 0 <= j && j < 3;
  @     //i |->^0
  @     ((e == 0) ==>
  @              // Reg^return,1
  @             (gst[1][2][cntr] <= gsgn_m[1][gnbLocs_m + 2]) &&
  @             (gse[1][2] <= gsgn_m[1][gnbLocs_m + 2])
  @     )
  @   )
  @ && //N(2)(2)
  @   (forall int e, j; 0 <= e && e <= 0 && 0 <= j && j < 3;
  @     ((e == 0) ==>
  @              // Reg^return,2
  @             (gst[2][2][cntr] <= gsgn_m[2][gnbLocs_m + 2]) &&
  @             (gse[2][2] <= gsgn_m[2][gnbLocs_m + 2])
  @     )
  @   )
  @*/
4:    ireturn
```

Figure 4.4: The result of the translation for the method m

Chapter 5

Strict Hoare Logic for Procedures

In this deliverable, UEDIN reports on work done towards encoding type-based analyses of Java source code into a program logic for Java bytecode. Our intended analyses are, for example, range constraints on method invocation, or dependent types like vector<n>; with target a Hoare logic such as the Mobius base logic [8, 23]. A description is given in a preliminary note from early in the task:

> I. Stark. J Minor. Draft note on proposed programming language analysis.

This sets out a small Java-like programming language J Minor, a minimal virtual machine to execute it, and a bytecode logic.

This initial investigation, however, revealed an unexpected but fundamental misalignment between the components involved: type systems; Hoare logic; and the generation of verification conditions from program annotations. This mismatch prevents the faithful representation in Hoare logic of properties known to hold from the program source.

More significantly, it also exposed a gap between existing formalizations of Hoare logic and program verification tools. Our work in this task has been towards rectifying this, by adapting Hoare logic to capture a stricter contract-style semantics. This is a new direction, unanticipated in the original task proposal, and we give here a brief review of the advances made so far. A summary slide, from material presented at a Dagstuhl workshop and the Mobius annual meeting in Munich, appears in Figure 5.1

5.1 Hoare Logic and procedures

The classic interpretation of a Hoare assertion $\models \{P\}\ c\ \{Q\}$ is that for any terminating execution of program code c, if property P holds before, then Q is true afterwards. Note that this interpretation considers only start and final states (not the progress between them) and is hypothetical (i.e., property P is not required to hold initially, it only says what should follow if it is).

The simplest rule of consequence for this interpretation is that from $\vdash \{P'\}\ c\ \{Q'\}$ we can deduce $\vdash \{P\}\ c\ \{Q\}$ if $(P \Rightarrow P')$ and $(Q' \Rightarrow Q)$. This is sound, and for a straightforward while language it is also complete: everything valid can be derived.

For a language with defined procedures (or functions, or methods), things are more complex. This was highlighted by various authors, beginning with Morris and Olderog. Nipkow gives a review of the history in [27, §5.2]. The simple consequence rule remains sound but is no longer complete. However the following rule

$$\text{CONSEQ} \qquad \frac{\vdash \{P'\}\ c\ \{Q'\}}{\vdash \{P\}\ c\ \{Q\}} \quad \forall s,t\ .\ (\forall z.P'(z,s) \Rightarrow Q'(z,t)) \implies (\forall z.P(z,s) \Rightarrow Q(z,t))$$

taken from Nipkow's definitive Isabelle formalization of Hoare Logic is complete. He proves its completeness in [27]. The additional parameter z accounts for auxiliary variables, following Kleymann. Note the nested implication in the side condition. Alternative presentations are possible: where a language has procedures,

Strict Hoare Semantics

Classical Hoare Logic is *hypothetical*: to derive $\{\phi\}\ P\ \{\psi\} \vdash \{\phi'\}\ C\ \{\psi'\}$ we only use that **if** ϕ holds before calling procedure P, then ψ holds after.

Type derivations capture a stricter property: deriving $F : A \to B \vdash M : T$ requires that every application of F **must** be to a value of type A.

This has particular force with refined type systems, where types capture additional information about values such as sign or range restrictions.

To correctly encode such types into a Hoare system like the Mobius base logic, we propose a *strict* semantics, where procedure declarations include preconditions that must be satisfied on entry.

Such a contract-style semantics invalidates existing Hoare rules: notably, the side condition on the rule of consequence must be strengthened.

We have formalized the strict semantics in Isabelle, building on Nipkow's existing presentation of Hoare logic for procedures. We propose a modified rule set, and have proved its completeness.

Figure 5.1: Summary from talks at Mobius meetings

the nested implication may appear in the procedure invocation rule instead. This is the case in the Mobius Base Logic, which is similarly complete [8, §3.2 $\mathsf{Post}_{\mathsf{sinv}}$ and §3.3 InvS].

Let $\{P\}$ *body* $\{Q\}$ denote a specification of a procedure body. A key consequence of this rule is that, if some other code invokes the procedure then we need not ensure that P is satisfied on invocation: just that when it is, then Q will hold on return. If a caller has no need of the postcondition, it needs not to meet the precondition. This reflects the extensional nature of Hoare logic, noted above: Hoare triples make statements only about the initial and final states of execution, not what course it takes internally.

In contrast, the generation of verification conditions from program annotations often uses a stricter rule, requiring that every call to a procedure must satisfy its precondition. This corresponds to the simplest rule described earlier: it is sound (and therefore safe), but overly conservative. For example, the Mobius bytecode VCgen does this [23, §5]. As a consequence, derivations in the Mobius logic that make full use of the more general invocation rule may give rise to proofs that cannot be validated by the bytecode VCgen: not that the verification conditions generated are hard to prove, but they will actually be false.

5.2 Types and procedures

When classic Hoare semantics is what we want, the incompleteness of the simple consequence rule is just an aside to be noted. A problem arises, though, if we really do wish to enforce preconditions strictly. For example, this happens with static type systems for procedure or method calls. If a function f is declared of type (int * int → int), or a method int M(int i,int j), then typechecking tells us that whenever f is called, or M invoked, then the arguments really are integers. In a more refined type system, with a declaration like int M(int i>0, int j:0..10), type safety guarantees that arguments always satisfy these constraints.

With the classic hypothetical Hoare semantics, direct translation will not fully capture these type guarantees; and if we use some other strict semantics instead, then the existing formalizations [27], derivation rules, and associated metatheoretic results like soundness or completeness, no longer apply.

It appears that this strict semantics underlies other settings, too. For example, it is implicit in the fault-avoiding semantics of separation logic [29], also used in Hoare Type Theory [26]. The contracts of Eiffel

clearly enforce strict preconditions. We are unsure about the position of JML, the Java Modeling Language: the requires keyword for preconditions suggests a strict interpretation, as does the following quote from the JML reference manual:

> ... verification tools, will try to prove that the assertion always holds at that program point, for every possible execution. [22, §12.3]

Similarly the JML semantics of Jacobs gives a strict interpretation [21], and tools implement that; but the JML reference manual also gives a hypothetical version:

> Suppose also that ... the precondition, P, from the requires clause, holds. [22, §9.6.2]

Whether JML demands strict preconditions, or whether verifiers simply implement it as a convenient approximation, remains unclear to us.

Our programme has been to formulate a semantics for Hoare Logic with strict preconditions, to develop a machine-checked formalization, and prove new results of soundness and completeness. This will support the full representation of rich Java type systems in a bytecode logic; and also provide sound reasoning techniques for other settings where strict preconditions are in use.

5.3 Strict Hoare semantics

The challenge in formalizing strict preconditions is to identify not only an appropriate semantic relation $\models \{P\}\ C\ \{Q\}$, but also derivation rules for a corresponding strict axiomatic relation $\vdash \{P\}\ C\ \{Q\}$, and to prove that these two correspond via soundness and completeness results.

We base our work on Nipkow's formalization of Hoare logic for a while language with a single recursive procedure [27, §5]. Our key extension is to augment procedure declarations with a *Req* state predicate, with the intention that it must be satisfied whenever the procedure is called.

To make concrete that intention we supplement Nipkow's big-step execution relation $s - c \to t$ with a relation $s \sim c \leadsto u$ for execution up to call: meaning that when started in state s, code c can run to a point where it is about to make a procedure call and the current state is u. This properly extends the reach of standard partial Hoare semantics in two ways: it exposes internal states of running code; and it examines the behaviour of nonterminating program runs. Note that $s \sim c \leadsto u$ may be one-to-many, as any run of c might cause multiple, possibly recursive, procedure calls; although as Nipkow's semantics is anyway nondeterministic, this introduces nothing new.

We can now introduce a strict notion of validity for Hoare triples:

$$\begin{aligned} \models_s \{P\}\ c\ \{Q\} \stackrel{def}{\iff} & \forall s,t\ .\ (s - c \to t) \implies (\forall z.P(z,s) \Rightarrow Q(z,t)) \\ & \&\ \forall s,u.(s \sim c \leadsto u) \implies (\forall z.P(z,s) \Rightarrow Req(u)) \end{aligned}$$

The first line here is exactly the classic definition, as in Nipkow; the second line adds the constraint of checking *Req* on all procedure calls.

The standard rules for Hoare reasoning, formulated in Nipkow [27, §5.2], are no longer correct for this relation: for example, classically the triple $\vdash \{true\}\ CALL\ \{true\}$ is always derivable, but the triple $\models_s \{true\}\ CALL\ \{true\}$ is false for any nontrivial *Req*.

Instead, we propose variant rules for procedure call and consequence:

S-CALL
$$\frac{(P, CALL, Q) \vdash_s \{P\}\ body\ \{Q\}}{\vdash_s \{P\}\ CALL\ \{Q\}} \qquad \forall s,z\ .\ P(z,s) \Rightarrow Req(s)$$

S-CONSEQ
$$\frac{C \vdash_s \{P'\}\ c\ \{Q'\}}{C \vdash_s \{P\}\ c\ \{Q\}} \qquad \begin{aligned} &\forall s,t\ .\ (\forall z.P'(z,s) \Rightarrow Q'(z,t)) \implies (\forall z.P(z,s) \Rightarrow Q(z,t)) \\ &\&\ ((\forall s.(\exists z.P(z,s)) \Rightarrow (\exists z'.P'(z',s))) \lor (\forall u.Req(u))) \end{aligned}$$

The rules themselves are the same as usual: the difference is in the side conditions. The condition on (S-CALL) states that *Req* must be satisfied on procedure call. That for (S-CONSEQ) takes the existing

condition from Nipkow in the first line and adds a precondition strengthening that P must imply P', except if it happens that *Req* is constantly true.

Interestingly, the precondition strengthening on (S-CONSEQ) is exactly the same as that given by Nipkow for total correctness [27, §5.4]. Of course, total correctness also requires other changes to (WHILE) and (CALL) rules; and our strict relation makes no statements about termination or non-termination.

We conjecture that the novel combination of strict validity $\models_s$ and strict derivation rules $\vdash_s$ correctly captures contract-style requirements for a while language with recursive procedures.

5.4 Isabelle formalization

We have extended Nipkow's Isabelle development to include two distinct notions of validity:

- Nonstrict $\models_{ns} \{P\}\ c\ \{Q\}$. Classic Hoare, preconditions are purely hypothetical, with no obligation to satisfy them on procedure entry.
- Strict $\models_s \{P\}\ c\ \{Q\}$. Contract-style, procedure precondition must be satisfied on entry throughout all terminating and nonterminating runs.

These are matched by two distinct rule sets:

- Nonstrict $\vdash_{ns} \{P\}\ c\ \{Q\}$. Classic Hoare, with the most powerful consequence rule.
- Strict $\vdash_s \{P\}\ c\ \{Q\}$. Contract Hoare, with a strict procedure call rule and the more limited consequence rule.

Nipkow already formalized a proof that $\vdash_{ns}$ is sound and complete of for $\models_{ns}$. We add:

- Proper containment: $\models_s \subsetneq \models_{ns}$ (proved)
- Completeness $\models_s \subseteq \vdash_s$ (proved using an adaptation of Nipkow's most general triples)
- Soundness $\vdash_s \subseteq \models_s$ (proof in progress)

We also have an intermediate relation of partial strictness, which treats only terminating runs and lies properly between $\models_s$ and $\models_{ns}$. We have proved that the strict rules are sound for partial strictness but not complete, with a concrete counterexample.

Chapter 6

Type-systematic description of bidirectional data-flow analyses

In [32, 31, 33, 30] (reported in [25]), we have previously proposed a type-systematic method for describing (unidirectional) data-flow analyses and optimizations based on such analyses. Crucially, the method emphasizes general valid analyses over strongest analyses, as the problem of deciding whether some analysis is valid amounts to type-derivation checking while that of computing the strongest analysis becoming principal type inference. As a major benefit, this design allows for simple soundness and improvement arguments for analyses and optimizations so described, based on similarity-relational type semantics, and also for mechanical transformability of Hoare logic proofs of programs.

In [16] (reported in [24], the first deliverable of Work Package 3.5) we showed that the type systems arising in this manner can be viewed as applied versions of more foundational Hoare logics for reasoning about the abstract property semantics underlying the analyses described. Such Hoare logics allow for precise reasoning about the property semantics (as opposed to the approximate reasoning offer by the type systems) as well as certification of analyses and optimizations in formalisms that are more basic and therefore more easier to trust than the more applied type systems as well as more universal. In addition type systems, applied versions of these Hoare logics include also hybrid formalisms (lying between type systems and Hoare logics) describing conditional analyses.

Although cascades of unidirectional analyses are enough for most analysis tasks, there do exist important analyses that do not fit into this standard framework because of their inherent bidirectionality. Such analyses have mostly been studied by Khedker and Dhamdhere, who have argued that they are not unnatural or complicated conceptually, provided one looks at them the right way, and not necessarily expensive to implement. However the main emphasis of their work has been on algorithmic descriptions that are based on transfer functions and focus on the notion of the strongest analysis of a program. By and large, these descriptions are silent about general valid analyses.

In the new article [17], which we report here, we extend our previous work to cover bidirectional data-flow analyses.

Our contribution is the following. We take our type-systematic account of unidirectional analyses and transfer it to the bidirectional case for structured (high-level) and unstructured (low-level) languages. Specifically, for both language flavors, we formulate a schematic type system and principal type inference algorithm and show them to agree. As a side result, we show a correspondence between declarative pre/post-relations and algorithm-oriented transfer functions. Crucially, differently from unidirectional analyses, principal type inference does not mean computing the weakest pretype of a program for a given posttype, since any choice of a pretype will constrain the range of possible valid posttypes and can exclude the given one. Instead, the right generalizing notion is the weakest pre-/posttype pair for a given pre-/posttype pair. This is the greatest valid pre-/posttype pair pointwise smaller than the given one (which need not be valid for the given program).

Further, we demonstrate the similarity-relational interpretation of types in action, by showing a schematic

soundness proof and explaining how mechanical transformability of Hoare logic proofs arises from this (meta-level) semantic argument.

As examples we use type inference (seen as a data-flow problem), for a structured language where a variables type can change over a program's run but type-errors are unwanted, and a stack usage optimization, namely load-pop pairs elimination, for a low-level language manipulating an operand stack. Both of these analyses are inherently bidirectional and their soundness leads to meaningful transformations of program proofs. In the first example, bidirectionality is imposed by our choice of the analysis domain (the inferred type of a variable can be either definite or unconstrained) and the notion of validity. In the second example, it is unavoidable for deeper reasons.

The load-pop pairs elimination example comes from our earlier paper [31], where we treated several stack usage optimizations. Here we elaborate this account and put it on a solid type-system-theoretic basis, discussing, in particular, type-derivation checking vs. principal type inference.

We conclude that focusing in the first place on general valid analyses rather than strongest analyses has clear benefits in terms of the transparency of the resulting theory of bidirectional data-flow analyses and its applicability in PCC, where we need to certify analyses or transform program correctness certificates along program optimizations.

Chapter 7

Program repair as sound optimization of broken programs

In the new submitted manuscript [15], we extend the type-systematic approach to program optimizations to program repair.

Programmers make mistakes. More often than not, these mistakes manifest themselves in run-time errors that are not caught and lead to abortion. Compilers can be made to detect the possibility of such mistakes by program safety analyses, but they cannot correct them because they cannot know the intended, non-erroneous meaning. Thus it may look as if program repair (i.e., transforming programs with abnormal evaluations into programs with meaningful normal evaluations) is inevitably a manual activity.

We contest this view by proposing a new, semantics-based approach to program repair. Our central idea is to define the intended meaning of broken programs (i.e., programs that may abort under a given error-admitting language semantics) by a special, error-compensating semantics under which programs have no or fewer abnormal evaluations. Program repair can then become a compile-time, automatic program transformation that turns a given program into one whose evaluations under the error-admitting semantics agree with those of the given program under the error-compensating semantics. We build such transformations upon data-flow analyses devised specifically to achieve this form of semantic soundness.

Technically, and this is our second key idea, we describe repairs with their underlying analyses as type systems with a transformation component as in our previous work on program optimizations.

We present our approach on two examples: repair of file-handling programs, which may have missing or superfluous open and close statements, and repair of programs manipulating a queue that can under- and overflow. Both examples show that program repair shares many features with program optimization: both are analyses-based automatic program transformations that are sound and improving in a clear mathematical sense (we use a similarity-relational semantics of types). What is more, while a repair repairs broken programs, it would typically also optimize already safe programs, at no additional cost. We can therefore expect that in program repair we may recognize problems and apply solutions from classical program optimization.

File access repair is strikingly similar to partial redundancy elimination à la Paleri et al. [28], which moves expression evaluations in order to minimize the number of times each expression is evaluated. This optimization builds on partial availability and anticipability analyses made conditional on the disjunction of availability and anticipability. Our file access repair transformations deals with programs opening, closing and reading files. Reading or closing a closed file lead to abortion and so does opening an open file. The repair turns a potentially broken program into a safe program. It builds upon two may-analyses. These detect, for any program point and any given file, if it was possibly read in the past and not closed or opened later and if it will be possibly read in the future and not opened or closed before. If the analyses establish that a file is possibly read in both the past and the future of a program point, the repaired program will have the file open at this point. In all other cases, the repaired program will be closed. In this way, not only are programs repaired wrt. file access errors but file session lengths are minimized.

The repair of queue-operating programs can be seen as enforcing portability. The error-compensating semantics arises from an implementation of the queue that resolves queue over/underflow in an ad-hoc way. The error-admitting semantics simply aborts on queue over/underflows. Repairing a program soundly with respect to this error-compensating semantics makes the program portable from that particular implementation to any implementation agreeing with the normal evaluations of the error-admitting semantics.

We also argue that a useful variation of program repair is enforcing of coding conventions. Here, the standard semantics of the programming language is taken to be the error-compensating semantics while as the error-admitting semantics one employs an instrumentation of the standard semantics that monitors adherence to the coding conventions and aborts as soon as a violation is detected.

Chapter 8

Conclusions

In this deliverable we reported about how to embed type-based analyses into program logics. We provided different methods to integrate type-based systems into a program logic with application areas ranging from information-flow to resource control to bi-directional dataflow analyses.

The work presented in the previous chapters throws light on the relationship between type-based analyses and program logics from different angles. Inspired by the approximate reasoning of type-based analyses, the lazy abstraction-based embedding presented in Chapt. 2 achieves full automation. At the same time it manages to maintain a reasonably degree of precision and, thus, value-sensitivity for a number of cases that can currently not be treated by type-based approaches.

Relational bytecode correlations presented in Chapt. 3 allow for a descriptive, extensional formulation of non-interference generalising current type-based methods. A particular property of the presented logic is to track copies of values and their flow through objects and methods. The presented approach features heap-local reasoning facilities and allows to prove interesting properties in particular about recursive data structures.

Chapt. 4 presented a prototype of a tool generating the byte-code modelling language annotations for encoding a type-based analyses. The chosen technology allows for a direct integration of the approach into the MOBIUS tool suite.

The work presented in Chapt. 5 investigates embeddings specialised to represent type-based approach used for reasoning about resources. They observed that for type-based systems that allow to annotate resource constraints on method parameters, a stricter version of a Hoare logic is needed.

Chapt. 6 and Chapt. 7 investigate foundational aspects of the interplay between Hoare logics and type-based systems. They discovered that the latter can be seen as applied versions of the first ones. Based on that fundamental insight they developed a type-systematic description of bi-directional dataflow analyses preparing one foundation for optimising, proof-preserving compilers developed in WP 4. Further, they demonstrate how to interpret program repair as a sound program optimisation.

The different approaches allow to translate type-based derivation into proofs of the program logic or extend program logics allowing to incorporate the advantages of type-based systems like automation or efficiency by simultaneously increasing the range of verifiable programs.

Bibliography

[1] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In G. Morrisett and S. Peyton Jones, editors, Principles of Programming Languages, pages 91–102. ACM, 2006.

[2] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In Principles of Programming Languages. ACM Press, 2000.

[3] A. Banerjee and D. Naumann. Stack-based access control for secure information flow. Journal of Functional Programming, 15:131–177, March 2005. Special Issue on Language-Based Security.

[4] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference Java bytecode verifier. In Programming Languages and Systems: Proceedings of the 16th European Symposium on Programming, ESOP 2007, number 4421 in Lecture Notes in Computer Science, pages 125–140. Springer-Verlag, 2007.

[5] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In N. D. Jones and X. Leroy, editors, Principles of Programming Languages, pages 14–25. ACM Press, 2004.

[6] L. Beringer and M. Hofmann. Secure information flow and program logics. In IEEE Computer Security Foundations Symposium, pages 233–248. IEEE Press, 2007.

[7] Lennart Beringer. Relational bytecode correlations. 2009. Submitted. Available at `http://www.tcs.informatik.uni-muenchen.de/~beringer/rbc.pdf`.

[8] Lennart Beringer, Martin Hofmann, and Mariela Pavlova. Certification using the mobius base logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, Formal Methods for Components and Objects: Revised Lectures from the 6th International Symposium FMCO 2007, volume 5382 of Lecture Notes in Computer Science, pages 25–51. Springer-Verlag, 2008.

[9] Lars Birkedal and Hongseok Yang. Relational parametricity and separation logic. In Helmut Seidl, editor, Foundations of Software Science and Computational Structures, 10th International Conference, (FOSSACS 2007), Proceedings, volume 4423 of Lecture Notes in Computer Science, pages 93–107. Springer-Verlag, 2007.

[10] R. Bubel, R. Hähnle, and P.H. Schmitt. Specification predicates with explicit dependency information. In Proceedings of the 5th International Verification Workshop, volume 372, pages 28–43, August 2008. Available at `http://www.cse.chalmers.se/~bubel/doc/`.

[11] Richard Bubel. Secure information flow: Encoding declassification in a program logic. Research report, 2009.

[12] Richard Bubel, Reiner Hähnle, and Benjamin Weiß. Abstract interpretation of symbolic execution with explicit state updates, 2009. Submitted to FMCO'08. Available at `http://www.cse.chalmers.se/~bubel/doc/`.

[13] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In Workshop on Formal Methods for Industrial Critical Systems, volume 80 of Electronic Notes in Theoretical Computer Science, pages 73–89. Elsevier, 2003.

[14] J. Chrzaszcz, M. Huisman, A. Schubert, J. Kiniry, M. Pavlova, and E. Poll. BML Reference Manual, December 2008. In Progress. Available from `http://bml.mimuw.edu.pl`.

[15] B. Fischer, A. Saabas, and T. Uustalu. Program repair as sound optimization of broken programs. In IEEE and IFIP Int. Symp. on Theoretical Aspects of Software Engineering, pages 165–173. IEEE Press, 2009.

[16] M. J. Frade, A. Saabas, and T. Uustalu. Foundational certification of data-flow analyses. In IEEE and IFIP Int. Symp. on Theoretical Aspects of Software Engineering, pages 107–116. IEEE Press, 2007.

[17] M. J. Frade, A. Saabas, and T. Uustalu. Bidirectional data-flow analyses, type-systematically. In ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, pages 141–149. ACM Press, 2009.

[18] J. Fulara, K. Jakubczyk, and A. Schubert. Supplementing java bytecode with specifications. In T. Hruka, L. Madeyski, and M. Ochodek, editors, Software Engineering Techniques in Progress, pages 215–228. Oficyna Wydawnicza Politechniki Wroclawskiej, 2008.

[19] Reiner Hähnle, Jing Pan, Philipp Rümmer, and Dennis Walter. Integration of a security type system into a program logic. Theoretical Computer Science, 402(2-3):172–189, 2008. Available at `http://www.cse.chalmers.se/~reiner/pub.html`.

[20] S. Hunt and D. Sands. On flow-sensitive security types. In Principles of Programming Languages, Charleston, South Carolina, USA, January 2006. ACM Press.

[21] Bart Jacobs. Weakest pre-condition reasoning for Java programs with JML annotations. Journal of Logic and Algebraic Programming, 58(1–2):61–88, 2003.

[22] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML reference manual. Draft of May 2008, obtained from `http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html`.

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

[24] MOBIUS Consortium. Deliverable 3.2: Intermediate report on embedding type-based analyses into program logics, 2007. Available online from `http://mobius.inria.fr`.

[25] MOBIUS Consortium. Deliverable 4.5: Report on proof transformation for optimizing compilers, 2008. Available online from `http://mobius.inria.fr`.

[26] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In International Conference on Functional Programming, pages 62–73. ACM Press, 2006.

[27] Tobias Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, Proof and System-Reliability, pages 341–367. Kluwer, 2002.

[28] V. K. Paleri, Y. N. Srikant, and P. Shankar. Partial redundancy elimination: a simple, pragmatic, and provably correct algorithm. Science of Computer Programming, 48(1):1–20, 2003.

[29] John C. Reynolds. An overview of separation logic. In Verified Software: Theories, Tools, Experiments: Selected Papers from VSTTE 2005, number 4171 in Lecture Notes in Computer Science, pages 460–469. Springer-Verlag, 2008.

[30] A. Saabas. Logics for Low-Level Code and Proof-Preserving Program Optimizations. PhD thesis, Tallinn University of Technology, 2008.

[31] A. Saabas and T. Uustalu. Type systems for optimizing stack-based code. In M. Huisman and F. Spoto, editors, Bytecode Semantics, Verification, Analysis and Transformation, volume 190(1) of Electronic Notes in Theoretical Computer Science, pages 103–119. Elsevier, 2007.

[32] A. Saabas and T. Uustalu. Program and proof optimizations with type systems. Journal of Logic and Algebraic Programming, 77(1–2):131–154, 2008.

[33] A. Saabas and T. Uustalu. Proof optimization for partial redundancy elimination. In ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, pages 91–101. ACM Press, 2008.

[34] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. Journal of Computer Security, 2007.

[35] A. Schubert, J. Chrzszcz, T. Batkiewicz, J. Paszek, and W. Ws. Technical aspects of class specification in the byte code of java language. In To be published in Bytecode'08 proceedings, Electronic Notes in Theoretical Computer Science. Elsevier, 2008.

[36] H. Yang. Relational separation logic. Theoretical Computer Science, 375(1-3):308–334, 2007.